Hardware-Assisted Dependable Systems

Dissertation
submitted for the degree of
Doktoringenieur (Dr.-Ing.)

by
Dmitrii Kuvaiskii
born 28.11.1987 in Chigirik, Uzbekistan, USSR

Technische Universität Dresden

Faculty of Computer Science
Institute of Systems Architecture
Chair of Systems Engineering

Supervisors:
Prof. Dr. (PhD) Christof Fetzer
Prof. Dr. (PhD) Pramod Bhatotia

Submitted November 5, 2017

Note on Title

Based on the recommendation of the reviewers and the PhD committee, the initial working title of this thesis ("Dependable Systems Leveraging New ISA Extensions") was adapted to the final title named "Hardware-Assisted Dependable Systems". This title better reflects the actual content of the thesis in its general sense.

Acknowledgements

This PhD thesis would not be possible without guidance and support of many, many people.

First and foremost, I would like to thank my supervisors, prof. Christof Fetzer and prof. Pramod Bhatotia. Without their invaluable counsel and advice, I would not be able to achieve good results and finish my PhD so soon. Specifically, I would like to express my gratitude to prof. Fetzer for allowing me to pursue topics I was most interested in and for deep technical discussions early on in my PhD. Also, I am immensely grateful to prof. Bhatotia for teaching me how to conduct research and for supporting me on a day-to-day basis. Finally, I would like to thank both my supervisors for arranging the defense of my PhD at short notice, due to the specific circumstances of my visa.

I am also grateful to prof. Thorsten Strufe for being my Fachreferent and to prof. Pascal Felber for frequent collaborations and discussions. Other professors that helped me a lot include Herbert Bos, Cristiano Giuffrida, Herman Haertig, Jeronimo Castrillon, and Christel Baier.

I would like to thank all my colleagues and close friends for their endless support, reviews, comments, and patiently listening to my constant complains about everything. Thanks to Oleksii Oleksenko, Do Le Quoc, Maksym Planeta, Sergei Arnautov, Rasha Faqeh, Franz Gregor, Bohdan Trach, Irina Karadschow, Robert Krahn, Diogo Behrens, Raluca Halalai, Andre Martin, Ute Schiffel, Jons-Tobias Wamhoff, Frank Busse, Lenar Yazdanov, Thordis Kombrink, and Thomas Knauth. Special thanks go to my colleagues and fellow interns during my stay at Intel Labs: Mona Vij, Somnath Chakrabarti, Shweta Shinde, Palak Jindal, Mohammed Karmoose, Noor Abbani, and many others. And thanks to everyone who I did not explicitly mention here.

I am blessed to have amazing parents and a loving sister. Thank you, Anastasia, for showing me how to put up a fight. Thank you, dad, for making me love math and physics and for that Pentium-II computer we bought in 1998. Thank you, mom, for forcing me out of my comfort zone and for everything you did.

Thank you, Daria, for the past six years and for being awesome.

Abstract

Unpredictable hardware faults and software bugs lead to application crashes, incorrect computations, unavailability of internet services, data losses, malfunctioning components, and consequently financial losses or even death of people. In particular, faults in microprocessors (CPUs) and memory corruption bugs are among the major unresolved issues of today. CPU faults may result in benign crashes and, more problematically, in silent data corruptions that can lead to catastrophic consequences, silently propagating from component to component and finally shutting down the whole system. Similarly, memory corruption bugs (memory-safety vulnerabilities) may result in a benign application crash but may also be exploited by a malicious hacker to gain control over the system or leak confidential data.

Both these classes of errors are notoriously hard to detect and tolerate. Usual mitigation strategy is to apply ad-hoc local patches: checksums to protect specific computations against hardware faults and bug fixes to protect programs against known vulnerabilities. This strategy is unsatisfactory since it is prone to errors, requires significant manual effort, and protects only against anticipated faults. On the other extreme, Byzantine Fault Tolerance solutions defend against all kinds of hardware and software errors, but are inadequately expensive in terms of resources and performance overhead.

In this thesis, we examine and propose five techniques to protect against hardware CPU faults and software memory-corruption bugs. All these techniques are *hardware-assisted*: they use recent advancements in CPU designs and modern CPU extensions. Three of these techniques target hardware CPU faults and rely on specific CPU features: Δ-encoding efficiently utilizes instruction-level parallelism of modern CPUs, Elzar re-purposes Intel AVX extensions, and HAFT builds on Intel TSX instructions. The rest two target software bugs: SGXBounds detects vulnerabilities inside Intel SGX enclaves, and "MPX Explained" analyzes the recent Intel MPX extension to protect against buffer overflow bugs.

Our techniques achieve three goals: transparency, practicality, and efficiency. All our systems are implemented as compiler passes which transparently harden unmodified applications against hardware faults and software bugs. They are practical since they rely on commodity CPUs and require no specialized hardware or operating system support. Finally, they are efficient because they use hardware assistance in the form of CPU extensions to lower performance overhead.

Publications

The content of this thesis is based on the following publications.

- **Δ-encoding: Practical Encoded Processing**. Dmitrii Kuvaiskii and Christof Fetzer. In proceedings of *the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2015. Best Student Paper award.
- **Elzar: Triple Modular Redundancy using Intel AVX**. Dmitrii Kuvaiskii, Oleksii Oleksenko, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. In proceedings of *the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016.
- **HAFT: Hardware-assisted Fault Tolerance**. Dmitrii Kuvaiskii, Rasha Faqeh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. In proceedings of *the European Conference on Computer Systems (EuroSys)*, 2016.
- **SGXBounds: Memory Safety for Shielded Execution**. Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. In proceedings of *the European Conference on Computer Systems (EuroSys)*, 2017. Best Paper award.
- **Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches**. Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Submitted to *the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2018.

Contents

List of Figures

List of Tables

1 Introduction

Modern software systems are complex beasts. From embedded systems powering autonomous cars to MapReduce frameworks crunching numbers in data centers, our software gets bigger, in terms of lines of code (LoC) and of interacting components [55].

Consider an example of Boeing 787 Dreamliner shipped in 2010: software that powers this plane contains 7 million LoC, spread among 10,000 sensors and ECUs, generating over 15GB of data for each flight [242]. Another example is the amount of software found in a modern high-end car like Tesla Model S first presented in 2012: at least 100 million LoC and growing each year [228]. If we look at Amazon or Google data centers, we will see thousands of nodes executing billions of lines of code and producing results in a manner of milliseconds [55]. Perhaps the best-known example of software complexity is the evolution of the Linux kernel: Linux version 1.0 was released in 1994 and contained 176 thousand LoC, steadily increased its code base and by 2003 (version 2.6.0) it contained almost 6 million LoC, jumped to 12 million LoC in 2011 (version 3.0) and currently contains around 20 million LoC (version 4.13) contributed by almost 14,000 developers [81].

All this code executes on commodity hardware powered by Intel, AMD, and ARM microprocessors (CPUs). Even domains that traditionally used specialized hardware solutions – e.g., cars and trucks with the advent of autonomous driving – now switch to commodity architectures based on ARM and x86 [112]. The complexity of modern CPUs may even eclipse that of modern software. During 45 years of CPU history, the number of transistors exploded from mere 2,300 in the Intel 4004 microprocessor to 7 billion in the recent Intel Broadwell-EP Xeon (a factor of 3,000,000). Not only the raw number of transistors on CPUs increased, but also their architectures evolved in complex ways. In addition to several big leaps – increasing the word size from only 4 bits to modern 64 bits, adding superscalar, out-of-order, and speculative execution, moving to multi-threads and multi-cores – there were a number of smaller-scale changes to CPU architectures, such as additional instructions for SIMD processing, virtualization, and security. Microprocessor designs became so complex that the Intel's Software Developer's Manual contains 4,744 pages of dense explanations on the x86 architecture [105] and the famous optimization manuals by Agner Fog span 628 pages describing obscure features of x86-based CPUs [221].

What does this huge complexity of software and hardware mean to application developers and end users? *Software complexity* directly translates to software bugs and vulnerabilities: the 2014 report from Coverity shows that there are around 0.7 bugs per each 1,000 lines of code [59]. This number would translate to almost 5,000 software bugs in a Boeing 787 plane and 70,000 in a Tesla car. Even the high-quality Linux codebase contains around 5,000 bugs, with around 1,000 of them classified as potential high-impact vulnerabilities [59]. *Hardware complexity*, together with shrinking transistor sizes (from 10,000 nm in Intel 4004 to 14 nm in Intel Skylake), leads to sporadic hardware glitches: bits flipped in RAM or CPU registers, stale values in CPU flip-flops, or stuck-at bits in CPU caches. Several large-scale studies indicate that hardware faults occur at a surprisingly high rate and tend to reappear more frequently after the first occurrence [103, 165, 203]. Furthermore, the advancements in dark silicon-based 8 nm chip technology with fluctuating voltages is forecasted to further deteriorate the reliability of CPUs [211].

There are numerous real-world examples of software bugs and hardware faults that cause

financial damage or even death of people; we will name only a few prominent examples.

Software bugs manifest themselves in all technology domains and are frequently escalated to software vulnerabilities exploited by hackers. In the aerospace domain, the aforementioned Boeing 787's software contained an integer overflow bug that could lead to a complete loss of control in the air and required a shutdown every 248 days [230]. Bugs and vulnerabilities of hardware/software systems of modern cars – including Tesla S – are considered a serious threat to life of a driver and passengers and the main impediment to wide adoption of autonomous driving [94]. A single software bug at the scale of such giants as Amazon and Google can lead to outages of entire data centers and millions of dollars lost in revenue [86, 220]. Especially threatening are the bugs in operating system kernels, which can be exploited to gain complete control over a machine and disrupt all computations or steal confidential data; such bugs are still routinely found in the Linux kernel [208].

Hardware faults are more intricate. Most of the time these faults lead to observable machine crashes and can be easily fixed. Indeed, many embedded systems employ triple/dual modular redundancy (TMR/DMR) and watchdogs to detect crashed nodes [23], while online services are increasingly using techniques such as state machine replication [201] for tolerating crashes [22, 41, 102]. However, more insidious faults are Silent Data Corruptions (SDCs) – hardware faults that lead to erroneous computation results. If not treated properly, these faults can result in catastrophic consequences. Anecdotal evidences from internet services show that data corruptions in hardware can lead to process state corruption [47], data loss [163], and in some unfortunate cases, errors propagate throughout the system causing outage of the entire service [8].

The increasing rate of hardware faults and software bugs is already changing the way software systems are designed today. Several studies show that it is common to use ad-hoc mechanisms to detect data corruptions caused by hardware faults, such as source code assertions, periodic background integrity checks, and message checksums throughout the system [2, 8]. Software bugs are usually fixed by applying a specific, ad-hoc patch in a particular piece of code; ironically, around 14–24% of these fixes introduce new bugs in the same application [253]. These ad-hoc solutions have their drawbacks: they notably require extra manual effort to write the checks and bug fixes and can only protect from errors anticipated by the programmer.

At the same time, the developers shy away from using comprehensive principled approaches such as Byzantine fault tolerance (BFT). Though BFT solutions can tolerate both hardware and software errors, they are cumbersome and inefficient, incurring high performance and maintenance overheads due to an overly pessimistic fault model [31, 222].

Thus, we come to the crux of the problem tackled in this thesis: *How to detect and tolerate hardware faults and software bugs in a disciplined manner and with low overhead?* To meet the low-overhead requirement, we rely on the following observation: modern commodity CPUs possess underutilized resources and unused extensions which can be re-purposed for fault tolerance (to protect against hardware faults) and security (to protect against software bugs). Thus, our proposed solutions are *hardware-assisted*: they are designed and implemented in such a way as to utilize underlying hardware – modern Intel x86-64 CPUs in particular – in a most efficient manner.

As we will see in the next chapters, hardware-assisted approaches significantly outperform their software-only counterparts, usually by more than 30%. But to appreciate the advantages of hardware assistance, we first need to understand the current state of hardware. In particular, we will briefly outline the history, current trends, and features of commodity CPUs.

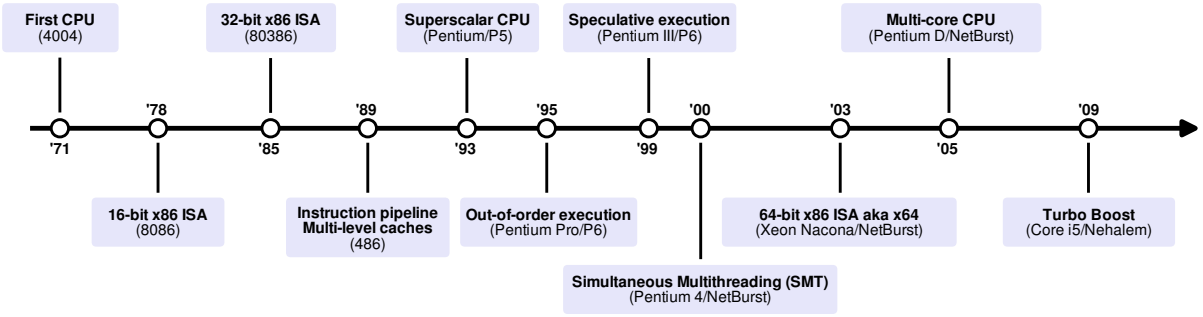

Figure 1.1 – Timeline of features introduced in commodity-hardware CPUs. Name of the first microarchitecture where a particular feature was implemented is given in brackets. For simplicity, only Intel CPUs are shown.

1.1 Brief history of CPUs

All solutions presented in this thesis rely on the features of and extensions to commodity-hardware CPUs. In this thesis, we dissect modern Intel microarchitectures and Instruction Set Architectures (ISAs) to understand their performance implications and propose low-performance-overhead systems based on this understanding. For example, Δ-*encoding* heavily relies on Instruction-Level Parallelism achieved via such CPU advancements as deep instruction pipelines, superscalar out-of-order execution, and speculative capabilities based on accurate branch prediction (Chapter 3); *Elzar* and *HAFT* build upon Intel AVX and TSX extensions respectively (Chapters 4 and 5); *SGXBounds* targets Intel SGX environments (Chapter 6); *Intel MPX Explained* discusses performance and security aspects of the Intel MPX feature (Chapter 7).

Before we delve into details of particular technologies we utilize in this thesis, we need to give a bigger picture and understand broader patterns in the development of CPUs and their extensions. What follows next is a very brief history of CPU evolution on the example of the biggest player in this field – Intel and its x86 microprocessors (see Figure 1.1).[1]

The very first CPU was developed in 1971. It was a tiny, crude, 4-bit Intel 4004 that could perform simple arithmetic operations and was primarily used in calculators. This CPU had 16 4-bit general-purpose registers, 46 instructions, and could directly address 4KB of ROM and 640B of RAM. In those early days, executing a single instruction took 5 cycles, which translates to 0.2 instructions per cycle (IPC). It was a long way to modern CPUs with Instruction-Level Parallelism (ILP), with their IPC of 4-5. Intel 4004 did not feature a CPU cache – the memory–CPU gap began to widen only in the 1980s, but in 1971, accessing ROM/RAM was *not* slower than reading from CPU registers.

Fast-forward to 1978 and we see a wide variety of sufficiently powerful and commercially available CPUs, with Intel, Motorola, Zilog, and other major players competing for the emerging Personal Computer (PC) market. Intel releases its 8086 CPU, the logical successor to the 8-bit 8080 described as "the first truly useful microprocessor", and the direct predecessor to 8088 used in the very first IBM PC.

With Intel 8086, the x86 Instruction Set Architecture (ISA) first comes into being. Even though x86 is widely considered inelegant and many attempts were made to replace it with better instruction sets (including attempts by Intel itself with its iAPX432 and recently Itanium), it remains the dominant family of ISAs to this day, with only one notable rival – RISC-based ARM. The x86 design follows the Complex Instruction Set Computing (CISC) model, meaning that a

[1] Contents of this and the following sections are based on data from `wikipedia.org` and `intel.com`.

single x86 instruction can execute several micro-operations, e.g., the `inc mem-addr` reads a value from memory, increments it, and stores it back at the same memory address. The prominent feature of x86 is its 100% backwards-compatibility, such that a modern 64-bit x86 incarnation (aka x86-64 aka x64) is still able to run unmodified 16-bit programs written in 1978.

Intel 8086 possessed 8 16-bit general-purpose registers, with familiar labels of AX, BX, SI, DI, etc. 8086 also featured x86 memory segmentation to access up to 1MB of memory. For performance, 8086 introduced the first step towards instruction pipelining: the fetch and execution stages were separated with the help of a prefetch queue, allowing to execute a previous instruction simultaneously with fetching a next one. Still, 8086 lacked many features we take for granted in modern CPUs: no security and isolation (privilege levels or rings in the protected mode arrived later with Intel 80286) and no CPU caches.

Another milestone in the processor design was Intel 80386 aka i386 aka 386 released in 1985. At that time, IBM PC already popularized affordable personal computers, and 80386 became the heart of many high-end PCs of the 1980s (in fact, Intel ceased its production only in 2007 because of the chip's astonishing success in the embedded market). In 80386, the major improvement over state-of-the-art was the debut of the 32-bit x86 ISA: its instruction set, programming model, and binary encodings are the de-facto standard of x86 CPUs to this day. 32-bit word sizes enabled native support for 4GB address space, single-precision floating point arithmetic, and integers up to several billions – enough for a vast majority of applications during 1980s and 1990s. Other important innovations included the addition of a fully-fledged Memory Management Unit (MMU) to support virtual memory, four privilege rings in protected mode allowing for clear separation of user/kernel spaces, and debug registers serving as hardware breakpoints.

After word sizes plateaued at 32 bits for the next two decades, CPU designers switched to solving the issues of growing CPU–RAM performance disparity and of Instruction-Level Parallelism (ILP). To solve the CPU–RAM disparity, first CPU caches were introduced in Intel 80386, with different chips containing up to 16KB of first (and only) level cache. The iconic Intel 80486 microprocessor aka i486 aka 486 introduced two-level CPU caches: two internal level-1 (L1) caches, one for instructions and one for data, and one off-chip L2 cache. It also introduced the first of the many developments towards high ILP – the tightly-coupled instruction pipeline. This pipeline introduced the usual five stages of x86 instruction execution: (1) instruction fetch, (2) main instruction decode, (3) operand prefetch and memory address computation, (4) execution, and (5) write-back. Thanks to this, up to five instructions could be executed tightly one after another, each occupying one of the above stages.

Instruction pipelining can significantly boost CPU throughput, roughly doubling CPU performance at the same clock rate. Moreover, pipelines can be made "deeper" by adding more and more stages; modern Intel CPUs have 14-16 stages. However, they still have two bottlenecks. (1) The execution stage may require much more than one CPU cycle to execute a computation-intensive instruction, thus holding off other pipeline stages. (2) A data dependency between two consecutive instructions – so-called hazard – may delay the processing of the second instruction. The next ten years of CPU evolution saw the solutions to both these issues.

The first bottleneck stemmed from the fact that Intel 80486 and its predecessors had only one execution unit (EU). In 1993, to solve the bottleneck, the first Pentium (dubbed P5, i.e., the fifth generation of Intel microarchitecture) introduced superscalar execution. Superscalar CPUs contain more than one EU and thus can execute several instructions simultaneously. In particular, Intel Pentium duplicated the whole instruction pipeline, with one pipeline able to handle any x86 instruction but the other one only the most common simple instructions. Modern CPUs have a more intricate design, with one pipeline and several buffers to pass instructions

and their data in-between stages. The number of execution units also increased dramatically – a modern Intel Skylake has four integer arithmetic EUs, three floating-point arithmetic EUs, two branch EUs, two EUs for loads, and one for store.

Two years later, in 1995, the Pentium Pro CPU (with a P6 microarchitecture) introduced a solution to the bottleneck of data hazards in the pipeline – out-of-order execution. Recall that a program is nothing more but a sequential stream of instructions. In this sequential mode of execution, if one instruction's input is the output of the preceding instruction, the whole pipeline has to stall and dispatches the second instruction only after the first one wrote-back its output in the final stage. Out-of-order execution breaks this paradigm and allows to execute instructions in the order other than the original order in a program. Achieving out-of-order execution is non-trivial and requires register renaming in hardware, reservation stations for execution units, and a re-order buffer to store instructions with unresolved dependencies. However, this technique drastically reduces the number of stalls and increases ILP.

The last bottleneck of instruction pipelining comes from branches: earlier CPUs had to halt the pipeline whenever a conditional branch instruction was executed since it was impossible to predict which instruction should be dispatched next. In 1999, Pentium III (based on the same P6 microarchitecture) added speculative execution to the arsenal of mechanisms designed to increase ILP. Speculative execution relies on branch prediction: the CPU simply guesses which branch will be taken and dispatches instructions from that branch in the pipeline. If the guess is correct, the CPU can continue its "speculated" execution path. If the guess was wrong, the CPU needs to flush the wrongly speculated instructions and start executing the right branch. Fortunately, most branches in programs are easily predictable, and branch predictors of current CPUs are correct in 97% of cases.

With this final invention, the flow of ideas to increase instruction-level parallelism stopped. In comparison to primitive CPU designs of 40 years ago which could achieve a maximum of 0.2 IPC, a modern Intel Skylake CPU has a theoretical maximum of around 5 IPC. Nowadays, even though each new generation of Intel CPUs increases the number of stages in the instruction pipeline, number of instructions held in decode queues and reorder buffers, number of execution ports, as well as the quality of branch predictors, these incremental improvements came to a point of diminishing returns.

One of the final advances in CPU design came with the advent of Simultaneous Multithreading (SMT) and multi-core CPUs. SMT – or Hyperthreading in Intel parlance – was introduced in 2000 in Pentium 4 (with a NetBurst microarchitecture) and allows to multiplex two logical threads of execution on a single physical core. This technique is completely oblivious to the software running on top: the CPU reports to the operating system that it has two cores instead of only one. SMT builds on a simple observation: normal programs do not exhibit a high level of instruction parallelism and thus cannot occupy all stages and execution units of a CPU pipeline. Thus, a superscalar CPU becomes underutilized. However, with SMT, instructions from two different programs are multiplexed on a single CPU and can be executed in any given pipeline stage at a time. SMT-enabled CPUs can see up to 30% performance improvement in comparison to non-SMT ones.

In contrast to SMT where a CPU "pretends" to have two physical cores, multi-core CPUs – first released in 2005 – actually integrate two or more separate cores in a single chip package. In a true dual-core CPU, two programs or two threads of the same program actually execute in parallel, not sharing any resources except for the L3 cache. Similar to Instruction-Level Parallelism (ILP), multi-core designs enable Thread-Level Parallelism (TLP) where multiple application threads pinned to separate cores increase performance several-fold. Modern Intel

CPUs such as Xeon Phis have up to 60 cores.

In parallel to the ILP/TLP chase, Intel CPUs slowly transitioned to the 64-bit architecture called x86-64 or x64, starting from 2003 and the NetBurst microarchitecture.[2] 64-bit wide words solved the problem of limited address space for long (in fact, modern CPUs use no more than 48 bits for memory addresses). It seems unlikely that commodity CPUs will require a switch to 128-bit word sizes in the near future.

Finally, one of the last significant innovations in CPUs was dynamic voltage scaling – the Turbo Boost technology for Intel CPUs introduced in 2009 in Core i5 processors. Turbo-Boost-enabled processors can automatically raise CPU's operating frequency depending on the workload. Under intensive load, the operating system requests higher frequency. During prolonged idle states, the operating system decreases the CPU's frequency to the minimum. This allows to conserve power and the amount of heat produced by the processor, at the same time achieving peak throughput when needed.

There are several new trends in CPU designs such as open-source hardware movements and tight integration of CPUs with GPUs and FPGAs (accelerators). However, we will not discuss them, since they are not relevant for the purposes of this thesis. In fact, the CPU features that will be important for us are (1) 64-bit word sizes, (2) multi-core processors, and (3) instruction-level parallelism via deep pipelines and superscalar out-of-order execution.

1.2 Brief history of CPU extensions

We have seen in the previous section how Intel CPUs evolved into high-performance high-complexity chips. The evolution was driven by two main factors. First, the increase in word size was the main source of innovation in the 1970s–1980s. However, with the advent of 32-bit CPUs, it took 20 years to move to 64 bits, and it will take another 30–50 years to 128-bit CPUs be required to meet our demands. Second, the pursuit of higher and higher instruction-level and thread-level parallelism (ILP and TLP) drove the innovation in 1990s and 2000s. However, with the ILP wall evident in the late 1990s and the extensive development of multi-core CPUs in the 2000s, CPU manufacturers moved to another design option: CPU extensions.

CPU extensions (or ISA extensions) are different from the CPU advancements reviewed in the previous section in the following. First, CPU extensions are limited, self-contained sets of new instructions with a clearly defined usage domain (in contrast to general-purpose instructions of the x86 ISA). For example, Intel MPX introduces 7 instructions for fast bounds-checking of memory ranges. Second, CPU extensions usually provide a set of new CPU registers targeted for use by this extension's instructions. Continuing the same example, Intel MPX provides 4 new registers to store bounds; they can be accessed only via MPX instructions. Third, CPU extensions are "opt-in": only a subset of CPUs from the same generation may support a specific extension, any CPU extension is fully compatible with other x86 instructions and other extensions, and a program may be built with or without a CPU extension without the need to rewrite its code (in the ideal case).

The timeline of CPU extensions is shown in Figure 1.2. Names of extensions are given in bold, their categories (usage domains) in italics, and the first CPU architecture that supported a particular extension is given in brackets. We concentrate only on Intel CPU extensions.

[2] Though we concentrate solely on Intel microarchitectures in this section, we must note that the first x86-64 architecture was actually developed by Intel's rival – AMD (that is why x86-64 is also sometimes called AMD64).

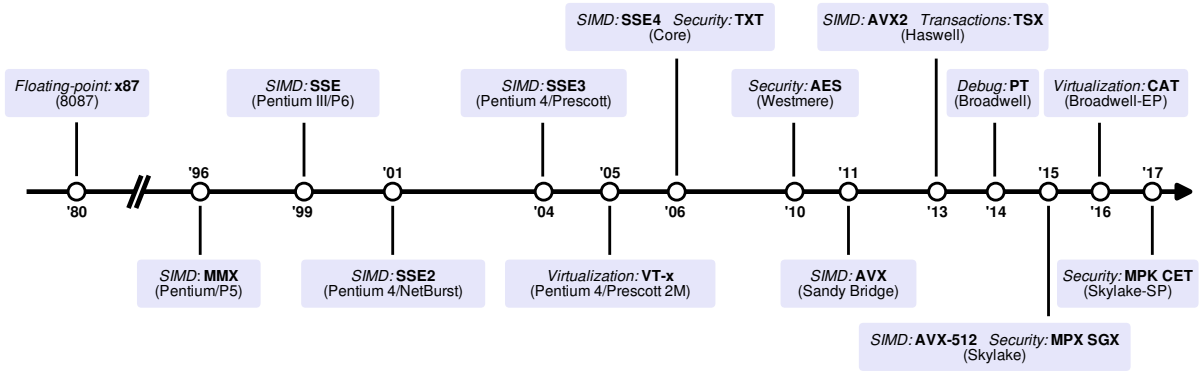

Figure 1.2 – Timeline of extensions introduced in commodity-hardware CPUs. Name of the first microarchitecture where a particular extension was implemented is given in brackets. For simplicity, only Intel CPUs are shown.

Arguably, the first CPU extension was the x87 floating-point instruction set, released in 1980 as a companion to the Intel 8086 CPU. x87 was quite literally a CPU extension – it was delivered as an optional floating point coprocessor. As any other CPU extension, x87 introduced a set of new registers and instructions, and was not strictly needed to run programs. Instead, with the help of a compiler, it allowed x87-enabled assembly to run much faster when executing math-heavy workloads. x87 extension lived through several generations and became redundant with the advent of SIMD extensions like SSE and AVX.

The most ubiquitous and widely used CPU extensions is the Single Instruction Multiple Data (SIMD) family of extensions. SIMD is a paradigm to execute the same instruction on several data points in parallel. For example, a 256-bit SIMD addition operates on four 64-bit pairs of integers and produces four 64-bit sums, all in parallel and in one CPU cycle. SIMD extensions introduce wide registers that keep more than one data point and wide instructions operating on these registers. Strictly speaking, SIMD extensions are not required to run programs, but they are so prevalent that even modern Linux kernels assume their presence by default.

The first incarnation of SIMD in Intel processors was Intel MMX in 1996. MMX introduced eight 64-bit registers able to hold two 32-bit integers, four 16-bit, or eight 8-bit ones (recall that at that time all CPUs were 32-bit). MMX also defined 40 new instructions to operate on these registers. Note that MMX did not provide floating-point operations and thus CPUs still relied on a slow x87 coprocessor.

The second incarnation of SIMD was Intel Streaming SIMD Extensions (SSE), first released in 1999 as an extension to Pentium III. Intel SSE contained 70 new instructions, with most of them targeted to single-precision (32-bit) floating point data (floats). It also added eight 128-bit registers XMM0–XMM7 able to hold either four floats or two 64-bit/four 32-bit/eight 16-bit/sixteen 8-bit integers. Intel SSE had four versions, with SSE2 introduced in 2001, SSE3 in 2004, and SSE4 in 2006. Each new version added new instructions and new registers, with SSE4 totaling 300 instructions and 16 registers.

The current SIMD incarnation is Intel Advanced Vector Extensions (AVX) first supported in Intel SandyBridge CPUs shipped in 2011. Intel AVX introduced 16 256-bit-wide YMM registers and a new three-operand instruction format. The second version of AVX, AVX2, added more instructions, and AVX-512 introduced 32 512-bit-wide ZMM registers. The number of instructions in AVX-512 is far above 500. Chapter 4 covers Intel AVX in more detail.

In-between new versions of SIMD extensions, in 2005, Intel released a set of instructions for better support of virtualization called VT-x. Intel VT-x adds 10 new instructions to enter and

exit a virtual execution mode in which the Virtual Machine Monitor (VMM) runs. Another recent extension for virtualization is Intel Cache Allocation Technology (CAT) that allows to dynamically partition the last-level cache (LLC) among several applications. This partitioning improves performance for workloads with real-time guarantees and strict isolation of data among several Virtual Machines (VMs). Both Intel VT-x and Intel CAT are out of scope of this thesis.

A rather small CPU extension called Intel Transactional Synchronization Extensions (TSX) was released in 2013 as part of the Haswell microarchitecture. Intel TSX implements Hardware Transactional Memory (HTM) – a technique to simplify concurrent programming by executing a sequence of load/store instructions in an atomic way [100]. TSX provides two interfaces for transactional execution: Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM). HLE adds two instruction prefixes (not new instructions!) which can be used with a subset of common x86 instructions with memory operands. RTM is an alternative to HLE that introduces three explicit instructions to start, abort, and end a hardware transaction. Both interfaces allow for optimistic transactional execution and can lead to 40% higher performance in comparison to traditional lock-based solutions. We provide detailed discussion of Intel TSX in Chapter 5.

The rest CPU extensions we examine serve debugging and security purposes. Intel had its first and largely unsuccessful attempt in security extensions with Intel Trusted Execution Technology (TXT) in 2006. Intel TXT allowed to attest the authenticity of a computer and its operating system to the end user and used a Trusted Platform Module (TPM) technology coupled with cryptographic techniques. For this, a single TXT instruction must be executed at boot-loading time: this instruction triggers the authentication process of TXT that securely measures code, data, configuration, and other information loaded into memory. Intel TXT was shown to be prone to various attacks and did not gain momentum; it is now superseded by a more elegant Intel SGX technology.

In 2010, Intel proposed a small Advanced Encryption Standard (AES) instruction set. Intel AES has 7 instructions that accelerate encryption and decryption operations of the Advanced Encryption Standard. This CPU extension was quickly incorporated in major crypto-libraries and showed performance improvements of up to 8 times.

Another interesting CPU extension was released in 2014 with Broadwell CPUs and targets the root cause of most security vulnerabilities – bugs in applications. Intel Processor Trace (PT) dynamically builds a detailed trace of all activity happening during program execution, including branches taken and memory accessed. Intel PT can be triggered on specific actions and can be configured with filtering capabilities to dump only the desired information. This thesis does not utilize the above three extensions, and we refer reader to appropriate Intel documentation for details.

The year 2015 witnessed several promising CPU extensions released as part of Intel Skylake microarchitecture. Two of them concerned security of applications: Intel Memory Protection Extensions (MPX) and Software Guard Extensions (SGX). The first of these two, Intel MPX, aims to protect legacy C/C++ applications from their bugs being exploited. In particular, Intel MPX includes 7 new instructions to perform explicit bounds checks on memory addresses and four registers that store these bounds. The second, Intel SGX, is a much broader extension that allows to create an opaque region of memory cryptographically protected from any other software (including privileged software like the operating system and the hypervisor) and remotely attest it. Thus, Intel SGX introduces a new mode of operation: user code and data can be completely protected even if executed in a malicious environment with compromised OS and physical attacks on RAM, network, and hard drive. We cover Intel SGX in Chapter 6 and Intel MPX in Chapter 7.

Finally, two very recent CPU extensions are Intel Memory Protection Keys (MPK) and Control-flow Enforcement Technology (CET). Intel MPK augments each page in the page table with four bits so each page can be assigned one of 16 "key" values. For example, one page may have an associated key write-disabled and another read-disabled. Thus, the application can dictate access permissions at the level of separate pages. The other extension, Intel CET, copes with the problem of Return-, Call-, and Jump-Oriented Programming (ROP, COP, and JOP) attacks. CET introduces a hardware-based shadow stack that negates ROP attempts and indirect branch tracking that negates COP- and JOP-based attacks. These two security extensions were not available at the moment of this writing and are not covered in the thesis.

Looking at Figure 1.2, we note two trends. The first one is that CPU extensions prior to 2010s were largely confined to the SIMD domain. In fact, these extensions are so common nowadays that they are frequently considered an integral part of CPUs. The second trend shows that prior to around 2013, all extensions (with an exception of an ill-fated Intel TXT) served to increase performance by optimizing certain operations. However, in the last five years, Intel released six CPU extensions that provide isolation, debugging, and security guarantees (PT, MPX, SGX, CAT, MPK, and CET). This trend reaffirms our conclusion that starting from 2010s, CPU designers shifted their focus to novel CPU extensions.

We must note that even though we discussed only Intel CPUs and their evolution of CPU extensions, other companies provide similar technologies. For example, SIMD extensions can be found in all other CPUs: IBM's AltiVec for PowerPC, AMD's 3DNow! and SSE/AVX implementations, and ARM's NEON. As another example, AMD revealed its answer to Intel SGX: Secure Memory Encryption (SME), Secure Encrypted Virtualization (SEV), and hardware-based SHA-powered security coprocessor. ARM released its attempt on security extensions called TrustZone already in 2005. Thus, it is generally possible to port programs written for a specific Intel extension to other CPUs and architectures, e.g., from AMD and ARM.

In this thesis, we build on the following Intel CPU features and extensions:

- Δ-encoding (Chapter 3) uses *Instruction-Level Parallelism* in the form of superscalar out-of-order execution with branch predictors and deep pipelines.
- Elzar (Chapter 4) uses *Intel AVX* to detect and mask CPU faults using triple modular redundancy.
- HAFT (Chapter 5) uses *Intel TSX* to detect and roll-back CPU faults using transactional memory.
- SGXBounds (Chapter 6) uses *Intel SGX* to protect applications from outsider (privileged-level) and insider (software bugs) attacks.
- "MPX Explained" (Chapter 7) discusses how *Intel MPX* detects buffer overflows in legacy applications.

1.3 Scope and goals

In this thesis, we propose and evaluate hardware-assisted systems that bring *dependability guarantees* to applications. We limit our scope to two specific problems in dependability: (1) hardware faults occurring in CPUs and leading to silent data corruptions, and (2) software faults (bugs) leading to memory errors that can be exploited in a hacker attack. As we described above, both these problems are ubiquitous in real world and lead to disastrous consequences.

We concentrate specifically on *CPU* hardware faults leading to data corruptions because these faults are the ones with no established protection mechanisms [85]. In contrast, hardware faults that lead to crashes are extensively researched, and numerous techniques are adopted in

practice, including consensus protocols like Paxos [47], ZooKeeper's Zab [102], and Raft [176], and redundant nodes for fail-over when one machine fails [2, 115]. Hardware faults happening in RAM, storage, and networks are also thoroughly investigated and have settled protection mechanisms: parity bits and Error-Correcting Codes (ECC) for RAM [103], Redundant Array of Independent Disks (RAID) for storage [180], retransmission protocols for networks, and checksums for all three kinds of faults. Thus, we target the class of faults not covered by previous techniques – hardware faults occurring in CPUs. Chapters 3–5 of this thesis describe our proposed solutions to prevent CPU faults.

We also concentrate specifically on *memory corruption* software bugs which are exploitable by malicious attackers [225, 233]. Memory corruption bugs (or memory-safety bugs) such as buffer overflows and out-of-bounds reads/writes are among the top-3 security risks according to the MITRE ranking [1]. Even though there are many proposals to protect against these bugs both from academia [6, 17, 63] and industry [110, 162, 207], the "silver-bullet" solution is yet to be developed. Chapters 6–7 describe two techniques to detect memory corruption bugs.

All dependability systems described in this thesis aim to achieve three major goals:

- **Transparency.** Most software that requires dependability guarantees is written in C/C++ and has a long history of development. Thus, we aim to develop systems that can be applied to legacy software, without any modifications to the code base. For this, we implemented our systems as compiler passes which transparently "harden" existing application code against hardware faults and software bugs.
- **Practicality.** Many previous approaches to detect hardware and software faults failed to gain attention either because of too-high overheads or unrealistic assumptions of special hardware/operating system support [85, 225]. Realizing this, we developed techniques that can work on existing OSes and CPUs and require no specialized hardware.
- **Efficiency.** As we described in previous sections, modern commodity-hardware CPUs possess features and extensions that can be beneficial for fault tolerance and security. Therefore, instead of developing generic systems that would work on a variety of hardware platforms but with high performance costs, we build solutions tailored to exercise and/or re-purpose existing CPU extensions with low overhead. This way we achieve efficient hardware-assisted execution without sacrificing practicality of our techniques.

1.4 Contributions

The contributions of this thesis are as follows:

- Design, implementation, and evaluation of three systems for fault tolerance:
  - **Δ-encoding**: source-to-source compiler to detect transient and permanent CPU faults in legacy C programs utilizing unused ILP resources of modern CPUs (Chapter 3);
  - **Elzar**: LLVM compiler pass to detect and mask transient CPU faults in multithreaded legacy C/C++ programs using Intel AVX extension (Chapter 4);
  - **HAFT**: LLVM compiler pass to detect and tolerate transient CPU faults in multi-threaded legacy C/C++ programs using Intel TSX extension (Chapter 5);
- Design, implementation, and evaluation of two systems for security:
  - **SGXBounds**: LLVM-based bounds checker to detect and tolerate security bugs in multithreaded legacy C/C++ programs inside Intel SGX enclaves (Chapter 6);
  - **MPX Explained**: detailed analysis of Intel MPX and discussion of its applicability in comparison to other bounds-checking approaches (Chapter 7).

2 General Background

Dependability of software systems has a long history.

On the one hand, dependability in the form of protection against hardware faults (i.e., fault tolerance) can be traced back to the very first, analog general-purpose computing devices. The first known fault-tolerant computer was built in 1951 in Czechoslovakia – the SAPO, "automatic computer" [215]. This computer used triple modular redundancy to vote on the outcomes of three Arithmetic Logic Units (ALUs) running in parallel. Since then, many fault-tolerant designs were proposed and implemented, especially for spacecrafts of NASA [18, 215]. Even though protection against hardware errors is most important in safety-critical systems such as aircrafts, nuclear power plants, and railroad systems, fault-tolerant features slowly made their way into commodity computers. Nowadays, commodity-hardware systems may include ECC-protected memory, store their data on RAID hard drives, and communicate via a reliable TCP/IP network stack. The only part of hardware for which there is no common fault-tolerant solution is the CPU itself. We devoted the first part of this thesis to exactly this remaining problem, where we presented Δ-encoding, Elzar, and HAFT techniques to combat sporadic CPU faults.

On the other hand, dependability in the form of protection against software faults (i.e., systems security) developed much later, as a reply to the quick spread of computer viruses [145]. Arguably, the first malicious program to exploit software bugs was the Morris worm, spread by Robert Tappan Morris in 1988 [177]. The Morris worm exploited several known vulnerabilities in Unix command-line tools such as "sendmail", "finger", and "rexec" to infect the victim computer and propagate it further on the Internet. Nowadays, the unsolved problem of zero-day exploits and ubiquitous software vulnerabilities leads to high-profile attacks such as Heartbleed [12] and Cloudbleed [185] that can affect 17% of all Internet's web servers and cost up to $500 million [227]. Among different kinds of software bugs exploited, memory corruption bugs that violate memory safety are the most important and disastrous ones [225]. The second part of this thesis describes memory-safety solutions to detect and tolerate memory corruptions, with SGXBounds and Intel MPX techniques discussed in detail.

In what follows, we introduce common terminology used throughout this thesis and briefly discuss the current landscape of techniques to defend against hardware and software faults.

2.1 Common terminology

For the sake of completeness, we introduce the terminology and taxonomy of dependable systems used throughout this thesis. For definitions and discussions, we heavily rely on the classic paper "Basic Concepts and Taxonomy of Dependable and Secure Computing" by Avizienis et al. [19]. For the purposes of this thesis, we present only a minimally needed subset of all definitions and illustrate them with examples directly related to our work. On several occasions, we cite directly from this paper, since some definitions are impossible to formulate better.

Throughout this thesis, we casually assume that *dependability* is an umbrella term for *fault tolerance* to protect against hardware faults and *systems security* to protect against software faults. Loosely, we could code it in math terms: "dependability = fault tolerance + systems

security". We provide more rigorous definitions for these terms and their applicability to our work below. First, however, we need to introduce the necessary concepts of a *computing system*, its *service* and its *state*, as well as the defitions for *faults*, *errors*, and *failures*.

2.1.1 Computing system, its service, and its states

In this thesis, we develop techniques that add dependability guarantees to existing, unmodified C/C++ programs. More rigorously, our developed techniques provide the dependability property to existing computing systems.

Computing system is a single entity interacting with other entities (other systems). We call other systems the *environment* of our system; these other systems include the physical world, other hardware, software, and humans. The computing system can be characterized by several fundamental properties: *functionality*, *performance*, *dependability*, and *cost*.

In this thesis, we present techniques that add dependability guarantees to arbitrary unmodified C/C++ programs. Thus, for our purposes, a computing system is a legacy C/C++ program that receives inputs from its environment (network, command-line arguments, files, or humans), processes them, and sends outputs back to the environment.

The properties most relevant to our thesis are performance and dependability. By the performance property we imply performance and memory overheads with respect to original, unmodified programs. In particular, we measure performance overhead as the overhead in runtime (in seconds, for standalone programs) or in throughput and latency (in messages/second or seconds respectively, for server programs). We measure memory overhead as the overhead of peak actively used (i.e., working set size) or reserved (i.e., total allocated) memory of a program. By the dependability property we imply the ability of the system to continue its correct execution in presence of faults. We measure dependability qualitatively as integrity/security guarantees (well-known, representative faults being prevented by our techniques) and quantitatively as fault coverage (number of faults successfully prevented). We are not interested in the functionality and cost properties mentioned above because we do not develop programs and do not change their original functions, but rather transparently add dependability.

Service delivered by a system is the way the system implements its functionality as perceived by user. Strictly speaking, to deliver its service, the system moves through a sequence of **states**. One state encompasses all computation, stored information, communication, and physical condition of the system. For example, the C program's state is comprised of the currently executed instruction, all CPU registers and the state of CPU execution units (computation), CPU caches, internal CPU buffers, RAM, files on a hard drive (stored information), open sockets for input/output, network connections and packets currently sent through network (communications), and the state of all involved peripherals (physical condition).

The part of the system state that is visible to the user and can be modified by him/her is called *external state*, while the part hidden from the user and environment is called *internal state*. In our example of the C program's state, the external state comprises open sockets, connections, and files; the internal state comprises CPU registers, caches, RAM, etc.

2.1.2 Faults, errors, and failures

The ultimate goal of dependability is to ensure *correct service* of the system, i.e., the system executes correct functions on correct data at all times. Unfortunately, systems may sometimes misbehave: the system can experience a *failure* – an event when the delivered service deviates from correct service. In real world, a failure of a system always has a root cause (a *fault*) that

corrupts the state of the system (an *error*) and ultimately manifests itself in the observable failure.

Fault is an initial cause of an error in the system state. There are many kinds and classifications of faults; here we give an overview of only the most relevant ones.

Development faults are the faults occurring during system development. The typical example of a development fault is a software bug – an accidental mistake in code done by a careless programmer. Opposite to development faults, *operational faults* occur during service delivery by the system. A spurious bit-flip in a DRAM cell is an example of an operational fault during execution of a program.

Internal faults are faults originating inside the system itself, due to its internal defects. Software bugs are internal faults since they lurk in the incorrectly written program code. Faults due to hardware aging (e.g., stuck-at bits in CPU registers) are also internal faults since the system includes the CPU chip in its boundary. *External faults* originate outside the system and penetrate it via communication channels or interference. Bit-flips in memory chips due to cosmic rays are examples of external faults. We refer to internal faults that enable external faults to harm the system as *vulnerabilities*. For example, a software bug (internal fault) that is triggered by malicious input from an attacker (external fault) is escalated to a security vulnerability.

Transient faults occur sporadically and randomly for a brief moment in time. They affect different parts of the system state, i.e., they activate uniformly random errors. In contrast, *permanent faults* occur in the system and do not disappear with time. Somewhere in-between lie *intermittent faults* which occur sporadically and stay for a short time, possibly re-occurring again later. Permanent faults activate deterministic errors, infinitely affecting the same part of the state. Hardware bit-flips are examples of transient faults, while hardware stuck-at bits and software memory corruption bugs are permanent faults.

Soft faults are faults which are hard to impossible to reproduce. *Hard faults*, on the contrary, are faults which are reproducible. Most transient faults are soft, and most permanent faults are hard. However, this matching becomes blurred in complex software, when permanent faults can lead to spontaneous, non-reproducible errors (thus, they are soft faults).

Faults can be further classified into *natural* and *human-made*, *hardware* and *software*, *malicious* and *non-malicious* faults. With this classification, we can give rigorous definitions of faults that we examine in this thesis:

- **Hardware CPU/RAM faults** are operational external natural hardware non-malicious faults occurring in CPU and RAM components of the computing system. They can be both transient, intermittent, and permanent in nature. They also can be soft or hard.
- **Software memory-corruption bugs** are development internal human-made software malicious permanent hard faults occurring during execution of the computing system. Their root causes are mistakes in program code.

Error is an incorrect state (part of the total state) of the system. One example of an error is execution of a wrong instruction in the program due to a hardware fault in the Instruction Pointer (IP) CPU register. Another example of an error is a wrong value of some critical variable in the program due to a software memory corruption bug.

Faults are the root causes of errors: faults are said to *activate* errors. Note that not every fault leads to an error; such faults are called *dormant*. For example, a bit-flip in a CPU register that is not used by the program does not do any harm to the system.

It is important to note that errors can *propagate* through successive states of the system and corrupt greater and greater parts of state. For example, a single buffer overflow bug may be exploited multiple times to eventually dump all program memory through a network (this can

result in a huge data leak such as Heartbleed [227]).

Failure is a visible to the user deviation from correct service behavior. Crash or hang of a computing system is a common kind of failure: the system does not respond to user requests and is thus useless. Systems that exhibit only this kind of failure are called *fail-stop*. More insidious kind of failure is a *Silent Data Corruption (SDC)*: the system continues its execution but provides incorrect results to the user. Systems that never crash and never produce SDCs are called *fail-safe*.

Errors (corrupted system states) are causes of system failures: errors are said to *propagate* to failures. As with faults, not every error leads to a failure; such errors are called *latent*. One example is when a buffer overflow bug is neutralized by a subsequent sanity check in the program.

Failures can be classified as *content failures* and *timing failures*. Content failures mean that the content delivered to the user is incorrect and deviates from the expected results. Timing failures mean that the content is delivered to the user either too early or too late. In this thesis, we only consider content failures and neglect timing ones.

For dependability, all failures happening in the system must possess the *detectability* property: the system must signal to the user that a failure occurred. It is important that dependable systems do not have false positives (*false alarms*) and false negatives (*unsignaled failures*). In this thesis, we develop techniques that always provide detectability of failures.

Failures can also by classified by their consequences. Some failures may affect only availability, i.e., the duration of outage of the system before it runs again. Other failures may compromise confidentiality, i.e., leak confidential data. Yet other failures may reduce integrity of the system state, i.e., corrupt state and force the system to produce nonsense data. In this thesis, we mainly aim to preserve system integrity in spite of failures.

Finally, failures can be *minor* or *catastrophic*. Minor failures entail consequences of similar cost to the benefits provided by the service. Catastrophic failures, in contrast, entail consequences that are very harmful: they are orders of magnitude higher than the benefits provided by the service. Techniques in this thesis concentrate on preventing catastrophic failures like the ones mentioned in Chapter 1.

2.1.3 Dependability, fault tolerance, and systems security

Now that we introduced all necessary definitions, we can discuss the concepts of dependability, fault tolerance, and systems security.

Dependability of a system is "the ability to avoid service failures that are more frequent and more severe than is acceptable" [19]. Dependability is a broad concept that encompasses the following characteristics of the system:

- **Availability** is readiness for correct service.
- **Reliability** is continuity of correct service.
- **Safety** is absence of catastrophic consequences for the user of the service.
- **Integrity** is absence of incorrect state and corrupt data.
- **Maintainability** is ability to undergo repairs and modifications.

Security is another concept that includes availability and integrity characteristics from dependability and adds *confidentiality* – absence of unauthorized disclosure of information. Note that even though we discuss confidentiality in SGXBounds (Chapter 6), our primary goal is never to develop protocols for confidentiality. In fact, by enforcing integrity in SGXBounds, we automatically enforce confidentiality.

In this thesis, we concentrate only on the reliability, safety, and integrity attributes of dependability (hence the title of this thesis). In particular, we develop techniques that enforce (1) reliability in the sense of tolerating faults and continuing correct execution, (2) safety in the sense that the service never leads to catastrophic loss of data, money, or lives, and (3) integrity in the sense that the service is always correct and never produces corrupt results.

We make a distinction between two kinds of dependability: *fault tolerance* and *systems security*. In case of fault tolerance, we assume only hardware CPU/RAM faults. Therefore, reliability, safety, and integrity of our fault-tolerance techniques guarantee correct and continuous execution of the program in spite of hardware faults. In case of systems security, we deal with software memory-corruption bugs. Thus, reliability, safety, and integrity of our systems-security techniques imply correct and continuous execution of the program in spite of software bugs.

Being it fault tolerance or systems security, we must separate two phases of protection against faults: *fault detection* and *fault recovery*. Fault detection is the first phase when a fault/error is detected by some form of redundancy built-in in the program execution. Fault recovery is the second phase when a detected fault/error is removed or masked to allow continuous execution. In this thesis, Δ-encoding and Intel MPX provide only fault detection, while Elzar, HAFT, and SGXBounds additionally support fault recovery.

Lastly, this thesis employs only one means to attain dependability: fault tolerance. Strictly speaking, fault-tolerance techniques *avoid* service failures in the presence of faults. Other means include *fault prevention* (preventing occurrence of faults), *fault removal* (reducing the number and severity of occurred faults), and *fault forecasting* (estimating the future likelihood and consequences of faults). All these means are out of scope of this thesis; refer to [19] for their discussion.

2.2 Hardware faults and fault tolerance

In the first part of this thesis, we introduce defenses against hardware faults such as bit-flips and stuck-at bits. In particular, we concentrate on faults in CPU and RAM as discussed in the previous chapter.

Hardware faults in CPU and RAM are probabilistic and occur in random parts of the program at random execution moments. This dictates uniform protection of the whole program during the whole execution. To provide such protection, a form of redundancy needs to be introduced at some level of the software-hardware stack. Protection against hardware faults using some form of redundancy is generally referred to as fault tolerance.

Two classes of redundancy are possible: spatial or temporal redundancy. Spatial redundancy implies replication of some hardware component of the system, e.g., running two or three separate machines that compute over the same data. Temporal redundancy implies replication in time, i.e., one hardware component computes over the same data twice or thrice, one computation after the other. (In fact, there is no strict division of techniques on only-spatial and only-temporal, e.g., a single technique can repeat the computation twice on one CPU but using different sets of CPU registers.) In either case, at some critical points in computation the outputs of replicas must be compared: if they are the same, then no fault happened and the system may continue its execution, otherwise the system must signal an error and crash (fail-stop) or try to tolerate it (fail-safe).

There are several classic approaches to detect hardware faults in CPUs and RAM: dual and triple modular redundancy, lock step CPUs, state machine replication, and local software-based hardening. In what follows, we briefly describe each of these approaches. Note though that the

defenses we develop in this thesis – Δ-encoding, Elzar, and HAFT – fall under the local-hardening class of techniques with mostly temporal redundancy.

2.2.1 Dual and Triple Modular Redundancy

Dual and Triple Modular Redundancy (DMR and TMR) are classical approaches for achieving fault tolerance in safety-critical systems [144]. Initially they were used only at hardware level, but later were also adapted to software applications.

DMR employs two separate replicas that perform the same computation on the same inputs and periodically compares outputs before critical operations. For example, DMR can be implemented using two CPUs working in tight lock step (see next section). Another example of DMR is the system with two cores on the same CPU chip which execute the same program and periodically synchronize before outputting results. No matter what the implementation, a DMR system assumes that only one of two replicas can be faulty at any moment in time, and that this discrepancy in the output is detected by simple comparison. Consequently, DMR can only detect hardware faults but not tolerate them.

TMR, on the other hand, runs three replicas and detects faults by comparing three replicas' outputs and additionally performs fault recovery by majority voting, i.e., by detecting which replica differs from the other two and correcting its state. It imposes an obvious restriction on the fault model: only one replica is assumed to be affected by the fault. In certain circumstances, any two or even all three replicas may be incorrect. In this case, TMR can only perform detection of faults but cannot tolerate them since replicas cannot agree upon one correct state.

2.2.2 Lock step CPUs

Traditionally, hardware faults such as bit-flips were detected via lock step CPUs, where two CPUs execute the same program in parallel and synchronize and compare their outputs. Clearly, lock step CPUs are a specific form of dual modular redundancy. Lock step CPUs are still actively used for critical applications in the embedded domain and on mainframes. As an example, HP NonStop [28] divides a multi-core in multiple logical single core systems and maps programs to two cores with the help of binary rewriting and support from specialized hardware.

Unfortunately, lock-stepping requires deterministic core behavior and is not readily applicable to modern CPUs that have become increasingly more non-deterministic [28]. Therefore, applications running in lock step cannot harness the power of multiple cores (which would lead to non-deterministic execution of multi-threaded programs).

Moreover, lock step CPUs provide only fault detection, requiring a separate mechanism for fault recovery. Finally, the cycle overhead of lock step CPUs/cores is at least 100%, i.e., we need twice the number of CPU cycles to execute a program. This overhead is usually prohibitively expensive in cloud and data center environments.

It must be noted that the same issues pertain to three CPUs working in lock step (a form of triple modular redundancy): it is impossible to run non-deterministic multithreaded programs and performance overhead becomes prohibitive.

2.2.3 State Machine Replication

To achieve high availability, many systems [22, 41, 102] use State Machine Replication (SMR) [201]. These systems typically assume a fail-stop model, where the only type of faults happening

are machine or process crashes. Unfortunately, this model does not cover transient faults which might lead to arbitrary state corruptions, which is the main focus of this thesis.

On the other hand, Byzantine Fault Tolerance (BFT) [45] tolerates machine crashes as well as transient hardware faults that lead to state corruptions (and even malicious attacks). Unfortunately, BFT incurs prohibitive overheads because of the overly pessimistic fault model. For example, PBFT [45] requires $3f + 1$ replicas to tolerate f faults. To reduce the number of replicas, systems like MinBFT [235] and CheapBFT [118] use a hybrid fault model and require only $2f + 1$ nodes. To additionally decrease the performance overhead of BFT, researchers explored the use of specialized trusted hardware [118, 235], relaxed network assumptions [181, 182], speculative execution of requests [124], and OS support [122]. Nonetheless, BFT techniques are still considered too expensive and are not used in practice.

A common challenge for all SMR solutions is multithreading. To support multithreaded programs, SMR techniques require some form of deterministic execution. For example, Crane [61] builds on top of deterministic multithreading [139, 175], Eve [119] speculatively executes requests and falls back to deterministic re-execution in case of conflicts, and Rex [89] enforces deterministic replay of the primary's trace on secondary replicas. Unfortunately, all these approaches are cumbersome and introduce additional performance overheads.

2.2.4 Local Software-Based Hardening

Local software-based hardening techniques can be broadly divided into three categories: Thread-Level Redundancy (TLR) also called Redundant Multithreading (RMT), Process-Level Redundancy (PLR), and Instruction-Level Redundancy (ILR).

Redundant Multithreading (RMT). In RMT approaches [153, 257], a hardened program spawns an additional trailing thread for each original thread. During runtime, trailing threads execute on separate spare CPU cores or take advantage of the Simultaneous Multithreading (SMT) capabilities of modern CPUs. RMT allows keeping only one memory state among replicas (assuming that memory is protected via ECC). Unfortunately, RMT approaches heavily rely on the assumption of spare cores or unused SMT, which is commonly not the case in multithreaded environments where programs tend to use all available CPU cores.

RMT approaches make use of multiple execution blocks available in modern CPUs by running redundant copies of a program on multiple threads. Before each memory operation, its operands have to be checked for consistency among threads, which causes a significant performance overhead (roughly 200%). DAFT [257] reduces the effect of this issue by having non-blocking memory accesses, that is, by executing memory operations asynchronously with checks. Its successor RAFT [258] goes even further and monitors replicas' behavior only at the system call level. This technique shows only 2.8% average overhead. That being said, all approaches from this category still require deterministic behavior of the program to perform consistency checking.

Process Level Redundancy (PLR). PLR implements the similar idea as RMT, but at the level of separate processes [214, 258]. In PLR, each process replica operates on its own memory state, and all processes synchronize on system calls. In multithreaded environments, allocating a separate memory state for each process raises a challenge of non-determinism because memory interleavings can result in discrepancies among processes and lead to false positives. Some PLR approaches resolve this challenge by enforcing deterministic multithreading [65]. PLR might incur a lower performance overhead than RMT but it still requires spare cores for efficient execution.

Instruction-Level Redundancy (ILR). In contrast to RMT and PLR, ILR performs replication *inside* each thread and does not require additional CPU cores [170, 191]. This is achieved by

replicating instructions of the original program and weaving them in the same thread for parallel execution. These instructions do not change the functionality of the program, but a fault in one replica will lead to a different result of the computation which can be detected by comparing replicas' results (checking). This in-thread replication seamlessly enables multithreading and requires no spare cores for performance.

EDDI [170] was the first implementation of ILR and exploited unused Instruction-Level Parallelism (ILP) available in modern processors to run replicated instructions in parallel. SWIFT [191] was the logical continuation of EDDI, adding control-flow protection and eliminating memory state replication by assuming an ECC-protected memory. With this set of optimizations, Swift showed impressively low performance overhead of 40% but used a VLIW-based Intel Itanium 2 processor[1]. Later research re-implemented Swift in x86 and observed overheads of 116% on average [255]. This high overhead indicates that ILR imposes high pressure on the CPU backend. Even worse, these numbers are reported for duplicated instructions, implying that instruction triplication results in significantly higher performance impact.

We should note that the techniques introduced in the first part of this thesis – Δ-encoding, Elzar, and HAFT – all rely on instruction-level redundancy for hardware fault detection.

2.3 Software faults and systems security

In the second part of this thesis, we develop and discuss defenses against software faults aka software bugs. In contrast to hardware faults, which are uniformly random and probabilistic in their nature, software faults are localized and deterministic: if a programmer introduced a bug in the program, then this bug will be triggered each time vulnerable code is executed. Thus, protection against software faults generally falls under the umbrella term of systems security.

The most common and arguably most important software bugs are *memory corruption bugs* [225]. This class of bugs pertains to unsafe languages such as C/C++ and covers all cases when a pointer in the program incorrectly points to a wrong object or to a garbage value. Whenever such a "poisoned" pointer is dereferenced, the value loaded from/stored to the pointed address is incorrect, i.e., this value is not what the programmer originally intended and not what the program expects. The result can be a segmentation fault and consequent crash of the program. Hackers can use this to launch Denial-Of-Service attacks (DOS attacks) and make programs unresponsive. However, a much worse result of an incorrectly dereferenced pointer is when the program continues execution but with the wrong loaded/stored value. In such cases, hackers can launch an attack to subvert execution, escalate their privileges to seize control of the whole system, or leak confidential data [185, 227]. Typical examples of memory corruptions include buffer overflows, off-by-one errors, direct and indirect out-of-bounds accesses, dangling and NULL pointers, use-after-free, etc.

To prevent memory corruption bugs, *memory safety* must be enforced: each and every pointer dereference (i.e., each and every memory access) must be checked that it still points to the valid and intended object in memory. Unsafe languages like C/C++ do not provide any means to add such checks per se, thus these checks must be retrofitted in legacy programs using a separate technique. In the second part of this thesis, we introduce two such memory safety techniques: SGXBounds (Chapter 6) and Intel MPX (Chapter 7).

In what follows, we give a quick overview of existing memory safety techniques as well as of other approaches to prevent (some subclasses of) memory corruptions.

[1] HP, the primary customer of Intel Itanium CPUs, switched to Intel Xeon in 2014. This effectively marks the end of the era of Itanium CPUs.

2.3.1 Memory safety

Memory-safety approaches prevent the very first step in any attack – exploiting a vulnerability, such as overflowing a buffer or freeing an already freed object. Thus, a comprehensive memory-safety defense can deterministically eliminate *all* memory corruption attacks.

Since we concentrate on memory corruptions in unsafe languages like C and C++, we focus solely on *spatial* and *temporal* bugs due to incorrect use of pointers to objects [225].

By spatial bugs we imply pointers going out of bounds of their initial object – buffer over-/underwrites and buffer over-/underflows (or simply buffer overflows) [158]. In the simplest cases, buffer overflows are caused by a pointer incremented past the initial object: the overflow is contiguous, because it can only read/write to immediately adjacent objects in memory. In the more complex cases, buffer overflows can corrupt objects located far away from the initial object, e.g., if there is a pointer arithmetic with an index variable that (maliciously) experiences integer overflow.

By temporal bugs we imply pointers which become dangling – dangling pointer dereferences, double frees, and invalid frees [157, 217]. In general, temporal bugs occur when an initial pointed-to object is deallocated and thus the pointer points to some invalid memory address. Dangling pointer dereferences access a memory region that probably contains a new object, and thus the new object may be maliciously overwritten or leaked. Double frees and invalid frees are more subtle but can also result in incorrect overwrites of objects.

Memory safety is the only approach that prevents the root cause of attacks, namely, memory corruption bugs made by careless programmers [225]. It is no surprise that memory-safety defenses made their way into specialized hardware [156] and recently culminated in Intel Memory Protection Extensions (MPX) [109].

All memory-safety approaches rely on additional metadata stored in shadow memory.[2] If metadata is associated with objects in memory, we call it memory-safety *object-based* defenses, e.g., AddressSanitizer [207], DieHard [27], Baggy Bounds [6]. In contrast, if metadata is associated with individual pointers to objects in memory, we call it memory-safety *pointer-based* defenses, e.g., SoftBound [158], CETS [157], Intel MPX [109].

Consider an example: a tiny program with one C struct object and ten different pointers into it. An object-based approach allocates one entry in shadow memory, marking the bounds (and liveness) of the object, usually by simply setting a bit for each 8-byte region of application memory. A pointer-based approach allocates ten shadow-memory entries, each associated with a specific pointer and containing its lower and upper bounds. For both object- and based-approaches, a check is inserted into the original program code before each pointer dereference. This check loads the corresponding metadata and compares the pointer value against the bounds and temporal information. If a check fails, an exception is generated and usually the program is crashed.

For memory-safety approaches to work correctly, they need to propagate metadata correctly throughout the whole program. It is rather straight-forward for object-based defenses: only memory allocation and deallocation functions must be augmented to populate and clear shadow memory. Since most C programs operate on memory using standard library functions “malloc” and “free”, it is enough to wrap only these functions. Pointer-based approaches require more instrumentation: each pointer-related instruction in the original program must be instrumented to also propagate pointer bounds information. In general, object-based approaches are far easier

[2] “Shadow memory” denotes additionally allocated application memory to store metadata for security checks. Usually, this is the dominant source of memory overhead exhibited by a particular defense. For some defenses, the base address of such shadow region must be randomized to protect shadow memory against malicious overwrites.

to implement and scale, while pointer-based approaches require whole-program instrumentation. In other words, pointer-based approaches suffer from worse modularity support and compatibility issues (e.g., dynamic or third-party libraries) [225].

At this point it must be clear that object-based techniques incur less memory overhead but are more coarse-grained than pointer-based ones. For example, Baggy Bounds Checking [6] incurs less than 10% memory overhead, while the most comprehensive pointer-based approach WatchdogLite [156] can require 4× more memory in the worst case. At the same time, object-based approaches cannot protect against intra-object overflows, e.g., in a struct that has an inner array. Pointer-based approaches allow so-called bounds narrowing to cope with exactly these cases. Moreover, pointer-based approaches implement temporal checks in a disciplined deterministic way, while object-based approaches resort to ad-hoc probabilistic protections.

2.3.2 Other approaches to prevent subclasses of memory corruption bugs

Memory safety is the only class of techniques to defend against *all* possible memory corruptions. Therefore, this thesis focuses only on memory-safety defenses. For the broader discussion, however, we mention other techniques that trade some security guarantees for better performance.

Address Space Randomization (ASR) [133, 225] is a very broad class of techniques that change the original address space layout of the program in a random way. The key observation is that many attacks require a precise knowledge of the address space, e.g., addresses of specific functions for return-into-libc attacks, addresses of gadgets for ROP attacks, layouts of the stack for stack-based attacks, etc. By randomizing the address space, we leave the attacker only tiny chances of guessing the correct layout.

ASR techniques can be classified into *coarse-grained* and *fine-grained*. Coarse-grained ASR randomizes only the base addresses of program and dynamic libraries segments. Its variants are currently deployed in most operating systems under the name Address Space Layout Randomization (ASLR). Fine-grained ASR, as the name suggests, performs much more fine-grained randomization of the layout: replacing instruction sequences with equivalent sequences, inserting garbage code (NOPs), permuting the order of basic blocks/functions, randomizing the layout of stack variables, heap objects, and struct representations, splitting basic blocks and functions, etc. [133].

Comparing these two classes, coarse-grained ASR incurs negligible performance overheads but can be easily broken via information leaks or relative-addressing attacks, whereas fine-grained ASR is harder to exploit but has observable overheads of $5 - 10\%$. Regarding coarse-grained ASR, its obvious weakness stems from the fact that only *base addresses* are randomized. Thus, a single leak of one address of a known-to-attacker function in the code segment is enough to reveal the complete virtual space layout of this segment and launch an attack. Regarding fine-grained ASR, this class significantly raises the bar for an attacker, since several information leaks are required for a successful exploit. On the negative side, fine-grained ASR introduces substantial memory overhead of up to $20 - 40\%$ [133].

In general, any ASR defense requires a high level of entropy so that the probability of an attacker guessing the correct layout is very low. In addition, ASR cannot fully protect against information leaks: the layout of data in memory can be changed, but the data is still stored in plaintext.

Code Pointer Integrity (CPI) [129] is a compiler-based defense to protect against control-flow hijacking attacks. Its main security goal is to prevent the attacker from modifying any code pointer (or, transiently, any pointer that can point to a code pointer) in the program. This

is achieved by splitting the whole address space of the application into disjoint *safe region* and *regular region*. All memory accesses that are proven at compile-time to operate on code pointers are redirected to the safe region and are instrumented with the usual memory-safety checks. The key idea in CPI is that code pointers constitute a minority (e.g., 6% for SPEC2006) of all pointers in the program. Thus, distinguishing these pointers and instrumenting only them significantly decreases performance and memory overheads in comparison to complete memory-safety approaches.

Performance overheads of complete CPI constitute 8% on average and up to 45% for pointer-intensive applications. However, the real problem is the introduction of the safe region: in the flat-array implementation and on x86-64, CPI requires doubling the size and also hiding the base address of the safe region using ASLR [74]. Moreover, CPI detects only control-flow hijacking and overlooks data-only attacks and information leaks.

Control-Flow Integrity (CFI) is a general approach to prevent ROP-style attacks and was introduced in mid-2000's by Abadi et al. [4]. In contrast to Code Pointer Integrity, CFI does not prevent the modification of a code pointer, but the *use* (dereference) of a maliciously modified pointer.

The initial workflow of CFI was as follows: (1) at compile-time, generate a precise control-flow graph (CFG); (2) based on CFG, assign equivalence classes to code pointer targets; (3) embed equivalence class identifier (a simple integer) in the code of each target; (4) insert a check before each code pointer dereference that compares the current code pointer value against the embedded-in-target equivalence class; (5) at run-time, the resultant executable self-checks itself.

This initial design assumes that the code is protected via Data Execution Prevention (DEP) [225] and cannot be modified by the attacker (that is why it is safe to keep identifiers directly in code). It also assumes that a fine-grained CFG can be obtained, which can only be satisfied with whole-program analysis. Unfortunately, even if these assumptions are satisfied, static CFI (also called *forward-edge CFI*) is not sufficient to protect from dynamic stack-based attacks. Original CFI suggests an additional *shadow stack* mechanism to prevent such dynamic attacks; such defenses are sometimes called *backward-edge* CFI.

After the introduction of the CFI concept, many implementations of CFI were introduced: some of them protect only forward edges, some add CFI at a binary level, some utilize virtual machines [40]. The initial definition of CFI became blurred, and nowadays many control flow hijacking defenses are referred to as CFI. For example, Cryptographically Enforced Control Flow Integrity (CCFI) [146] does not even make use of CFG, instead encoding all pointers at run-time. As of 2017, there are around 25 different CFI approaches, all with different design choices affecting their applicability, compatibility, performance and security guarantees.[3]

In general, CFI approaches introduce low performance overhead of $1 - 10\%$. Despite all CFI implementations differing in their respective threat models and security guarantees, the general consensus is that fine-grained CFI is a strong defense against ROP-attacks. Unfortunately, CFI does not protect against data-only attacks or leaks of confidential data.

Data-Flow Integrity (DFI) defends against control flow and non-control data attacks by instrumenting all writes and reads in the program [44]. DFI is a generalization of CFI because it inserts checks not only on code pointer dereferences, but on *all memory reads*.

DFI works in three phases: (1) at compile-time, a static points-to whole-program analysis is performed to identify sets of write-instructions for each read instruction; (2) the identified sets are compiled together into one shadow-memory table; (3) an update to a corresponding entry in

[3]We refer the reader to a survey of state-of-the-art CFI approaches [40].

the shadow-memory table is inserted before each write; (4) a check against the corresponding entry in the shadow-memory table is inserted before each read; (5) at run-time, the resultant executable detects all reads of incorrectly written objects.

Interestingly, even though DFI dictates checks only on reads, it must employ an additional mechanism to be sure that unchecked writes do not intentionally overwrite the shadow-memory table. Roughly speaking, a really protected DFI implementation must insert some form of checks on writes as well.

DFI is a very strong technique to protect against control-flow hijacking or data-only attacks. However, DFI cannot stop information leaks: even though a check is executed before an out-of-bounds read, the read value was correctly written by the program, and no alarm is raised. In addition, due to extensive instrumentation and the need to consult the shadow-memory table, DFI exhibits performance overheads of 100% and more. Memory overhead can peak up to 50% because of the shadow-memory table.

Data Integrity is a defense that (similar to Code Pointer Integrity) prevents the attacker from modifying any variables in the program. Naturally, Data Integrity is a superset of CPI, adding protection against non-control data attacks.

We are aware of only one implementation of Data Integrity, namely, Write Integrity Testing (WIT) [7]. WIT is a compile-time technique that builds on a whole-program points-to analysis to compute control-flow and data-flow graphs (CFG and DFG). Based on CFG and DFG, WIT identifies sets of objects that can be written by each write instruction in the program. At each write, WIT inserts a check to validate if a current written-to object belongs to the pre-calculated set. This way, WIT enforces write integrity, i.e., no instruction in the program can modify an incorrect object. To embrace dynamically allocated objects, WIT introduces a metadata table ("color table") which is stored in shadow memory and consulted on each appropriate write instruction.

Interestingly, WIT is an improvement over Data-Flow Integrity (DFI) discussed above. In contrast to DFI, WIT does not insert checks on read instructions but rather on writes. Thus, WIT introduces no instrumentation on reads at all, which leads to better performance at the cost of no protection against malicious reads.

WIT has average performance overheads of 15% and memory overheads of 13%. WIT does not prevent out-of-bound reads and thus suffers from information leakage. Additionally, points-to analyses may provide inaccurate over-approximated results and weaken WIT's security guarantees (similar to CFI).

Data Space Randomization (DSR) is a probabilistic defense that tries to prevent all possible attacks including information leaks [30, 42]. In contrast to ASR which strives to randomize only addresses where code and data reside, DSR randomizes the data itself in order to obfuscate it for a malicious attacker. This randomization usually implies XORing all pieces of data with a predefined secret mask on writes (XOR encryption) and XORing back with the same mask on reads (XOR decryption). DSR can be thought of as a weak but sufficient form of cryptographic encryption.

In contrast to other data-based defenses such as DFI and Data Integrity, DSR does not insert checks which crash the program on attack detection. Instead, DSR instruments the program such that all sensitive data is stored XOR-encrypted in memory and thus is useless for an attacker (she cannot understand the read values and she cannot inject her own values other than randomly). This interesting property of DSR allows for low-overhead instrumentation without any checks.

Ideally, DSR should generate a distinct XOR mask for each data object. However, due to the same imprecisions of a static points-to analysis as in cases of Data Integrity and DFI, DSR

introduces over-approximation of sets of data objects with the same mask.

Performance overhead of DSR is $10 - 15\%$ [42]. DSR does not produce and maintain any shadow-memory metadata and thus incurs no memory overhead.

One obvious limitation of DSR is its reliance on XOR masks being kept secret. If the attacker is able to read XOR masks from the code segment (where they are kept), she can launch any attack by injecting correctly XORed values. Likewise, the attacker can infer XOR masks by analyzing the leaked obfuscated data, thus obtaining confidential data in plain-text. Another issue is the treatment of dynamically allocated objects on heap. In general, these objects will end up in the same equivalence class and assigned the same XOR mask. Thus, corruptions from one heap object to another will not be detected.

Instruction Set Randomization (ISR) [120] is a class of binary-based defenses that obfuscate the original assembly of the program. The key idea is to XOR each instruction's opcode with a secret integer such that the attacker cannot inject his own malicious assembly (without knowing the secret, she can only inject some random instructions). This defense was invented to deflect code injection vectors. However, modern defenses such as Data Execution Prevention (DEP) [225] allow for non-writable code pages and non-executable data pages, which already removes any possibility of code injection. Therefore, ISR defenses became obsolete and we mention them here only for completeness.

3 △-encoding: Leveraging Instruction Level Parallelism

The first class of faults we concentrate on in this thesis are *hardware CPU and RAM faults*. More specifically, we are interested in those faults that lead to silent data corruptions (SDCs) without crashing the computer. Faults that result in a crash or a hang are out of scope of our research: these faults are trivially detected by a watchdog and tolerated by a simple reboot of a machine.

As mentioned in the previous chapters, there is no established commodity-hardware solution to protect against CPU faults—these faults are considered too rare and too insignificant to be worthy of attention. RAM faults, in contrast, are widely acknowledged as one of the major contributors to hardware glitches and wrong computations, with Error Correcting Codes (ECC) being a conventional technique to tolerate them. However, as we show in the next section, current assumptions about the rates and effects of CPU and RAM faults are routinely violated. CPU faults are more common than one would expect and their rate is predicted to increase drastically in the near future. RAM faults too are more intricate than one would anticipate, and ECC alone does not provide sufficient level of protection.

Thus, the first technique we describe in this thesis is Δ-encoding to detect transient, intermittent, and permanent CPU and RAM faults in legacy C programs. Δ-encoding provides very high (four-nines) fault coverage even in case of hard errors in CPU and error bursts in RAM. To achieve this, Δ-encoding relies on underutilized Instruction Level Parallelism (ILP) resources of modern CPUs, in particular, on deep instruction pipelining, out-of-order execution, and sophisticated branch prediction. In short, Δ-encoding effectively employs advanced features of commodity-hardware CPUs described in §1.1.

The content of this chapter is based on the paper "Δ-encoding: Practical Encoded Processing" presented at DSN'2015 [125]. The paper was a joint collaboration with Christof Fetzer.

3.1 Rationale

A dramatic decrease in hardware reliability, most importantly in CPUs and RAM, was forecast already in the 2000s [35]. This is due to the decrease of feature sizes with each new hardware generation, causing variations in transistor behavior. These variations, if not masked at the hardware level, can lead to silent data corruptions (SDCs) in a program. Moreover, additional effects such as transistor aging and soft errors (due to alpha particles and cosmic rays hitting silicon chips) increase the probability of a program to produce wrong results.

Recent studies provide supporting evidence for this forecast. Google analyzed DRAM failure patterns across its server fleet [203]. The research concluded that (1) DRAM failure rates are higher than previously expected[1], (2) memory errors are strongly correlated, and (3) memory errors are dominated by hard errors rather than by soft errors. Another study shows that even

[1] One third of machines under study experienced at least one correctable memory error per year; the annual rate of uncorrectable errors amounted to a significant 1.3%. Note that all memory modules were equipped with error correcting codes (ECC).

ECC-enabled DRAM chips do not provide adequate protection from the emerging problem of *disturbance errors*, when accesses to one DRAM row corrupt data in adjacent rows [123].

Similar findings were revealed in regard to modern CPUs. Microsoft conducted analysis of hardware failures on a fleet of 950,000 machines [165]. This work showed that (1) failure rates of modern CPU subsystems are non-negligible[2], (2) failure rates increase with the increasing CPU speed, and (3) CPU faults tend to be intermittent rather than transient. Unfortunately, the study considers only crash failures and not data corruptions in applications; other studies, however, indicate that CPU faults result in a non-trivial number of SDCs [137].

Many hardware errors, either in CPU or in memory, lead to a process or machine crash. Still, some hardware faults induce programs to output incorrect results, which can propagate further and lead to catastrophic consequences. One anecdotal evidence is the famous Amazon S3 unavailability incident, when a single bit corruption in a few messages caused an 8-hour outage [8].

The consequences are even more disastrous in safety-critical applications. As one example, Toyota Motor Corporation was forced to recall its automobiles in the years 2009–2011 after several reports that Toyota cars experienced unintended acceleration [243]. The number of victims was estimated to be 37, and financial loss for Toyota $2,470 million. Though the exact causes of the problem were not found out, insufficient protection against hardware errors could be one of them:

> "Michael Barr of the Barr Group testified that ... Toyota did not follow best practices for real-time life-critical software, and that a single bit flip which can be caused by cosmic rays could cause unintended acceleration."

Detecting hardware faults of all types is a necessity for applications from different domains. *Tolerating* faults once they are found can often be achieved by simply restarting the process or rebooting the machine: in most cases it is enough that incorrect computation results are not propagated to the outside. Therefore, we concentrate on *hardware error detection* in this chapter. (More precisely, we concentrate on detection of data corruptions that occur due to hardware errors changing a program's data flow.)

The conservative error detection approach, widely used in automotive and aerospace systems, is to employ some form of hardware-based fault tolerance. Usual mechanisms include triple/dual modular redundancy (TMR/DMR), flip-flop hardening, watchdogs, etc. [23]. The hardware-based approach, however, implies higher hardware costs and lower performance in comparison to modern commodity hardware. For example, Intel conjectures that future self-driving cars will require greater computing power and suggests to use commodity CPUs [104].

Another approach called Software-Implemented Hardware Fault Tolerance (SIHFT) [85] achieves fault tolerance via software-only methods; thus, it does not require specialized hardware. However, in spite of the experimental studies clearly indicating the prevalence of permanent and intermittent errors in CPU and memory, most SIHFT techniques assume only transient errors. In this sense, these techniques favor performance over fault coverage and cannot be relied upon in safety-critical systems.

One notable SIHFT technique that can detect both permanent and transient errors in underlying hardware is *encoded processing* [198]. It is based on the theory of arithmetic codes (AN-encoding) and was used in fault-tolerant computing [80]. Unfortunately, pure AN-encoding has limited fault coverage. Advanced variants of AN-encoding exist [198], but programs encoded with

[2]For example, the chance of a crash is 1 in 190 for machines with the total CPU time of 30 days.

them – namely with the *ANBD* variant – experience slowdowns of up to 250×. Thus, though ANBD-encoding yields very high fault coverage, it is impractical in terms of performance.

As a result, existing SIHFT techniques either do not detect all possible hardware errors or incur prohibitive performance penalties. This work makes a step towards hardening critical computations against permanent and transient hardware errors with a moderate performance penalty.

Our approach, called Δ-*encoding*, is based on the combination of AN-encoding and duplicate execution of instructions. The original program data flow is duplicated and AN-encoded at compile-time; at run-time, the program effectively works on two copies of data encoded in two different ways. The careful choice of AN-encoding parameters coupled with execution duplication greatly simplifies AN-encoded operations, improving the performance; moreover, the combination of approaches facilitates detection of all types of hardware errors.

We implemented Δ-encoding as a source-to-source transformer. Our fault injection experiments reveal that Δ-encoding can detect, on average, 99.997% of injected errors. Our performance evaluation shows that Δ-encoding incurs an acceptable slowdown of 2–4× in comparison to native execution.

3.2 Background

The Δ-encoding technique proposed in this chapter combines two existing approaches: *AN-encoding* and *duplicated instructions*. In this section, we briefly discuss both of them.

3.2.1 AN-encoding

AN-encoding is a technique to protect program execution from transient and permanent errors in the underlying hardware. It is based on AN codes – error correcting codes suitable for arithmetic operations [37]. Schiffel [198] describes AN-encoding and its variants in detail.

With AN codes, to encode an integer n, one multiplies it by a constant A. The resultant integer $\hat{n} = A \cdot n$ is called a *code word*; all words that are not multiples of A are *invalid*. If a hardware error alters $\hat{n}$, it becomes an invalid word with high probability; this probability depends on A [198]. If $\hat{n}$ is still a code word, $\hat{n}$ mod A equals 0; if the result of this operation is not 0, a hardware error is detected. To decode, a division $\hat{n}/A$ is used.

AN-encoding exploits information redundancy, i.e., additional bits are required to store an encoded integer. In practice, the number of bits to represent encoded integers is doubled.

As an example, consider the addition of two integers 5 and 3 (see Figure 3.1a). For simplicity, we choose $A = 11$. AN-encoded integers are thus $A \cdot 5 = 55$ and $A \cdot 3 = 33$. These code words can be directly added and result in a code word: $55 + 33 = 88$. Now, if a hardware error would cause any of the terms to become invalid, the sum will also be an invalid code. Listing Figure 3.1b shows an AN-encoded version of the original addition.

This example highlights two main properties of AN-encoding: first, operations on encoded inputs directly produce encoded outputs, second, errors in inputs propagate to outputs. The first property means that by substituting all original operations with encoded operations, the data flow of a program is protected against hardware faults. The second property implies that the encoded execution of a program does not require intermediate checks.

One drawback of AN-encoding is that not all operations on encoded values are easily implemented. As the previous example shows, encoded addition corresponds to the usual arithmetic addition; subtractions and comparisons are also trivial. However, encoded multiplication, division,

(a) Native program

```
1 int32_t a = 5;
2 int32_t b = 3;
3 int32_t c = a + b;
4 printf("%d", c);
```

(b) AN-encoded program

```
1 #define A 11
2 int64_t a = 5 * A;
3 int64_t b = 3 * A;
4 int64_t c = a + b;
5 if (c % A != 0)  raise_error();
6 printf("%d", c/A);
```

(c) Duplicated instructions program

```
1 int32_t a1 = 5;        int32_t a2 = 5;
2 int32_t b1 = 3;        int32_t b2 = 3;
3 int32_t c1 = a1 + b1;  int32_t c2 = a2 + b2;
4 if (c1 != c2)  raise_error();
5 printf("%d", c1);
```

(d) Δ-encoded program

```
1 #define A1 9           #define A2 7
2 int64_t a1 = 5 * A1;   int64_t a2 = 5 * A2;
3 int64_t b1 = 3 * A1;   int64_t b2 = 3 * A2;
4 int64_t c1 = a1 + b1;  int64_t c2 = a2 + b2;
5 if (c1 % A1 != 0 || c2 % A2 != 0 || c1/A1 != c2/A2)  raise_error();
6 printf("%d", (c1 - c2) >> 1);
```

Figure 3.1 – Example illustrating how a native program (a) is transformed using (b) AN-encoding, (c) duplicated instructions, and (d) our Δ-encoding.

bitwise operations, etc. require more sophisticated implementations. These complex encoded operations can hamper performance and/or require intermediate decoding of operands.

Another drawback of pure AN-encoding is that it does not detect all types of hardware errors. In our previous example, if the addition operation is erroneously substituted by subtraction, the result is still a code word, since $55 - 33 = 22$. Moreover, if one of the operands is replaced by some other code word (due to a fault on the address bus), the result is also a code word, e.g., $55 + 11 = 66$. To detect these types of errors, variants of AN-encoding were developed, namely ANB- and ANBD-encodings [199]. Unfortunately, they incur very high performance penalties (up to 250×) rendering them impractical in most use cases.

AN codes should not be confused with conventional linear codes such as Hamming codes or Reed-Solomon codes. Firstly, the linearity property does not hold in AN codes; secondly, linear codes are suitable for storage and transmission whereas AN codes are used in data processing.

In general, AN-encoding has the advantage of detecting both transient and permanent errors during program execution; a severe disadvantage is its low performance. Pure AN-encoding cannot detect all kinds of hardware errors and thus it does not provide high fault coverage. ANB- and ANBD-encodings do provide full fault coverage, but at the price of even higher performance overheads.

3.2.2 Duplicated Instructions

Fault tolerance can also be achieved by duplicating all original instructions in a program. The duplicates work with a second set of registers and variables, i.e., all data is also duplicated. During execution, "master" and "shadow" instructions are issued on the same processor; their results are compared periodically to check for hardware errors. Oh, Shirvani and McCluskey [170] provide detailed information about error detection by duplicated instructions.

Concerning our previous example of $5 + 3$, the addition operation is issued twice on the CPU, such that two copies use two different sets of registers. The check operation makes sure that both copies calculated 8, and if not, a hardware error is detected. Listing Figure 3.1c illustrates this.

The duplicated instructions approach assumes that hardware faults are transient and affect only one data-flow copy. For example, this approach cannot detect hard errors in the CPU. If the addition operation is permanently faulty, then $5 + 3$ can result in an incorrect value for both copies.

The duplicated instructions technique incurs only modest performance penalty of 60% [191], since additional instructions can be effectively scheduled by the compiler and executed by the CPU in an out-of-order fashion. Indeed, since "master" and "shadow" execution paths are independent of each other and require synchronization only at rare check points, the execution runs essentially in parallel on modern super-scalar processors.

On the whole, the approach of duplicated instructions enables comprehensive protection from transient errors, incurring only modest execution slowdowns. However, this approach cannot cope with permanent errors affecting both "master" and "shadow" copies of data flow.

3.3 Fault Model

We adopt a *data-flow software-level symptom-based fault model* from [198]. This model provides an abstraction of the underlying hardware and works on the "symptoms" caused by hardware errors at the software level. Such a model has several advantages: (1) it is independent from specific hardware models and thus applies to any combination of CPU/RAM, (2) it does not account for *masked hardware faults*, i.e., faults that are neutralized at hardware level, and (3) this fault model can be easily adapted for fault injection campaigns.

The model consists of the following symptoms:

Modified operand: One operand is modified, e.g., $55 + 33$ is changed to $51 + 33$. This happens due to a bit flip in memory/CPU register.

Exchanged operand: A different but valid operand is used, e.g., $55 + 33$ is changed to $55 + 11$. This happens due to a fault on the address bus.

Faulty operation: An operation produces incorrect results on specific inputs, e.g., $55 + 33$ results in 87. A CPU design flaw can lead to such a fault.

Exchanged operation: An operation that was not intended is executed, e.g., $55 + 33$ is changed to $55 - 33$. This happens due to a fault in the CPU's instruction decoder.

Lost update: A store operation is omitted, e.g., the result of $55+33$ is not stored in memory/CPU register; an outdated value from the memory/register is then erroneously used. This happens due to a fault on the address bus.

Many fault-detection approaches assume a *Single Event Upset (SEU)* fault model, where exactly one bit is flipped throughout program execution; in contrast, we make no assumptions on the number of bits affected by a hardware error or on the number of hardware errors during execution.

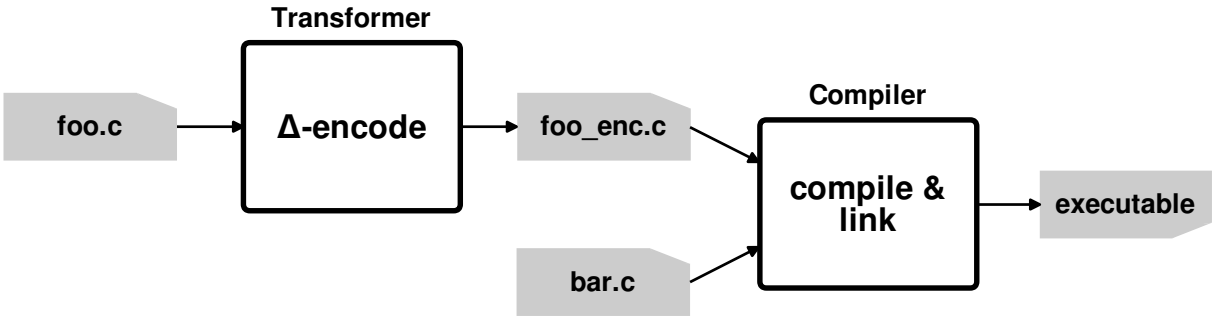

Figure 3.2 – Δ-encoded program.

We argue that the SEU model is unrealistic. First, as studies show ([203], [123]), modern RAM experiences not only transient bit flips, but also permanent faults. Second, another study [197] reveals that about 17% of hardware faults affecting the combination logic result in double or multiple bit errors. These results motivate the adoption of a fault model that has no error rate assumption: *any number* of errors of *any type* can happen during program execution. Our only assumptions are that errors occur randomly and corrupt a random number of bits.

Our fault model does not cover *control flow errors*, when a corrupted instruction pointer (IP) points to an unintended instruction address. Such faults have a very low probability of resulting in SDC. Nevertheless, our approach can be coupled with a control flow checker to detect both data and control flow errors.

Finally, the sphere of replication (SoR) [189] assumed in this work is the CPU and the memory directly used by the encoded program (or the encoded part of a program). The operating system as well as the disk and network subsystems are out of SoR; errors in these systems cannot be detected.

3.4 Δ-encoding

In this section, we describe Δ-*encoding*, a novel technique that combines AN-encoding and duplicated instructions. Δ-encoding borrows the ability to detect hard errors from AN-encoding; it uses the idea of duplicated instructions to achieve full fault coverage without sacrificing performance. Moreover, a clever combination of approaches allows to simplify AN-encoding, improving its performance.

Conceptually, Δ-encoding performs two compile-time transformations on the original program: first, all data is AN-encoded and all original operations are substituted by AN-encoded operations, second, all encoded data and operations are duplicated and checks are inserted at synchronization points. The result is a hardened program with two copies of a completely encoded data flow, as shown in Figure 3.2.

3.4.1 Encoded Data

To encode data in Δ-encoding, we set two different constants for the two copies of data: A_1 for the first encoded copy and A_2 for the second copy. Thus, the two copies of data flow operate on different values, i.e., our approach employs *data diversity*, which is beneficial for fault tolerance [10]. In particular, if a hard CPU fault triggers on some specific inputs, it will corrupt only one copy of the data, but not the other.

The key idea behind Δ-encoding is a smart choice of A_1 and A_2:

$$A_1 - A_2 = 1 \tag{3.1}$$

This choice of the constants enables us to decode values quickly, by subtracting the second encoded copy $\hat{n}_2$ from the first encoded copy $\hat{n}_1$ (hence the name Δ-*encoding*):

$$n = \hat{n}_1 - \hat{n}_2 = n \cdot A_1 - n \cdot A_2 = n \cdot (A_1 - A_2) \tag{3.2}$$

Note that this decoding requires only one instruction cycle; in contrast, decoding in pure AN-encoding is much more expensive, since it requires a division instruction. The division instruction is one of the most costly operations in modern CPUs. For example, according to the Intel IA-64 architecture manual, division takes 60-80 cycles to finish [107]. Our quick decoding is especially beneficial for programs that make heavy use of pointers because all pointers are kept encoded and must be decoded at each pointer dereference.

The choice of A_1 and A_2 in Equation 3.1 has a drawback: both copies of a value are decoded in the same way (by subtracting the A_2-encoded copy from the A_1-encoded copy). This can lead to SDC since a permanent fault affects both decoding operations in the same way. Thus, we push the idea further and use the following scheme to choose A_1 and A_2:

$$A_1 = 2^k + 2^i \qquad A_2 = 2^k - 2^i \tag{3.3}$$

where k and i are non-negative integers, $k > i$.

We notice that:

$$A_1 - A_2 = 2^k + 2^i - 2^k + 2^i = 2^{i+1} \tag{3.4}$$

$$A_1 + A_2 = 2^k + 2^i + 2^k - 2^i = 2^{k+1} \tag{3.5}$$

Based on Equations 3.4 and 3.5, there are two ways to decode a value:

$$n = (\hat{n}_1 - \hat{n}_2)/2^{i+1} = n \cdot (A_1 - A_2)/2^{i+1} \tag{3.6}$$

$$n = (\hat{n}_1 + \hat{n}_2)/2^{k+1} = n \cdot (A_1 + A_2)/2^{k+1} \tag{3.7}$$

The division by a power of 2 corresponds to the right shift instruction. Since we fix k and i beforehand, the number of bits to shift by is known at encoding time. As a result, decoding schemes 3.6 and 3.7 require only two cycles: one for subtraction/addition and one for right shift.

For example, let $k = 3$ and $i = 0$. Then $A_1 = 9$ and $A_2 = 7$; their difference is $A_1 - A_2 = 2$ and their sum is $A_1 + A_2 = 16$, and to decode one needs to shift right by $i + 1 = 1$ and $k + 1 = 4$ correspondingly. Our original code snippet from Figure 3.1a can be Δ-encoded with these parameters and results in an encoded program from Figure 3.1d.

Δ-encoding uses this scheme, with A_1 and A_2 chosen as in Equation 3.3 and decoding as in Equations 3.6 and 3.7. This scheme has two advantages: (1) decoding is much faster than in pure AN codes and (2) two different ways to decode a value will fail differently in reaction to the same permanent error.

In our final implementation, we chose $k = 13$, $i = 0$ and thus $A_1 = 8193$, $A_2 = 8191$ and shifts of 1 and 14. We introduce these parameters here for clarity of description; the justification for the parameters is given in §3.5.1.

(a) Encoding

```
1 int64_t encode(int32_t n, int64_t a) {
2     return n * a;
3 }
```

(b) Decoding

```
1 int32_t decode(int64_t n1_enc, int64_t n2_enc, int64_t a) {
2     if (a == A1)
3         return (n1_enc - n2_enc) >> 1;
4     else
5         return (n1_enc + n2_enc) >> 14;
6 }
```

Figure 3.3 – Encoding and decoding operations in Δ-encoding.

(a) Fully encoded operations: Addition

```
1 int64_t add_enc(int64_t x_enc, int64_t y_enc) {
2     return x_enc + y_enc;
3 }
```

(b) Partially encoded operations: Left shift

```
1  int64_t shl_enc(int64_t x1_enc, int64_t x2_enc, int64_t y1_enc, int64_t y2_enc, int64_t a)
2  {
3      if (a == A1) {
4          int32_t y = (y1_enc - y2_enc) >> 1;
5          return x1_enc << y;
6      }
7      else {
8          int32_t y = (y1_enc + y2_enc) >> 14;
9          return x2_enc << y;
10     }
11 }
```

(c) Fully decoded operations: XOR

```
1  int64_t xor_enc(int64_t x1_enc, int64_t x2_enc, int64_t y1_enc, int64_t y2_enc, int64_t a)
2  {
3      if (a == A1) {
4          int32_t x = (x1_enc - x2_enc) >> 1;
5          int32_t y = (y1_enc - y2_enc) >> 1;
6      }
7      else {
8          int32_t x = (x1_enc + x2_enc) >> 14;
9          int32_t y = (y1_enc + y2_enc) >> 14;
10     }
11     int32_t res = x ^ y;
12     return res * a;
13 }
```

Figure 3.4 – Encoding and decoding operations in Δ-encoding.

3.4.2 Encoded Operations

Δ-encoding works on AN-encoded values. This implies that all original operations – addition, subtraction, multiplication, bitwise AND, OR, XOR, shifts, comparisons, etc. – are substituted with the corresponding *encoded operations*. In this section, we provide examples of some typical Δ-encoded operations. For clarity, we introduce them as functions in the C language.

Encoding and decoding operations were already described conceptually. Figure 3.3 shows their

practical implementations. It is worth mentioning that encoding could be implemented through shifts and addition/subtraction, as shown by Equation 3.3; however a simple multiplication exhibits similar performance. The decoding operation corresponds to Equations 3.6 and 3.7.

Most arithmetic operations stay the same in AN codes and also in Δ-encoding; the operations include addition, subtraction, comparisons, modulo, etc. Figure 3.4a exemplifies this. Note that since no encoding/decoding takes place, there is no notion of A in the code snippet.

Some operations require partial decoding. One example is a left shift operation: the number of bits by which an integer is shifted to the left must be decoded, but the integer itself can stay encoded (see Figure 3.4b). Another example is multiplication, where it is enough to decode only one operand.

Finally, bitwise operations (AND, OR, XOR, one's complement) as well as division are notoriously slow if implemented using encoding. In these cases, the only reasonable strategy is to decode operands, perform the original operation, and re-encode the result. Figure 3.4c exemplifies this using the XOR operation.

Encoded operations must be not only fast, they must also propagate possible errors to the resultant integer. This holds for operations like addition. Operations like left shift and XOR rely on duplicated instructions, since it is unlikely that the result of the first operation execution (with A_1) will be corrupted exactly in the same way as in the second execution (with A_2). Moreover, the sum of two encoded copies $\hat{x}_1 + \hat{x}_2$ has zeros in the lower 14 bits by Equation 3.5 (otherwise it indicates that an error occurred during the operation); we use this property to propagate errors in some operations.

3.4.3 Accumulation of Checks

As any fault detection mechanism, Δ-encoding inserts periodic checks of calculated values. An example of such a check is shown in Figure 3.1d, Line 5. It includes checking if both copies of a variable are code words and if they correspond to the same original value. If any of the conditions fails, then an error must have happened, and the execution is terminated.

A naive approach to detect errors would be to check the result of each encoded operation. This would lead to a tremendous slowdown, since each operation would then be accompanied by a heavy-weight check with divisions and branches.

On the other side, one could check only final results, i.e., check only output values right before decoding them. Indeed, if the property of error propagation would hold for all encoded operations, it would be sufficient to check only the results of the computation. In real-world scenarios, however, this property is frequently violated; the XOR operation from Figure 3.4c is one example.

The practical solution would be to analyze the program's data flow and insert checks only at critical points (e.g., after each XOR operation, but not after additions). Even in this case, the number of inserted checks incurs significant overhead.

To achieve a better trade-off between performance and fault coverage, we introduce the *accumulation of checks*. We allocate a pair of integers called *accumulators* and we substitute all intermediate checks with a simple addition to the accumulators. The principle is illustrated in Figure 3.5. The original program performs two operations: addition $x + y$ and subtraction $x - y$. The encoded program adds two accumulations and one subsequent check instead of two expensive checks.

Using accumulators instead of direct checks is beneficial for performance: accumulation requires only two additions instead of several divisions and branches. Moreover, it does not decrease the error detection capabilities of Δ-encoding, because the addition operation propagates any

```
1  int128_t accu1 = 0;
2  int128_t accu2 = 0;
3  void accumulate(int64_t n1_enc, int64_t n2_enc) {
4      accu1 += n1_enc;
5      accu2 += n2_enc;
6  }
7  ...
8  a1_enc = x1_enc + y1_enc;
9  a2_enc = x2_enc + y2_enc;
10 accumulate(a1_enc, a2_enc);
11 b1_enc = x1_enc - y1_enc;
12 b2_enc = x2_enc - y2_enc;
13 accumulate(b1_enc, b2_enc);
14 if (accu1 %A1 != 0 || accu2 % A2 != 0 || accu1/A1 != accu2/A2)  raise_error();
```

Figure 3.5 – Example of checks' accumulation in Δ-encoding.

erroneous value to the accumulator. One last non-obvious advantage is that accumulations are less susceptible to the "who guards the guardians" problem: a check could be erroneously skipped due to a single CPU fault, but quietly skipping both accumulator updates is highly improbable.

3.4.4 Fault Coverage

Δ-encoding provides very high fault coverage. Here we explain how our approach covers all symptoms from the symptom-based fault model described in §3.3. We provide a quantitative analysis only for the case of modified operand; other symptoms can be analyzed in a similar way.

Modified operand: AN codes guarantee that, given a modified operand fault, the probability of a SDC is $1/A$ ([198]). In duplicated instructions, given that a random fault (corrupting a random number of bits) affected both copies of the operand, the probability of a SDC is $1/2^n$, where n is the number of bits of the operand.
With Δ-encoding, given that a fault affected both copies of the operand, a SDC may happen only if (1) the first copy is a code word and (2) the second copy corresponds to the first copy (i.e., produces the same original value when decoded). Combining these requirements together and taking into account that AN codes double the number of bits in operands, we get the probability of a SDC equal to $\frac{1}{A \cdot 2^{2n}}$.

Exchanged operand: Since Δ-encoding performs each operation twice, SDC happens only if two exchanged operand faults substitute two correct copies with two incorrect but valid copies. The probability of such chain of events is negligible.

Faulty operation: Two copies of data are encoded differently (with $A1$ and $A2$) in Δ-encoding; thus, two executions of a faulty operation would work on different operands and would fail in different ways. This means that the probability that two faulty operations produce two corresponding code words is negligible.

Exchanged operation: Since Δ-encoding performs each operation twice, two copies of the operation must be substituted by two exactly the same non-intended operations. This scenario is highly improbable.

Lost update: In Δ-encoding, two store operations are used to update two copies of data; thus, two stores must be omitted to result in a lost update. Such scenario has negligible probability.

As this analysis shows, Δ-encoding provides high fault coverage for all types of faults. Notice

that modified operand faults happen more frequently than other types, because the underlying hardware errors – memory and CPU register bit-flips – occur with perceptible regularity. But other types of faults, however improbable they are, must also be accounted for in safety-critical systems.

The combination of duplicated instructions, AN codes and heuristic accumulation of checks also provides high guarantees against intermittent and permanent errors. For example, using duplicated instructions alone, it is possible that both copies of a variable are stored in the same physical CPU register which experiences a stuck-at fault, and thus the fault remains undetected. In Δ-encoding, a stuck-at fault in a register results in an invalid word (with high probability).

Interestingly, the approach of duplicated instructions cannot detect permanently faulty operations. If the same inputs are fed to two executions of a faulty operation, both executions produce the same incorrect output. In Δ-encoding, the two copies of data are diverse, leading to two different incorrect results. Thus, Δ-encoding can detect permanent faults which would lead to a SDC in the case of simple duplicate execution.

3.5 Implementation

We implemented Δ-encoding as a source-to-source C transformer in Python (see Figure 3.6). Original C programs are encoded at the level of an Abstract Syntax Tree (AST) built by PycParser[3]. Our transformer walks through the AST, substituting all inputs and constants by encoded values and all original C operators by the corresponding encoded operations. The transformer also produces function-wrappers to perform libc/system calls from encoded source (e.g., *malloc()*) and vice versa.

Δ-encoded programs preserve the original code structure, i.e., original control flow as well as variable and function names. This is possible because our transformer does not employ any code optimizations, working as close to the original source as possible. Preserving the original information greatly facilitates debugging and manual changes in encoded programs.

The Δ-encoded code emitted by the transformer does not rely on a specific compiler and is not influenced by compiler optimizations. The structure of Δ-encoding itself prevents the compiler from optimizing duplicate instructions away. (Compiler optimizations are a constant threat for fault-tolerant high-level transformations, since they can be very efficient at eliminating code and data redundancy; some techniques even require all compiler optimizations to be disabled, as in [188].) As an example, consider the decoding operation from Figure 3.3b: the compiler has no knowledge of inherent interdependency between two encoded copies and cannot figure out that the two ways of decoding produce the same original value.

The Δ-encoded code can be intermingled with unencoded sources. First, the programmer can manually add calls to unencoded functions in the emitted encoded code (e.g., adding *printf()* calls for debug purposes). Second, the transformer generates wrappers for unencoded functions used by the encoded code (e.g., libc functions such as *malloc()* and *free()*).

3.5.1 Encoding Data

Since Δ-encoding expands the original domain of values to accommodate all encoded values, our implementation restricts all integer variables to be at most 48 bits wide. We chose $A_1 = 8193$, $A_2 = 8191$ such that the encoded values never exceed the 64-bit range, since the maximum encoded value $(2^{48} - 1) \cdot 8193$ is less than 64 bits wide.

[3] `https://github.com/eliben/pycparser`

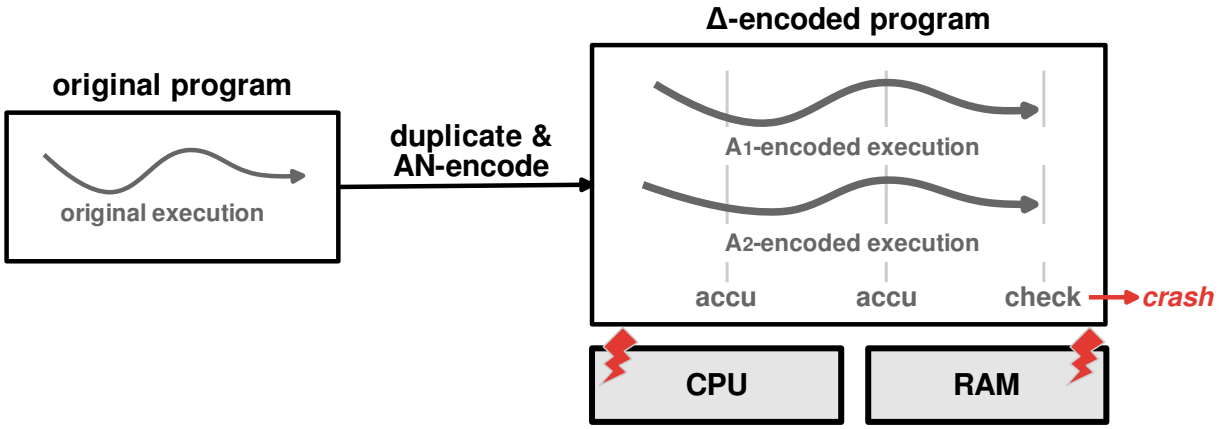

Figure 3.6 – Δ-encoding implementation.

In general, original integer types are limited by at most 32-bit data types. 64-bit types are also supported, but the original program must guarantee that the values never exceed the 48-bit bound. This is the case for pointer types: on modern 64-bit systems, pointers are 64 bits wide but virtual address formats use only the 48 low-order bits [9]. Therefore, our implementation supports pointer types on current 64-bit architectures.

The Δ-encoding transformer implements two copies of variables as two-item arrays. For example, *int32_t n* is transformed into *int64_t n_enc[2]*. This implementation is not optimal with respect to fault detection, because the two copies of the variable are adjacent to each other, and one fault changing bits in-between can corrupt both copies. A better implementation would require separate "shadow" stack and heap for second copies of data. Unfortunately, such separation would require compiler support and thus is impossible in our current C-to-C transformer approach.

One interesting feature of Δ-encoding is the prohibition of silent integer over- and underflow. AN codes modulo arithmetic is not isomorphic to the original modulo arithmetic, e.g., $2^{32} \cdot A$ would not wrap to 0; Δ-encoding would therefore require expensive checks to support integer overflow behavior. Wishing to keep Δ-encoded programs as fast as possible, we disallow all silent under- and overflows. If a programmer wishes to support such wraparounds, she is required to implement them explicitly. Our decision is also partially justified by security reasons: many integer overflows are unintended and can be a source of vulnerabilities [38]. In Δ-encoding, silent integer over- and underflows raise a run-time error.

There is one subtle issue when encoding local loop variables. Modern compilers are particularly good at optimizing loops; in several occasions we noticed that the compiler removed the second copy of a loop variable, weakening the protection. Indeed, the compiler has full right to perform such an aggressive optimization: it knows an initial value and the complete data flow of a loop variable and ascertains uselessness of the second copy. To prevent the compiler from removing the variable, we insert inline pseudo-assembly that clobbers both copies of the loop variable. This example illustrates how careful one should be when enabling fault tolerance without changing the compiler behavior.

3.5.2 Encoding Operations

Some of the encoded operations were already described in §3.4.2. The final implementations follow closely the examples from Figure 3.3a to Figure 3.4c. The Δ-encoding transformer provides the complete set of encoded C operators, including arithmetic, comparison, logical, bitwise,

member and pointer operators, casts, etc.

All encoded operations are inlined in the final executable. This enables the compiler to choose the specific computation path. For example, the decode() operation from Figure 3.3b will be inlined two times in the code (first with $A1$ and then with $A2$): first time stripped to Line 3, second time – to Line 5.

The code emitted by the Δ-encoding transformer must be compiled with the SSE extensions disabled. Otherwise the compiler can glue two move-to-memory instructions of adjacent data copies into one SSE-move. If a hardware error affects this SSE-move, both copies of data are affected, which can lead to undetected SDCs. This flaw in our data representation, where variables are encoded as two-item arrays, was already described in the previous subsection. Note that if copies of data would be completely decoupled, SSE extensions could be enabled again.

The AN codes approach is not able to detect incorrect branching resulting from faults in branching logic. Indeed, the decision of which branch to take is based on the flag bit values of a status register. Flag bits cannot be encoded, and a single bit-flip can lead to an incorrect branch. Fortunately, duplicated instructions suggest a way to detect errors in branching logic: our transformer inserts a "shadow" branch for each original branch. The original branch is encoded to work on the first copy of data, the "shadow" branch works on the second copy. If the branching decisions differ in the two branches, an error is detected.

3.5.3 Accumulation of Checks

The idea of accumulation was defined in §3.4.3; here we describe some implementation issues.

As explained previously, accumulations are a low-overhead substitute for checks, such that the frequency of the checks themselves is significantly decreased. In fact, our experiments showed that checks can be done in the very end of computation, and all intermediate steps are sufficiently protected via accumulations. In the final implementation, we introduced checks only at the end of encoded computations and in wrapper functions.

In its turn, the frequency of accumulations can be tuned. Ideally, data flow analysis must be done to pinpoint critical places. Currently, we adopt a simple strategy: accumulations are inserted after each assignment in original C code. This straightforward technique yields satisfactory results.

As shown in Figure 3.5, accumulation corresponds to one addition operation. Accumulators are 128-bit integers. We use *int128_t* data type provided by gcc; under the hood, this data type is treated as two 64-bit integers. It is tempting to use 64-bit accumulators, but they overflow fast; the accumulation operation would require an additional overflow check. We opted for 128-bit accumulators instead. Since encoded values can be maximum 64 bits wide, 2^{64} accumulations must happen before the accumulators overflow in the worst case. This number of accumulations is enough for any conceivable program; overflow checks are not required for 128-bit accumulators.

Unfortunately, signed 128-bit addition is much slower than its 64-bit counterpart on modern CPUs. It requires one sign extension, one 64-bit addition and one 64-bit add-with-carry – 3 operations in total. Our performance evaluation highlights this slowdown.

Interestingly, it can be meaningful to remove all accumulations completely and perform only one check in the very end of the encoded computation. Remember that Δ-encoding (ideally) propagates all hardware errors to outputs. One can rely on this property and get rid of all intermediate accumulations, in the hope that any error will be detected by the final check. Our evaluation shows that such a trade-off between performance and fault tolerance is acceptable in some scenarios.

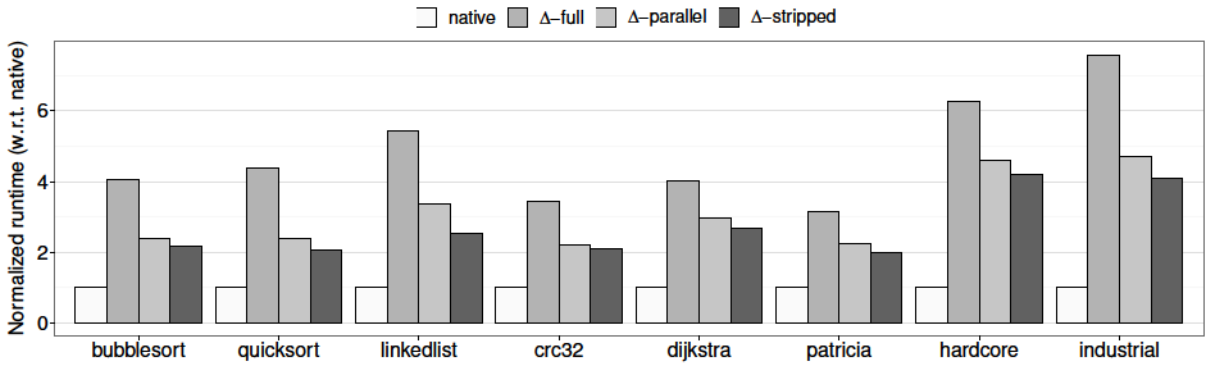

Figure 3.7 – Performance overhead in comparison to native execution.

Moreover, accumulations and checks could be done in parallel to the program's execution. The program could send encoded values for accumulation/check asynchronously and immediately continue execution. Accumulation/check functionality could run on another CPU core or in the dedicated hardware module. If the system allows certain latency between the actual corruption of data and its detection, this parallel approach could be used. For example, automotive embedded systems allow for such latency and are usually equipped with a special hardware watchdog[4]; it would be reasonable to add the accumulation/check functionality in the watchdog and run the encoded program on the main CPU.

3.6 Evaluation

In this section, we evaluate a set of programs encoded with the Δ-encoding transformer in terms of performance and fault coverage. The set of programs under test consists of several microbenchmarks and two use cases. Microbenchmarks give an estimation of the provided fault coverage versus performance slowdown. The first use case is taken from the field of distributed systems and exemplifies the so-called *trusted modules* – small safety-critical parts of applications which need to be robust against hardware errors. The second use case comes from the field of automotive embedded systems and exemplifies *X-by-wire* systems, where a program processes data from sensors and controls actuators such as car brakes.

3.6.1 Methodology

Performance Experiments

All performance experiments were run on a computer with Intel Core i5-3470 CPU (Ivy Bridge architecture), 4GB RAM, L1, L2 and L3 caches of 32KB, 256KB and 6MB. All programs and their variants (including native) were compiled using gcc version 4.8.2, with all optimizations enabled except for SSE (flags *-O3 -mno-sse*). For all programs, execution time was calculated as the number of cycles to perform the processing of data. All programs were run for at least one second, with predefined inputs. The final results are an average of 5 runs. All performance figures show a slowdown compared to the native execution.

Each Δ-encoded program was tested in 3 variants: with 128-bit accumulation, without accumulation, and with parallel accumulation (simulation). These variants were described in

[4]The automotive E-Gas Monitoring Concept has 3 levels of design, with the third, "controller monitoring" level implemented as an independent hardware module (watchdog) [70].

Faults	Program	Variant	masked	OS-det	Hang	Δ-detected	SDC
Transient	**Bubblesort**	**native**	18.265	13.208	0.000	—	68.527
		Δ-stripped	11.488	17.776	0.002	70.730	0.004 (4)
		Δ-full	16.588	19.541	0.001	63.866	0.004 (4)
	HardCore	**native**	16.488	52.300	0.011	—	31.201
		Δ-stripped	18.728	26.033	0.220	55.019	0.000 (0)
		Δ-full	20.416	26.068	0.520	52.996	0.000 (0)
	Industrial	**native**	43.945	19.652	0.041	—	36.362
		Δ-stripped	39.163	21.111	0.000	38.991	0.735 (735)
		Δ-full	28.308	18.100	0.001	53.588	0.003 (3)
Intermittent	**Bubblesort**	**native**	58.315	5.265	0.010	—	36.410
		Δ-stripped	53.000	19.170	0.000	27.825	0.000 (0)
		Δ-full	55.710	11.395	0.000	32.880	0.015 (3)
	HardCore	**native**	59.675	29.155	0.000	—	11.165
		Δ-stripped	57.725	19.345	0.130	22.795	0.000 (0)
		Δ-full	57.020	15.705	0.305	26.970	0.000 (0)
	Industrial	**native**	68.940	12.630	0.045	—	18.385
		Δ-stripped	64.250	13.700	0.000	21.580	0.470 (94)
		Δ-full	58.050	12.075	0.000	29.875	0.000 (0)
Permanent	**Bubblesort**	**native**	61.645	5.540	0.290	—	32.525
		Δ-stripped	49.110	22.375	0.665	27.840	0.010 (2)
		Δ-full	53.510	14.470	0.000	32.015	0.005 (1)
	HardCore	**native**	59.415	29.475	0.060	—	11.045
		Δ-stripped	56.905	21.975	0.725	20.395	0.000 (0)
		Δ-full	54.310	19.175	0.645	25.870	0.000 (0)
	Industrial	**native**	49.935	13.310	1.005	—	35.750
		Δ-stripped	48.005	20.635	0.110	30.865	0.385 (77)
		Δ-full	44.215	19.190	0.075	36.520	0.000 (0)

Table 3.1 – Fault injections: transient multi-bit, intermittent with duration of 100 instructions, and permanent with stuck-at faults. Results are shown as percentages of all injected faults. In *SDC* column, parentheses show absolute numbers of silent data corruptions.

§3.5.3. The variant with 128-bit accumulation (Δ-full) provides full-fledged protection from hardware errors. The variant with no accumulation (Δ-stripped) reduces fault coverage and increases performance, and can be an appropriate trade-off for some scenarios. Finally, "parallel accumulation" (Δ-parallel) is a simulation of hardware-implemented accumulation; we simulate it by moving encoded values to a predefined memory address instead of adding them to the accumulator.

Fault Injection Experiments

For fault injection campaigns, we used Intel Pin[5] and the BFI plug-in[6]. BFI is able to inject random faults and was used in other research [25]. We improved BFI to also inject stuck-at intermittent/permanent faults.

BFI injects single- and multiple-bit faults in: CPU register file, memory cells, address bus, and code segment. These hardware faults trigger software-level symptoms of our fault model. Modified operands are caused by bit-flips in registers and memory. Exchanged operands are due to faults on the address bus or in registers holding addresses. A faulty operation is represented as a fault in operation's output register/memory cell. Exchanged operations are transient faults

[5] `http://www.intel.com/software/pintool`
[6] `https://bitbucket.org/db7/bfi`

in the code segment. A lost update is a direct consequence of address corruption during a move instruction.

We conduct three fault injection campaigns:

Transient Faults: A single multiple-bit transient fault is injected per run, with 100,000 runs in total. This is similar to the Single Event Upset model, but the fault can corrupt multiple adjacent bits.

Intermittent Faults: The same stuck-at fault is triggered for the duration of 100 instructions, with 20,000 runs in total. For example, it simulates an intermittent stuck-at-1 fault in a RAX register.

Permanent Faults: The same stuck-at fault is triggered for the whole duration of the computation, with 20,000 runs in total. For example, it simulates a permanent stuck-at-1 fault in a RAX register.

We inject hardware errors at random and uniformly distributed. In the case of intermittent faults, the fault is injected at a random instruction and reoccurs in 100 subsequent instructions. In the case of permanent faults, the fault is injected at a random instruction and reoccurs until the computation is finished.

The results of fault injections are sorted in 5 categories: *masked faults* (do not affect execution), *OS-detected* (detected by OS, e.g., segmentation fault), *hang* (the program hanged because of the fault), Δ-*detected* (detected by Δ-encoding), *SDC* (undetected; led to silent corruption of data).

Each Δ-encoded program was tested in 2 variants: with accumulation (Δ-full) and without it (Δ-stripped).

3.6.2 Microbenchmarks

As a proof of concept, we chose several microbenchmarks: bubblesort, quicksort, linked list, CRC32, dijkstra, and patricia trie. CRC32, Dijkstra, and patricia trie are taken from MiBench [91]. These three benchmarks perform a significant number of I/O operations to read inputs; in contrast, bubblesort, quicksort and linked list work purely on memory values.

The performance results of the benchmarks are shown on Figure 3.7 (first six). Δ-full versions incur the overhead of 4.08× on average, Δ-stripped and Δ-parallel – 2.26× and 2.59× correspondingly. Δ-parallel performs two times better than Δ-full on some benchmarks, which indicates that a hardware-assisted approach of Δ-parallel could bring a significant performance improvement.

As for fault coverage, we performed fault injection experiments on one representative benchmark – bubblesort. The results are shown in Table 3.1. The native program experiences a significant number of SDCs (from 32% for permanents up to 68% for transients). Δ-encoding variants drastically reduce the rate of SDCs to almost 0%.

It is interesting to examine the few SDCs not detected by Δ-encoding. In the case of transient faults, all 8 undetected faults happened on the address bus such that the injected corrupted bits were written in-between two copies of data, corrupting them both in the same way. This issue was discussed in §3.5.1 and is a deficiency of our implementation.

In the case of intermittent and permanent faults, all 6 SDCs resulted from the same corrupted register. This register was allocated by the compiler for the same encoded operation on two copies of data, such that two copies were affected by the same permanent fault. This is yet another disservice of a compiler (the first one was discussed in §3.5.1); these faults could be detected if we would have control over the compiler's backend.

Variant	2AN	Δ-full	Δ-parallel	Δ-stripped
Leader	42.0	6.3	4.6	4.2
Follower	32.7	3.5	2.8	2.6

Table 3.2 – HardCore's performance overhead in comparison to native.

3.6.3 Use Case: Trusted Modules

The first use case, the *HardCore* trusted module, comes from the field of dependable distributed systems. HardCore is a small safety-critical part of a bigger system – HardPaxos [25]. HardPaxos is a version of the Paxos consensus protocol which enables the service on top to tolerate hardware errors; fault tolerance of the whole service depends solely on HardCore. That is, HardCore is required to have very high fault coverage.

We encoded HardCore using Δ-encoding and reproduced the experiments from [25]: for the leader and for the follower scenarios. Note that the version of HardCore described in [25] was hardened with a variant of AN-encoding called 2AN, incurring very high overheads compared to native execution. The performance numbers for 2AN and our Δ-encoding are shown in Table 3.2; the slowdown for the worst case-scenario (for leader) is presented in Figure 3.7. HardCore's slowdown is higher in comparison to microbenchmarks: HardCore makes heavy use of small loops which the compiler unrolls for the native version but not for Δ-encoded versions (see 3.5.1). In general, our evaluation shows that the Δ-encoded HardCore is *one order of magnitude faster* than the 2AN-encoded version.

The results of fault injections can be seen in Table 3.1. The native version has a significant number of SDCs (31% in case of transients, 11% in case of permanents), while the Δ-encoded HardCore detects *all injected errors in all experiments*. Note that Δ-stripped performs no worse than Δ-full: the reason is the small size of HardCore functions, such that the injected error propagates directly to the outputs. This means that the Δ-stripped version provides complete fault coverage with the average performance benefit of 70% compared to the Δ-full encoding.

3.6.4 Use Case: Safety-Critical Embedded Systems

Our second use case, which we refer to as *industrial*, is a real-world X-by-wire controller from the automotive embedded systems domain. The program makes heavy use of arithmetic operations, working on a small set of variables and spanning over 900 lines of code. We consider this program a typical example of safety-critical embedded applications which can benefit from Δ-encoding.

The performance slowdown is shown in Figure 3.7. We would like to stress the slowdown of 4.7× for the Δ-parallel variant: parallel accumulation on a separate hardware module is well-suited for embedded systems, since this functionality can be put in the already existing hardware watchdog. The relatively high slowdown is due to division operations, which require decoding to the original values, their division and subsequent encoding (see §3.4.2).

Table 3.1 shows the fault injection results for the industrial program. The Δ-full variant shows very high fault coverage, with 3 SDCs in the case of transients and 0 in other cases. The Δ-stripped variant, however, results in a significant number of SDCs: the industrial program has a long and complex execution path such that errors do not propagate to the outputs. This is in contrast to HardCore where Δ-stripped had the same fault coverage as Δ-full. The reasons for SDCs are the same as for bubblesort and HardCore.

DI	Δ-stripped	Δ-parallel	Δ-full	ANBD
1.6	2.1	2.4	4.4	16.0

Table 3.3 – Quicksort's performance overhead: comparison of approaches.

Characteristic	**Program**	native	Δ-full	Δ-parallel	Δ-stripped
Instructions/cycle	**Bubblesort**	1.25	2.27	2.34	2.26
	HardCore	1.78	2.73	2.61	2.70
	Industrial	1.46	2.70	2.75	2.82
Branch misses, %	**Bubblesort**	9.31	4.82	6.14	6.00
	HardCore	0.00	0.00	0.00	0.00
	Industrial	3.08	1.02	0.92	0.77

Table 3.4 – Performance characteristics.

3.6.5 Discussion

Δ-encoding was developed to provide a high level of fault tolerance. In this sense, we favor fault coverage over performance. Δ-encoded programs must be protected from all error types: transient, intermittent and permanent, single-bit and multiple-bit, single faults and multiple faults. Our experiments show that Δ-encoding (namely the Δ-full variant) achieves an average fault coverage of 99.997%.

We consider performance slowdowns of 3–4× acceptable for our use cases. First of all, safety-critical computations are usually limited in size and not resource-demanding. Second, a software-only encoded processing approach is inherently slow, and a slowdown of several times is a significant improvement compared to the previous works on AN-encoding.

Unfortunately, we could not obtain the implementations of AN-encoding [198] or duplicated instructions [170]. However, we can perform an indirect comparison on the mutual quicksort benchmark to put Δ-encoding into perspective (see Table 3.3). The duplicated instructions approach (DI in the table) reveals a slowdown of 1.6× in the best case [170]; the ANBD-variant of AN-encoding has a slowdown of 16× [198]. Δ-encoding shows performance numbers closer to duplicated instructions, with the slowdowns of 2–4×. This indicates that Δ-encoding outperforms previous AN-encoding techniques, adding only a moderate overhead on top of duplicate execution.

For performance, our approach relies on deep instruction pipelining, out-of-order execution and sophisticated branch prediction in modern CPUs. All these techniques enable effective scheduling of instructions. Programs usually do not utilize instruction pipeline and branch prediction fully. Δ-encoding takes advantage of an underutilized pipeline and branch predictor, such that the two copies of data can be processed in parallel. Table 3.4 shows that the number of instructions per cycle roughly doubles in Δ-encoded programs, while the number of branch misses drops drastically. In case of HardCore, branch predictor shows perfect results, and there are 0% of branch misses even in native execution. These numbers prove that Δ-encoding benefits from heavily utilized pipeline and branch predictor.

3.7 Related Work

Local error detection research has a long history. It began in 1960s with pure hardware approaches used in highly available servers and space industry; starting from late 1990s, research focus shifted to software-only approaches, commonly known as software-implemented hardware fault

tolerance (SIHFT).

3.7.1 Hardware-based approaches

Hardware-implemented error detection is exemplified by the evolution of two mainframe systems: IBM S/360 (now called IBM System z) and Tandem NonStop (now HP NonStop) [23]. These systems provide massive redundancy to achieve high availability: lockstepped proprietary CPUs, redundant CPU logic, ECC-protected memory and caches, and redundant hardware components and paths. The two systems guarantee very high fault coverage, but hardware implementation implies very high economic costs. Δ-encoding can be seen as a much cheaper alternative to harden only a small subset of software stack run on commodity hardware.

A cost-effective hardware approach is to use simple checkers which observe activities of commodity hardware units and raise exceptions in case of errors. For example, the DIVA checker [16] commits CPU outputs only after it verified their correctness. Argus [149] implements four independent checkers to validate four CPU/memory tasks: control flow, data flow, computation, and memory accesses. Nostradamus [159] is yet another checker that compares an instruction's expected impact on the CPU state to the actual impact on the state. Though the approaches incur low performance overhead (5-10%), they require significant changes in hardware, whereas Δ-encoding is purely software-based and provides the same error detection guarantees.

Symptom-based detection (e.g., ReStore [239]) analyzes anomalous behavior of hardware such as memory access exceptions, mispredicted branches and cache misses. However, the approach cannot offer adequate fault coverage required in safety-critical systems, detecting only about a half of propagated faults.

3.7.2 Software-based approaches

Redundant Multithreading (RMT) [153] protects from transient faults by executing two copies of the program on two cores, periodically comparing their outputs. However, the technique assumes existence of a spare core, therefore typical embedded systems with single-core CPUs cannot benefit from RMT. In contrast, Δ-encoding requires only one core for computations.

In duplicated instructions approach, program flow executes twice on the same core. The approach was first proposed in EDDI [170] and later refined in SWIFT [191]. Both solutions concentrate on transient errors and favor performance over fault coverage; moreover, SWIFT has an assumption of ECC-protected memory which does not hold for commodity and embedded hardware. Interestingly, EDDI's offshoot called ED4I [169] is similar to Δ-encoding: it combines data diversity and duplicated instructions, protecting from permanent faults. Unfortunately, ED4I was a theoretical attempt and was not even evaluated for performance, whereas Δ-encoding is a complete and practical solution.

Encoded processing uses AN codes theory and was first used as a pure hardware approach; an example is a STAR computer designed for spacecrafts [18]. Forin [80] laid the foundations of software-implemented encoded processing, which was later extensively researched by Schiffel [198]. However, AN-encoding and variants thereof, which were used in these works, reveal imbalance in fault coverage versus performance: pure AN encoding has low fault coverage, ANB- and ANBD-variants have low performance. Our proposed Δ-encoding provides balance between the two metrics.

3.8 Conclusion

We presented Δ-encoding, a fault detection mechanism that covers not only commonly assumed Single Event Upsets, but also multiple-bit, intermittent and permanent faults. To achieve high fault coverage, Δ-encoding combines two approaches: AN codes and duplicated instructions. As our evaluation shows, Δ-encoding achieves fault coverage of 99.997% at the cost of an average slowdown of 2–4×.

Our prototype is a source-to-source transformer. As we mentioned before, it would be more beneficial to implement Δ-encoding as a compiler plug-in. In this way, we would be able to perform sophisticated data flow analysis to remove redundant accumulations and make the compiler Δ-encoding-aware.

Another interesting direction is a software-hardware Δ-encoding approach. Accumulations and checks can be moved out of the critical path and encapsulated in a separate hardware module. Δ-encoding could also benefit from additional instructions in Instruction Set Architecture (ISA).

Another interesting implication of Δ-encoding is the recovery ability. If a fault affected only one copy of data, it is detected via AN codes. The second copy of data can be used to recover the first copy, masking the fault, and the execution can continue.

We envisage security-related applications of Δ-encoding. Data diversity and the ability to use different pairs of As for different parts of a program could enable protection against malicious attacks.

4 Elzar: Leveraging Advanced Vector Extensions

While Δ-encoding introduced in the previous chapter covers all kinds of CPU and RAM faults, it also exhibits high overheads of 2–4×. These overheads are acceptable in safety-critical embedded domains, but are prohibitive for large-scale online services.

Thus, in the following two chapters we concentrate only on the prevailing subclass of hardware faults – transient CPU faults that result in silent data corruptions (SDCs). We omit all other subclasses of faults from our fault model since they do not usually occur in data center environments (in contrast to aerospace and land transport which are exposed to a wide range of external factors). In particular, we assume that intermittent and permanent CPU faults ultimately exhibit themselves as machine crashes and are trivially tolerated (by rebooting a machine). We also assume that all memory is sufficiently protected via some form of error detecting/correcting codes: DRAM and Last-level CPU cache are usually protected by Single-Error Correcting and Double-Error Detecting (SECDED) Hamming codes, and Level-1 CPU cache is protected by parity bits. Therefore, our solutions need to protect only against transient bit-flips in CPU registers and execution units.

In this chapter, we present our first attempt to tolerate transient CPU faults with low performance overhead – Elzar. (As we shall see, this first attempt was not very successfull.) Elzar achieves fault tolerance by utilizing Intel Advanced Vector Extensions (AVX) to provide triple modular redundancy for legacy C/C++ applications. In other words, Elzar relies on Intel AVX – the latest implementation of the Single Instruction Multiple Data (SIMD) family of extensions described in §1.2.

The content of this chapter is based on the paper "Elzar: Triple Modular Redundancy using Intel AVX" presented at DSN'2016 [126]. The paper was a joint collaboration with Oleksii Oleksenko, Pramod Bhatotia, Pascal Felber, and Christof Fetzer.

4.1 Rationale

Transient faults in CPUs can cause arbitrary state corruption during computation. Therefore, they pose a significant challenge for software systems reliability [197]. The causes for transient faults are manifold, including radiation/particle strikes, dynamic voltage scaling, manufacturing variability, device aging, etc. [35]. Moreover, the general trend of ever-decreasing transistor sizes with lower operating voltages only worsens the reliability problem [98, 211].

The unreliability of CPUs is especially threatening at the scale of data centers, where tens of thousands of machines are used to support modern online services. At this sheer scale, CPU faults happen at a surprisingly high rate and tend to increase in frequency after the first occurrence, as reported by a number of large-scale in-the-field studies [88, 165, 202]. Since the machines in data centers operate in tight collaboration, a single CPU fault can propagate to the entire data center, leading to catastrophic consequences [8, 163].

To overcome the problem of transient CPU faults, large-scale online services started using ad-hoc

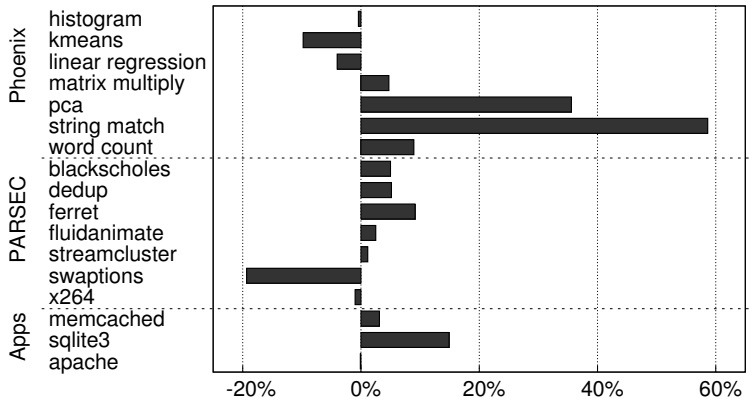

Figure 4.1 – Performance improvement with SIMD vectorization enabled (maximum runtime speedup for Phoenix and PARSEC benchmarks, maximum throughput increase for Memcached, SQLite3, and Apache).

mechanisms such as integrity checks, checksums, etc. For instance, Mesa [2], a data warehousing system at Google, makes use of application-specific integrity checks to detect transient faults during computation. Unfortunately, ad-hoc mechanisms have two major limitations: (1) they require manual effort to design and implement application-specific integrity checks, and (2) they can only protect from errors that are anticipated by the application programmer.

As an alternative to ad-hoc checking techniques, one can make use of a principled approach like Byzantine Fault Tolerance (BFT). BFT-based systems do not only tolerate transient faults, but also malicious adversaries. Unfortunately, BFT yields high performance and management overheads because of its broad assumptions on the type of faults and the power of the adversary [222]. Since most online services run behind the security perimeter of a data center, the "pessimistic" BFT fault model is considered overkill. Therefore, BFT-based systems are rarely adopted in practice.

To find a good compromise between ad-hoc mechanisms and BFT-based systems, a number of light-weight *hardening* techniques were proposed (see §4.2). These hardening techniques transform the original program to locally detect and correct faults. A well-known hardening approach is Instruction-Level Redundancy (ILR) [127, 170, 191]. ILR is a compile-time transformation that replicates original instructions to create separate data flows and inserts periodic checks to detect divergence caused by transient faults in these data flows. In particular, ILR duplicates instructions to achieve fault detection [170, 191] and triplicates them to tolerate faults by majority voting [190].

As a result, with ILR the CPU executes *the same instruction* two or three times on *several data copies*. We notice that, in fact, this corresponds to the very definition of Single Instruction Multiple Data (SIMD) processing. SIMD exploits data level parallelism, i.e., a single instruction operates on several pieces of data in parallel. Given that most modern CPUs have support for SIMD processing (Intel x86's SSE and AVX, IBM Power's AltiVec, and ARM's Neon), we can naturally ask the following question: *Can we utilize SIMD instructions to tolerate transient CPU faults and achieve better performance than ILR with three copies?*

Before answering this question, we first need to understand how much of the SIMD potential of modern CPUs is *actually* being used in real-world applications. To investigate this, we tested applications from the Phoenix [187] and PARSEC [32] benchmark suites, as well as several real-world applications, namely Memcached, SQLite, and the Apache web server. We compiled all

applications in two versions: "native" with all optimizations enabled, and "no-SIMD" where we disable SSE, AVX, and all vectorization optimizations in LLVM. The performance improvements of native over no-SIMD, shown in Figure 4.1, indicate that most applications do not utilize the benefits of SIMD processing. Indeed, most of them exhibit less than 10% improvement, with only *string match* significantly benefiting from AVX.[1] One can therefore conclude that SIMD processing units are currently largely underutilized CPU resources and could hence be used for fault tolerance.

To this end, we propose Elzar,[2] a compiler framework to harden *unmodified multithreaded* programs by leveraging SIMD instructions available in modern CPUs (§4.3). Elzar is built on the Intel AVX technology to achieve triple modular redundancy. Since AVX possesses 256-bit wide registers and regular programs operate on at most 64-bit ones, it is possible to operate with four replicas in parallel, which is more than enough to harden applications and mask faults with majority voting. Consequently, if a hardware fault affects one of the four replicas in an AVX register, it can be detected and outvoted by the other correct replicas.

We implemented Elzar as an extension of the LLVM compiler framework (§4.4). It executes as a pass of the usual build process right before the final code generation. In particular, Elzar transforms all the regular instructions of an application into their AVX-based counterparts, replicating data throughout AVX registers. To achieve such transparent transformation, we use a mix of LLVM vectors and low-level AVX intrinsics.

We evaluated our approach by applying Elzar to the Phoenix and PARSEC benchmark suites (§4.5), as well as three real-world case-studies: Memcached, SQLite3, and Apache (§4.6). To our disappointment, our evaluation showed mostly negative results, with an average normalized runtime slowdown of 4.1–5.6× depending on the number of threads. When compared against a straightforward instruction triplication approach [190], Elzar performed 46% worse on average. At the same time, Elzar was better on CPU-intensive benchmarks with few memory accesses and many floating-point operations.

We attribute poor performance of Elzar to two main causes. First, there is a significant discrepancy between the regular CPU instructions and their AVX counterparts. This discrepancy forced us to introduce additional wrapper instructions that significantly hamper performance. Second, AVX instructions in general have higher latencies and are less optimized than the regular CPU instructions. Nonetheless, we believe there is potential in using AVX for fault tolerance, and discuss how future implementations of this technology could boost Elzar's performance via minor modifications to the AVX instruction set (§4.7). Our rough estimation suggests that Elzar could achieve overheads as low as 48% with the changes we propose.

4.2 Background and Related Work

Our approach is based on three ideas: software-based hardening for fault detection, triple modular redundancy for fault recovery, and Intel AVX technology for SIMD-based fault tolerance.

[1] Some applications (e.g., *kmeans* and *swaptions*) actually perform worse when SIMD is enabled. This counter-intuitive result is explained by the fact that compilers have only rough cycle-cost models and sometimes produce suboptimal instruction sequences.

[2] Named after a four-armed character of Futurama. Similarly, Intel AVX has 4×64-bit wide registers for SIMD processing.

4.2.1 Software-Based Hardening

Software-based hardening techniques can be broadly divided into three categories: Thread-Level Redundancy (TLR) also called Redundant Multithreading (RMT), Process-Level Redundancy (PLR), and Instruction-Level Redundancy (ILR).

Redundant Multithreading (RMT). In RMT approaches [153, 257], a hardened program spawns an additional *trailing* thread for each original thread. At runtime, trailing threads are executed on separate spare cores or take advantage of the Simultaneous Multithreading (SMT) capabilities of modern CPUs. Similar to Elzar, RMT allows keeping only one memory state among replicas (assuming that memory is protected via ECC). However, RMT approaches heavily rely on the assumption of spare cores or unused SMT, which is commonly not the case in multithreaded environments where programs tend to use all available CPU cores.

Process Level Redundancy (PLR). PLR implements the similar idea as RMT, but at the level of separate processes [214, 258]. In PLR, each process replica operates on its own memory state, and all processes synchronize on system calls. In multithreaded environments, allocating a separate memory state for each process raises a challenge of non-determinism because memory interleavings can result in discrepancies among processes and lead to false positives. Some PLR approaches resolve this challenge by enforcing deterministic multithreading [65]. PLR might incur a lower performance overhead than RMT but it still requires spare cores for efficient execution.

Instruction-Level Redundancy (ILR). In contrast to RMT and PLR, ILR performs replication *inside* each thread and does not require additional CPU cores [170, 191]. This in-thread replication seamlessly enables multithreading and requires no spare cores for performance. We present ILR in detail in §4.3.2.

Recent work on ILR mainly concentrated on optimizations to trade-off fault coverage for lower overheads [75, 255]. In contrast to these new approaches, Elzar aims to utilize SIMD technology available on modern CPUs to achieve low performance overhead without compromising on fault coverage. A recent proposal has shown promising initial results when applying SIMD instructions to parallelize ILR [51]. The scope of the work is however limited: (1) it only detects faults and does not provide recovery; (2) it only protects the floating-point unit; (3) it targets only single-threaded programs; and (4) hardening is performed manually at the level of the program's source code. In contrast, Elzar targets detection *and* recovery of transient CPU faults for unmodified *multithreaded* programs. Furthermore, Elzar protects the whole CPU execution including pointers, integers, and floating-point numbers.

Our own HAFT approach is a fault tolerance technique that couples ILR with Hardware Transactional Memory (HTM) (see Chapter 5). In this work, instructions are duplicated to provide fault detection, and an HTM mechanism roll-backs failed transactions to provide fault recovery. Elzar does not rely on a separate rollback mechanism, but rather masks faults using Triple Modular Redundancy.

4.2.2 Triple Modular Redundancy

Triple Modular Redundancy (TMR) is a classical approach for achieving fault tolerance in mission-critical systems [144]. TMR detects faults by simple comparison of three replicas and performs fault recovery by majority voting, i.e., by detecting which replica differs from the other two and correcting its state. Consequently, it imposes an obvious restriction on the fault model: only one replica is assumed to be affected by the fault.

While most of the software-based hardening techniques discussed above utilize only Dual Modular Redundancy (DMR), i.e., they can only detect but not correct faults, there are still a

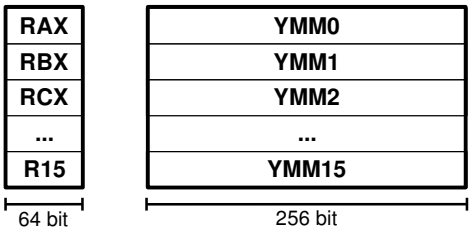

Figure 4.2 – General purpose (GPR) and AVX (YMM) registers.

Figure 4.3 – AVX addition (left): original values `r1` and `r2` are replicated throughout the AVX registers; AVX shuffle (right): original values are rearranged.

number of techniques based on TMR [65, 190]. In the context of ILR, SWIFT-R [190] extends the fault detection mechanisms of SWIFT [191] by inserting three copies (instead of two) for each instruction and performing periodic majority voting to detect and correct faults. ELZAR, in contrast, implements TMR without an increase in the number of instructions, since AVX registers are large enough to hold at least 4 copies of the data.

4.2.3 Intel AVX

Our solution relies heavily on the Single Instruction Multiple Data (SIMD) technology and its specific implementation, Intel AVX. The main idea behind it is to perform the same operation on multiple pieces of data simultaneously (data level parallelism). Figure 4.3 illustrates this concept and how it relates to replication for fault tolerance. AVX adds new wider registers (YMM registers) that are capable of storing several elements and the corresponding new instructions that operate on these elements in parallel. Initially, AVX was targeted for applications that perform parallel data processing such as image or video processing; in this work, we (ab)use it for fault recovery. Note that we do not use the previous generation of Intel's SIMD implementation, SSE, since it can only operate on two 64-bit values and we need at least three copies to be able to correct faults.

Hardware implementation. The x86-64 architecture provides 16 256-bit wide YMM registers available for AVX instructions. Figure 4.2 compares them with general-purpose registers (GPRs). It should be noted, however, that even though only 16 registers are visible at the assembly level, many more registers are implemented physically and used at runtime (e.g., 168 YMM registers in Intel Haswell).

In modern implementations, AVX has several dedicated execution units. It provides a high level of parallelism and allows programs to avoid some common bottlenecks.

Instruction set. The AVX instruction set consists of a large number of instructions, including special-purpose extensions for cryptography, multimedia, etc. ELZAR uses only a subset of AVX instructions, which we discuss in the following.

Most arithmetic and logic operations are covered by AVX, except for integer division and modulo. For example, Figure 4.3 (left) illustrates how addition is performed with AVX.

AVX-based comparisons act differently than their counterparts in the general instruction set.

Instead of directly affecting the flags in the x86 FLAGS register as normal comparisons do, AVX comparisons return either all-1 (if result is "true") or all-0 ("false") values for each YMM element. This behavior is explained by the fact that the comparison is performed in parallel on multiple pieces of data, with possibly conflicting outcomes that would affect the flags differently. On the other hand, there are no control flow instructions in the general instruction set that could operate on such sequences of 1s and 0s. Therefore, a `ptest` AVX instruction was introduced that sets the ZF and CF flags in FLAGS by performing an `and`/`andn` operation between its operands. (We omit the detailed explanation of how `ptest` works for the sake of simplicity and refer the reader to the Intel architecture manuals.) As a result, a branch is encoded in AVX as a sequence of an AVX comparison followed by a `ptest` and a subsequent jump based on the ZF and CF flags.

In this work, we use `shuffle`, a specific AVX operation that performs data rearrangement inside a YMM register. One example of a shuffle is shown in Figure 4.3 (right). In combination with other operations, it allows us to get much of the functionality that is not implemented in hardware. For example, we can get a horizontal test for equality using a combination of `shuffle`, `xor` and `ptest` (see §4.3.3 for more details).

4.3 Design

In this section, we introduce the design of ELZAR and describe the principle of ILR upon which it is based.

4.3.1 System Model

Fault model. ELZAR uses the Single Event Upset (SEU) fault model [191], where only one bit-flip in a CPU is expected to occur during the whole execution of a program. A bit-flip means an unexpected change in the state of a CPU register or a wrong result of a CPU operation. The SEU is transient, i.e., it does not permanently damage the hardware and lasts only for several clock cycles.

We fully protect the AVX register file and the AVX operations; recall that they are completely decoupled from the regular GPR registers and scalar instructions (§4.2.3). We do not consider faults in the memory subsystem since it is assumed to be protected by ECC. Our fault model also does not cover control flow errors, assuming some orthogonal control flow checker.

In general, ELZAR protects from more than single faults. Indeed, four copies of data can tolerate two independent SEUs with a high probability: If any two copies agree and each of the other two copies disagree with the former ones, the majority voting can still mask the faults in the latter copies (we elaborate more on that in §4.3.3). In what follows, we focus on tolerating single faults for simplicity.

Memory and synchronization model. ELZAR imposes no restriction on the underlying memory and synchronization model, and even works with programs containing data races. ELZAR does not replicate nor modify the original memory-related operations (loads, stores, atomics) in any way, therefore the program's memory access behavior is unchanged. As a result, ELZAR allows for arbitrary thread interleavings in multithreaded programs and supports all kinds of synchronization primitives.

(a) Native

```
1  loop:
2    r1 = add r1, r2
3
4
5
6
7    cmp r1, r3
8
9
10   jne loop
```

(b) ILR

```
loop:
  r1 = add r1, r2
  r1' = add r1', r2'
  r1'' = add r1'', r2''
  majority(r1, r1', r1'')
  majority(r3, r3', r3'')
  cmp r1, r3


  jne loop
```

(c) Elzar

```
loop:
  y1 = add y1, y2




  y4 = cmpeq y1, y3
  ptest y4
  ja recover(y4)
  je loop
```

Figure 4.4 – Original loop (a) increments r1 by r2 until it is equal to r3. Usual ILR transformation (b) triplicates instructions and adds majority voting before comparison. AVX-based Elzar (c) replicates data inside YMM registers, inserts `ptest` for comparison, and jumps to majority voting only if a discrepancy is detected in y4.

4.3.2 Instruction-Level Redundancy

We base Elzar on Instruction-Level Redundancy (ILR) [170, 190, 191], a software-based technique to detect and tolerate transient hardware faults. As other software-based approaches, ILR transforms the original program by replicating its computation and inserting periodic checks on computation results. An example of an ILR-transformed code snippet is shown in Figure 4.4b.

Replication. ILR replicates programs at the level of instructions. At compile-time, ILR inserts "shadow" copies for each instruction except for a few instructions classified as "synchronization" instructions. The shadow copies operate on their own set of shadow registers. At runtime, the program effectively executes the original and the shadow instructions, creating mostly independent original and shadow data flows which synchronize only on specific instructions.

The synchronization instructions include all memory-related operations (loads, stores, atomics) and control-flow operations (branches, function calls, function returns). Memory-related operations are not replicated for two reasons: (a) the memory subsystem contains only one copy of the state and there is no need to store twice, and (b) ILR keeps the memory access behavior unmodified in order to allow for non-determinism in multithreaded applications. Control-flow operations are not replicated because ILR protects only data integrity and assumes no control-flow faults. Note that by not replicating function calls, ILR requires no changes in function signatures and no wrappers for system calls and third-party non-hardened libraries.

To create a shadow data flow, ILR replicates all inputs: values loaded from memory, values returned by function calls, and function arguments. This is achieved by a simple move of an input value in one of the shadow registers.

If only fault detection is required, it is sufficient to *duplicate* the instructions and signal an error or simply crash if two data flows diverge [170, 191]. If fault tolerance is needed, the instructions must be *triplicated* and majority voting must be used to mask faults in one of the three data flows (see Figure 4.4b) [190].

Checks. To be able to detect faults, ILR additionally inserts checks right before synchronization instructions. As one example, a load address must be checked before the actual load, otherwise a wrong value could be undetectably loaded and used by the subsequent instructions. As another example, all function arguments must be checked before the function call to prevent the callee from computing with wrong values. Finally, it is important to check the branch condition before branching or else the program could take a wrong path.

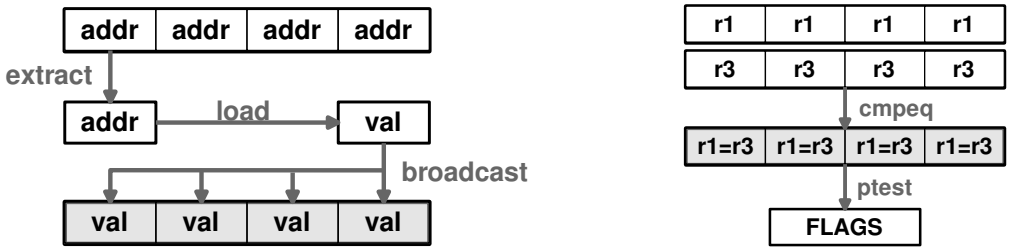

Figure 4.5 – Elzar load (left): original `load` is wrapped by AVX-based `extract` and `broadcast`; Elzar branching (right): original `cmp` for equality is transformed in a sequence of `cmpeq` and `ptest`.

The checks themselves are straightforward. If crash-stop behavior is sufficient, a check compares two copies of data and crashes the program if the copies diverge. For availability (fault tolerance), ILR requires majority voting on three replicas to mask a possible fault (as depicted in Figure 4.4b). During majority voting, three copies of data are compared to each other, and if one copy differs from the other two it is overwritten with the majority value.

4.3.3 Elzar

As appears clearly in Figure 4.4, ILR requires three times more instructions than the original program plus expensive majority voting on synchronization events. As a result, a simple 3-instruction loop may require around 13 instructions under ILR. Such a blow-up in instructions can quickly saturate CPU resources and result in high performance overhead.

Elzar, on the other hand, does not replicate instructions but rather data and thus increases the total number of instructions only modestly. Figure 4.4c shows that Elzar inserts only 2 additional instructions to perform a check on a branch condition. The replication is achieved by utilizing wide YMM registers, with `y1`–`y4` each containing four copies of the original values. The `add` and `cmp` instructions in this snippet are actually AVX instructions which operate on four copies inside the YMM registers in parallel. The somewhat peculiar check consists of the `ptest` AVX instruction and a subsequent jump to recovery code if a discrepancy in branch condition `y4` is detected; we cover AVX-based checks in detail below.

In general, Elzar transforms a program as follows: it (1) replicates the data in YMM registers, (2) inserts periodic checks, and (3) inserts recovery routines. In the following, we discuss each of these steps in detail.

Step 1: Replication. AVX provides an almost complete set of arithmetic and logical instructions: addition, subtraction, multiplication, bitwise operations, shifts, etc. For floating point data, all the usual instructions are present in AVX. For integers, the only missing instructions are integer division and modulo; Elzar falls back to basic ILR in these cases. In general, Elzar achieves replication by simply replacing the original arithmetic and logical instructions with their AVX counterparts, as in Figure 4.3.

The situation is more complicated for (most) non-replicated synchronization instructions. These are the regular loads, stores, function calls, etc., which do not operate on YMM registers. Thus, Elzar has to extract one copy of each instruction's argument from YMM registers and use this copy in the instruction. If a synchronization instruction returns a value (e.g., load), this value must then be replicated inside a YMM register. AVX provides dedicated instructions for such purposes: `extract` and `broadcast`. Unfortunately, these additional instructions must wrap every single load, store, etc., which leads to high overheads. An example of such "wrapping" for

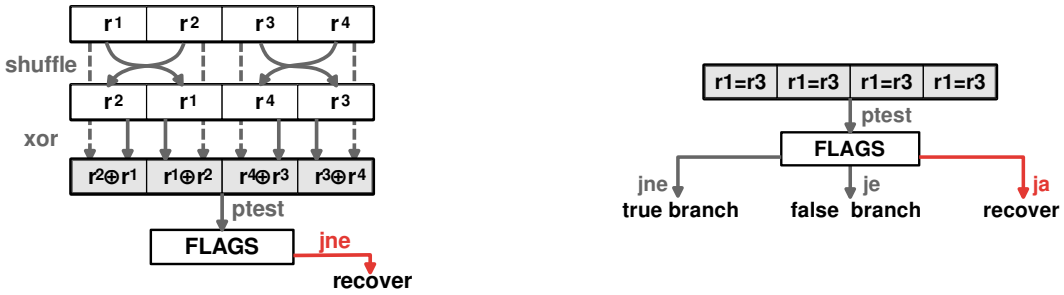

Figure 4.6 – Checks on synchronization instructions (left) and on branches (right).

a load is shown in Figure 4.5 (left).

A special case of a synchronization instruction is a branch. A typical x86 branching sequence consists of one comparison (`cmp`) which toggles the FLAGS register and the subsequent jump instruction (`je` for "jump if equal", `jne` for "jump if not equal", etc.). This is exemplified in Lines 7–10 of Figure 4.4a. Unfortunately, as explained in §4.2.3, AVX lacks instructions affecting control flow except for `ptest`. Moreover, the AVX-based comparison instructions (e.g., `cmpeq`) do not toggle the FLAGS register but instead fill the elements of a YMM register with true/false values. Therefore, Elzar inserts an additional `ptest` to examine the result of `cmpeq` and only then proceeds to a jump (see Figure 4.5 and also Figure 4.4c, Lines 7, 8, and 10).

Step 2: Adding checks. In order to detect faults, Elzar inserts checks before each synchronization instruction. If a check succeeds, i.e., all copies of a YMM register contain the same value, the program continues normally, otherwise the YMM register must be recovered via majority voting. Note that the check itself must be as efficient as possible since it executes on the fast path. The recovery routine, however, resides on the slow path and can hence be less efficient.

Similar to replication, Elzar distinguishes between branches and all other synchronization instructions. Because of implementation choices in AVX, checks turn out to be very effective for branches but not for other operations. To support efficient checks in Elzar, we rely on the assumption that a fault corrupts only one copy in a YMM register (see §4.3.1).

In general, a check on the arguments of a synchronization instruction requires a pair-wise (horizontal) comparison of copies inside a YMM register. For example, upon a function call, all function arguments replicated in the corresponding AVX registers must be checked for discrepancies. Interestingly, AVX provides a horizontal subtraction instruction called `hsub`, but it is not implemented for 64-bit integers and is generally slow. Hence, we opted for another implementation of checks that involves a `shuffle` and a subsequent `xor`. This idea is illustrated in Figure 4.6 (left). In an error-free case, `xor` produces all-0s which is easily ruled out by `ptest`. In the case of a fault in one of the copies, the result of `xor` is a mix of 0s and 1s, which triggers the `jne` path and leads to recovery.

A check on a branch is cheaper and conceptually simpler. As evident in Figure 4.5 (right), branching in AVX already requires an AVX-based comparison and a `ptest`. We notice that in error-free case, comparisons in Elzar can produce only all-true or all-false results (see §4.2.3). Thus, a mix of true and false indicates a fault. Fortunately, `ptest` is a versatile instruction that allows us to check for an all-true, all-false, or true-false mix outcome simultaneously, as shown in Figure 4.6 (right). Therefore, to add a check before a branch, it is sufficient to augment the AVX-based branching with just a single jump instruction, `ja` ("jump if above"), as shown in Figure 4.4c, Line 9.

Step 3: Adding recovery. Checks on branches and other synchronization instructions trigger a recovery routine when a fault is detected. The task of this routine is to mask a fault. Because

of the assumption that a fault is localized in only one copy of the YMM register (see §4.3.1), it is sufficient to identify two identical replicas in the register and blindly broadcast their value to the whole register. This can be performed efficiently by a single comparison of the low elements of the faulty YMM register (depicted in gray in Figure 4.6) and, depending on the result of the comparison, copying either the lowest or the highest element to the rest of the register.

We note, however, that we can easily implement a smarter recovery strategy that would support more complex fault patterns involving multiple bit flips. As the recovery procedure is on the slow path, i.e., it is triggered only rarely, it does not need to be optimized for speed and this added reliability can be implemented without compromising performance.

The idea of the extended recovery procedure is to check all four elements and consider three scenarios: (1) if three elements are identical, then the last one is faulty and can be overwritten with the value of the former; (2) if two elements are identical and the other two have each a different value, then the latter elements are both faulty and can be overwritten with the value of the former; finally, (3) if we have two groups of two elements, with each group agreeing on a different value, then the same fault has affected two elements and we have no majority, hence program execution must stop. This recovery strategy can tolerate all single bit flips, all flips of two bits of different order in the replicas, as well as a wide variety of more complex fault patterns that leave at least two elements identical.

4.3.4 Data Types Support

AVX natively supports 8-, 16-, 32-, and 64-bit integers as well as single- and double-precision floating points. However, up to this moment the discussion implied 64-bit integers replicated four times across a 256-bit YMM register.

There are three options to support smaller types: (1) cast all smaller integer types to 64-bit integers and 32-bit floats to 64-bit doubles, (2) replicate all types only four times in the low bits of YMM registers, leaving upper bits nullified, or (3) replicate smaller types so many times as to fill up the whole YMM register. The first approach obviously breaks semantics of integer overflows and floating point precision, possibly leading to unexpected computation results. The second approach is better but requires additional care for AVX instructions that compute across the whole YMM register, e.g., results of comparisons may differ in lower and upper bits. Therefore we chose the third approach which leads to extreme settings of up to 32-modular redundancy for 8-bit integers but is conceptually clean.

Compilers like LLVM sometimes produce esoteric integer types like 1-bit or 9-bit integers, usually for sign-extension and truncation purposes. Such data types are rare but still present in many applications, therefore we extend them to the AVX-supported bit width and treat them as "usual" integers. We take special care whether to zero- or sign-extend them, depending on the associated semantics.

4.4 Implementation

We implemented Elzar as an LLVM compiler pass [134] that takes unmodified source code of an application and emits an AVX-hardened executable. We also implemented a fault injection framework to be able to test Elzar's fault tolerance capabilities.

4.4.1 Compiler Framework

Tool chain. We developed ELZAR as a compiler pass in LLVM 3.7.0 (∼ 600 LOC). Additionally, we extract the implementation of checks and recovery in a separate LLVM IR file (∼ 250 LOC). This separation allowed us to write the pass in a (mostly) target-independent way, i.e., AVX can be substituted by another similar technology (e.g., ARM Neon) by rewriting only the IR file with checks and recovery.

ELZAR is plugged in the usual build process of an application, i.e., there is no need to modify the source code or the makefiles/configuration scripts. To achieve this, we employ the LLVM gold linker plugin that can save the final optimized and linked LLVM bitcode to a file. ELZAR takes this file as input, adds AVX-based redundancy, and emits the hardened executable. Thus, ELZAR performs its transformation after all optimization passes and right before assembly code generation.

In order to be able to use AVX for replication, we disallow any vectorization in original programs. All other optimizations are enabled. Additionally, we run the *scalarrepl* pass to replace all aggregate data types (structs, arrays) because they are not natively supported by LLVM vectors we employ.

Pass details. The usual way to write AVX-enabled programs is to use AVX intrinsics or directly AVX inline assembly. This approach is the closest to "bare metal" and allows for fine performance tuning, but it is also time-consuming and error-prone. Moreover, using intrinsics or inline assembly would make it impossible to directly port ELZAR to a different technology than Intel AVX.

Fortunately, LLVM provides first-class *vector* types that were specifically introduced for SIMD programming and come with an extensive support for vector operations. The x86 code generator recognizes vectors and transforms them into AVX instructions. LLVM also introduces three special instructions to work with vectors, `extractelement`, `insertelement`, and `shufflevector` that are respectively mapped to AVX's `extract`, `broadcast`, and `shuffle`. Generally, we found vectors to be a very powerful abstraction, with the quality of the generated AVX code improving with each LLVM release.

With LLVM vectors, the process of AVX hardening becomes fairly trivial: (1) all data types of a program are transformed into corresponding vector types, (2) each of the synchronization instruction's arguments is extracted from a vector using `extractelement`, (3) each synchronization instruction's return value is broadcast to the whole vector using `insertelement`, (4) all other instructions are substituted to work on the corresponding vectors, and (5) checks and recovery routines are inserted before synchronization instructions. An example of ELZAR-transformed program is shown in Figure 4.7.

A nice feature of this vector-based approach is that one can abstract away from the underlying AVX implementation. As such, we do not need to care about most corner cases like vector-based integer division which is not implemented in AVX. We can still write it in an LLVM vector form, and the x86 code generator automatically converts it to four regular division instructions.

The careless use of vectors, however, may seriously hamper performance in some cases. For example, a straightforward implementation of branches with LLVM vectors results in a convoluted and ineffective instruction sequence; this is related to the fact that ELZAR uses `ptest` in an unusual manner that was not anticipated by the developers of the x86 code generator and is not efficiently supported in the pattern-matching rules. For such corner cases, we explicitly insert boilerplate code patterns as shown in gray in Figure 4.7b. This code actually generates the

(a) Native

```
1 loop:
2   r1 = add i64 r1, r2
3   c = cmp eq i64 r1, r3
4
5
6
7   br i1 c, exit, loop
```

(b) ELZAR

```
loop:
  r1 = add <4 x i64> r1, r2
  c1 = cmp eq <4 x i64> r1, r3
  c64 = sext c1 to <4 x i64>
  t = call ptest(<4 x i64> c64)
  c = cmp eq i32 t, 0
  br i1 c, exit, loop
```

Figure 4.7 – Example from Figure 4.4 as represented in simplified LLVM IR. Original code (a) operates on `i64` 64-bit integers. ELZAR (b) transforms the code to use `<4 x i64>` vectors of four integers. Since LLVM-based comparisons do not directly map to AVX, ELZAR inserts some boilerplate code (shown in gray).

`ptest-je` instruction sequence in the final executable, exactly as in Figure 4.4c.[3]

As discussed previously (§4.3.3), AVX natively supports only 8-, 16-, 32-, and 64-bit integers and 32- and 64-bit floating points. Since LLVM sometimes produces types with unsupported widths, we have no other choice but to extend them to supported types. In the case of integers, we take special care to sign- or zero-extend them. In some other cases (e.g., for SQLite3), we had to switch off the long-double type using predefined macros in the source code.

Libraries support. Most previous research in the area of ILR focused on hardening only the program's source code and left third-party libraries unprotected [75, 190, 191, 255]. This leads to better performance but also to lower fault coverage, because a fault in library code can go undetected. We notice however that many programs from the Phoenix and PARSEC benchmark suites, which are used in our evaluation, heavily utilize the standard C (libc) and math (libm) libraries. Therefore, to report more accurate numbers, we also harden a significant part of libc and libm. We decided not to harden the I/O, OS, and pthreads-related functions for our prototype implementation because their execution takes less than $\sim 5\%$ of the overall time. As a reference implementation, we chose the musl library with inline assembly disabled.

Limitations. Our prototype does not support inline assembly because LLVM treats assembly code as calls to undefined functions and provides no information about such code. Furthermore, our prototype does not have support for C++ exceptions.

4.4.2 Fault Injection Framework

For time budget reasons, we ran our fault injection experiments on a medium-sized cluster of computers without AVX installed. We therefore needed a fault injection tool that can emulate Intel AVX. Since available tools do not provide such support, we developed our own binary-level fault injector ($\sim$ 320 LOC) using Intel Software Development Emulator (SDE), which provides support for AVX instructions and gdb debugger. In the following, we give a high-level overview of our fault injector.

Basically, a fault injection campaign for each program proceeds in two steps. First, a program instruction trace is collected via the Intel SDE debugtrace tool. This preparatory step is required to automatically find and demarcate the boundaries of the hardened part of the program (remember that ELZAR does not harden external libraries and we do not want to inject faults

[3] To construct the boilerplate LLVM code, we consulted the source code of LLVM codegen's regression tests. These tests gave us a good understanding of how specific LLVM constructs are mapped to AVX assembly. This was literally a "test-driven development" experience.

FI outcome	Description	System
Hang	Program became unresponsive	Crashed
OS-detected	OS terminated program	
Elzar-corrected	Elzar detected and corrected fault	Correct
Masked	Fault did not affect output	
SDC	Silent data corruption in output	Corrupted

Table 4.1 – Fault injection outcomes classified.

into them). Knowing these boundaries, our fault injection tool can narrow down the set of instructions in which the fault can be injected.

Second, the program is executed repeatedly and, in each run, a single fault is injected (§4.3.1). To that end, a program-under-test is started under Intel SDE with a gdb process attached. To inject a fault, we dynamically create a new gdb script that sets a random breakpoint for a given occurrence of a particular instruction (otherwise gdb would always stop at the first occurrence of the instruction). When the program runs under Intel SDE with gdb attached, it stops at the breakpoint, the fault injection happens, and the now-faulty program continues execution. After the program terminates, our fault injection tool examines the program output, assigns a corresponding outcome (see below), and proceeds to another fault injection run.

Each fault injection run results in one of the outcomes listed in Table 4.1. To distinguish between the correct and corrupted system states, each program-under-test is run first without fault injections to produce a reference output ("golden run"). Consequently, after each run, the program output is compared against this reference output, and a SDC is signaled if two outputs differ.

We inject faults by overwriting an output register of an instruction where the breakpoint was set. We inject not only in AVX (YMM) registers but also in regular (GPR) registers. For YMM registers, we inject faults only in one element of the register to match our fault model (§4.3.1).

4.5 Evaluation

In this section, we answer the following questions:

- What is the performance overhead incurred by Elzar, and what are the causes for high overheads (§4.5.2)?
- How many faults are detected and corrected by Elzar during fault injection experiments (§4.5.3)?
- How does Elzar perform compared to a state-of-the-art ILR implementation (§4.5.4)?

4.5.1 Experimental Setup

Applications. Elzar was evaluated on two benchmark suites: Phoenix 2.0 [187] and PARSEC 3.0 [32]. Results are reported for all 7 Phoenix benchmarks and 7 out of 13 PARSEC benchmarks. The remaining 6 benchmarks from the PARSEC suite were not evaluated for the following reasons: *bodytrack* and *raytrace* use C++ exceptions not supported by Elzar, *facesim* crashes with a runtime error when built with LLVM, *freqmine* is based on OpenMP and does not compile under our version of LLVM, *canneal* has inline assembly and *vips* has long-double floats not supported by Elzar.

All applications were built with LLVM 3.7.0 and Elzar as described in §4.4.1. The native versions were built with `msse4.2` and `mavx2` flags to enable SIMD vectorization. The Elzar

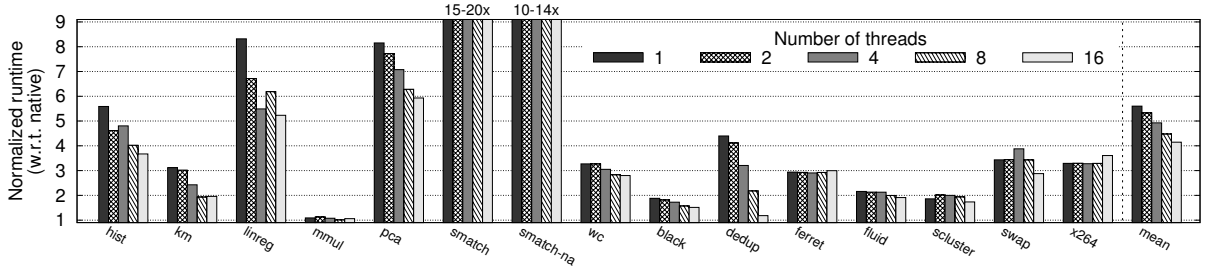

Figure 4.8 – Performance overhead over native execution with the increasing number of threads.

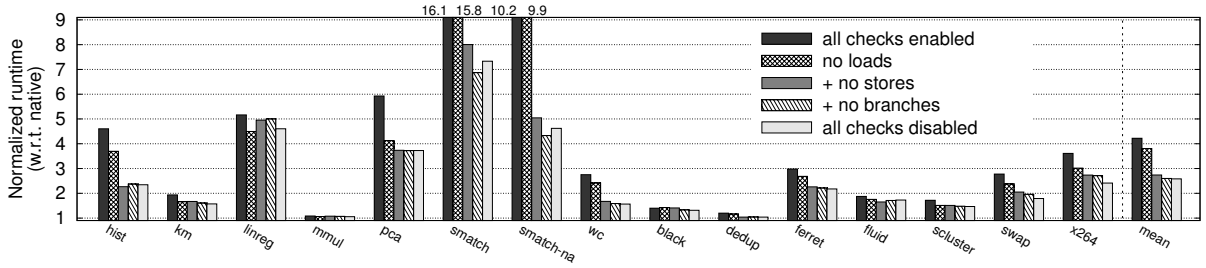

Figure 4.9 – Performance overheads breakdown by disabling checks (with 16 threads).

versions were built with all vectorization disabled, i.e., with `no-sse`, `no-avx`, `fno-vectorize`, and `fno-slp-vectorize` flags. For all versions, all other compiler optimizations were enabled (`O3` flag). Additionally, we used the `fno-builtin` flag to transparently link against our versions of libc and libm.

Note that we compare Elzar against the native version with all AVX optimizations enabled. As Figure 4.1 indicates, most benchmarks do not benefit from AVX. However, *string match* shows a 60% increase in performance. Therefore, we decided to also show how Elzar performs in comparison to the native version with AVX optimizations disabled; we refer to this experiment as *smatch-na* (for "string match no AVX").

Datasets. For the performance evaluation, we use the largest available datasets provided by Phoenix and PARSEC. However, for the fault injection experiments, we use the smallest available inputs due to the extremely slow fault injection runs.

Testbed. The performance evaluation was done on a machine with two 14-cores Intel Xeon processors operating at 2.0 GHz (Intel Haswell microarchitecture[4]) with 128 GB of RAM, a 3.5 TB SATA-based SDD, and running Linux kernel 3.16.0. Each core has private 32 KB L1 and 256 KB L2 caches, and 14 cores share a 35 MB L3 cache. For performance measurements, we report an average of 10 runs.

For fault injections, we used a cluster of 25 machines to parallelize the experiments. We injected a total of $2,500$ faults in each program. All programs-under-test were run with two threads to account for the impact of multithreading.

4.5.2 Performance Evaluation

Impact of Elzar and scalability. The performance overheads incurred by Elzar are shown in Figure 4.8. There is significant variability in behavior across benchmarks, with some showing

[4]We also performed experiments on Intel Skylake but the results were similar to Intel Haswell. Therefore, we omit them in our evaluation.

Benchmark	L1-miss	br-miss	loads	stores	branches
hist	0.66	0.01	53.21	26.67	9.56
km	1.48	0.33	20.83	0.48	14.96
linreg	2.05	0.01	18.02	0.21	3.82
mmul	62.39	0.14	40.16	0.07	10.10
pca	12.19	0.27	14.21	0.21	3.79
smatch	0.12	0.70	11.61	14.35	22.40
wc	10.94	3.31	29.75	23.63	13.67
black	0.40	1.21	9.38	2.84	15.63
dedup	4.30	3.80	30.08	13.55	12.01
ferret	4.69	12.65	14.47	2.28	17.42
fluid	1.17	14.70	11.77	2.58	14.29
scluster	4.17	1.47	32.60	0.43	9.33
swap	0.82	0.97	30.98	4.80	11.05
x264	0.34	0.31	26.83	8.32	21.00

Table 4.2 – Runtime statistics for native versions of benchmarks with 16 threads: L1D-cache and branch miss ratios, and fraction of loads, stores, and branches over executed instructions (all numbers in percent).

overheads as low as 10% (*matrix multiplication*) and some exhibiting up to 20× worse performance (*string match*). On average, the normalized runtime of Elzar is 4.1–5.6× depending on the number of threads.

For some benchmarks, there is also variability across the number of threads. Ideally, if a program has linear scalability, Elzar should incur exactly the same performance overhead with any number of threads, e.g., as in case of *word count* or *ferret*. However, some benchmarks such as *dedup* are well-known to have poor scalability, i.e., with many threads they spend a lot of time on synchronization [29]. Thus, Elzar's overhead is partially amortized by the sub-linear scalability of these benchmarks.

To gain better understanding on the causes of high overheads as well as the causes of high variability across benchmarks, we gathered runtime statistics for native and Elzar versions of all benchmarks. The results are shown in Tables 4.2 and 4.3. The benchmarks were run with 16 threads (and in the case of Elzar, with all checks enabled) and profiled using perf-stat to collect hardware counters of raw events such as the number of loads, stores, branches, all instructions and AVX instructions only, etc.

Based on the information from Tables 4.2 and 4.3, we can highlight several causes of high performance overheads. Firstly, as Table 4.3 shows, Elzar leads to an increase in the total number of executed instructions of 4–8× on average. This disappointingly high number is explained by the fact that Elzar adds wrapper instructions for loads, stores, and branches, as well as expensive checks on synchronization instructions (see §4.3.3).

Second, looking at the achieved Instruction-Level Parallelism (ILP) in Table 4.3, we notice that current x86 CPUs provide much better parallelization for regular instructions as compared to AVX instructions. As one example, *linear regression* achieves a high ILP of 6.51 instructions/cycle in native execution, but the AVX-based version reaches only a disappointing ILP of 1.7. Combined with the 10.49× increase in number of instructions for the AVX-based version, it is no surprise that *linear regression* exhibits an overhead of $\sim$ 5–8×.

Two benchmarks that show the lowest overheads are *matrix multiplication* and *blackscholes*. In the case of *matrix multiplication*, almost all of Elzar's overhead is amortized by a very

	Instruction-Level Parallelism (ILP), instr/cycle			Increase in # of instr w.r.t. native	
Benchmark	**Native**	**Elzar**	**SWIFT-R**	**Elzar**	**SWIFT-R**
hist	1.59	2.13	4.30	8.56	6.17
km	3.48	2.58	3.85	6.37	4.34
linreg	6.51	1.70	3.46	10.49	4.33
mmul	0.22	0.96	1.71	4.47	7.77
pca	2.61	2.28	3.89	6.82	9.45
smatch	2.38	3.26	3.46	32.72	11.56
wc	1.31	2.24	3.05	6.14	3.42
black	1.83	1.77	2.97	1.70	5.18
dedup	1.04	1.75	2.00	4.64	3.68
ferret	1.11	1.81	2.57	4.32	6.33
fluid	1.22	1.54	2.77	2.43	6.02
scluster	0.68	1.22	1.34	3.77	3.87
swap	1.97	2.06	2.68	3.50	4.40
x264	2.11	2.00	3.44	3.26	3.71

Table 4.3 – Runtime statistics for Elzar and SWIFT-R versions of benchmarks with 16 threads: Instruction-Level Parallelism (ILP) and increase factor in the number of executed instructions w.r.t. native.

poor memory access pattern that leads to 62.39% of all memory references missing L1 cache; in other words, *matrix multiplication* spends more time in waiting for memory than in actual computation. In the case of *blackscholes*, the main cause for low overheads is the small fraction of loads/stores (12.22%) and branches (15.63%).

Finally, we inspected the causes for extremely high overheads in *string match*. First of all, *string match* by itself significantly benefits from AVX vectorization (see Figure 4.1). Indeed, Elzar is $\sim$ 15–20$\times$ slower than the native version, but $\sim$ 10–14$\times$ slower than native with AVX vectorization disabled. Second of all, Elzar increases the total number of executed instructions by a factor of 32. Upon examining the source code of *string match*, we noticed that it spends most of the time in `bzero` to nullify some chunks of memory. LLVM produces a very effective assembly for this helper routine, but Elzar inserts wrappers and checks for the store and branch instructions in `bzero`, leading to much longer and slower assembly code.

Impact of checks. We also investigated the impact of checks inserted by Elzar (see §4.3.3). Figure 4.9 shows the results of successively disabling checks on loads, stores, branches, and all other instructions (e.g., function calls, function returns, atomics). Note that the results are shown for benchmarks run with 16 threads.

We observe that checks constitute a significant part of the overall performance overhead of Elzar. For example, disabling checks on loads and stores decreases the overhead from 4.2 to 2.7$\times$ on average, a difference of 55%. Disabling checks on branches leads to a negligible overhead reduction of 4%, which proves that our branch checking scheme is very efficient (§4.3.3).

We also observe that disabling checks on loads and stores respectively reduces the overhead by 11% and 40%, i.e., checks on stores have higher overheads than checks on loads. The reason is that stores require to check both the address and the value to store whereas loads only need to check the address.

Floating point-only protection. As AVX was initially developed to accelerate floating-point calculations, it is interesting to study the overheads when applying Elzar only to floating-point data. We thus developed a stripped-down version of Elzar that replicates floats and doubles but not integers and pointers, and ran tests on several PARSEC benchmarks that

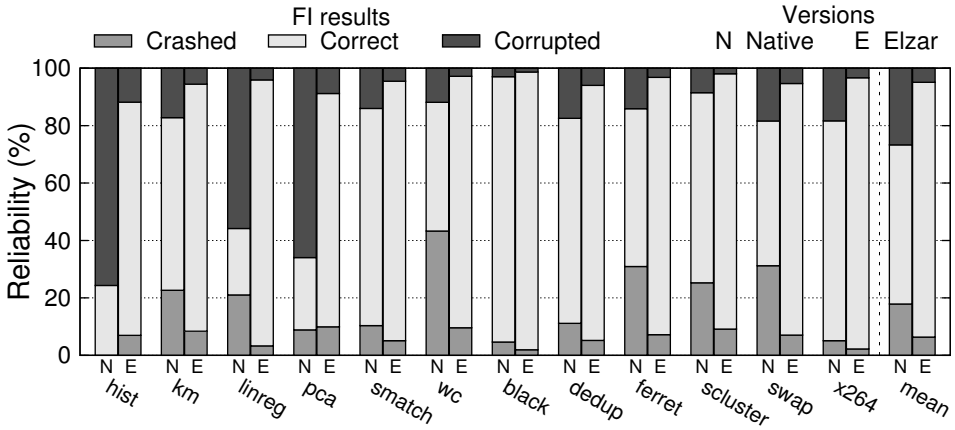

Figure 4.10 – Reliability of Elzar (fault injections done on benchmarks with 2 threads).

contain sufficiently many floating-point operations: *blackscholes* (47% of all instructions are floating-point), *fluidanimate* (32%), and *swaptions* (34%) [32].

Our results prove that Elzar hardens floating points with a low overhead. Depending on the number of threads, we observe a 9–35% performance overhead over native for *blackscholes*,[5] 10–18% for *fluidanimate*, and 40–60% for *swaptions*. The overhead is mainly caused by the checks on synchronization instructions.

4.5.3 Fault Injection Experiments

The results of the fault injection experiments are shown in Figure 4.10. On average, Elzar reduces the SDC rate from 27% to 5% and the crash rate from 18% to 6%.

Histogram has the worst result with 12% SDC. It highlights Elzar's window of vulnerability: address extractions before loads and stores. If a fault occurs in the extracted address, it will be used to load a value from the wrong address, and this value will then be broadcast to all replicas. In other words, the fault will remain undetected and may lead to SDC (similarly, such a fault may lead to a segmentation fault and therefore to a system crash). Indeed, Table 4.2 tends to confirm this observation since *histogram* has the highest number of memory accesses among all benchmarks. Similarly, *blackscholes* has the least number of loads/stores and thus has only 1% SDC.

4.5.4 Comparison with Instruction Triplication

Lastly, we compare Elzar against a common ILR approach based on triplication of instructions. More specifically, we compare Elzar against SWIFT-R [190] as shown in Figure 4.11. We re-implemented SWIFT-R because its source code was not publicly available; we employed manual assembly inspection to make sure our implementation of SWIFT-R produces fast and correct code.

In general, SWIFT-R incurs lower overheads than Elzar, 2.5× against 3.7× on average. Interestingly, Elzar performs better in three benchmarks, namely *kmeans*, *blackscholes*, and *fluidanimate*. To understand the differences between these approaches, we also report runtime statistics of SWIFT-R (Table 4.3).

[5] This is in line with the numbers reported by Chen et al. [51] where a single-threaded, manually written SSE-based version of *blackscholes* exhibits ∼ 30% overhead.

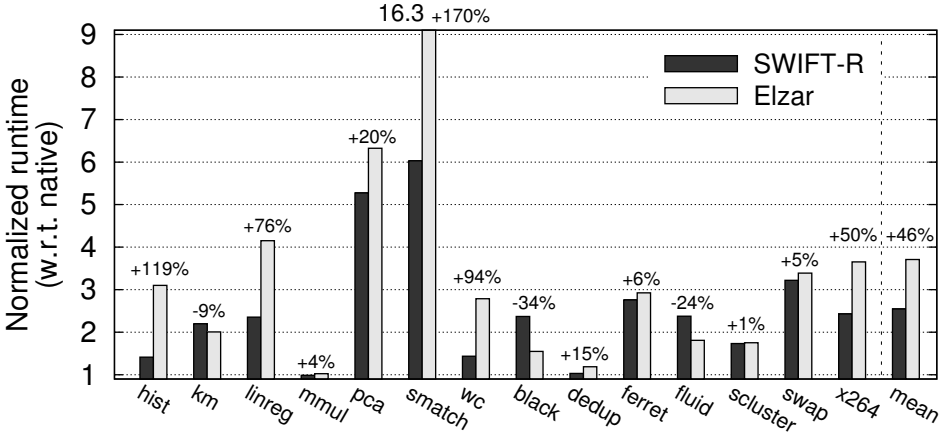

Figure 4.11 – Performance comparison of Elzar and SWIFT-R (with 16 threads).

We can draw two conclusions. First, SWIFT-R benefits from higher ILP, which is the key for its low performance overhead. As discussed before, Elzar takes a different stance and replicates not instructions but data; that is why it exhibits lower ILP but still performs on par with SWIFT-R in many cases.

Second, SWIFT-R significantly increases the number of instructions, which hampers its performance. Elzar has a smaller increase, proving our hypothesis that AVX-based ILR leads to less code blow-up. For example, Elzar outperforms SWIFT-R on *blackscholes* and *fluidanimate* exactly for this reason: even though SWIFT-R's ILP is almost 2× higher than Elzar, SWIFT-R produces ∼ 2.5–3× more instructions.

At the same time, SWIFT-R significantly outperforms Elzar in benchmarks that are dominated by memory accesses. In these cases, Elzar inserts a plethora of checks and wrappers, which results in a much higher number of instructions compared to SWIFT-R. This is exemplified by *histogram*, *string match*, and *word count*.

4.6 Case Studies

In this section, we report our experience on applying Elzar to three real-world applications: Memcached, SQLite3, and Apache.

Memcached key-value store. We evaluated Memcached v1.4.24 with all optimizations enabled, including atomic memory accesses. The evaluation was performed locally on the same Haswell machine used for other experiments, with 1–16 cores dedicated to the Memcached server and all other cores to the YCSB clients [56] for generating workload. We opted to show the local performance of Memcached because the performance in a distributed environment is limited by the network and not by the CPU.

Figure 4.12a shows the throughput of native and Elzar versions of Memcached run with two extreme YCSB workloads: A (50% reads, 50% writes, Zipf distribution) and D (95% reads, 5% writes, latest distribution). We observe that Elzar scales on par with native, achieving up to 72% of native throughput for workload A and up to 85% for workload D. We also observed in our experiments that the latency of Elzar is ∼ 25% worse than native (not shown here). Such good results are explained partially by Memcached's poor memory locality, which amortizes the costs of Elzar.

SQLite database. We evaluated SQLite3 using an in-memory database and YCSB workloads, similar to Memcached. We should note that SQLite3 has a reverse scalability curve because it was

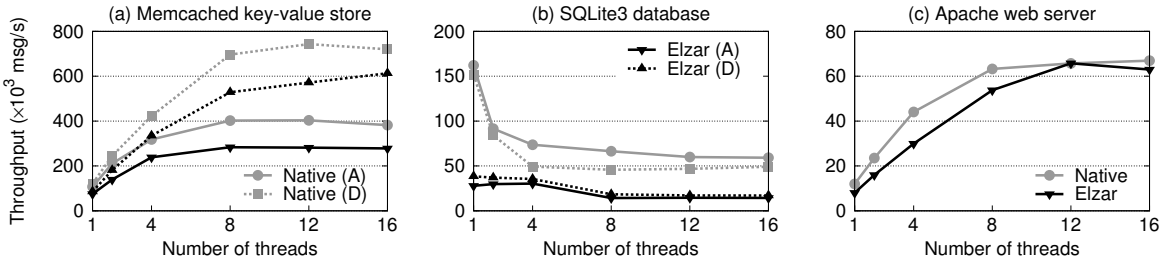

Figure 4.12 – Throughput of case studies: (a) Memcached key-value store, (b) SQLite3 database, and (c) Apache web server. Two extreme YCSB workloads are shown for Memcached and SQLite3: workload A (50% reads, 50% writes, Zipf distribution) and workload D (95% reads, 5% writes, latest distribution).

designed to be thread-safe and *not* concurrent. Therefore, SQLite3 exhibits worse throughput with higher numbers of threads.

The performance results are shown in Figure 4.12b. ELZAR performs poorly, achieving only 20–30% of the throughput of the native version. This overhead comes from the high number of locally near loads and stores, as well as function calls and function pointers. In all these cases, ELZAR inserts additional checks and wrappers that significantly degrade performance.

Apache web server. We evaluated the Apache web server using its "worker multi-processing module" with a single running process and a varying number of worker threads. As a client, we used the classical *ab* benchmark which repeatedly requests a static 1MB web page.

Figure 4.12c shows the throughput with varying number of threads. ELZAR performs very well, with an average throughput of 85% compared to native. We attribute this good performance to the fact that Apache extensively uses third-party libraries that are not hardened by ELZAR.

4.7 Discussion

In this section, we highlight performance bottlenecks in the current AVX implementation and discuss the possible remedies.

4.7.1 Performance Bottlenecks

Loads, stores, and branches. Even not taking into account the overhead of checks, ELZAR still performs 160% worse than the native version (see Figure 4.9, "all checks disabled"). This performance impact stems mainly from the three bottlenecks: loads, stores, and branches.

To understand the impact of each of the three main bottlenecks, we created a set of microbenchmarks. Each microbenchmark has two versions: one with the regular instruction (e.g., regular load) and one with the AVX-based instruction (e.g., AVX-based load as shown in Figure 4.5). In each microbenchmark, the instruction is replicated several times to saturate the CPU and wrapped in a loop to get execution time of at least 1 second. We wrote the microbenchmarks using volatile inline assembly to be sure that our instructions are not optimized away by the compiler; all tests were performed on our Intel Haswell machine.

The results of microbenchmarks are shown in Table 4.4. We conclude that adding `extract` and `broadcast` wrappers for AVX-based loads results in a $\sim 2\times$ increase of load execution time. Similarly, adding `ptest` for AVX-based branches leads to an overhead of $\sim 1.9\times$. Interestingly, AVX-based stores do not exhibit high overhead, which is explained by the fact that our Intel

	Loads	Stores	Branches
average-case	1.96	1.00	1.86
worst-case	2.06	1.14	1.89

Table 4.4 – Normalized runtime of AVX-based versions of microbenchmarks w.r.t. native versions.

Haswell has only one port to process data stores and thus the store operation itself is a bottleneck even in the native version.

Checks on loads and stores. As can be seen from Figure 4.9, ELZAR's checks on synchronization instructions contribute a significant amount of the overhead (39% on average). Specifically, checks on loads and stores account for most of the overhead because of the complicated sequence of check instructions. At the same time, checks on branches add only 5% overhead due to an efficient re-use of `ptest` already needed for branching itself (see Figure 4.6).

Missing instructions. Our Intel Haswell supports the AVX2 instruction set. Though AVX2 provides instructions for almost all operations, some classes of operations are missing. Two prominent examples are integer division and integer truncation. In the case of integer divisions, ELZAR generates at least four regular division instructions and the corresponding wrappers to extract elements from the input YMM registers and insert elements in the output YMM register; with truncations, the situation is similar. Clearly, emulating such missing instructions via a long sequence of available AVX instructions can lead to tremendous slowdowns.[6] For example, our microbenchmark for truncation exhibits overheads of 8×.

4.7.2 Proposed AVX Instructions

ELZAR could greatly benefit from a rather restricted set of new AVX instructions as proposed next. The instructions we propose are not ELZAR-specific and other applications can find use for them. Moreover, some of them are already introduced in the AVX-512 instruction set which will be available in Intel's upcoming CPUs.

Loads and stores (gathers and scatters). As is clear from Figure 4.5, regular load instructions are restricted in that they require an address operand specified in a general-purpose register (GPR). ELZAR would need an instruction that can load the elements of an output YMM register from several addresses specified in the corresponding elements of an input YMM register.

The current implementations of AVX already support a similar instruction called `gather` (Figure 4.13, left). Unfortunately, gather instructions still require a base address from a GPR and do not yet support all data types. Moreover, the current implementation is slower than a simple sequence of several loads [101]. Nonetheless, we can expect that future AVX implementations will provide better support for gathers so that they can be successfully exploited in ELZAR. Interestingly, introducing gathers could also close a window of vulnerability discussed in §4.5.3.

A similar argument can be made regarding stores. AVX-512 introduces `scatter` instructions that can store elements from a YMM register based on the addresses in another YMM register. Thus, ELZAR could advantageously substitute current implementations of stores with scatters.

Comparisons affecting FLAGS. Currently, AVX exposes only one instruction, `ptest`, that can affect control flow by toggling the FLAGS register. Accordingly, ELZAR inserts an AVX-based

[6] One simple optimization would be to identify missing instructions and emit a sequence of only 3 divisions/truncations. However, this solution still requires extracting elements and then combining them again. For our prototype, we had no need to implement such an optimization because these instructions are rare.

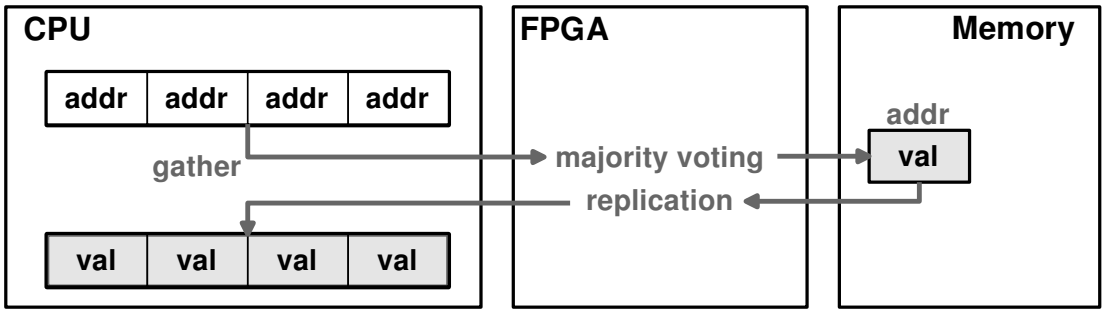

Figure 4.13 – Offloading checks to a FPGA accelerator via gather/scatter AVX instructions.

comparison followed by a `ptest` to implement branching, as shown in Figure 4.5. Table 4.4 indicates that this additional operation leads to an overhead of almost 2×.

The only way to improve performance of branches is to re-implement the logic of the usual comparison instructions. In x86, a `cmp` instruction performs both the comparison *and* the toggling of FLAGS. We would propose a similar family of AVX-based comparisons which could output the result of comparison (§4.2.3) *and* set the corresponding flags in FLAGS. Such improved comparisons could be also beneficial for vectorized applications that rely heavily on `ptest`.

Checks on loads and stores. Checks on loads and stores are implemented via an inefficient `shuffle-xor-ptest` sequence (see Figure 4.6). Having a single comparison instruction similar to the comparisons described above would greatly decrease the overheads of checks. Such an instruction would perform a pair-wise comparison of neighboring elements in a YMM register (so-called "horizontal" comparison) and toggle FLAGS. Thus, a long sequence of instructions from Figure 4.6 would be replaced by a single instruction.

The benefits of such an instruction for other applications than Elzar are unclear. Thus, in the next section we propose a more viable alternative involving an FPGA accelerator.

Truncations, divisions, and others. Curiously, a family of truncation operations (`vpmov`, `vcvt`) is already implemented in AVX-512. Integer division and modulo operations are quite rare and their absence is unlikely to lead to significant overheads; thus we believe these instructions are no candidates for future AVX implementations. We probably missed some other instructions that are not present in AVX, but we believe they are sufficiently uncommon to not provide much benefit for Elzar.

4.7.3 Offloading Checks

In order to decrease the overhead of checks, we can take advantage of the upcoming FPGA accelerators that will become part of CPUs [90]. These FPGAs will be tightly coupled with the CPU and both will share the virtual memory of a process. As such, it will likely be possible to offload some functionality from the CPU to the FPGA. As of the moment of this writing, details on the Intel FPGA accelerators are not public and our speculations may prove wrong when the final products are released.

We propose to offload the checks on loads and stores to the FPGA as follows (see Figure 4.13). For an Elzar-hardened program, all loads and stores are tunneled through the FPGA. The FPGA checks all copies of the address (for loads) and all copies of the value (for stores) and implements majority voting to mask possible faults. After that, the FPGA performs a load from a correct address or a store of a correct value. For loads, the FPGA also replicates the loaded value and sends it back to the CPU.

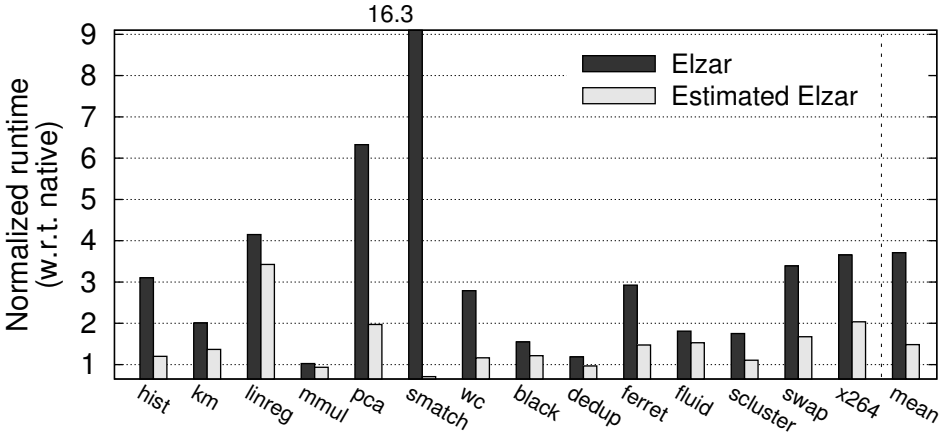

Figure 4.14 – Estimation of performance overhead of Elzar with the proposed changes to AVX (with 16 threads).

4.7.4 Expected Overheads

To summarize, our proposed set of changes in the underlying hardware is as follows: (1) using AVX-based gathers/scatters for loads/stores, (2) using AVX-based comparisons that can directly toggle FLAGS, and (3) offloading checks on loads/stores onto an FPGA.

To understand the synergistic effect of the proposed changes, we performed the following experiment. First, we note that it is not possible to substitute AVX-based loads, stores, and branches with cheaper alternatives without disrupting the original flow of our benchmarks. Thus, we do a "reverse" comparison, i.e., instead of accelerating Elzar, we decelerate the native versions by adding dummy inline assembly around loads, stores, and branches. The assembly we add consists of instructions that Elzar uses as wrappers (see §4.3.3), e.g., we add dummy `extract` and `broadcast` for each load and a dummy `ptest` for each branch. (Adding dummy assembly can affect code generation and the CPU pipeline, but on average produces an adequately accurate estimation.) Consequently, the overhead of Elzar with regard to this impaired native version serves as a rough estimate of Elzar overheads with our proposed changes.

The results of this experiment are shown in Figure 4.14. The average performance overhead is estimated to be 48%, i.e., an improvement of 150% over current Elzar. Many benchmarks exhibit very low overhead of 10–20%. The case of *string match* is peculiar, since it turns out to be faster than the native version in our experiment. Upon reading the disassembly, we found out that our dummy inline assembly in the "decelerated" native version prevented an optimization of function inlining: this led to a faster execution time of the Elzar version than the "decelerated" version.

4.8 Conclusion

We presented Elzar, an AVX-based implementation of ILR. Elzar achieves fault tolerance not by replicating instructions, but by replicating data inside AVX registers. To our disappointment, we found out that AVX suffers from several limitations that lead to poor performance when used for ILR. The observed performance bottlenecks are primarily caused by the lack of suitable control flow and memory access instructions in the AVX instruction set, which necessitates the introduction of wrappers and ineffective checks for some types of instructions. We believe that these limitations can be overcome by simple extensions to the AVX instruction set (see §4.7).

We proposed improvements for the future generations of AVX that can lower the overheads of ELZAR down to $\sim 48\%$ according to our study.

5 HAFT: Leveraging Transactional Synchronization Extensions

The previous chapter discussed Elzar that is able to detect and mask transient CPU faults. Elzar has a high variance in performance overhead: while in some cases Elzar performs well, many workloads suffer an unacceptable overhead of 4–5×. Therefore, in this chapter we introduce our second and more successful attempt to tackle the same problem – Hardware-Assisted Fault Tolerance (HAFT).

HAFT efficiently utilizes Intel Transactional Synchronization Extensions (TSX) to roll-back erroneous program execution and exhibits only 2× slowdown on average. As we show in the following, performance overhead of HAFT is significantly lower than other state-of-the-art approaches to tolerate CPU faults. This is possible due to hardware-assisted nature of HAFT.

The content of this chapter is based on the paper "HAFT: Hardware-assisted Fault Tolerance" presented at EuroSys'2016 [127]. The paper was a joint collaboration with Rasha Faqeh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer.

5.1 Rationale

Transient faults, or soft errors, in CPUs can cause arbitrary state corruptions during computation. Several studies suggest that transient errors are a pervasive cause of software systems failures [88, 165, 202]. These studies point to a wide range of reasons for such transient faults in CPUs, including manufacturing problems, overheating, dynamic voltage scaling, hardware/software incompatibility, or power supply faults.

These issues are amplified in the new processor architectures that are continuously boosting performance with higher circuit density using ever-shrinking transistor sizes, and are simultaneously achieving higher energy efficiency by operating at lower voltages [35]. These trends negatively affect the reliability of the underlying hardware [213]. Furthermore, the advancements in the 7 nm chip technology with near-threshold computing (dim silicon) will only worsen the reliability of CPUs [211].

The unreliability of CPUs becomes a particularly serious concern for modern online services running in data centers. Given the sheer scale at which these services operate, the transient faults occur at a surprisingly high rate and tend to reappear more frequently after the first occurrence [88, 165, 202]. Anecdotal evidence indicates that a single transient fault in the hardware can lead to process state corruption [47, 62], data loss [163], and in some unfortunate cases, errors can propagate throughout the system causing outage of the entire service [8].

As a result, software systems running in modern data centers are being increasingly adapted to tolerate transient faults. For instance, Mesa [2], a data warehousing system at Google, uses application-specific integrity checks to deal with data corruptions during computation. In fact, many large-scale systems employ ad-hoc mechanisms to detect data corruptions, such as source code assertions, periodic background integrity checks, and message checksums throughout the system [2, 8, 57, 115]. However, these ad-hoc solutions can only protect from errors anticipated

by the programmer and may fail to detect arbitrary hardware faults.

Researchers have proposed a series of disciplined *hardening* approaches to protect software systems against transient faults [126, 191, 214, 257, 258]. In particular, these hardening approaches add redundancy at the level of program instructions, threads, or whole processes, and insert periodic comparisons of redundant copies to detect transient faults. While these approaches have been an active area of research for decades, almost all of the existing solutions in this domain target sequential programs only, making them impractical for ubiquitously deployed multithreaded software systems.

To support multithreaded programs, a few hardening systems have been recently proposed [26, 31, 65]. However, all these systems still have at least one of the following limitations: *(i)* they require manual efforts to modify the application, e.g., to annotate the protected code regions, *(ii)* they target restrictive programming models, e.g., assuming only event-based applications, *(iii)* they rely on application-specific checks leveraging the high-level programming languages such as Apache Pig [174], *(iv)* they require operating system support, deterministic multithreading and/or spare cores for redundant execution, *(v)* they provide only fail-stop semantics without providing recovery from faults.

In this chapter, we propose a Hardware-Assisted Fault Tolerance (HAFT) technique that overcomes the aforementioned limitations. HAFT applies to unmodified applications on the existing operating systems running on commodity hardware. HAFT targets the general shared-memory multithreaded programming model supporting the full range of synchronization primitives. Moreover, HAFT neither enforces deterministic execution nor requires spare cores, and thereby, it does not limit the available application parallelism, which is crucial for imposing low performance overheads. Finally, HAFT achieves high availability by providing fault detection *as well as* recovery from faults.

HAFT is a compiler-based hardening approach that leverages two techniques: Instruction-Level Redundancy (ILR) [191] for fault detection and Hardware Transactional Memory (HTM) [254] for fault recovery. To achieve fault tolerance, HAFT transforms an application in the following way. First, the instructions of the application are replicated and periodic integrity checks are inserted. The replicated instructions create a separate data flow along the original one, and both flows are efficiently scheduled via instruction-level parallelism of modern CPUs. Next, the whole execution of a program is covered with HTM-based transactions to provide fault recovery. When a fault is detected by ILR, the transaction is automatically rolled back and re-executed. The HTM implementation we employ is best-effort, which renders HAFT's recovery guarantees best-effort as well. Nonetheless, our evaluation shows that clever placement of transactions allows HAFT to achieve high availability even in the presence of frequent faults.

We implemented HAFT as an extension of the LLVM compiler framework to transform unmodified application code. In our evaluation, we applied HAFT to the Phoenix and PARSEC benchmark suites. The fault injection experiments show that the average number of data corruptions decreases from 26.2% to 1.1% and on average, 91.2% of the data corruptions can be corrected. In terms of performance, applications hardened with HAFT run on average 2× slower than native versions. We also applied HAFT to a set of real-world applications including Memcached, Apache, and SQLite. Furthermore, a comparative evaluation revealed that HAFT imposes 30–40% less performance overhead than the state-of-the-art solution for multithreaded programs [26].

5.2 Background and Related Work

We discuss below existing approaches to fault tolerance and uses of hardware transactional memory for fault recovery.

5.2.1 Fault Tolerance Approaches

State Machine Replication (SMR). To achieve high availability, some software systems [22, 41, 102] use State Machine Replication (SMR) [201]. These systems typically assume a crash fault model. However, this model does not cover transient faults which might lead to arbitrary state corruptions.

Byzantine Fault Tolerance (BFT) [45] tolerates not only crashes, but also transient hardware faults (and even malicious attacks). Unfortunately, BFT incurs prohibitive overheads because of the overly pessimistic fault model. To decrease the performance overhead of BFT, researchers explored the use of specialized trusted hardware [118, 235], relaxed network assumptions [181, 182], speculative execution of requests [124], and OS support [122]. In contrast, HAFT imposes low overheads by assuming a *more restrictive* fault model: it protects only against hardware non-malicious faults.

To support multithreaded programs, all SMR solutions require some form of deterministic execution. Crane [61] builds on deterministic multithreading [139, 175], Eve [119] speculatively executes requests and falls back to deterministic re-execution upon conflicts, and Rex [89] enforces deterministic replay of the primary's trace on secondary replicas. HAFT supports non-determinism because it requires no replicas, achieving fault tolerance *locally*.

Due to its local fault tolerance, we consider HAFT to be not a substitute for SMR, but rather a complementary approach. Indeed, SMR is usually applied only to the "control path" of distributed software systems, e.g., coordination services such as Chubby [41] and ZooKeeper [102]. HAFT can, in particular, be used to protect the data path, ensuring that the main computation itself is not affected by transient faults.

Local hardening approaches. Due to lack of adoption of BFT [222], researchers actively explored local hardening approaches that protect against data corruptions. These approaches *harden* programs by adding redundancy at the level of program instructions (see §5.3.2), threads, or processes.

Redundant Multithreading (RMT) [153, 238, 257] spawns an additional, trailing thread for each original thread in a program and redundantly executes it on a spare core. In the same spirit, Process-Level Redundancy (PLR) [65, 214, 258] uses redundant processes instead of threads, with processes-replicas having their own private memory space and synchronizing on system calls. Both of these approaches require spare cores for redundant execution and are thus not suitable for multithreaded programs that tend to occupy all available cores. Moreover, they only support deterministic program executions.

Scalable Error Isolation (SEI) [26, 57], a recently proposed fault detection technique, is the only approach we are aware of that does not require deterministic execution of multithreaded programs. It assumes an event-driven programming model, executing each event handler twice and appending a CRC signature to all output messages. Thereby, SEI guarantees end-to-end protection from data corruptions in a distributed environment. Unfortunately, SEI requires manual effort to adapt existing code bases. HAFT, in contrast, applies to unmodified programs and targets the common shared-memory programming model. Finally, the authors of SEI assume a broader fault model than HAFT, with no bound on the number of corrupted variables per

one event handler, and formally prove the correctness of SEI under this model. HAFT provides weaker guarantees with the benefit of better performance (§5.6.1).

Most of the approaches above only provide fault detection and fail-stop behavior. Coupling them with fault recovery mechanisms [184, 190, 218, 223] is considered a non-trivial task. HAFT, on the other side, seamlessly combines fault detection and fault recovery.

Lock step CPUs. Traditionally, incorrect execution of programs has been detected via lock step CPUs, where two CPUs execute the same application in parallel and synchronize their outputs. Lock step CPUs are still actively used for critical applications in the embedded domain and on mainframes. By its very nature, lock-stepping requires deterministic core behavior and cannot be applied to modern CPUs that have become increasingly more non-deterministic [28]. Moreover, lock step CPUs provide only fault detection, requiring a separate mechanism for recovery. Being a more light-weight technique, HAFT supports automatic recovery and non-determinism both on the application as well as on core level.

5.2.2 Leveraging HTM for Fault Recovery

Transactional memory was first proposed as a better alternative for traditional lock-based synchronization in concurrent shared-memory applications [100, 148]. However, it also provides strong isolation guarantees and local rollback and can be exploited as a recovery technique [76].

Intel TSX. In this chapter, we focus on a recent HTM implementation called Intel Transactional Synchronization Extensions (TSX) [254]. More specifically, we use the Intel Restricted Transactional Memory (RTM) interface.

RTM introduces a set of new instructions to explicitly begin, commit, and abort transactions. Applications can mark the boundaries of transactions using XBEGIN and XEND, explicitly abort them using XABORT, and check if a CPU core is currently executing in a transaction using XTEST.

In Intel TSX [135, 240, 254], transactions utilize the L1 data cache as a local buffer to track their read- and write-sets. An optimized cache coherency protocol is used to detect collisions between concurrent transactions. Read- and write-sets are implemented at the (64-byte) cache line granularity. A cache line that is part of the read-set can be evicted *without* necessarily causing the transaction to abort, while evicting a cache line that is part of the write-set *always* aborts the transaction.

Internally, XBEGIN commands the core to take a snapshot of its register state and to start tracking the changes done by the transaction in the read- and write-sets. If the core detects a conflict with another transaction (or even with non-transactional code), it aborts its transaction. Otherwise, upon execution of XEND, the transaction commits by atomically flushing its write-set to RAM. If the transaction was aborted (either implicitly or explicitly via XABORT), its read- and write-sets are discarded, the registers' state is restored from the snapshot, and the execution jumps to an abort handler specified as argument to XBEGIN. The abort handler is usually implemented to retry a transaction several times before resorting to a fallback path.

Applicability to fault tolerance. Given that Intel TSX is targeted primarily for synchronization, it is not immediately obvious whether it can be also used for fault tolerance. Although some research has recently shown promising results when using HTM for recovery [92, 250, 251], the question remains: can commodity-hardware HTM implementations provide efficient and comprehensive support for fault recovery?

In HAFT, the whole application must be wrapped in hardware transactions to support fault recovery. Yet, several design choices of Intel TSX are driven by the assumption that transactions cover only a handful of small critical sections. This limits TSX's applicability

(a) Native

```
11
12 z = add x, y
13
14
15
16
17 ret z
```

(b) ILR

```
z = add x, y
z2 = add x2, y2
d = cmp neq z, z2
br d, crash

ret z
```

(c) HAFT

```
xbegin
z = add x, y
z2 = add x2, y2
d = cmp neq z, z2
br d, xabort
xend
ret z
```

Figure 5.1 – HAFT transforms original code (a) by replicating original instructions with ILR for fault detection (b) and covering the code in transactions with Tx for fault recovery (c). Shaded lines highlight instructions inserted by ILR and Tx.

for the whole-application fault recovery in the following ways. Firstly, Intel TSX provides no guarantees that a transaction will eventually commit even when applied to sequential code [254]. Secondly, transaction size is limited by the CPU cache size and by the interval between timer interrupts. For example, TSX has the following rough thresholds after which more than 10% of transactions abort: 16 KB for the write set, 1024 KB for the read set, and 1 million CPU cycles (approx. 0.3 ms) [135, 240]. Thirdly, all interrupts/signals (including page faults) and so-called "unfriendly" instructions (x87 floating-point, TLB or EFLAGS manipulation, system calls) force a core to abort any active transactions.

Thus, to guarantee forward progress, HAFT needs a non-transactional fallback path in case transactional execution does not succeed. Consequently, if a fault happens during one of these non-transactional fallbacks, it cannot be recovered. Moreover, a HAFT transaction must be sufficiently small to finish before a timer interrupt happens or the L1 cache overflows. Finally, several factors such as CPU hyper-threading, memory false sharing, and unfriendly instructions also negatively affect HAFT's recovery capabilities.

5.3 HAFT

HAFT is a compiler-based transformation that consists of two components: ILR for fault detection and Tx for fault recovery. Figure 5.1 shows an example of HAFT transforming a simple code snippet. ILR is applied first, replicating all instructions except control-flow ones (Figure 5.1b). To achieve fault detection, ILR inserts a check before returning the result; if two copies of data diverge, then a fault is detected and an error is reported by enforcing program termination. To achieve fault recovery, Tx is applied next, covering the code in transactions and substituting crashes by transaction aborts (Figure 5.1c). In this case, if a fault is detected at run-time, the current transaction is rolled back and re-executed. HAFT attempts to re-execute aborted transactions for a bounded number of times (three by default in our implementation), after which the code executes non-transactionally until a new transaction begin is encountered. If a fault occurs during such a non-transactional part of code, ILR has no other choice but to terminate the program. Therefore, HAFT provides best-effort fault recovery, falling back to fail-stop semantics in rare cases when the limit of re-executions is exhausted.

5.3.1 System Model

Before we explain the basic design of HAFT, we present the system model assumed in this work.

Fault model. HAFT protects against single event upsets (SEU), i.e., a corruption of a single CPU register or a single miscomputation in a CPU execution unit that would otherwise lead to

Silent Data Corruptions (SDC) [35]. The SEU model covers transient hardware faults due to particle strikes, aging, dynamic voltage scaling, device variability, etc. We assume that at most one SEU fault occurs during one hardware transaction. HAFT can probabilistically protect against bursts of faults as long as duplicated data flows result in differing corrupted state. Due to the choice of ILR for fault detection, HAFT cannot tolerate common-mode failures; however, single uncorrelated bit-flips are considered to be the dominant cause of CPU faults [35].

Additionally, HAFT assumes that RAM and caches are already protected by ECC [191]. This assumption usually holds for data center servers, e.g., our experimental machine has memory ECC support and all cache levels are protected by ECC or parity.

The design of HAFT assumes correct execution of Intel TSX. The TSX transactional state resides in the L1 cache and thus is protected by ECC. However, if XBEGIN, XEND, or XABORT perform an erroneous operation (e.g., not all cache lines are flushed to RAM or rolled back), the program state becomes inconsistent.

Memory model. HAFT relies on the Release Consistency (RC) memory model [83], which requires that all shared memory accesses are done via synchronization primitives. The RC model guarantees correctness for data-race free programs and enables the optimizations on shared memory accesses (§5.3.3) which would not be feasible under stricter memory models such as sequential consistency [132]. Indeed, a data race would lead to a discrepancy in results under our optimized ILR that in turn would lead to either a transaction abort (if executed inside an HTM transaction) or a program crash (if executed in non-transactional part of code). To allow for the shared memory accesses optimization, we assume data-race free executions.

Synchronization model. Our current implementation supports POSIX threads API and C/C++ atomic synchronization primitives. In fact, HAFT works with any synchronization mechanism that maps directly to LLVM atomic instructions [140]. Thus, even lock-free programming patterns are supported as long as they are explicitly implemented via atomics. Ad-hoc synchronization mechanisms such as user-defined spin locks are not supported, but they are error-prone and not recommended for use [247].

HAFT is not readily applicable to HTM-enabled applications. Our current prototype does not expect TSX instructions in the native program and therefore could break semantics assumed by the programmer. However, in §5.6.1 we show that HAFT can be efficiently expanded to applications that use lock elision as their main synchronization primitive.

5.3.2 Basic Design

In the following, we describe the basics of ILR and Tx. For simplicity of presentation, we first consider sequential applications. We then show in §5.3.3 that HAFT's basic design naturally extends to multithreaded programs and we further enhance it with optimizations to improve performance and reliability.

Instruction Level Redundancy (ILR). HAFT utilizes Instruction Level Redundancy (ILR) for fault detection [75, 126, 170, 191]. ILR operates on one copy of the memory state and checks the results of computations before each update to memory. This way, ILR does not increase the memory footprint and allows non-determinism in applications, selective hardening of functions, and interoperability with legacy libraries.

To add redundancy, ILR creates a second, shadow data flow along the master flow, with shadow instructions working on their own registers (see Figure 5.1b). Note that the shadow instructions are executed in the same thread. Since there are no dependencies between master and

```
int c = 123;                                                    ;; Original C code
void foo() { while (c < 1000) c++; }
```

```
1  entry:                                                       ;; Basic block 1
2  tx-begin()
3  dup c.init = load c.adr
4  loop:                                                        ;; Basic block 2
5  tx-cond-split()
6  dup c = phi [c.init, entry], [c.new, loop]
7  dup c.new = add c, 1
8  dup cnd = cmp eq c.new, 1000
9  tx-counter-inc(7)
10 br cnd, end, loop
11 end:                                                         ;; Basic block 3
12 store c.new, c.adr
13 c.tmp = load c.adr2                                          ;; ILR check ⎤
14 d = cmp neq c.tmp, c.new2
15 br d, xabort                                                              ⎦
16 tx-end()
```

Figure 5.2 – HAFT transactification example: original C code (top) and LLVM IR generated for it (bottom). Lines 3 and 6-8 show original instructions replicated by ILR, lines 12-15 show a check on store inserted by ILR. Shaded lines highlight calls to HTM helper functions inserted by Tx.

shadow instructions, they can execute in parallel, benefiting from the instruction-level parallelism present in all modern CPUs.

The basic version of ILR replicates all instructions except control flow (branches, function calls, returns) and memory-related (loads, stores, atomics) instructions. If a non-replicated instruction returns a value, as in case of loads and function calls, this value is immediately replicated for later use in the shadow data flow using a register-to-register move.

To achieve fault detection, ILR inserts checks on every instruction that updates memory or control flow. Each check compares a master and shadow data copies, reporting an error upon detecting a discrepancy (Figure 5.1b, lines 4-5). ILR has a few *windows of vulnerability*, i.e., it cannot detect faults occurring in-between the checks and the checked instructions [191].

In the context of this work, the important advantages of ILR are its fine-grained checking and in-thread redundancy. As we utilize HTM for recovery, we are restricted to transactions of small size operating on a single core. The small size of transactions implies that the checks must be inserted as close as possible to the potential sources of transient faults. The single-core requirement implies that the fault detection mechanism must not use additional cores. ILR fulfills both these requirements.

Transactification (Tx). In addition to ILR for fault detection, HAFT also employs transactification (Tx) to achieve fault recovery. The Tx pass of HAFT inserts transaction boundaries in an application so that it always executes inside HTM transactions. The challenge here is to determine correct transaction boundaries. HTM is traditionally used to protect critical sections, with tiny transactions scattered around the code. In that case, the programmer herself assigns transaction boundaries and ensures the optimal transaction size. HAFT, however, is a fully automated technique that transparently covers the whole application with transactions at compile-time. Thus, an algorithm to efficiently put transaction boundaries—a *transactification* algorithm—is required.

To best illustrate the mechanisms underlying the transactification process, consider the simple

example shown in Figure 5.2.[1] It consists of a single function incrementing a global variable within a loop.[2] Here, ILR is first applied on original code: instructions on lines 3 and 6-8 are replicated, and a store instruction is augmented with a check on lines 12-15; for simplicity, we omit the check before a branch on line 10; refer to §5.3.3 for details. Next, Tx is invoked to insert transaction boundaries.

A simple transactification algorithm would be to insert boundaries *only* at the level of separate functions (lines 2 and 16). But in reality functions can be arbitrarily large and can in turn call other functions, whereas hardware transactions are severely restricted in size as discussed in §5.2.2. Therefore, transactions are bound to abort under this naïve approach, i.e., the rate of successfully committed transactions would be prohibitively low.

Another extreme is to cover each *basic block* (single entry single exit section of code) in a separate transaction. In this case, since basic blocks usually contain just a handful of instructions, all transactions should eventually commit. In our example, we would have three transactions covering the three basic blocks (lines 1–3, 4–10, and 11–16). However, the second basic block corresponds to the body of the loop that executes several hundreds of times, creating several hundreds of tiny transactions at run-time. Unfortunately, producing that many transactions introduces high performance penalty (see §5.5.3).

Therefore, to achieve high commit rate and low performance overhead, Tx takes a balanced approach and inserts hardware transactions at the granularity of functions *and* loops. The algorithm tries to maximize the size of transactions, while at the same time keeping it less than a predefined *threshold* to avoid capacity aborts and ensure that the majority of transactions can commit successfully. To that end, given that the size of transactions is not always known at compile-time because the number of loop iterations is not always known statically, Tx keeps track of the number of instructions executed inside transactions at run-time using per-thread instruction counters.

Tx inserts transactions at compile-time by inspecting all functions in the application and applying a transformation pass that adds transaction demarcations at specific locations. It relies upon the following helper functions that embed the low-level HTM instructions necessary for transactional execution:[3] *(i)* `tx-begin()` starts a new hardware transaction and resets the thread-local counter. If the transaction does not succeed after a number of retries (default is three), the code executes non-transactionally. *(ii)* `tx-end()` commits the current transaction. *(iii)* `tx-cond-split()` if the thread-local counter exceeds a predefined threshold, commits the current transaction, starts a new hardware transaction, and resets the counter. *(iv)* `tx-counter-inc()` increments the thread-local counter by the number of instructions given as parameter.

For each function in an application, Tx first inserts a transaction begin at function entry (line 2) and a transaction end before function return (line 16).

After that, loops are transformed. For each loop, Tx inserts a conditional statement at the entry point to commit the current transaction and start a new one only when the instruction counter exceeds a predefined threshold (line 5). This optimization yields significant performance gains since the counter check is significantly cheaper than systematically starting a new transaction at each iteration.

The instruction counter is incremented at each loop latch, i.e., at each point where the execution

[1] We use a simplified LLVM IR notation; the `phi` instruction selects a value depending on the predecessor of the current block.

[2] Note that, for the sake of illustration, we have simplified the generated LLVM code and discarded certain compiler optimizations.

[3] The code of these functions consists of just a few instructions that are subsequently inlined by the optimizer for performance reasons.

(a) Unoptimized

```
   ;; Load (atomic)
1  d = cmp neq adr, adr2
2  br d, xabort
3  val = load adr
4  val2 = move val
   ;; Store (atomic)
5  d = cmp neq val, val2
6  br d, xabort
7  d = cmp neq adr, adr2
8  br d, xabort
9  store val, adr
```

(b) Optimized

```
;; Load (race-free)
val = load adr
val2 = load adr2


;; Store (race-free)
store val, adr
tmp = load adr2
d = cmp neq tmp, val2
br d, xabort
```

Figure 5.3 – Memory accesses in ILR. Unoptimized (a) is used for atomic accesses while optimized (b) is safe for race-free programs. Shaded lines highlight instructions of the original master flow.

can jump back to the entry point of the loop (line 9). The increment value is computed as the longest path in the loop body leading to the latch, i.e., it corresponds to a worst-case scenario and the counter represents an upper bound of the transaction size. In the example, the increment value of 7 corresponds to 3 original instructions in the loop, 3 shadow instructions added by ILR, and one branch instruction. Note that a fault in the instruction counter is benign: the corrupted counter can force a transaction to prematurely commit or to unexpectedly abort. In either case, the counter will be reset as soon as a new transaction starts.

Using this loop transformation, several loop iterations can be executed at run-time before the threshold is reached and a new transaction begins. Thereby, this technique minimizes the number of required hardware transactions. Note that these transformations are applied recursively to nested loops.

Finally, Tx inserts transaction boundaries around function calls. In the general case, Tx does not know which function is called and for how long it executes, therefore it pessimistically ends the current transaction before the call and begins a new one after it.

5.3.3 Advanced Features and Optimizations

To reduce the performance overhead of HAFT and increase its reliability, we apply a number of optimizations on ILR and Tx.

Shared memory accesses. In basic ILR, each load and store requires expensive checks (Figure 5.3a). This can yield significant overheads since, in an average application, approximately 10% of instructions are stores and 30% are loads [32, 33]. In other words, around 40% of the original instructions need checks under the naïve ILR interpretation.

To reduce the number of checks, previous research has assumed a very relaxed memory model with two consecutive loads on the same address always returning the same value [191]. This assumption holds for sequential applications but is violated in multithreaded environments. In contrast, our refined ILR distinguishes between different types of memory accesses and applies optimizations only when they are safe.

The key enabler for our optimizations is the RC memory model (see §5.3.1). Our design assumes data-race free programs, where all accesses to shared memory are protected via locks or done explicitly using atomics. As such, we can separate memory accesses into atomic and regular ones. Atomic operations are not replicated and require (expensive) checks, while regular memory accesses optimize away most checks by relying on (cheaper) memory loads.

(a) Naïve

```
1 d = cmp neq cnd, cnd2
2 br d, xabort
3 br cnd, trueblk, falseblk
4
5
```

(b) Safe

```
br cnd, strueblk, sfalseblk
strueblk:                    ;; Shadow blocks
br cnd2, trueblk, xabort
sfalseblk:
br !cnd2, falseblk, xabort
```

Figure 5.4 – Control flow protection in ILR. The naïve approach (a) does not protect the condition while the safe one (b) does.

This optimization is illustrated in Figure 5.3b. By replicating regular loads, we can eliminate the checks of load addresses. Indeed, the data-race freedom assumption guarantees that both master and shadow loads read the same value in the error-free case. A fault happening during one of the loads will result in a wrong value being read and will propagate further until it is detected at some later point. Since almost all loads are considered regular, this optimization alone leads to up to 40% reduction in overhead (see §5.5.3). On the contrary, for the rare cases of *atomic loads*, we cannot perform any optimizations and fall back to an expensive address check and a shadow move for each load (Figure 5.3a, top).

The case of stores is more sophisticated. As atomic stores are considered irreversible externalization events, all checks must be performed before the store (Figure 5.3a, bottom). The effects of regular stores are, however, thread-local or protected by locks, which enables us to place the check after the store and simplify it with the help of an extra load (Figure 5.3b, bottom). Performance-wise, the load and check operations are coalesced in an effective `cmp` x86-instruction, and the additional load does not introduce any latency since it utilizes the store-buffer forwarding feature available on modern CPUs.

Control flow protection. ILR protects against the important class of transient faults that affect the status register (EFLAGS in x86) and result in taking incorrect branches. These faults are especially threatening in control flow intensive applications. For example, 20% of data corruptions in one of the benchmarks (*linearreg* in Figure 5.9, right) are due to such faults.

Since there is no way to replicate the status register, the basic version of ILR checks branch conditions before a branch instruction (Figure 5.4a). However, if the condition variable `cnd` becomes faulty in-between the check and the actual branch, the program flow can diverge undetectably and lead to further data corruptions.

Our refined ILR removes an explicit check on the condition and substitutes it with shadow basic blocks that evaluate the shadow condition and signal an error if a mismatch is detected (Figure 5.4b). The `strueblk` shadow basic block is taken if the master condition `cnd` evaluates to true, and therefore the shadow condition `cnd2` must also evaluate to true; otherwise an error is signaled. The same reasoning applies to the `sfalseblk` block, which operates on an inverse shadow condition. The destinations of the original branch are rewired to the shadow blocks and a transient fault in the status register cannot remain undetected.

Note that ILR does not protect against arbitrary control-flow errors, in particular transient faults that set the program counter (PC) register to some invalid value. Saggese et al. [197] show that a random value in the PC virtually never leads to data corruptions, i.e., there is no need to protect the PC.

Fault propagation check. The design of HAFT assumes that a fault happening in a transaction is quickly detected and handled. There is, however, one corner case when the fault can propagate to a subsequent transaction: the compiler can move stores as part of the loop hoisting optimization, as the example in Figure 5.2 shows. Here the global variable `c` is incremented in a loop. For

performance reasons, the compiler has moved the load of the initial value before the loop (line 3) and the store of the final value after the loop (line 12).

In this scenario, a fault corrupting `c` in one transaction may propagate to the next transaction if the fault happens during loop execution. This problematic case arises from the fact that ILR inserts a check on `c` only at the final store (lines 12–15).

To limit the propagation of faults inside such loops, we developed the following optimization, called a *fault propagation check*. ILR analyzes each loop induction variable and, if it is not covered by in-loop checks, adds an explicit check at the loop entry. Tx recognizes these additional checks and moves them inside the conditional transaction split, such that the checks are performed directly before committing the previous transaction. In this case, if a fault corrupts a variable, it will be detected by the newly added checks and the transaction will abort without the fault propagating further.

Local function calls. As described in §5.3.2, Tx inserts unconditional transaction begins and ends at each function entry, function call, and function return. This is a very conservative stance which does not rely on any knowledge of the relationship between different program functions. We notice, however, that most program functions are local, i.e., they are always called from other HAFTed program functions. At the same time, there are some functions that are called from third-party libraries, e.g., `main`.

Tx exploits this distinction between local and externally called functions by performing the following optimization. If a function is marked as local, calls to this function are surrounded merely by a counter increment and a follow-up conditional transaction split. Similarly, a local function uses a conditional transaction split at its entry and a counter increment upon return. With this caller-callee interaction in place, Tx eliminates two unnecessary transactions at each function call. In our current implementation, the developer is required to provide a black-list of externally called functions for this optimization.

Lock elision. HAFT also supports an original approach for lock elision, which consists in substituting (*eliding*) locks with hardware transactions to gain better performance [186]. The key observation is that at run-time locks are often unnecessary because many critical sections do not overlap in time and could execute safely without locks. In this case, speculative execution of a critical section in a transaction is faster than lock-based execution.

The lock elision optimization in HAFT relies on the fact that hardware transactions can be used for fault recovery *and* lock elision at the same time. We implement this optimization in the following way. Whenever HAFT detects a call to a lock function (acquire or release), it does not surround it with a transaction end and begin, but instead it calls a corresponding wrapper. The wrapper checks if a thread already executes in a transaction. If so, the critical section is executed under the protection of the active transaction without acquiring the lock. Otherwise, HAFT falls back to the original conservative locking scheme. We found this optimization to be particularly helpful in case of Memcached, and we investigate its gains in §5.6.1.

5.4 Implementation

We implemented HAFT as a LLVM-based compiler framework [134] that takes unmodified source code of an application and produces a HAFTed executable (§5.4.1). Additionally, we implemented a software-based fault injection framework compatible with Intel TSX (§5.4.2).

5.4.1 HAFT Compiler Framework

Tool chain. We developed HAFT based on LLVM 3.7.0. In particular, we implemented HAFT as two independent LLVM passes: ILR to add fault detection capabilities (∼830 LOC) and Tx to add fault recovery (∼540 LOC). Both passes abstract away the underlying details of the architecture; the architecture-specific functionality is extracted in separate LLVM IR files that are queried during compilation.

Overall, the build process proceeds as follows. First, all source files are compiled separately and linked to produce a single LLVM bitcode file [134]. Thereafter, all regular LLVM compiler optimizations are performed on the bitcode representation. We then take the optimized bitcode and pass it through the two implemented compiler passes, namely, ILR followed by Tx. Finally, the target machine code is generated. Note that we neither impose restrictions on the traditional compiler optimizations, nor do we require changes to the build parameters.

ILR pass. For the implementation of the ILR compiler pass, we had to modify the LLVM CodeGen module. In particular, since ILR introduces redundant shadow registers and shadow instructions, the LLVM compiler is free to optimize away these shadow copies. To prevent LLVM from doing it, we decouple the master and shadow data flows by introducing CodeGen-level `move` pseudo-instructions and corresponding LLVM intrinsics. These instructions and intrinsics are opaque to all LLVM optimization passes and are replaced by real x86 register moves only at the very last stage of code generation.

Furthermore, the LLVM optimizer can also remove shadow loop induction variables in cases when the initial (constant) value for the variable is known. We prevent this optimization by moving the initial value to a global volatile variable and reading it before the loop body. This trick has negligible performance impact since the initial value is loaded only once before the loop.

For the shared memory access optimization of ILR described in §5.3.3, we insert a volatile shadow load to prevent the compiler from optimizing it away or moving it around other memory-related operations.

Tx pass. The Tx pass follows closely the description in §5.3.2. We introduce thread-local instruction counters and four helper functions, as well as wrappers for the acquire and release functions from the lock elision optimization in §5.3.3, in a separate LLVM IR file. The Tx pass queries this file during compilation. This way, we can abstract the Tx pass from the underlying hardware and pthreads implementation.

The threshold for transaction sizes (§5.3.2) and a black-list of non-local functions (§5.3.3) are specified using additional LLVM compiler flags.

Collaboration of ILR and Tx. The fault propagation check described in §5.3.3 requires a tight collaboration between otherwise independent ILR and Tx. To achieve this, ILR adds checks with associated LLVM metadata in the loop. Tx recognizes these checks and moves them in a conditional transaction split right before the previous transaction's commit. The fault propagation check currently works only on innermost loops. Only induction variables from the loop header that are not checked in the loop body are covered by this check.

Both ILR and Tx introduce some basic peephole optimizations, e.g., ILR removes checks that immediately follow a creation of a shadow copy and Tx removes pairs of transaction starts followed immediately by transaction ends.

Libraries support. HAFT can transform only the source code available during compilation. This becomes a problem for applications that rely heavily on external libraries such as libc or libstd++. In such case, these unprotected libraries constitute a significant part of runtime execution and faults happening in their code go undetected. To increase fault coverage for C/C++

FI result	Description	System
Hang	Program became unresponsive	Crashed
OS-detected	OS terminated program	
ILR-detected	ILR detected, Tx did not recover	
HAFT-corrected	ILR detected, Tx recovered	Correct
Masked	Fault did not affect output	
SDC	Silent data corruption in output	Corrupted

Table 5.1 – Classification of fault injection results.

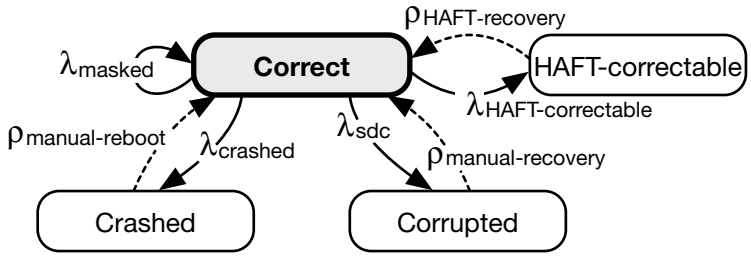

Figure 5.5 – HAFT probabilistic model. System transits from correct state to other states at predefined fault rates λ and returns back to correct state at predefined recovery rates ρ.

applications, we applied HAFT to a part of the libc library and link it to the final executable. We use the musl library [154] with assembly support disabled as reference implementation. We opted not to include dynamic memory allocation, I/O, OS, and pthreads-related functions for our prototype. Firstly, they account for a small fraction of runtime (less than 5%) for most programs, and secondly, they use system calls and unfriendly instructions prohibited in hardware transactions. Notice that most previous systems [75, 190, 191] did not apply their hardening techniques to external libraries, which impedes a direct comparison.

Limitations. Our HAFT prototype does not transform inline assembly code nor assembly functions: LLVM treats assembly as black-box function calls with no additional knowledge of their behavior. Furthermore, our prototype does not protect the C++ exception handling mechanism which requires a tight collaboration of LLVM IR and libstd++.

5.4.2 HAFT Fault Injection Framework

Fault injection tool. For conducting fault injection experiments of HAFT, we need a software-based fault injection tool that works with Intel TSX. As other tools [26, 200, 241] do not have such support, we developed our own binary-level fault injector (∼320 LOC).

Our fault injector is based on the Intel SDE emulator [111], which allows us to attach the GDB debugger to an emulated program. We leveraged this feature to design a simple GDB script-based fault injection tool. Intel SDE *emulates* all TSX instructions and thus enables us to perform fault injections on machines that do not have hardware support for TSX. It has an additional benefit that attaching GDB during a hardware transaction does not lead to a transaction abort.

The fault injection experiments proceed in two steps. In the first preparatory step, a reference execution trace of a tested program is generated using Intel SDE's debugtrace tool. This trace contains all the instructions executed by the program and all the registers updated by these instructions. Additionally, the program is run without any fault injections to produce a reference output.

From the obtained execution trace, at each fault injection, we choose a random occurrence of a random instruction that updates at least one register. We use weighted random numbers to inject faults uniformly across the whole execution of a program. After the specific occurrence of an instruction is chosen, one of its output registers is randomly selected to inject a fault into. The injection of a fault is simulated by XORing the value of this register with a random integer. Such faults imitate both sporadic corruptions of CPU registers and miscomputations in CPU execution units. The fault occurs right after the selected instruction. Faults are injected in general-purpose registers, as well as in the status and x86-64-specific registers.

In the second step, we start the program under Intel SDE with GDB attached and inject a single fault. To inject a fault, we construct a GDB script to set a conditional breakpoint in the program based on the specified instruction address and its occurrence number. Whenever the breakpoint is triggered by any thread, the script injects a fault and resumes execution. After the program terminates, the output is examined to study the effect of the fault injection (see Table 5.1). The second step is repeated until a sufficient number of runs (fault injections) is reached.

Fault injection probabilistic model. Our fault injection tool injects only one fault per run and requires smallest inputs to finish one experiment in a reasonable amount of time. Hence, we also built a probabilistic fault injection framework to investigate reliability of HAFTed programs working for a longer time and under different fault rates. We use a probabilistic model checker tool called PRISM [130] to construct a continuous-time Markov chain model of HAFT (∼130 LOC) and verify its properties probabilistically. Figure 5.5 represents the model for the native, ILR, and HAFT architectures. The architectures differ in the transition rates, which are selected from our fault injection experiments (see §5.5.5).

The system starts with a correct state. A transient fault can transfer the system to a correct, corrupted, crashed, or HAFT-correctable state. If a system is not in a correct state, then it is unavailable and needs recovery. A crashed system can be recovered by rebooting, and a corrupted system by manual recovery. The system in a HAFT-correctable state is recovered by restarting a transaction; this state exists only in the HAFT architecture.

5.5 Evaluation

Our evaluation answers the following questions:

- What are the performance overheads of HAFT? (§5.5.2)
- How effective are the optimizations in improving the performance and reliability of HAFT? (§5.5.3)
- What is the effect of hyper-threading on HAFT? (§5.5.4)
- What is the level of fault tolerance achieved by HAFT, and how efficient is it under different fault rates on long-running programs? (§5.5.5)
- What is the code coverage provided by HAFT, i.e., what fraction of the run-time execution is protected? (§5.5.6)

5.5.1 Experimental Setup

Applications. We evaluated HAFT with applications from two multithreaded benchmark suites: Phoenix 2.0 [187] and PARSEC 3.0 [32]. We report results for all 7 applications in the Phoenix benchmark and 8 out of 13 applications in the PARSEC benchmarks. The remaining five applications are not supported for the following reasons: *bodytrack* and *raytrace* make use

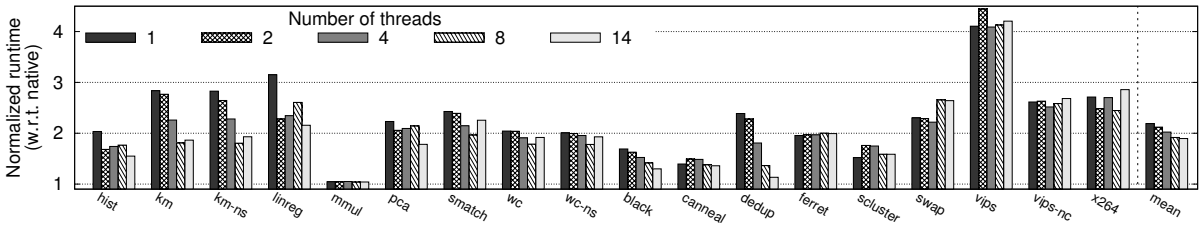

Figure 5.6 – Performance overhead over native execution with the increasing number of threads (on a machine with 14 cores).

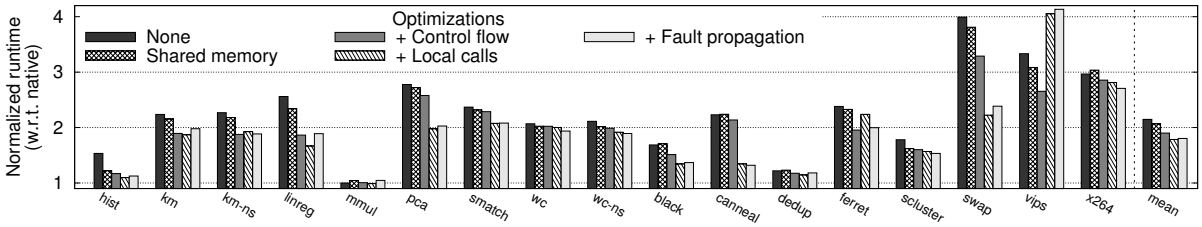

Figure 5.7 – Performance overhead over native execution with different optimizations (with 14 threads).

of C++ exceptions, which are currently not supported by our implementation; *freqmine* is an application based on OpenMP, which did not compile under our version of LLVM; *fluidanimate* produces nondeterministic output and thus makes it impossible to check the correctness of the results; and finally, the native version of *facesim* crashes with a runtime error when compiled with LLVM.

All applications were compiled with the HAFT compiler based on LLVM 3.7.0 with `-O3`, `-mrtm` (to support Intel TSX), and `-fno-builtin` (to transparently link against our own version of libc) flags and linked using the LLVM gold plugin.

Modified applications. Two applications from the Phoenix benchmark, *wordcount* and *kmeans*, have a high level of cache conflicts, which results in frequent transaction aborts. Therefore, we modified 47 LOCs in the former and 5 LOCs in the latter to mitigate this problem. We report results for both modified and unmodified versions. We refer to the modified ("no sharing") versions as *wordcount-ns* and *kmeans-ns*.

Datasets. For the performance evaluation, we used the largest available datasets provided by Phoenix and PARSEC benchmark suites. However, fault injection experiments were carried out using the smallest available input because they are extremely time consuming.

Testbed. We carried out the performance evaluation experiments on a machine with two 14-cores Intel Xeon processors operating at 2.0 GHz with hyper-threading enabled (Intel Haswell microarchitecture) with 128 GB of RAM, a 3.5 TB SATA-based SDD, and running Linux kernel 3.16.0. Each core has private 32 KB L1 and 256 KB L2 caches, and 14 cores share a 35 MB L3 cache. Due to hyper-threading, two logical threads sharing the same core also share the L1 and L2 caches. For fault injections, we used a cluster of 25 machines to parallelize the experiments.

Methodology. For all measurements, we confined our experiments to one processor, thus the maximum number of threads is restricted to 14 for all benchmarks. Note that we pinned application threads to separate physical cores in all experiments to avoid the effects of hyper-threading. In addition, we conducted an experiment to estimate how hyper-threading affects abort rates of HAFT (see §5.5.4).

For performance experiments, we ran programs with 1–14 threads. For fault injections, we

	Overheads				
Benchmark	**ILR**	**Tx**	**HAFT**	**Hyperthreading**	**Coverage (%)**
histogram	1.46	1.02	1.55	1.0	95.7
kmeans	1.60	1.28	1.86	2.6	95.8
kmeans-ns	1.63	1.28	1.93	5.4	—
linearreg	2.03	1.12	2.16	1.2	97.2
matrixmul	1.04	1.01	1.04	377	88.9
pca	1.35	1.14	1.78	2.4	95.1
stringmatch	1.50	1.46	2.26	1.8	98.7
wordcount	1.35	1.39	1.92	1.5	95.1
wordcount-ns	1.45	1.31	1.93	8.9	—
blackscholes	1.17	1.06	1.30	2.9	93.9
canneal	1.16	1.13	1.36	1.3	67.6
dedup	0.99	1.02	1.13	1.1	75.1
ferret	1.32	1.25	1.99	12.6	96.9
streamcluster	1.46	1.18	1.59	1.9	92.7
swaptions	1.98	1.57	2.64	11.4	89.6
vips	2.16	2.29	4.21	1.5	85.1
vips-nc	2.19	1.46	2.68	1.3	—
x264	2.32	1.33	2.86	4.9	85.5
mean	**1.52**	**1.27**	**1.89**	**24.5**	**90.2**

Table 5.2 – *First three columns*: Normalized runtime w.r.t. native of HAFT and its components (§5.5.2). *Fourth column*: Increase in abort rate when moving from the non-hyper-threaded to the hyper-threaded configuration (§5.5.4). *Fifth column*: Code coverage of HAFT in % (§5.5.6). All experiments with 14 threads.

fixed the number of threads to two. For each Phoenix benchmark, we performed a warm-up run to load input files into the main memory to stress-test the CPU overheads of HAFT (otherwise Phoenix benchmarks would be dominated by I/O). For PARSEC benchmarks, we reused the provided framework.

Measurements. For all performance measurements, we report the average over 10 runs. Fault injection experiments were conducted by injecting 2,500 faults for each program.

5.5.2 Performance Overheads

We first present the performance overheads of HAFT over the native execution. Figure 5.6 shows the overheads for a varying number of threads ranging from 1 to 14 threads.

The average overhead across all applications is 2× (see bar *mean*). The best case for HAFT is *matrixmul* due to the very low instruction-level parallelism (ILP) of 0.2 instructions/cycle for the native execution; thereby, HAFT effectively utilizes these spare ILP resources, with a runtime overhead of just 5%. The worst case for HAFT is *vips*, which incurs a slowdown of 4×, where two factors negatively affect HAFT's performance. First, the native version already has high ILP of 2.6 instructions/cycle such that there are no spare cycles left for HAFT. Second, *vips* has many calls to tiny functions such that the Tx local function calls optimization leads to a high performance penalty. If we disable this optimization, the performance overhead drops to 2.5× (*vips-nc* in Figure 5.6; see also next section).

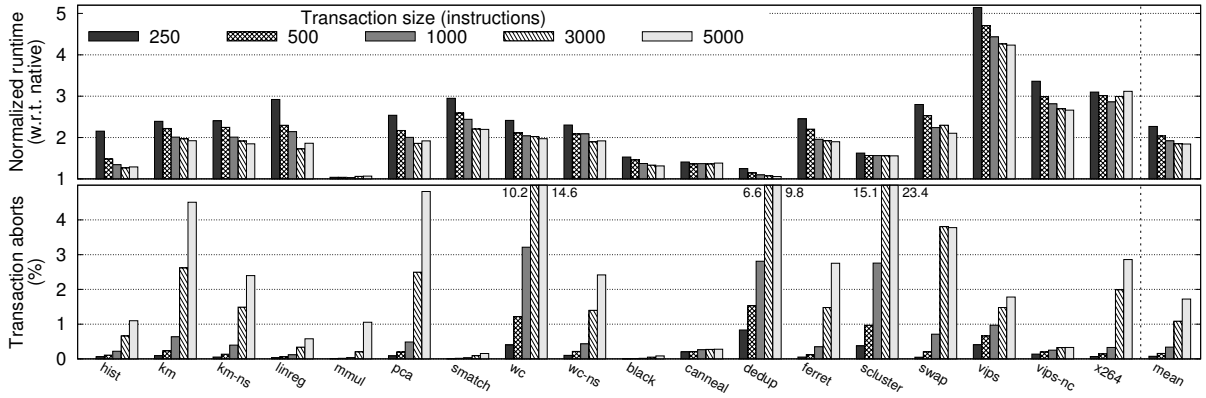

Figure 5.8 – Performance overhead over native execution (top) and percentage of aborts (bottom) vs. transaction size (with 14 threads).

HAFT benefits from the suboptimal scalability of native versions of programs. For example, the native version of *ferret* scales linearly, so the overhead of HAFT stays at the same level with the increasing number of threads. In contrast, the native version of *dedup* scales poorly with more than 2 threads and the overhead of HAFT is amortized in this case.

Table 5.2 (first three columns) highlights the contribution of HAFT components: ILR and Tx. ILR alone incurs performance overhead of 52% on average; this low overhead indicates that ILR efficiently uses spare ILP to hide additional instructions and checks inserted at compile-time. Tx incurs 27% overhead on average. Interestingly, the overhead of Tx is higher than that of ILR in the case of *vips*; as explained in the previous paragraph, this is due to the high number of calls to tiny functions. As soon as we remove this bottleneck, the overhead of Tx decreases by 60% (*vips-nc*).

5.5.3 Effectiveness of Optimizations

Impact of optimizations. The impact of different optimizations (§5.3.3) on performance is shown in Figure 5.7. We compare HAFTed benchmarks without any optimizations and then apply the following optimizations successively: ILR shared memory accesses, ILR control flow protection, Tx local function calls, and fault propagation check. Note that the fault propagation check is targeted to increase reliability at the price of some performance degradation.

This set of optimizations leads to an average performance improvement of 20% and in some cases achieves 70%. Interestingly, the addition of control flow checks, which are introduced to increase reliability, has a positive impact on performance: this happens because the check of a condition is substituted by a sequence of jumps, thus decreasing the number of executed instructions and benefiting from branch prediction.

Another somewhat surprising result is the Tx local function calls optimization: performance of most benchmarks improves significantly, whereas it degrades for *vips*. In the case of *vips*, the overhead of updating and checking the dynamic counter turns out to be higher than simply starting a new transaction on each function call. We decided to also show the results of *vips* with this optimization disabled (*vips-nc*) in other experiments.

Impact of transaction size.

We show the impact of different transaction sizes (maximum number of instructions in one transaction) on the performance overhead and the number of aborts in Figure 5.8 respectively. Note that the number of threads is fixed to 14 in these experiments. Performance overhead

Benchmark	Abort rate (%)	Abort causes (%)		
		Capacity	Conflict	Other
histogram	1.10	0.48	30.16	69.36
kmeans	4.51	0.01	99.90	0.09
kmeans-ns	2.40	0.03	95.68	4.29
linearreg	0.58	0.00	0.13	99.87
matrixmul	1.05	66.21	0.06	33.73
pca	4.82	0.72	82.97	16.31
stringmatch	0.15	2.53	0.32	97.15
wordcount	14.60	1.27	94.90	3.83
wordcount-ns	2.42	16.24	20.80	62.96
blackscholes	0.08	2.20	0.50	97.30
canneal	0.28	1.34	2.70	95.96
dedup	9.84	16.29	1.50	82.21
ferret	2.75	80.40	0.62	18.98
streamcluster	23.40	0.11	99.89	0.00
swaptions	3.78	90.87	0.01	9.12
vips	1.78	40.40	41.75	17.85
vips-nc	0.33	2.36	97.64	0.00
x264	2.86	64.22	6.72	29.06

Table 5.3 – Transaction abort rate and causes (with 14 threads). The worst-case transaction size of 5,000 is fixed for each benchmark.

decreases with greater transaction sizes, from 2.2× to 1.8× on average, due to the lower number of transactions. At the same time, the number of aborts grows with increasing transaction sizes. Aborts happen due to the following two reasons: first, longer transactions overflow the L1 cache more often, and second, longer transactions lead to higher probability of conflicts between threads.

Peculiarly, increasing transaction sizes (and thus higher abort rates) does not result in any clear pattern of performance overheads. Indeed, with increasing transaction sizes, two factors compete: (1) longer transactions amortize the cost of Tx instrumentation, and (2) the number of aborts increases because transactions start to overflow or conflict. The first factor decreases performance overhead while the second factor increases it.

This is evident from Figure 5.8. In the case of *streamcluster*, the number of aborts goes up to 23.4%, but longer (and fewer) transactions counterbalance this factor, and thus the performance overhead stays roughly the same. Compare it with *histogram*, where the number of aborts is low and the amortization factor dominates, thus decreasing the overhead. Finally, in the case of *x264*, the number of aborts drastically increases with transaction sizes greater than 1000, resulting in a change of the performance pattern.

The huge negative impact of cache sharing is clearly seen when comparing *kmeans* and *kmeans-ns* (removed true sharing), as well as *wordcount* and *wordcount-ns* (removed false sharing). In a demonstrative case of *wordcount*, rewriting the application with no cache sharing results in a 7× decrease of transaction aborts.

Table 5.3 shows the breakdowns of abort rates and their causes for each benchmark, measured with the worst-case transaction size of 5,000. The low abort rates (less than 1%) are largely dominated by the residual spontaneous ("other") aborts. Higher abort rates are caused either

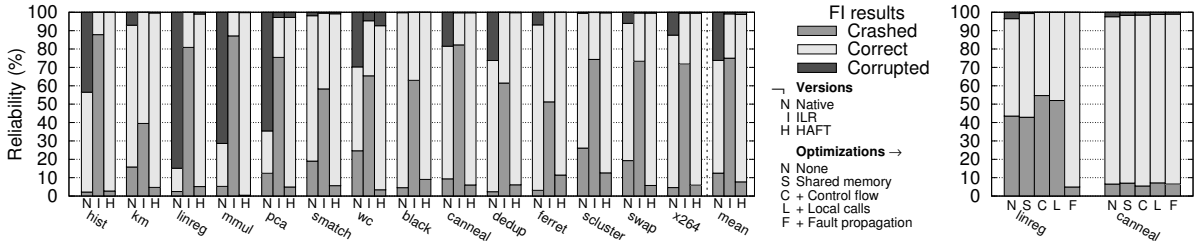

Figure 5.9 – Reliability of HAFT (left) and impact of different optimizations on two benchmarks (right) with 2 threads.

Fault probabilities	Native	ILR	HAFT
Masked (%)	61.3	24.2	24.2
SDC (%)	26.2	0.8	1.1
Crashed (%)	12.5	75.0	7.7
HAFT-correctable (%)	—	—	67.0

Table 5.4 – Parameters for the HAFT model.

mostly by capacity overflows or by conflicts among simultaneous transactions. For example, all aborts in *kmeans* are due to high conflict rates, whereas *matrixmul* experiences many capacity overflows due to its cache-unfriendly behavior.

For all other plots, we set for each benchmark the transaction size to the greatest value such that the percentage of aborts is sufficiently low, in order to achieve the best trade-off between performance and reliability. For example, we set transaction size to 1000 for *kmeans* and *pca*, and to 5000 for *stringmatch* and *blackscholes*.

5.5.4 Effect of Hyper-threading

To estimate the effect of hyper-threading on HAFT, we conduct the experiment with 14 threads (similar to Figure 5.6, last bar). However, in this experiment we pin 14 logical threads to 7 physical cores. Thus, each pair of threads shares CPU execution units and L1 and L2 caches.

Table 5.2 (fourth column) highlights the increase in abort rates compared to the baseline configuration of 14 logical threads on 14 physical cores. Many benchmarks still have low abort rates (*histogram*, *linearreg*, *canneal*, etc.), but some exhibit dramatic increase in transaction aborts (*matrixmul*, *ferret*, *swaptions*, etc.). In the former case, transactions are sufficiently small to peacefully co-exist in the shared L1 cache. In the latter case, transactions compete for the limited capacity of the cache and abort each other.

The case of *matrixmul* is peculiar, with an abort rate increasing by 377× from negligible 0.07% aborts in non-hyper-threaded scenario to 24% with hyper-threading. Our analysis indicates that aborts happen due to frequent overflows of a cache on read accesses – *matrixmul* is cache-unfriendly, and the sharing of L1- and L2-caches by two threads only exacerbates this problem.

5.5.5 Fault Injections

Fault injection experiments. The results of our fault injection experiments are shown in Figure 5.9. The faults were injected uniformly at random across the whole execution trace of each benchmark, including the parts not protected by HAFT (§5.4.1). Note that we were not

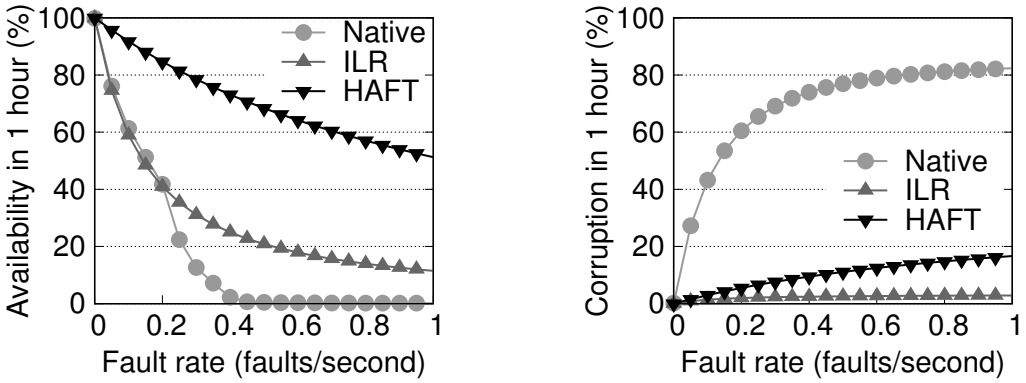

Figure 5.10 – HAFT fault injection modeling. Plots show fractions of time when system is available (left) or corrupted (right) in a time span of one hour w.r.t. the fault rate.

able to perform fault injections into *vips* as injecting one fault under Intel SDE took more than an hour even under the smallest inputs.

We also performed the experiment where the faults were injected only in the protected parts of the benchmarks, with very similar outcomes. This is expected: Our statistics indicates that the faults landing in unprotected parts constitute less than 1% of all injected faults in almost all cases except for *wordcount* and *x264*. Therefore, we do not show the results of this experiment.

Even in native execution, most of the faults (61.3%) are masked and programs remain correct after a fault is injected. However, around 26% of faults lead to data corruptions (see bar *mean*). When applying ILR, almost all faults (99.2%) are detected, but programs exit prematurely 75% of the time, leading to low availability (this can be explained by the fact that ILR sometimes detects also those faults that would be masked in native execution). Finally with HAFT, program reliability increases to approximately 91.2%. (Program reliability with HAFT reaches 92% on average if the faults are injected only in the protected parts of benchmarks.)

Figure 5.9 (right) shows the impact of different optimizations on the reliability of HAFT. As conducting these experiments is highly time-consuming, we chose only one benchmark from Phoenix (*linearreg*) and one from PARSEC (*canneal*). Note that the non-optimized versions (N) have a non-negligible number of data corruptions. In the case of *canneal* optimizations only slightly decrease the number of data corruptions, while for *linearreg* the shared memory optimization (S) and the addition of control flow protection (C) lead to SDC-free executions, but also slightly increase the proportion of crashes. The local calls optimization (L), which is only intended for performance improvement, has no effect on reliability. Finally, the fault propagation check (F) improves the availability of *linearreg* dramatically, reducing the number of crashes from 50% to less than 5%.

Fault injection modeling. To measure the reliability of HAFT, we use the model from §5.4.2 and parameters from Table 5.4. Fault probabilities are extracted from the fault injection experiments. We choose the following recovery rates: 6 hours for manual recovery, 10 seconds for machine reboot, and 2.5 μs for transactional recovery in HAFT. The rate of manual recovery is based on the Amazon report where it took 6 hours between the first noticed corruption and the renewal of processing of requests [8]. The rate of machine reboot is based on the time needed for a complete reboot of our server. The rate of HAFT recovery is based on the maximum transaction size of $5,000$ instructions, which corresponds to the maximum latency of recovery of 2.5 μs on a 2.0 GHz CPU.

Figure 5.10 (left) shows the fraction of time when the system is available in a time span of

one hour with regard to the fault rate. The fault rate varies from once every hour to once every second (0.00028 to 1 fault/second). HAFT significantly increases program availability compared to ILR and native. For example, under a fault rate of 1.0, HAFT's availability is around 50%, i.e., 30 minutes out of one hour. In contrast, availability of native and ILR versions is 0% and 10% (6 minutes) respectively. In addition, Figure 5.10 (right) indicates that ILR and HAFT drastically reduce the number of data corruptions. Native spends more than 80% of the time in a corrupted state, while both ILR and HAFT stay in this state for less than 20%.

5.5.6 Code Coverage

Lastly, we analyzed what fraction of the run-time execution is protected with HAFT. Remember that our prototype of HAFT does not protect external libraries except for partial support of libc (see §5.4.1). To this end, we measured the fraction of dynamic execution spent inside transactional execution (Table 5.2, fifth column). The fraction is calculated as the number of cycles executed in transactions to the number of all cycles executed, as reported by the perf tool. Each program was built with all HAFT optimizations enabled and with the number of threads fixed to 14; the number of retries was set to three. The mean code coverage across all benchmarks is 90.2% indicating a high level of protection for almost all applications. Two exceptions are *canneal* and *dedup*: the former extensively uses containers from libstd++ while the latter spends many cycles in unprotected parts of libc for thread management and dynamic allocation.

5.6 Case Studies

We successfully applied HAFT on five real-world applications without any source code modifications. We present detailed results for Memcached (§5.6.1) and present summarized results for the others (§5.6.2). All applications were run in a local deployment on a single Haswell machine: we deployed each server application on one 14-core processor and its client applications on the other processor.

5.6.1 Memcached Key-Value Store

We evaluated Memcached [78] v1.4.24 using workloads from the YCSB benchmark [56] with 1 million key-value queries, each key being 16 B and each value 32 B. Figure 5.11 (left two graphs) shows the throughput of Memcached increasing with the number of threads, with two extreme YCSB workloads corresponding to the best and worst case for HAFT: A (50% reads, 50% writes, Zipf distribution) and D (95% reads, 5% writes, latest distribution). We evaluated Memcached with all available variants for synchronization using pthreads locks and atomic operations. For both native and HAFT, we tested two versions, one with locks only (native-lock and HAFT-lock) and one with atomics enabled (native-atomics and HAFT-atomics). Note that HAFT-lock has the optimization of lock elision (see §5.3.3). We also show the version with this optimization disabled (HAFT-lock-noelision).

The lock elision optimization allows HAFT-lock to perform 30% better than HAFT-lock-noelision and on par with native-lock, i.e., the overhead of HAFT is completely amortized by this optimization. Indeed, when configured to use locks, Memcached spends most of the time acquiring and releasing the locks. Since HAFT already uses transactions for recovery, removing the overhead of these locks comes for free. Moreover, HAFT-lock performs similar to HAFT-atomics, indicating that an application can achieve the same performance improvement with lock elision as when using atomics.

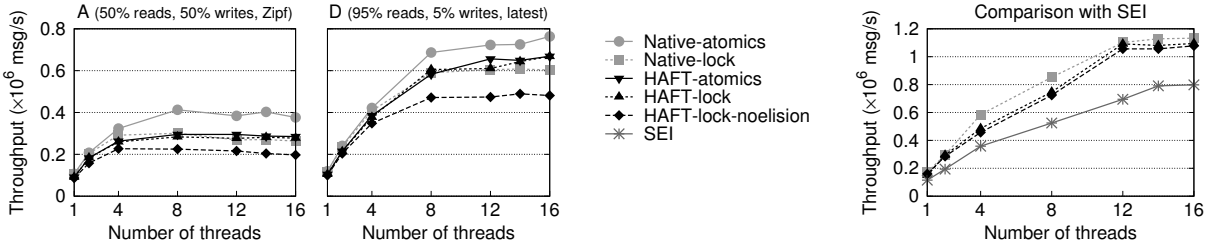

Figure 5.11 – Memcached throughput. Left two graphs: workloads A and D. Right graph: comparison of HAFT and SEI using a mcblaster client, a key range of 1,000, and values of size 128 B (same experimental setup as in [26]).

Our experiments also show that the latency of HAFT is 30% worse than in native on average and the percentage of committed transactions remains above 95% in all runs. Finally, the fault injection experiments indicate that HAFT decreases the percentage of data corruptions from 2% to 0.09% (two SDCs). Both lingering data corruptions happened in the very beginning of two functions responsible for shaping a reply message (namely, `add_bin_header` and `add_iov`). In both cases, the "length" function argument was corrupted exactly before its shadow copy was created; as a result, the reply string was incorrectly truncated.

Comparison with SEI. We also compared HAFT against SEI [26], another state-of-the-art approach, using Memcached.[4] We deployed SEI locally on our Haswell machine and reproduced the experiments from the SEI paper with the mcblaster client, a key range of 1,000, and values of size 128 B. Since SEI performs modifications to Memcached, we apply HAFT on the modified version.

Figure 5.11 (right graph) shows that HAFT performs on par with the native version (similar to graphs on the left) and outperforms SEI by 30–40%. The lower performance of SEI is explained by the local deployment; in the experiments with remote clients in the original paper [26], SEI's overhead was amortized by the network. Also note that the lock elision optimization of HAFT provides no benefit in this experiment. This is due to an older version of Memcached (namely, version 1.4.15) that supports only coarse-grained locks and thus is not amenable to our simple lock-elision heuristics.

We conclude with an indirect comparison of fault coverage, based on the numbers reported in [26].[5] As shown earlier, HAFT leaves 0.09% of data corruptions, whereas SEI with a similar configuration cannot detect 0.15% of corruptions.

5.6.2 Additional Case-Studies

LogCabin (RAFT). LogCabin [141] is an implementation of a consistent storage mechanism built on the RAFT [176] consensus protocol. For the evaluation, we used the benchmark shipped together with LogCabin that repeatedly writes 1,000 values into a memory-mapped file.

Apache web server. Apache is a popular web server [14]. For multithreading, we use a "worker multi-processing module" with a single running process and a varying number of worker threads. We used the Apache *ab* benchmark tool that queries a static 1 MB Web page for the evaluation.

LevelDB key-value store. LevelDB is a fast embedded key-value storage library developed by Google [136]. We evaluated LevelDB on an in-memory database using the same YCSB workloads

[4]Note that Memcached is the only multithreaded application evaluated in [26].

[5]These numbers should be treated with care because of the differences in Memcached versions, fault models and fault injection frameworks used.

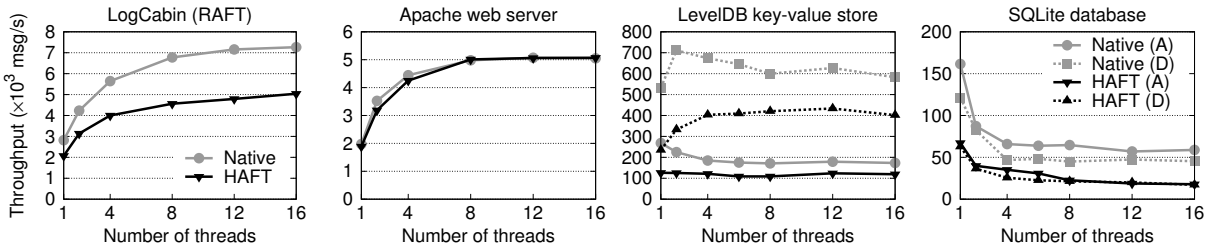

Figure 5.12 – Throughput of additional case-studies: LogCabin (RAFT), Apache web server, LevelDB key-value store, and SQLite database. Two extreme workloads are shown for LevelDB and SQLite: workload A (50% reads, 50% writes, Zipf distribution) and workload D (95% reads, 5% writes, latest distribution).

used for Memcached (workloads A & D).

SQLite database. SQLite is an SQL database engine implemented as an embeddable software library [179]. We evaluated SQLite on an in-memory database using again YCSB workloads A and D.

The scalability plots are shown in Figure 5.12. LogCabin and LevelDB are well-behaved applications, performing 25–35% worse than native versions. Apache exhibits an overhead of just 10%; this good result is due to Apache's extensive use of external libraries that are not transformed by HAFT. SQLite shows the poorest results, with HAFT performing 3–4× worse than the native version. We attribute this poor performance mainly to the extensive use of function pointers that are conservatively treated as external functions by HAFT.

We performed fault injection experiments on LevelDB and SQLite. Though their native versions are already tolerant to data corruptions, the faults lead to a high number of crashes, 42% and 28% respectively. HAFT decreases these numbers to only 10% and 3.7%, providing significantly higher availability.

5.7 Conclusion and Future Work

Many software systems require very high level of reliability. Alas, adding fault tolerance capabilities to existing applications inevitably degrades their performance. Fortunately, modern commodity hardware with its increased instruction level parallelism and new extensions such as hardware transactional memory enables cheap and efficient fault tolerance solutions. In this chapter, we presented HAFT, a novel approach to software hardening that provides low-cost fault detection via instruction-level redundancy and fast fault recovery via HTM. Our evaluation shows that HAFT significantly increases reliability and availability at the cost of 2× performance overhead.

In the current design of the transactification algorithm, a single threshold value is chosen for the entire execution of a program (§5.3.2). In reality, different code paths of the same program exhibit different behavior with respect to hardware transactions. In this case, some form of static/dynamic adjustment of the threshold could prove beneficial.

Our current implementation of HAFT does not protect all program code. While adding protection to the most of the functionality that standard libraries provide seems straightforward, supporting inline assembly and the C++ exception mechanism would require substantial engineering effort. Another problem is unfriendly instructions which inevitably lead to TSX transaction aborts. We believe this can be fixed in the future implementations of TSX. Fortunately, once

these issues are resolved, all programs written in LLVM-backed programming languages could be transparently hardened.

Hardware transactional memory can be found in architectures other than x86-64. For example, IBM POWER8 [43] provides not only regular TSX-like transactions, but also *rollback-only transactions* which buffer stores without detecting data conflicts. Moreover, transactions in POWER8 can be suspended and resumed to avoid aborting on interrupts.

6 SGXBounds: Leveraging Software Guard Extensions

The previous three chapters of this thesis presented three solutions to one specific class of faults – hardware faults occurring in CPU and RAM and manifesting in incorrect execution of the program and erroneous outputs. In particular, we introduced Δ-encoding to detect transient, intermittent, and even permanent faults in CPU and RAM (Chapter 3), Elzar to detect and mask only transient CPU faults (Chapter 4), and HAFT to detect and roll-back transient CPU faults (Chapter 5). These three techniques constitute the first part of this thesis: protecting against hardware faults. We can conclude that Δ-encoding is an efficient solution for safety-critical embedded applications, while HAFT is tailored for cloud environments with lower requirements on fault coverage. (Elzar can be considered an efficient replacement for HAFT if future versions of Intel AVX incorporate our modifications proposed in §4.7.)

Now we switch our attention to the second class of faults detailed in this thesis – software faults aka software bugs, manifesting in incorrect execution of the program, erroneous outputs, and potential leaks of confidential data. In other words, in this and the following chapters, we shift our focus from fault tolerance to systems security.

As mentioned in §1.3, we concentrate on a specific subclass of software bugs – *memory corruption* bugs. Recall that memory corruption bugs occur in unsafe languages like C/C++, when a developer writes erroneous code that incorrectly manipulates pointers in the program. Buffer overflows, out-of-bounds memory accesses, off-by-one errors, dangling pointers, use-after-free errors are all examples of memory corruptions. In many cases, these bugs trigger a segmentation fault and crash the program; this is annoying since the program needs to be restarted with all intermediate data lost, but not too harmful. In other cases, memory corruption bugs can be exploited by hackers to modify program behavior, gain root privileges to the underlying system, or steal confidential data. Such bugs are called software vulnerabilities. In this and the following chapters, we discuss solutions to *memory corruption vulnerabilities*; these solutions enforce *memory safety* by protecting each and every memory access by a bounds check to detect potential bugs.

The first approach we discuss is SGXBounds: a bounds checker to detect and tolerate security vulnerabilities in multithreaded legacy C/C++ programs inside Intel SGX enclaves. SGXBounds is specifically tailored to Intel SGX (described in §1.2) and provides a fast and simple means to protect against memory corruptions such as buffer overflows.

The content of this chapter is based on the paper "SGXBounds: Memory Safety for Shielded Execution" presented at EuroSys'2017 [128]. The paper was a joint collaboration with Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer.

6.1 Rationale

Software security is often cited as a key barrier to the adoption of cloud services [53, 54, 226]. In this context, trusted execution environments provide mechanisms to make cloud services more

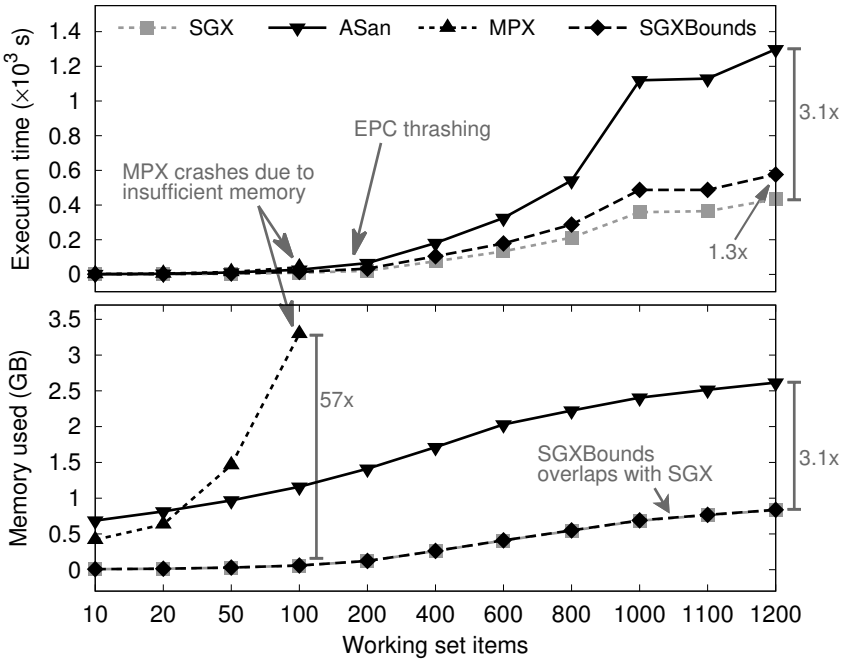

Figure 6.1 – Performance and memory overheads of SQLite.

resilient against security attacks [147, 204].

In this chapter, we focus on Intel Software Guard Extensions (SGX) [147], a recently proposed set of ISA extensions for trusted execution. Intel SGX provides an abstraction of *secure enclave*—a memory region opaque to other software including the hypervisor and the OS—that can be used to achieve *shielded execution* for unmodified legacy applications on untrusted infrastructure.

Shielded execution aims to protect confidentiality and integrity of applications when executed in an untrusted environment [15, 24]. The main idea is to isolate the application from the rest of the system (including privileged software), using only a narrow interface to communicate to the outside, potentially malicious world. Since this interface defines the security boundary, checks are performed to prevent the untrusted environment from attacks on the shielded application in an attempt to leak confidential data or subvert its execution.

Shielded execution, however, does not protect the program against *memory safety* attacks [225]. These attacks are wide-spread, especially on legacy applications written in unsafe languages such as C/C++. In particular, a remote attacker can violate memory safety by exploiting the existing program bugs to invoke out-of-bounds memory accesses (aka buffer overflows). Thereafter, the attacker can hijack program control flow or leak confidential data [87, 227].

To validate our claim, we reproduced many publicly available memory safety exploits inside the secure enclave (see §6.7 for details), including the infamous Heartbleed attack in Apache with OpenSSL [227] as well as vulnerabilities in Memcached [151], Nginx [164], and in 16 test cases from the RIPE security benchmark [244]. These examples highlight that a single exploit can completely compromise the integrity and confidentiality properties of shielded execution.

To prevent exploitation of these bugs, a number of memory safety approaches have been proposed to automatically retrofit bounds checking in legacy programs [6, 17, 39, 63, 155, 162]. Among these, we experimented with two prominent software- and hardware-based memory protection mechanisms in the context of shielded execution: AddressSanitizer [207] and Intel Memory Protection Extensions (MPX) [110], respectively.

Unfortunately, these approaches exhibit high performance and memory overheads, thus ren-

dering them impractical for shielded execution. For instance, consider the motivating example of SQLite evaluated against the `speedtest` benchmark (shipped with SQLite) with increasing working set items. Figure 6.1 compares the performance and memory overheads of SQLite hardened with AddressSanitizer and Intel MPX running inside an SGX enclave.

The experiment shows that Intel MPX performs so poorly that it crashes due to insufficient memory already after tiny working set of 100 (corresponding to memory consumption of 60MB for the native SGX execution). AddressSanitizer is more stable, but performs up to 3.1× slower than SGX on larger inputs (with virtual memory consumption of 700 − 800MB for the native SGX execution). Additionally, AddressSanitizer consumes 3.1× more virtual memory which can quickly exhaust available memory inside the enclave.

These overheads illustrate a drastic mismatch between memory needs of current memory-safety approaches and the architectural limitations of Intel SGX (high encryption overheads and limited enclave memory, as explained in §6.2.1). In particular, both AddressSanitizer and Intel MPX incur high memory overheads due to additional metadata used to track object bounds, which in turn leads to poor performance. (We detail the reasons behind the SQLite overheads in §6.2.3.)

In this chapter, we present SGXBounds—a memory-safety approach for shielded execution. Our design takes into account architectural features of SGX and reduces performance and memory overheads to the levels acceptable in production use. For instance, in the case of SQLite, SGXBounds outperforms both AddressSanitizer and Intel MPX, with performance overheads of no more than 35% and almost zero memory overheads with respect to the native SGX execution.

The SGXBounds approach is based on a simple combination of tagged pointers and efficient memory layout to reduce overheads inside SGX enclaves. In particular, we note that SGX enclaves routinely use only 32 lower bits to represent program address space and leave 32 higher bits of pointers unused.[1] We utilize these high bits to represent the upper bound on the referent object (or more broadly the beginning of the object's metadata area); the lower bound value is stored right after the object. Such metadata layout requires only 4 additional bytes per object and does not break cache locality—unlike Intel MPX and AddressSanitizer. Additionally, our tagged pointer approach requires no additional memory lookups for simple loop iterations over arrays—one of the most common cases for memory accesses [52].

Furthermore, we show that our design naturally extends for: (1) "synchronization-free" support for multithreaded applications, (2) increased availability instead of the usual fail-stop semantics by tolerating out-of-bounds accesses based on failure-oblivious computing [193, 194], and lastly, (3) generic APIs for objects' metadata management to support new use-cases.

SGXBounds is targeted but not inherently tied to SGX enclaves. Our approach is also applicable to programs that use 64-bit registers to hold pointers but can fit in 32-bit address space. However, as we show in our evaluation, SGXBounds provides no tangible benefits in traditional, unconstrained-memory environments in comparison to other techniques.

We implemented SGXBounds as an extension to the LLVM compiler with several optimizations for performance. Our compiler framework targets unmodified legacy multithreaded applications and thus requires no source code modifications. We evaluated SGXBounds using two multithreaded benchmark suites, Phoenix and PARSEC, and four real-world applications: SQLite, Memcached, Apache, and Nginx. On this set of benchmarks, AddressSanitizer and Intel MPX exhibit high performance overheads of 51% and 75% respectively; memory consumption is 8.1× and 1.95× higher than native SGX. In contrast, SGXBounds shows an average performance slowdown of 17% and an increase in memory consumption by just 0.1%. At the same time, it

[1] Current SGX implementations allow 36-bit address space. However, we believe that SGX enclaves spanning more than 4GB of memory are improbable.

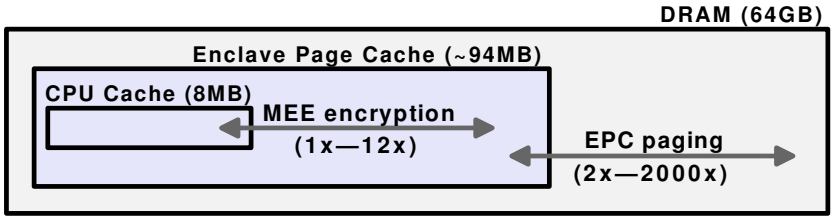

Figure 6.2 – Memory hierarchy and relative performance overheads of Intel SGX w.r.t. native execution [15].

provides similar security guarantees. Additionally, we evaluated SGXBounds on a CPU-intensive SPEC CPU2006 suite, both inside and outside SGX enclaves.

6.2 Background and Related Work

6.2.1 Shielded Execution

Our work builds on SCONE [15], a shielded execution framework to run unmodified applications. SCONE utilizes Intel SGX to provide confidentiality and integrity guarantees.

Intel SGX is a set of ISA extensions for trusted computing released with recent Intel processors [58, 147]. Intel SGX provides an abstraction of *enclave*—a memory region for which the CPU guarantees confidentiality and integrity.

A distinctive trait of Intel SGX is the use of a memory encryption engine (MEE). Enclave pages are located in the Enclave Page Cache (EPC)—a dedicated memory region protected by the MEE (Figure 6.2). While in main memory, EPC pages are encrypted. When such a page is accessed, the processor verifies that the access originates from the enclave code, fetches the requested data and copies it into the CPU cache. The MEE performs decryption and verifies the integrity of the data. This allows protecting enclaves from attacks launched by privileged software (e.g., by the OS or hypervisor) as well as from physical attacks (e.g., memory bus snooping), thus reducing the Trusted Computing Base (TCB) to the enclave code and the processor.

The EPC is a limited resource and is shared among all enclaves. Currently, the size of the EPC is 128 MB. Approximately 94 MB are available to the user while the rest is reserved for the metadata. To enable creation of enclaves with sizes beyond that of the EPC, SGX features a paging mechanism. The operating system can evict EPC pages to an unprotected memory using SGX instructions. During eviction, the page is re-encrypted. Similarly, when an evicted page is brought back, it is decrypted and its integrity is checked. Paging incurs high overhead, from 2× for sequential memory accesses and up to 2000× for random ones [15].

SCONE is a shielded execution framework that enables unmodified legacy applications to take advantage of the isolation offered by SGX [15]. With SCONE, the program is recompiled against a modified standard C library (SCONE libc), which facilitates the execution of system calls. The address space of an application is confined to only enclave memory, and the untrusted memory is accessed only via the system call interface. Special wrappers copy arguments of system calls inside and outside the enclave and provide functionality to transparently cryptographically protect any data that might otherwise leave the enclave perimeter in plaintext (so-called *shields*).

Clearly, the combination of SCONE and SGX is not a silver bullet. As we showcase in §6.7, bugs in the enclave code itself can render these mechanisms useless: we reproduced bugs in Memcached, Nginx, and the infamous Heartbleed attack, all *inside* the SGX enclave and running

	CF	DO	IL
Control Flow Integrity [40, 73, 146, 256]	✔	✘	✘
Code Pointer Integrity [129]	✔	✘	✘
Address Space Randomization [121, 133, 142, 206, 210]	✔*	✘	✘
Data Integrity [7]	✔	✔	✘
Data Flow Integrity [44]	✔	✔	✘
Software Fault Isolation [73, 237]	✔	✔	✔
Data Space Randomization [30, 42]	✔*	✔*	✔*
Memory safety [6, 17, 39, 63, 110, 155, 162, 207]	✔	✔	✔

*SGX enclaves do not provide sufficient bits of entropy in random offsets/masks

Table 6.1 – Current defenses against attacks [225]. **CF** – control flow hijack, **DO** – data-only attack, **IL** – information leak.

under SCONE. Thus, it is necessary to defend against data leaks such that the attacker cannot reveal confidential information even in the presence of exploitable vulnerabilities.

To choose the right defense against information leaks, we first discuss the applicability of state-of-the-art defenses for shielded execution and SGX (based on the classification by Szekeres et al. [225]). Table 6.1 highlights that most state-of-the-art defenses target control-flow hijack attacks only. Even if a proposed defense claims to protect against information leaks, it usually implies that the attacker *can* obtain confidential data in plaintext but *cannot* launch a hijacking attack based on these leaks [20, 60, 210, 219, 224]. Also note that Address Space Randomization (ASR) and its fine-grained variants [49, 60, 84, 142] do not have sufficient bit entropy in SGX enclaves (recall that SGX restricts enclave address space to only 36 bits) and thus can be easily broken [210, 219]. Concurrent and independent from our work, SGX-Shield investigated the use of fine-grained ASR in the context of small enclaves [206].

Most of the listed approaches do not prevent information leaks. The only exceptions are Software Fault Isolation (SFI) [73, 237], Data Space Randomization (DSR) [30, 42] and memory-safety techniques [6, 17, 63, 158, 162, 166, 196, 248]. Unfortunately, SFI requires manual separation of the enclave address space into fault domains and is too coarse-grained to guarantee high security (nevertheless, our preliminary evaluation using Intel MPX instructions indicates overheads of 3%, making it a viable low-cost alternative). DSR techniques rely on a simple XOR mask to obfuscate data, and a determined attacker can infer these masks by analyzing leaked data.

Therefore, we concentrate on memory-safety approaches proved to completely prevent data leaks and other attacks [225]. These approaches prevent the very first step in any attack—exploiting a vulnerability, such as overflowing a buffer or freeing an already freed object. We must note that even though we concentrate on memory safety, there are other, insider attack vectors (orthogonal to our work) where a malicious OS tries to deceive the shielded application [48, 171, 212, 249].

6.2.2 Memory Safety

The foundation of all memory attacks is getting access to a prohibited region of memory [150, 233]. Hence, memory safety can be achieved by enforcing a single invariant: memory accesses must always stay within the bounds of originally intended (referent) objects. For legacy applications written in C/C++, this invariant is enforced by changing (*hardening*) the application to perform additional bounds checks.

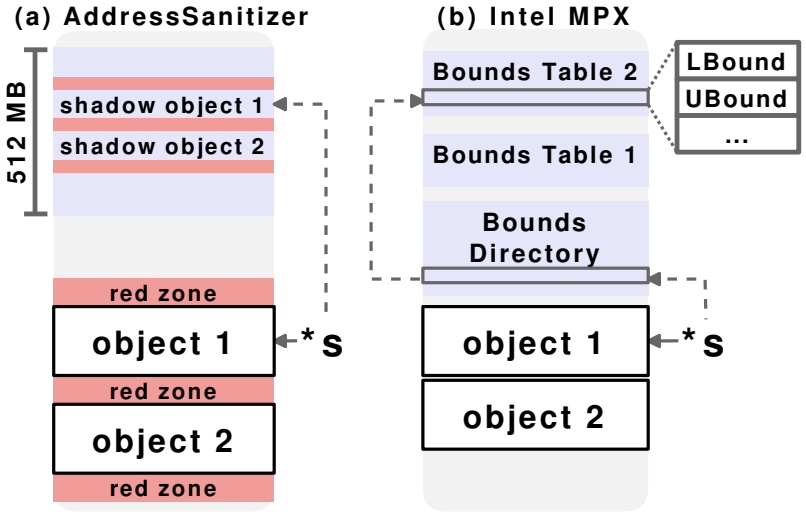

Figure 6.3 – Memory protection mechanisms.

A number of memory-safety approaches have been implemented either in software [6, 68, 162, 166, 207] or in hardware [110, 114, 131, 245]. We analyze two open-source and stable approaches in order to put our own results into perspective: software-based AddressSanitizer and hardware-based Intel MPX.

AddressSanitizer is an extension to GCC and Clang/LLVM that detects the majority of object bounds violations [207]. It keeps track of all objects, including globals, heap, and stack variables, and checks whether the address is within one of the used objects on each memory access. For that, it utilizes *shadow memory* – a separate memory region that stores metadata about main memory of an application (shown in pale-blue in Figure 6.3a). In particular, shadow memory shows which regions are allocated and used (i.e., safe to access) and which are not. AddressSanitizer does that by allocating *redzones* around all main memory objects and marking them inaccessible in the shadow memory. Hence, if an application tries to read or write out of object limits, this can be detected by checking the corresponding shadow address. On top of that, AddressSanitizer provides a quarantine zone for freed objects, thereby detecting temporal errors such as use-after-free and double free.

Execution of the hardened program is supported by a run-time library that initializes the shadow region and replaces memory management functions. It redefines memory-allocation functions (e.g., `malloc`) to allocate redzones and mark them unaddressable (poisoned) in shadow memory and memory-deallocation functions (e.g., `free`) to poison the whole object after it has been freed. The library also maps 1/8th of virtual address space for the shadow memory at startup.

The hardening is performed by a compile-time instrumentation pass. To understand how it works, consider an example in Figure 6.4a, which copies elements of one array ("s" for source) to another ("d" for destination). The first task of the pass is to set metadata for global, heap, and stack variables. In this example, it creates shadow objects for both arrays and sets the redzones by calling `init_shadow` (Figure 6.4b, lines 2–3). The pass also enforces the memory access correctness by computing the shadow addresses of all pointers (lines 7 and 11) and checking if they are within a redzone (lines 8 and 12). If a violation is detected, the application is crashed with a debugging message (lines 9 and 13).

Intel MPX is a recent set of ISA extensions of Intel x86-64 architecture for memory protection

(a) Original

```
1  int *s[N], *d[N]
2
3
4  for (i=0; i<M; i++):
5    si = s + i
6    di = d + i
7
8
9
10   val = load si
11
12
13
14   store val, di
15
```

(b) ASan

```
int *s[N], *d[N]
init_shadow(s, N)
init_shadow(d, N)
for (i=0; i<M; i++):
  si = s + i
  di = d + i
  ssi = get_shadow(si)
  if *ssi != 0:
    crash(si)
  val = load si
  sdi = get_shadow(di)
  if *sdi != 0:
    crash(di)
  store val, di
```

(c) Intel MPX

```
int *s[N], *d[N]
sbnd = bnd_create s
dbnd = bnd_create d
for (i=0; i<M; i++):
  si = s + i
  di = d + i

  if bnd_check si, sbnd:
    crash(si)
   val = load si
  val_bnd = bnd_load si
  if bnd_check di, dbnd:
    crash(di)
  store val, di
  bnd_store val_bnd, di
```

(d) SGXBounds

```
int *s[N], *d[N]
s = specify_bounds(s, s + N)
d = specify_bounds(d, d + N)
for (i=0; i<M; i++):
  si = s + i
  di = d + i
  sp, sLB, sUB = extract(si)
  if bounds_violated(sp, sLB, sUB):
    crash(si)
  val = load si
  dp, dLB, dUB = extract(di)
  if bounds_violated(dp, dLB, dUB):
    crash(di)
  store val, di
```

Figure 6.4 – Memory safety enforcement of original code in (a) via: (b) AddressSanitizer, (c) Intel MPX, and (d) SGXBounds.

[110]. By design, Intel MPX detects all possible spatial memory vulnerabilities including intra-object ones (when one member in a structure corrupts other members). The approach to achieving this goal is different from AddressSanitizer—instead of separating objects by unaddressable redzones, Intel MPX keeps *bounds metadata* of all pointers and checks against these bounds on each memory access. Since metadata bookkeeping and checking is implemented partly in hardware, such protection is supposed to be highly efficient.

From the developer perspective, Intel MPX adds new 128-bit registers for keeping upper and lower addresses (bounds) of a referent object. It also provides instructions to check if a pointer is within these bounds, along with instructions to manipulate them. To illustrate how Intel MPX works in practice, consider an example in Figure 6.4c. After the objects are created (line 1), their bounds have to be stored for future checks (lines 2–3). Then, on each memory access, we check if the accessed address is within the bounds of the referent object (lines 8 and 12) and crash if the check fails (lines 9 and 13). Unlike AddressSanitizer, we have to copy not only the arrays' elements but also their bounds (lines 11 and 15), which causes additional performance overhead. Note that this copying of bounds is required because the elements of arrays are pointers themselves.

One major limitation of the current Intel MPX implementation is a small number of bounds registers. If an application contains many distinct pointers, it will cause frequent loads and stores of bounds in memory. To make this interaction more efficient, bounds are stored in tables with an index derived from the pointer address, similar to a two-level page table structure in x86: a 2GB intermediate table (Bounds Directory) is used as a mediator to the actual 4MB-sized Bounds Tables, which are allocated on-demand by the OS when bounds are created (see Figure 6.3b). Thus, the constant memory overhead is minimal and the total overhead depends mainly on the number of pointers in the application.

Other memory-safety approaches. Apart from AddressSanitizer and Intel MPX, relevant memory-safety approaches include Baggy Bounds [6] and Low Fat Pointers [67, 68].

Baggy Bounds [6] solves the problem of high memory consumption and broken cache locality by enforcing allocation bounds via buddy allocator. Thus, all objects become power-of-two aligned, allowing simple and efficient checks against the base and bounds. The approach maintains minimal metadata for the bounds table, and the authors introduce tagged pointers with 5 bits holding the size. However, even with tagged pointers Baggy Bounds incurs perceivable overheads: 70% performance and 12% memory (on SPECINT 2000) [6].

Low Fat Pointers [67, 68] are conceptually similar to Baggy Bounds: they also introduce a special allocator that divides the virtual address space in regions of fixed sizes and derive base and bounds from the unmodified pointer. Overheads are also comparable to Baggy Bounds: 54% performance and 12% memory [68]. Yet, to support sparse memory regions, Low Fat Pointers assume a complete 64-bit address space, incompatible with the current version of SGX. Also, the prototype of Low Fat Pointers protects only stack and heap but not globals.

Given their tagged-based nature and low memory consumption, Baggy Bounds and Low Fat Pointers seem proper candidates for usage in SGX enclaves. Unfortunately, neither of them are publicly available.

6.2.3 Memory Safety for Shielded Execution

Now that we have covered the necessary background, we explain the overheads for the SQLite case study introduced in §6.1.

In the normal environment—outside of the SGX enclave—Intel MPX exhibits performance overheads of up to 2.5× and AddressSanitizer of up to 2.1× (not shown in Figure 6.1). These are reasonable overheads expected from these approaches.

Inside the enclave the picture changes dramatically (Figure 6.1). Intel MPX crashes due to insufficient memory even on tiny input sizes. The cause for this behavior is the amount of bounds tables created to support pointer metadata ($800 - 900$ tables each 4MB in size), leading to memory exhaustion. We should note however that SQLite is a worst-case example for MPX since it is exceptionally pointer-intensive; pointerless programs, e.g., those using flat arrays, perform significantly better under MPX (see §6.6).

AddressSanitizer performs up to 3.1× slower than the native SGX execution on bigger inputs. Performance deteriorates mainly due to the EPC thrashing caused by additional metadata accesses to shadow memory. Moreover, AddressSanitizer also has a constant memory overhead of 512MB for shadow memory plus some overhead for redzones around objects. This can lead to situations when the application prematurely suffers from insufficient memory.

For the same experiment, SGXBounds shows performance comparable to native SGX ($30-35$% slower) with almost no memory overhead. This motivates our case for a specialized memory safety approach for shielded execution.

6.3 SGXBounds

We built SGXBounds based on the following three insights. First, as shown in §6.2.1, shielded application memory (more specifically, its working set) must be kept minimal due to the very limited EPC size in current SGX implementations. This is in sharp contrast to the usual assumption of almost endless reserves of RAM for many other memory-safety approaches [6, 27, 68, 110, 143, 158, 207]. Second, applications spend a considerable amount of time iterating through the elements of an array [52], and a smartly chosen layout of metadata can significantly reduce the overhead of bounds checking. Third, we rely on the SCONE infrastructure [15] with its monolithic build process: all application code is statically linked without external dependencies, which removes the requirements for compatibility and modularity. The first and second insights dictate the use of per-object metadata combined with *tagged pointers* [6, 39] to keep memory overhead minimal, and thanks to the monolithic application assumption, SGXBounds avoids problems of interoperability with uninstrumented code [225].

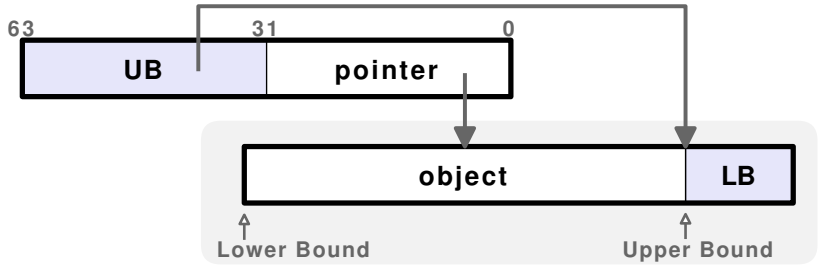

Figure 6.5 – Tagged pointer representation in SGXBounds.

6.3.1 Design Overview

All modern SGX CPUs operate in a 64-bit mode, meaning that all pointers are 64 bits in size. In SGX enclaves, however, only 36 bits of virtual address space are currently addressable [108], and even this amount of space is not likely to be used due to performance penalties. Thus, SGXBounds relies on the idea of *tagged pointers*: a 64-bit pointer contains the pointer itself in its lower 32 bits and the referent object's upper bound in the upper 32 bits (Figure 6.5). Note that with SCONE, all application code and data are stored inside the enclave address space and thus all addressable memory is confined to 32 bits and all original pointers can be replaced by their tagged counterparts.

The value stored in the higher 32 bits (UB) serves not only for the upper-bound check, but also as a pointer to the object's other metadata (lower bound or LB). The metadata is stored right after the referent object.

This metadata layout has important benefits: (1) it minimizes amount of memory for metadata, (2) it requires *no* additional memory accesses while iterating over arrays with a positive increment, and (3) it alleviates problems of fat pointers concerning multithreading and memory layout changes (see §6.4.1).

Figure 6.4d shows how SGXBounds instruments memory accesses. First, global arrays `s` and `d` are initialized with their respective bounds, and `s` and `d` pointers are transformed into tagged pointers (lines 2–3). For the sake of clarity, we show pointer increments on lines 5–6 uninstrumented (details are in §6.3.2). Next, before the first memory access at line 10, SGXBounds inserts a bounds check. For this, the original pointer value and its upper bound are extracted from the tagged `si` as well as the lower bound, and the bounds check is performed (lines 7–9). The second memory access (line 14) is instrumented in the same way.

Looking at Figure 6.4, we can highlight the differences between SGXBounds, AddressSanitizer and Intel MPX. Unlike AddressSanitizer, SGXBounds does not rely on a vast amount of shadow memory, allocating only 4 additional bytes per object. Also, AddressSanitizer requires adjacent objects to be separated by fixed-size unaddressable redzones and checks whether the memory access lands on one of these redzones. In contrast, SGXBounds extracts pointer bounds and compares the current value of the pointer against them—similar to Intel MPX. But unlike Intel MPX, SGXBounds does not maintain a bounds table and does not explicitly associate each pointer with its own bounds metadata: the newly created pointer implicitly inherits all associated metadata.

6.3.2 Design Details

Pointer creation. Whenever an object is created, SGXBounds associates a pointer with the bounds of this object.

For global and stack-allocated variables, we change their memory layout so they are padded with 4 bytes and initialize them at run-time. More specifically, we wrap such variables in two-member structures, e.g., `int x` is transformed into `struct xwrap {int x; void* LB}` (similar to [207]). At program initialization, we set the lower and upper bounds of each object with `specify_bounds(&xwrap, &xwrap.LB)`:

```
void* specify_bounds(void *p, void *UB):
    LBaddr = UB
    *LBaddr = p
    tagged = (UB << 32) | p
    return tagged
```

For dynamically allocated variables, SGXBounds wraps memory-management functions such as `malloc`, `calloc`, etc. to append 4 bytes to each newly created object, initialize these bytes with the lower-bound value, and make the pointer tagged with the upper bound:

```
void* malloc(int size):
    void *p = malloc_real(size + 4)
    return specify_bounds(p, p + size)
```

Note that there is no need to instrument `free` as the 4 bytes of metadata are removed together with the object itself.

Lastly, a pointer can be assigned a value of another pointer. If we would use fat pointers or pointers with disjoint metadata, we would need to instrument such pointer assignments, as in Intel MPX (see Figure 6.4c). However, in SGXBounds no instrumentation is needed, since the newly assigned pointer will also inherit the upper bound and thus all associated object metadata.

Run-time bounds checks. SGXBounds inserts run-time bounds checks before each memory access: loads, stores, and atomic operations (we revise this statement in §6.4.4). For this, first the original pointer and the upper and lower bounds are extracted. To extract the original pointer, it is enough to use only the lower 32 bits:

```
void* extract_p(void* tagged):
     return tagged & 0xFFFFFFFF
```

Similarly, to extract the upper bound, the higher 32 bits of the tagged pointer must be extracted:

```
void* extract_UB(void* tagged):
    return tagged >> 32
```

If a check against a lower bound is also required then this bound is read from the memory at the upper-bound's address:

```
void* extract_LB(void* UB):
    return *UB
```

Finally, SGXBounds adds the bounds check which crashes the application in case the bounds are violated (in the implementation, we take into account the size of the accessed memory while checking against the upper bound; here we omit it for clarity):

```
bool bounds_violated(void* p, void* LB, void* UB):
    if (p < LB or p >= UB):
        return true
```

Pointer arithmetic. There is a subtle issue with tagged pointers when it comes to pointer arithmetic. Take, for example, increment of a pointer as shown in Figure 6.4d, lines 5–6. In the ordinary case, pointer arithmetic affects only the lower 32 bits of a tagged pointer. However, it is possible that a malicious/buggy integer value overflows 32 bits and changes the upper bound bits. In this case, the attacker can manipulate the upper bound value and bypass the bounds check. To prevent such corner cases, SGXBounds instruments pointer arithmetic so that only

32 low bits are affected:

```
UB = extract_UB(si)
si = s + i
si = (UB << 32) | extract_p(si)
```

Type casts. Pointer-to-integer and integer-to-pointer casts are a curse for fat/tagged pointer approaches. Some techniques break applications with such casts [6, 117], others suffer from worse performance or lower security guarantees [39, 110, 158]. Unfortunately, arbitrary casts are common in real-world [52].

SGXBounds proved itself immune to arbitrary type casts. It does not perform any instrumentation on type casts and survives integer-to-pointer casts by design. Indeed, when a tagged pointer is casted to an integer, the integer inherits the upper bound. Unless the integer deliberately alters its high 32 bits, the upper bound will stay untouched and the later cast back to a pointer will preserve this bound.

Function calls. SGXBounds does not need to instrument function calls or alter calling conventions. Unlike other approaches [6, 39, 64, 110, 158], SGXBounds is not required to interoperate with possibly uninstrumented, legacy code: the only uninstrumented code is the standard C library (libc) for which we provide wrappers. This implies that any tagged pointer passed as a function argument will be treated as a tagged pointer in the callee. In other words, bounds metadata travels across function and library boundaries together with the tagged pointer.

As already mentioned, we leave libc uninstrumented and introduce manually written wrappers for all libc functions, similar to other approaches [6, 110, 158, 207]. Most wrappers follow a simple pattern of extracting original pointers from the tagged function arguments, checking them against bounds, and calling a real libc function. Others require tracking and extracting the pointers on-the-fly (e.g., the `printf` family), writing proxies for callbacks (`qsort`), or iterating through complex objects (`scandir`).

6.4 Advanced Features of SGXBounds

6.4.1 Multithreading support

Bounds checking approaches usually hamper multithreaded applications. AddressSanitizer does not require any specific treatment of multithreading, but, as we illustrate in §6.6.4, it can negatively affect cache locality if a multithreaded application was specifically designed as cache-friendly (recall that AddressSanitizer inserts redzones around objects). On the other hand, current implementations of Intel MPX instrumentation may suffer from false positives and false negatives in multithreaded environments, introducing a possibility of false alarms or, even worse, of undetected attacks [52, 173].

In fact, all fat-pointer or disjoint-metadata techniques similar to Intel MPX suffer from multithreading issues [52, 157]. An update of a pointer and its associated metadata must be implemented as one atomic operation which requires some synchronization mechanism. This inevitably hampers performance as this is necessary for each pointer/metadata update.

For example, in Figure 6.4c, lines 10–11, the pointer `val` and its bounds metadata `val_bnd` are copied to `di`. After the first thread loaded `val` on line 10, the second thread can jump in and change `val` to point to some other object. This will also change `val_bnd`. Next, the first thread continues its execution and loads the wrong `val_bnd` on line 11. Now `val` and `val_bnd` do not match, which might result in a false positive. This is a realistic failure scenario for current

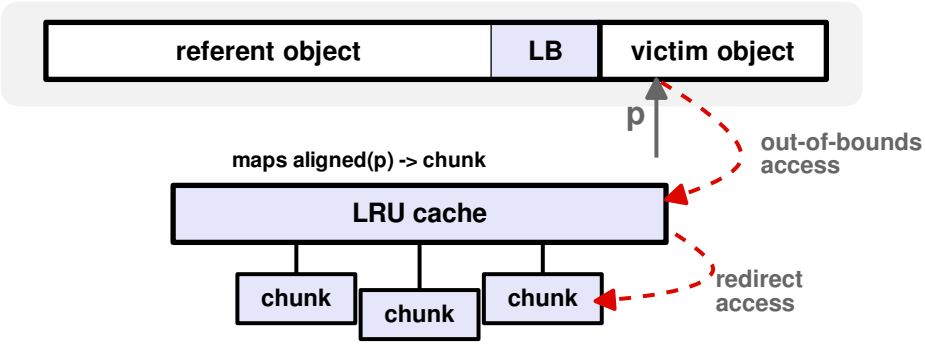

Figure 6.6 – Boundless memory blocks for SGXBounds.

implementations of Intel MPX since it does not enforce atomicity of metadata updates.[2]

SGXBounds does not experience this problem. Indeed, the pointer and the upper bound are always updated atomically since they are stored in the same 64-bit tagged pointer. Additionally, the lower bound is written only once (at object creation) and is read-only for the whole object's lifetime.

6.4.2 Tolerating Bugs with Boundless Memory

Up to this point, we assumed that an application crashes with a diagnostic error whenever SGXBounds detects an out-of-bounds access. This fail-fast strategy is simple and prevents hijacks and data leaks, but lowers availability of the system. Even in benign cases of off-by-one buffer overflows, the whole application is crashed and must be restarted.

To allow applications to survive most bugs and attacks and continue correct execution, SGXBounds reverts to failure-oblivious computing [194] by using the concept of boundless memory blocks [193]. In this case, whenever an out-of-bounds memory access is detected, SGXBounds redirects this access to a separate "overlay" memory area to prevent corruption of the adjacent objects, creating the illusion of "boundless" memory allocated for the object (see Figure 6.6).

This overlay area is implemented as a bounded least-recently-used (LRU) cache—a hash table that maps out-of-bounds memory addresses to spare chunks of memory (similar to [193]). These chunks are allocated on-demand, each being 1KB in size. The whole LRU cache is bounded, i.e., it cannot grow more than a certain predefined size (in our implementation, 1MB). This is required to prevent bugs and attacks that span gigabytes of out-of-bounds memory—a frequent consequence of integer overflows due to negative buffer sizes.

Consider an example of a classical off-by-one bug from Figure 6.4d. If `M` is greater than `N` by one, the last iteration of the loop will trigger bound violations on lines 8 and 12.

With boundless memory feature enabled, SGXBounds consults the LRU cache and redirects the load from `si` (line 10) to a load from an overlay address that corresponds to `si`. If there is no hit for `si` in the cache, SGXBounds falls back on a failure-oblivious approach and simply returns zero values.

Additionally, SGXBounds redirects the store to `di` (line 14) to a corresponding overlay address. If there is no overlay address in the LRU cache, then a new chunk of overlay memory is allocated and is associated with this address. If there is no space left for a new chunk in the LRU cache, the least recently used chunk is evicted (freed) and the new chunk is added instead.

[2] We demonstrate how multithreaded code fails in MPX and discuss this and other issues in more detail in the next chapter.

`on_create(objbase, objsize, objtype)`	called after object creation (globals, heap, or stack)
`on_access(address, size, metadata, accesstype)`	called before memory access (read, write, or read-write)
`on_delete(metadata)`	called before object destruction (only for heap)

Table 6.2 – SGXBounds metadata management APIs.

6.4.3 Metadata Management Support

So far, we discussed only one metadata type kept per object—the lower bound (see Figure 6.5). However, our memory layout allows us to add arbitrary number of metadata items for each object to implement additional functionality.

All instrumentation in SGXBounds is implemented as calls to auxiliary functions described in §6.3.2, which we refer to as instrumentation hooks. One can think of these hooks as a metadata management API (see Table 6.2). The API consists of only three functions: (1) `on_create` is called at run-time whenever a new object is created, either a global variable during program initialization or a local variable during stack frame creation or a dynamically allocated variable, e.g., via `malloc`. In the context of SGXBounds, it corresponds to the `specify_bounds` function which initializes our only metadata (lower bound). (2) `on_access` is called at each memory access, be it a write, read, or both (for atomic instructions such as compare-and-swap). In SGXBounds, the hook roughly corresponds to the `bounds_violated` function. (3) `on_delete` is called whenever an object is deallocated; we support this hook only for heap-allocated objects, since global variables are never deleted and there is no way to track deallocation of variables on stack. SGXBounds does not use this hook because we do not focus on temporal safety (also note that the metadata is removed automatically with the object).

With this API, it is straightforward to implement additional functionality. For example, SGXBounds can be expanded to probabilistically protect against double-free bugs using an additional metadata item acting as a "magic number" to compare with. Another example would be providing debug information about where a detected out-of-bounds access originates from.

6.4.4 Optimizations

Safe memory accesses. Many pointer arithmetic operations and memory accesses are always-safe. For example, the calculation of the member's offset in a structure is guaranteed to be in-bounds and never overflows 32 low bits. The memory access at a predefined index in a fixed-size array is also safe.

In these cases, there is no need for instrumentation of pointer arithmetic or bounds checks on memory accesses. We employ the built-in compiler analysis to detect all safe cases and do not instrument them. This is a standard optimization for many approaches [6, 64, 207] and yields significant performance gains for some applications, up to 20% (§6.6.5).

Hoisting checks out of loops. Many programs spend a lot of time iterating over arrays in simple loops. The array-copy example in Figure 6.4a is a good illustration.

The straightforward instrumentation with SGXBounds, as depicted in Figure 6.4d, inserts bounds checks before each memory access (on lines 7–9 and 11–13). It is immediately obvious from the code that the lower-bound check is useless: `si` and `di` start from the base addresses of

the corresponding arrays and increment on each iteration. Thus, it is safe to remove the check against the lower bound, which renders the extraction of the lower bound redundant. In the end, this optimization can save two memory accesses per iteration (to extract LBs).

The upper-bound check cannot be removed: in general case the value of `M` is unknown and can exceed the upper bound of the two arrays (`N`). But it is sufficient to perform only one check for each array outside of the loop, namely, the check of `s+M` and of `d+M` against their respective upper bounds.

Such optimization is applied only for loops with small increments (up to 1,024 bytes) – which is virtually all loops encountered in regular applications. We mark the last 4K page of an enclave as unaddressable, which protects from integer over- and underflows of the loop counter variable. These simple precautions protect against overflowing pointer arithmetic inside loops when lower- or upper-bound checks are hoisted out.

To perform these optimizations, we reused classical scalar evolution analysis. We observed performance gains of up to 22% in some cases (§6.6.5).

6.5 Implementation

6.5.1 SGXBounds Implementation

SGXBounds is a compile-time transformation pass implemented in LLVM 3.8. For greater modularity, we implement the functionality outlined in §6.3.2 as always-inlined functions in a separate C file. The pass inserts calls to these functions during instrumentation. We refer to this set of auxiliary C functions as the run-time for SGXBounds.

We do not alter the usual build process of an application, but rather use the Link-Time Optimization feature of LLVM.

Compiler support. SGXBounds compiler pass works under LLVM 3.8 [134] and was implemented in 951 lines of code (LOC). Its functionality closely follows the description in §6.3.

We treat inline assembly as an opaque memory instruction: all pointer arguments to inline assembly are bounds checked. To minimize the risk of misbehaving assembly, we disabled inline assembly in all tested applications which had such a flag.

To support C++, we opted to instrument the whole C++ standard library. We used libcxx (libc++) implementation for this purpose. SGXBounds does not yet completely support C++ exception handling: it runs C++ applications correctly only if they do not throw exceptions at run-time.

Run-time support. Next we describe implementation details of the SGXBounds auxiliary functionality. The complete implementation of the run-time functions spans 320 LOC, and the libc wrappers contain 4289 LOC.

We implemented boundless memory feature (§6.4.2) completely in the run-time support library in 68 LOC. It is based on uthash lists which we extend to a simple LRU cache [231]. To prevent data races, all read/update operations on the cache are synchronized via a global lock. Such implementation is slow, but since it is triggered on supposedly rare events of out-of-bounds memory accesses (and thus it lies on a slow path), we can ignore this possible performance bottleneck.

Furthermore, SGXBounds does not fall back to a failure oblivious approach for libc function wrappers, but rather returns an error code through `errno` where applicable (e.g., `EINVAL` for the `recv` function). This allows applications to quickly drop offending requests.

For the tagged pointer scheme, SGXBounds relies on SGX enclaves (and thus the virtual address space) to start from 0x0. To allow this, we set the Linux security flag `vm.mmap_min_addr` to zero for our applications. We also modified the original Intel SGX driver (5 LOC) to always start the enclave at address 0x0.

6.5.2 AddressSanitizer, Intel MPX, and SGX Enclaves

To integrate AddressSanitizer and Intel MPX into SGX enclaves, we had to solve three main issues. (1) SCONE disallows dynamic linking against shared libraries, so AddressSanitizer and Intel MPX must be compiled statically into the application. (2) The virtual address space is restricted to 32 bits. (3) The OS is not allowed to peek into the address space of the enclave.

Adapting AddressSanitizer for SGX enclaves. We had to solve issues (1) and (2) for AddressSanitizer. First, the current implementation of AddressSanitizer relies on libc being dynamically linked at application start-up (the usual function interposition scheme). Trying to statically link libc into the application would result in a compilation error due to multiple definitions of the same function.

Every function in SCONE libc has an alias (a second name which is used to denote the real function). We modified the interception layer of AddressSanitizer such that its wrapper functions call aliases (real libc functions), therefore solving the problem of multiple definitions. This is similar to SGXBounds (see `malloc` in §6.3.2).

Second, by default AddressSanitizer is compiled in 64-bit mode and reserves $\sim$ 16TB of memory for its shadow space. Fortunately, it also has a 32-bit mode where only 512MB of memory is carved for shadowing. We changed the build system of AddressSanitizer to always use the 32-bit mode. Also, we disabled "leak detection" flag that broke SCONE.

Adapting Intel MPX for SGX enclaves. To put Intel MPX inside SGX, we solved issues (2) and (3). Intel MPX operates in the 64-bit mode, and this affects its address translation to store and load bounds (Figure 9 in [110]). In the 64-bit mode, Intel MPX allocates a 2GB Bounds Directory (BD) table at start-up and 4MB-sized Bounds Tables (BT) on-demand.

We discovered that this address translation also works with 32-bit addresses. In the 32-bit address case, only 12 bits are used for indexing in the BD table, and the rest for BT tables. Thus, we were able to restrict the size of BD to 32KB by changing the corresponding constants in the MPX compiler pass and run-time libraries. We did not change the address translation logic of BT allocation.

For issue (3), we had to move the kernel logic into the SGX enclave. In the normal case, on-demand allocation of BTs requires support from the Linux kernel. Whenever an application fires a "bounds store" exception (meaning the application needs to allocate a new BT to store some pointer metadata), the kernel handles it: it examines the pointer address that raised the exception, calculates the correct BT, and allocates it on behalf of the application. Then the execution of the application continues, and the metadata is stored in the newly allocated BT.

This kernel-application cooperation is impossible in SGX. The kernel cannot examine the failing pointer and cannot peek into or modify memory inside the SGX enclave. To alleviate this problem, we moved all the BT-allocation logic from the kernel into the Intel MPX run-time library. We also instructed the kernel not to try to cooperate with the application, but only to forward the exception to the application itself. At this point the enclave takes control and handles the exception. Note that this logic does not compromise security because SGX double-checks the exceptions forwarded by the kernel. Our adaptation also does not influence performance since BT-allocation is a rare event, and the kernel-to-application forwarding adds negligible overhead.

6.6 Evaluation

Our evaluation answers the following questions:

- What are the performance and memory overheads of SGXBounds and how do they compare to AddressSanitizer and Intel MPX? (§6.6.2)
- How does the increasing working set affect the performance of SGXBounds? (§6.6.3)
- How does multithreading affect the performance? (§6.6.4)
- How effective are the optimizations in improving the performance? (§6.6.5)
- What level of security is achieved by SGXBounds according to the RIPE benchmark? (§6.6.6)
- How does the performance of SGXBounds change outside of SGX enclaves? (§6.6.7)

6.6.1 Experimental Setup

Applications. We evaluated SGXBounds using Fex [172] with applications from two multi-threaded benchmark suites: Phoenix 2.0 [187] and PARSEC 3.0 [32], as well single-threaded SPEC CPU2006 [99]. We report results for all 7 applications in the Phoenix benchmark, 9 out of 13 applications in PARSEC, and 13 out of 19 in SPEC. The remaining applications are not supported for the following reasons: *raytrace* depends on the dynamic X Window System libraries not shipped together with the benchmark; *freqmine* is based on OpenMP, *facesim* and *canneal* fail to compile under SCONE due to position-independent code issues, *dealII*, *omnetpp*, and *povray* fail due to incomplete support of C++, *perlbench* triggered an unsupported corner case of a specific loop optimization, and *gcc* and *soplex* violate C memory model and cannot be protected via bounds-checking [173].

Methodology. In all experiments (except §6.6.3) the numbers are normalized against the native SGX version, i.e., a version compiled under the SCONE infrastructure and not instrumented with any memory-safety techniques. For all measurements, we report the average over 10 runs and geometric mean for the "gmean" across benchmarks. For memory measurements, since the Linux kernel does not provide statistics on the Resident Set Size inside SGX enclaves, we show the maximum amount of reserved virtual memory.

Testbed. We used the largest available datasets provided by Phoenix, PARSEC, and SPEC benchmark suites. The experiments were carried out on a machine with a 4-core (8 hyper-threads) Intel Xeon processor operating at 3.6 GHz (Skylake μarchitecture) with 64GB of RAM, a 1TB SATA-based SDD, and running Linux kernel 4.4. Each core has private 32KB L1 and 256KB L2 caches, and all cores share a 8MB L3 cache.

Compilers. We used LLVM 3.8 for native SGX, AddressSanitizer, and SGXBounds versions and gcc 5.3 for the Intel MPX version. We use default options for AddressSanitizer but disable leak detection (see §6.5.2). We also disable "narrowing of bounds" feature in Intel MPX to remove false positives in some programs.

6.6.2 Performance and Memory Overheads

Figure 6.7 shows performance and memory overheads of Intel MPX, AddressSanitizer, and SGXBounds normalized against the uninstrumented SGX version. All benchmarks were run with 8 threads to fully utilize our machine.

Performance overheads of Intel MPX significantly vary across benchmarks, reaching up to $5-6\times$ in some cases. For example, consider *pca*. Its working set is 70MB (77MB for Intel MPX

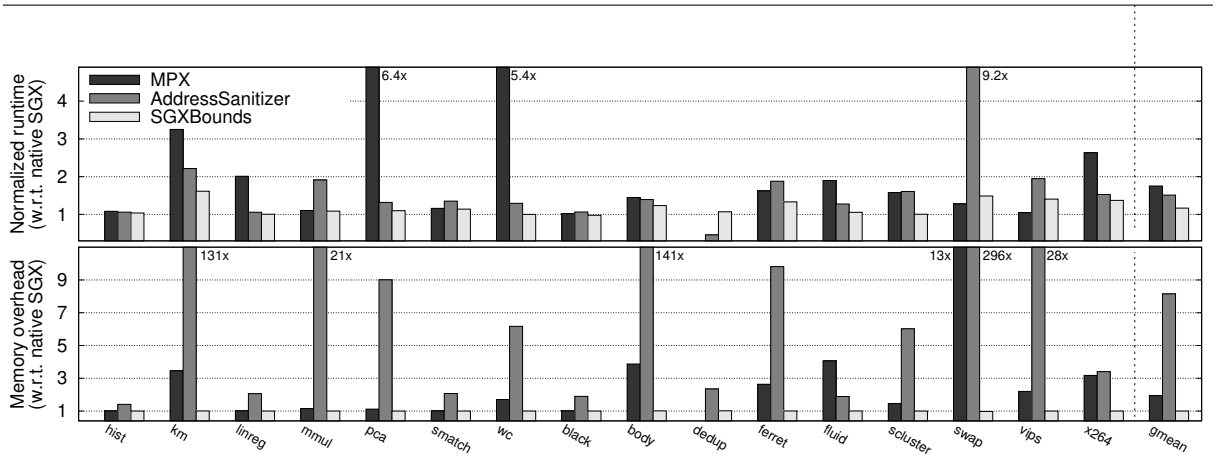

Figure 6.7 – Performance (top) and memory (bottom) overheads over native SGX execution (with 8 threads).

due to additional metadata), thus all data fits into EPC and performance is dominated by the on-die characteristics like CPU cache accesses and number of retired instructions. Indeed, Intel MPX leads to 10× more instructions, 5× more branches, and 25× more L1 cache accesses (*pca* operates on a large array-of-pointers and is thus pointer-intensive). Together, this leads to an overhead of 6.3×. On the other hand, pointer-free benchmarks like *histogram* and *blackscholes* exhibit almost zero overhead (observe that memory overheads in these cases are also close to zero).

Memory overheads of Intel MPX also vary. For benchmarks working with large arrays and/or using no pointer-based structures (almost all Phoenix benchmarks), pointer bounds metadata occupies relatively small amount of space and overheads are negligible. However, for pointer-intensive cases like *bodytrack* and *fluidanimation*, Intel MPX allocates a lot of metadata, leading to $\sim 4\times$ memory overhead. In degenerate cases, overheads can reach up to 13× (*swaptions*) or even crash the application (*dedup*, note the missing MPX bar).

AddressSanitizer has more reasonable and expected performance overhead of around 51%.[3] The *kmeans* benchmark has one of the highest overheads of 2.2×. Since the working set of *kmeans* is only 5MB (AddressSanitizer blows it up to 643MB but does not use most of it), the overhead is dominated by the CPU instructions and cache: 2.4× more instructions, 2.6× more branches, and 2.2× more L1 cache accesses.

In terms of memory usage, AddressSanitizer is a poor choice for SGX enclaves. By reserving 512MB of memory for its shadow space, AddressSanitizer reduces the available memory to 3.5GB (§6.2.2). Moreover, AddressSanitizer pads objects with redzones and uses so-called "quarantine" which obstructs reuse of memory [207]. All this can lead to memory blow-ups of $50 - 100\times$.

The most dramatic example of memory overheads is *swaptions*. This benchmark has a working set of only 3.3MB, but it constantly allocates and frees tiny objects. For Intel MPX, it results in a flood of pointers and a constant need for more and more bounds tables (12 BTs or 48MB). For AddressSanitizer with its quarantine feature, the reuse of memory is restricted and new objects are allocated in more and more pages ($103,250$ pages or 413MB). Note that the excessive amount of metadata does not seriously hamper performance of Intel MPX because the working set still fits into EPC, but AddressSanitizer suffers from EPC thrashing and thus exhibits poor performance.

[3]Except *dedup* which performs better than the baseline SGX version. Our investigation revealed that AddressSanitizer accidentally changes the memory layout of *dedup* such that it has much less LLC cache misses at runtime.

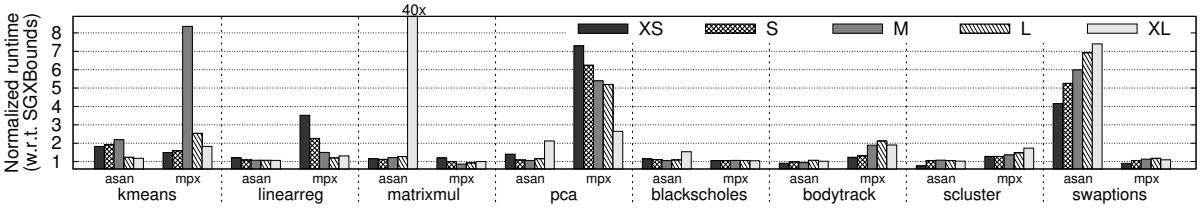

Figure 6.8 – Performance overheads over SGXBounds execution with increasing sizes of working sets (with 8 threads).

Working set (MB)		LLC misses (%)		Page Faults (×)		# of
		ASan	MPX	ASan	MPX	BTs
kmeans						
XS	(17)	5.8	-0.3	3.9	1.2	6
S	(34)	12.4	1.3	3.1	2.0	9
M	(68)	17.2	9.7	3.9	**44**	15
L	(**135**)	19.7	1.3	1.2	2.9	27
XL	(**270**)	11.3	1.5	1.2	1.9	52
matrixmul						
XS	(2)	1.7	1.0	9.4	1.5	1
S	(7)	-0.5	-1.2	5.8	1.4	1
M	(26)	-3.6	-13.8	2.9	1.2	1
L	(103)	125	-11.5	1.9	1.0	1
XL	(**412**)	**4367**	-0.1	1.2	1.0	1

Table 6.3 – Overheads w.r.t. SGXBounds for experiment of increasing working set size. Col. 4–5: page faults due to EPC thrashing. Col. 6: num. of bounds tables allocated in MPX.

Finally, SGXBounds performs the best, with an average performance overhead of 17% and average memory overhead of 0.1%. In comparison to Intel MPX, SGXBounds does not choke on pointer-intensive programs (*pca*, *wordcount*, *x264*). In comparison to AddressSanitizer, SGXBounds has much better memory consumption. It also does not exhibit corner-case performance drops like AddressSanitizer in *swaptions* and does not eat up all memory like Intel MPX in *dedup*.

6.6.3 Experiments with Increasing Working Set

To understand the behavior of different approaches with increasing sizes, we created five input sizes ranging from tiny (XS) to extra-large (XL) for several benchmarks (Figure 6.8). Note that we normalize against SGXBounds for clarity; SGXBounds itself performs $\sim 15\%$ worse than native SGX and has a maximum deviation of 2.1% across different sizes. We observed different patterns across approaches and benchmarks. In most cases, increasing the size did not influence the overheads of AddressSanitizer and Intel MPX in comparison to SGXBounds, indicating no changes in memory access patterns due to CPU cache or EPC thrashing. Next, we elaborate on the patterns for some other cases.

Kmeans has the following pattern: the overheads over SGXBounds grow until a certain point ("M"), reach a maximum and then drop. Looking at Table 6.3, we note that the working set fits completely in EPC at first and then spills out to RAM at large inputs. This means that before the "L" value, overheads are dominated by the on-die characteristics, and after it by the paging

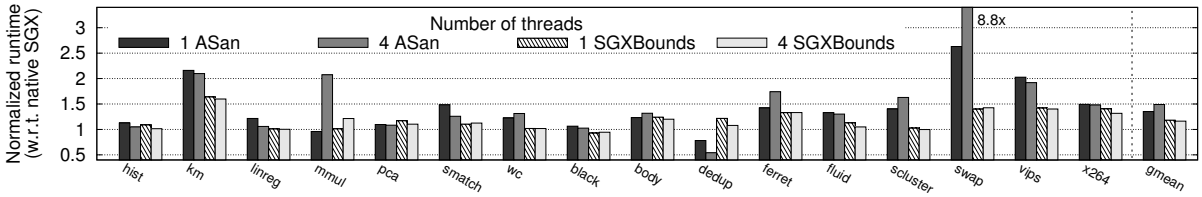

Figure 6.9 – Performance overheads of AddressSanitizer and SGXBounds over native SGX with different number of threads.

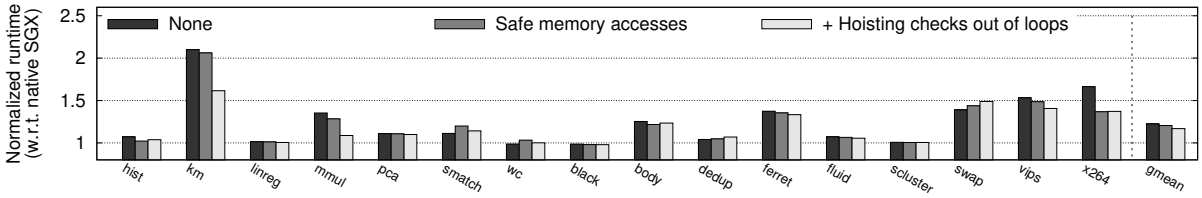

Figure 6.10 – Performance overheads of SGXBounds over native SGX execution with different optimizations (with 8 threads).

mechanism. In the case of *kmeans*—a benchmark which iteratively goes through its working set—the number of page faults explains the spikes and subsequent drops in both Intel MPX and AddressSanitizer.

Note the outlier number of page faults for "M" in Intel MPX: the working set increases to 127MB due to bounds tables. At the same time, the original SGX version and SGXBounds both have the working set of 68MB. Thus, SGXBounds fits completely into EPC while Intel MPX must evict and load-back pages (AddressSanitizer also has a working set that fits into EPC). Since such constant EPC thrashing is expensive (§6.2.1), performance of Intel MPX becomes 8.3× worse.

On "L" and "XL" sizes, all approaches do not fit into EPC and experience EPC thrashing, and this dominates the performance overheads of all of them. Note how the number of page faults from Table 6.3 correlates with the overhead in Figure 6.8.

Matrixmul performs a simple (cache-unfriendly) multiplication of two matrices and writes the result into a third matrix.

Intel MPX performs on par with SGXBounds. Looking at the number of bounds tables allocated (Table 6.3), we see that only one table was enough for any input size. This is trivially explained by the fact that *matrixmul* requires only three bounds entries—one for each matrix. Moreover, Intel MPX holds these bounds in CPU registers such that there are *no* additional memory accesses and thus no overhead.

Note that *matrixmul* exhibits sequential pattern of memory accesses. This implies that even when the working set does not fit in EPC, there is no EPC thrashing (old EPC pages are evicted and never accessed again) – in other words, page faults do not dominate performance overheads. In this scenario, CPU cache misses play a major role. AddressSanitizer breaks cache locality since it inserts additional accesses to shadow memory. On "XL" size, this effect is exacerbated by matrices not fitting in EPC, leading to 44× more LLC cache misses. This explains the 40× spike in overhead in Figure 6.8.

Approach	Prevented attacks
MPX	**2/16** (except return-into-libc on heap & data)
AddressSanitizer	**8/16** (except in-struct buffer overflows)
SGXBounds	**8/16** (except in-struct buffer overflows)

Table 6.4 – Results of RIPE security benchmark.

6.6.4 Effect of Multithreading

As discussed in §6.4.1, SGXBounds supports multithreading by design. To highlight the fact that SGXBounds does not impose additional performance overhead with more threads, we conducted an experiment with one and four threads (Figure 6.9). Also, the overheads with 8 threads are shown in section 6.6.2. We compare SGXBounds with AddressSanitizer which also has an efficient support for multithreading. We do not compare against Intel MPX since it lacks real support for multithreading; we believe that future versions of MPX might have deteriorated performance due to synchronization overheads.

On average, overhead of AddressSanitizer increases from 35% with one thread to 49% with four threads while overhead of SGXBounds decreases from 17% to 16%. In most cases however, both SGXBounds and AddressSanitizer do not exhibit any additional overhead. This is reasonable since both approaches do not require additional synchronization primitives and introduce lightweight wrappers around `pthreads`.

However, AddressSanitizer can break (1) memory layout due to redzones around objects, and (2) cache locality due to additional memory accesses to shadow memory. This happens in *matrixmul*: AddressSanitizer worsens cache locality on four threads and has 6.7× more LLC cache misses than SGXBounds. Note that SGXBounds adds only 12 bytes in *matrixmul* (4B for each matrix) which preserves the original memory layout. Thanks to this, SGXBounds performs 70% better than AddressSanitizer on 4 threads. A similar explanation holds true for *swaptions*.

6.6.5 Effect of Optimizations

We evaluated gains of optimizations as detailed in §6.4.4. The results are shown in Figure 6.10. On average, applying all optimizations yields a modest performance improvement of 2%.

Unfortunately, our optimizations are limited in scope. Our implementation relies on Scalar Evolution and SizeOffsetVisitor LLVM analyses. However, they do not yet support inter-procedural (whole-program) analysis. Therefore, the results turned out to be not as impressive as we originally hoped; we believe that enabling inter-procedural analysis in future implementations could greatly improve performance.

Nonetheless, our optimizations can give significant performance boost in some cases. For example, the hoisting checks optimization is helpful for *kmeans* and *matrixmul*, with performance improvements of up to 20%. Similar gains are seen for *x264* when the safe checks optimization is applied.

6.6.6 Security Benchmark (RIPE)

To evaluate security guarantees of SGXBounds, we employed the RIPE security benchmark [244]. RIPE claims to perform 850 working buffer-overflow attacks. However, under our native configuration, only 46 attacks were successful: through the shellcode that creates a dummy file and through return-into-libc. When building RIPE under SCONE infrastructure, this number

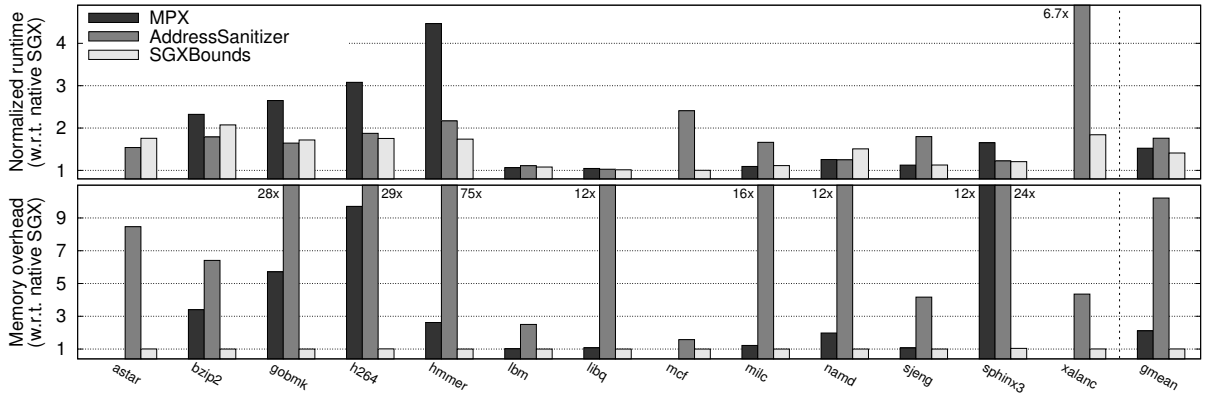

Figure 6.11 – SPEC inside of SGX enclave: Performance (top) and memory (bottom) overheads over native SGX execution.

decreased to 16 attacks: the shellcode attacks failed because SGX disallows the `int` instruction used in shellcode.

Table 6.4 shows the security results of all approaches. Intel MPX could not detect 14 out of these 16 attacks: the two attacks detected were both stack-smashing attacks trying to overwrite an adjacent function pointer. AddressSanitizer detected 8 out of 16 attacks: the remaining 8 attacks were all in-struct buffer overflows, when the same object contained a vulnerable buffer and a target-of-attack function pointer. Finally, SGXBounds showed the exact same results as AddressSanitizer. The in-struct overflows could not be detected because both AddressSanitizer and SGXBounds operate at the granularity of whole objects.

6.6.7 SPEC CPU2006 Experiments

To facilitate comparison with other approaches, we also report the overheads of SGXBounds over the SPEC CPU2006 benchmark suite. Note that all programs in SPEC are single-threaded and more CPU-intensive than Phoenix and PARSEC, such that the restrictions of SGX have less impact for SPEC. We performed two experiments to measure performance and memory consumption: inside of SGX enclaves (similar to previous evaluation) and outside them (to understand overheads in normal, unconstrained environments).

SGXBounds, being a bounds-checking approach, has false positives in some legitimate programs that implement custom memory management. For example, we could not run *soplex* because it directly updates referent objects of pointers. SGXBounds can also break on programs that manipulate high bits of pointers, e.g., *gcc* contains unions of pointers-ints and manipulates high bits. Note that other approaches have the same problems with these programs, e.g., MPX [173], Baggy Bounds [6], and Low Fat Pointers [131] – they all require manual modifications to misbehaving programs.

Figure 6.11 shows the results for our in-enclave scenario. In agreement with experiments on Phoenix and PARSEC (see Figure 6.7), SGXBounds shows the lowest performance and memory overheads on average, 41% and 0.4% respectively. Again, SGXBounds adds negligible overhead in memory consumption which in many cases leads to better cache and EPC locality. Consider *mcf*: AddressSanitizer exhibits performance overhead of 2.4× whereas SGXBounds–only 1%. This is explained by EPC thrashing: AddressSanitizer has $3,400\times$ more page faults than both original and SGXBounds versions. Similar explanations hold for other extreme cases such as *milc*, *sjeng*, and *xalanc*.

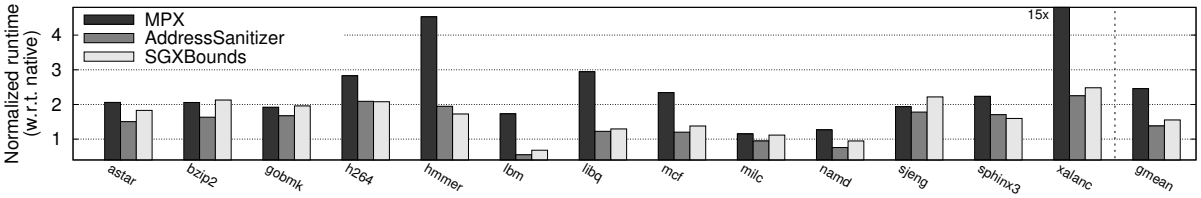

Figure 6.12 – SPEC outside of SGX enclave (normal unconstrained environment): Performance overhead over native execution.

Intel MPX performed slightly better than AddressSanitizer (52% performance and 110% memory overhead against 76% and 10× respectively) but failed to finish on *astar*, *mcf*, and *xalanc*. Just like in cases of *SQLite* and *dedup*, these programs crash due to insufficient memory for MPX Bounds Tables.

In addition, we show results for outside-enclave, unconstrained environment in Figure 6.12. As expected, SGXBounds performs not that well outside of enclaves, with a higher average performance overhead (55%) than AddressSanitizer (38%).[4] In unrestricted-memory environments, the benefits of a cache-friendly layout of SGXBounds are effectively wiped out, even though the memory consumption of SGXBounds is only 0.1% in contrast to $2-4\times$ of MPX and AddressSanitizer (not shown on plots). Also, the 55% performance overhead of SGXBounds is comparable to the ones incurred by Baggy Bounds (70%) and Low Fat Pointers (43%)[5]; see also §6.2.2.

6.7 Case Studies

In addition to SQLite, we evaluated three other applications. Our evaluation of the case-studies is based on: (1) performance and memory overheads; and (2) security guarantees. All applications were evaluated on the machine described in §6.6; clients connected via a 10Gb network.

Memcached. We evaluated Memcached v1.4.15 [78] using the memaslap benchmark shipped together with libmemcached v1.0.18 client [138]. Performance and memory overheads are shown in Figure 6.13 and Table 6.5. The uninstrumented SGX version performs significantly worse than the native version ($60-75\%$ throughput of native). This is due to the Memcached working set not fitting in the CPU cache; SGX spends some cycles on encrypting and decrypting data leaving the cache as well as checking its integrity. AddressSanitizer performs very close to SGX; even though it introduces additional memory accesses, the original memory latency is already high enough to hide this overhead. The performance of SGXBounds can be explained similarly. Finally, Intel MPX has an abysmal drop in throughput: MPX bounds tables consume so much memory that the working set exceeds the EPC and requires paging (we observed 100× more page faults than for SGXBounds).

For security evaluation, we reproduced a denial-of-service attack, CVE-2011-4971 vulnerability [151], in the SGX environment. All approaches—AddressSanitizer, Intel MPX, and SGXBounds—detected buffer overflow in the affected function's arguments. AddressSanitizer and Intel MPX halted the program, while SGXBounds with its boundless memory feature discarded the overflowed packet's content but went into an infinite loop due to a subsequent bug in the

[4] *lbm* and *namd* under AddressSanitizer perform better than the native version. This is due to changes in memory layout and similar to *dedup*; also see [173].

[5] For Low Fat Pointers, we took the same subset of 13 programs as in our evaluation and calculated the geomean. For Baggy Bounds, we resorted to specifying the reported mean over SPEC CPU2000.

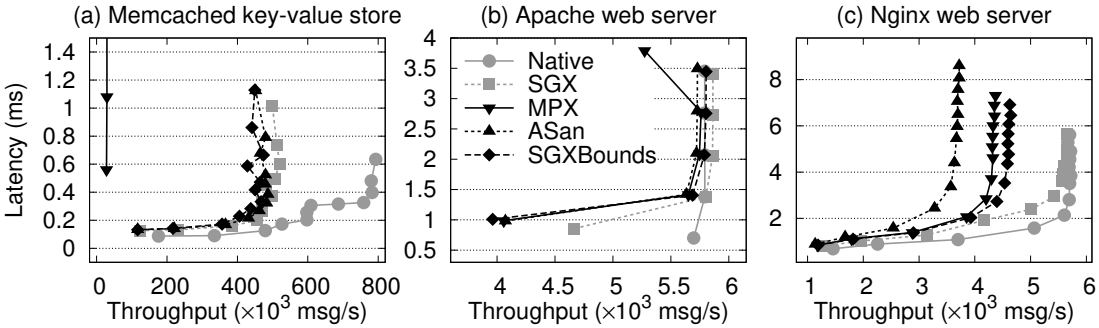

Figure 6.13 – Throughput-latency plots and peak memory usage of case studies: (a) Memcached, (b) Apache, and (c) Nginx.

	Memcached	Apache	Nginx
SGX	71.6	15.4	0.9
MPX	641	144	37.0
ASan	649	598	893
SGXBounds	71.8	23.2	1.0

Table 6.5 – Memory usage (MB) for peak throughput of case studies.

program's logic.

Apache. We evaluated Apache v2.4.18 [14] with OpenSSL v1.0.1f using the ab benchmark [3]. The performance results are plotted in Figure 6.13b; the memory usage is shown in Table 6.5. The SGX version of Apache performs slightly and consistently better than the native version. We attribute this to the SCONE features of user-level scheduling and asynchronous system calls [15]. Intel MPX quickly deteriorates with more clients; looking at the number of page faults, we conclude that this is due to the increasing overheads of bounds tables. (In Apache, each new client requires around 1MB of memory which bloats the bounds metadata for Intel MPX.) AddressSanitizer performs 2% worse than SGX, and SGXBounds—on par with SGX.

The unexpected 50% increase in memory use for SGXBounds in comparison to SGX is due to the custom memory allocator of Apache. It allocates only page-aligned amounts of memory, and the additional 4B of metadata forces our `mmap` wrapper to allocate a whole additional page.

To evaluate security, we looked at the infamous Heartbleed bug [12, 227]. AddressSanitizer, Intel MPX, and SGXBounds all detect Heartbleed attack. Additionally, SGXBounds does not crash the application, but—thanks to boundless memory—copies zeros into the reply message in accordance to the failure-oblivious computing policy. Thus, SGXBounds prevents confidential data leaks, at the same time allowing Apache to continue its execution.

Nginx. We evaluated Nginx v1.4.0 [164] using the ab benchmark. Figure 6.13c and Table 6.5 show performance and memory overheads. The $5-20\%$ difference in throughput between the native version and SGX is due to the overhead of copying the 200KB web page twice, first to the SCONE's syscall thread and then further to the socket. Note that this overhead was hidden by the overhead of thread synchronization in Apache (Apache uses 25 threads while Nginx is single-threaded).

AddressSanitizer performs the worst, achieving only $65-70\%$ throughput of that of SGX. In comparison to Apache, Intel MPX performs better than AddressSanitizer. The reason for this is a smarter memory management policy of Nginx, with as little memory copying as possible [164]. Because of this, Intel MPX does not spill bounds metadata as extensively as in Apache,

and gains better performance as a result. Finally, SGXBounds achieves $80-85\%$ throughput of SGX thanks to its efficient metadata scheme.

For security evaluation, the bug under test was a stack buffer overflow CVE-2013-2028 that can be used to launch a ROP attack [11]. All three approaches detect this bug. With SGXBounds boundless memory feature, Nginx can drop the offending request and continue its execution.

6.8 Discussion and Concluding Remarks

In this work, we presented SGXBounds—a memory-safety approach tailored to the specifics of Intel SGX. We conclude by discussing the limitations of our approach, future work, and peculiarities of SGX and MPX.

EPC Size. SGXBounds mandates the use of a limited 32-bit address space. This is in accordance with current SGX implementations which allow only 36-bit address space. SGXBounds could be refined to allow 36-bit pointers, hinged on the correct alignment of newly allocated objects (which is already provided by compilers and memory allocators).

It is possible that future SGX enclaves will have larger address spaces, decreasing the number of spare bits in pointers and negating the premise of SGXBounds. We believe enclaves spanning more than 4GB of memory are doubtful as they will suffer huge performance penalty. In addition, SGX is best suited for programs with small TCB and working sets.

Limitation of static linking. SGXBounds and the underlying SCONE infrastructure currently require the program to be statically linked. There is a decades-long debate on static vs dynamic linking [13, 195, 229, 232]. We strongly believe that dynamic linking is detrimental for security for a variety of reasons, including `LD_PRELOAD` issues, ldd and linker exploits. In addition, static linking enables powerful whole-program optimizations. Yet, SGXBounds could be used with dynamic libraries, though it would require additional wrapper functions for interoperability with them.

Catching intra-object overflows. SGXBounds keeps bounds for whole objects and therefore cannot detect intra-object overflows (similar to AddressSanitizer). Researchers currently explore the ability to catch such overflows using narrowing of bounds: whenever SGXBounds detects an access through a struct field, it updates the current pointer bounds to the bounds of this field. The main difficulty here is to keep additional lower-bound metadata for each object field; for this, we extend our metadata space and utilize metadata hooks.

Intel MPX. Considering that Intel MPX is a hardware extension, its low performance was surprising to us. Intel MPX performs well if the protected application works only with a small portion of pointers, but in the opposite case the overheads may get very high. To understand the underlying reasons of poor MPX performance, we conducted a more extensive and rigorous evaluation, results of which can be found in the next chapter.

7 Intel MPX Explained: Leveraging Memory Protection Extensions

In the previous chapter we noticed that Intel MPX exhibits high overheads in a restricted SGX environment. In particular, MPX showed performance overheads of up to $5-6\times$ in our experiments (§6.6.2). Even when we analyzed the performance of MPX in a normal, non-SGX environment, we were surprised to see overheads of $2.4\times$ on average, way higher than AddressSanitizer and SGXBounds (§6.6.7). Given that Intel MPX is a hardware-assisted technique, with all heavy bounds checking replaced by presumably fast CPU instructions, these performance numbers were underwhelming.

To understand the reasons behind such poor performance, this chapter analyzes Intel MPX in greater detail and discusses its applicability in comparison to other bounds-checking approaches. We identify the root causes for performance problems of MPX, ranging from contention on a single execution port while performing bounds checks to poor software-level support in GCC and ICC compilers. Even worse, we show how MPX can have false positives (false alarms) and false negatives (undetected bugs) in multithreaded programs. We conclude this chapter with lessons learned and a set of guidelines on the usage of this technique.

The content of this chapter is based on the paper "Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches" [173]. The paper was a joint collaboration with Oleksii Oleksenko, Pramod Bhatotia, Pascal Felber, and Christof Fetzer.

7.1 Rationale

The majority of systems software is written in low-level languages such as C or C++. These languages allow complete control over memory layout, which is especially important for systems development. Unfortunately, the ability to directly control memory often leads to violations of *memory safety*, i.e., illegal accesses to unintended memory regions [233].

In particular, memory-safety violations emerge in the form of *spatial* and *temporal* errors. Spatial errors—also called buffer overflows and out-of-bounds accesses—occur when a program reads from or writes to a different memory region than the one expected by the developer. Temporal errors—wild and dangling pointers—appear when trying to use an object before it was created or after it was deleted.

These memory-safety violations may result in sudden crashes, data losses, and other nasty bugs [233]. Moreover, these vulnerabilities can also be exploited to build a *memory attack*—a scenario when an adversary gets access to an illegal region of memory and can hi-jack the system or steal confidential data. This attack vector is prevailing among low-level languages, with almost 1,200 memory vulnerabilities published only in 2016 according to the US National Vulnerability Database [160].

Given the importance of the problem, there are numerous solutions for enforcing memory safety in unsafe languages, ranging from static analysis to language extensions [6, 27, 64, 114, 128, 131, 152, 158, 161, 162, 166, 207, 246]. In this work, we concentrate on *deterministic dynamic*

bounds-checking since it is widely regarded as the only way of defending against *all* memory attacks [155, 225]. Bounds-checking techniques augment the original unmodified program with metadata (bounds of live objects or allowed memory regions) and insert checks against this metadata before each memory access. Whenever a bounds check fails, the program is aborted and thus the attack is prevented. Unfortunately, state-of-the-art bounds-checking techniques exhibit high performance overhead (50–150%) which limits their usage to development stages only.

To lower runtime overheads, Intel recently released a new ISA extension—Memory Protection Extensions (Intel MPX). Its underlying idea is to provide hardware assistance, in the form of new instructions and registers, to software-based bounds checking, making it more efficient.

Yet, to our knowledge, there is no comprehensive evaluation of Intel MPX, neither from the academic community nor from Intel itself. Therefore, the goal of this work was to analyze Intel MPX in three dimensions: performance, security, and usability. *Performance* is important because only solutions with low (up to 10–20%) runtime overhead have a chance to be adopted in practice [225]. It was also crucial to investigate the root causes of the overheads to pave the way for future improvements. *Security* assessment on a set of real-world vulnerabilities was required to verify advertised security guarantees. *Usability* evaluation gave us insights on Intel MPX production quality and—more importantly—on application-specific issues that arise under Intel MPX and need to be manually fixed.

To fully explore Intel MPX's pros and cons, we put the results into perspective by comparing with existing software-based solutions. In particular, we compared Intel MPX with three prominent techniques that showcase main classes of memory safety: trip-wire Address Sanitizer [207], object-based SAFECode [64], and pointer-based SoftBound [158] (see §7.2 for details).

Our investigation reveals that Intel MPX has high potential, but is not yet ready for widespread use. Some of the lessons we learned are:

- New Intel MPX instructions are not as fast as expected and cause up to 4× slowdown in the worst case, although compiler optimizations amortize it and lead to runtime overheads of ~50% on average.
- The supporting infrastructure (compiler passes and runtime libraries) is not mature enough and has bugs, such that 3–10% programs cannot compile/run.
- In contrast to other solutions, Intel MPX provides no protection against temporal errors.
- Intel MPX may have false positives and false negatives in multithreaded code.
- By default, Intel MPX imposes restrictions on allowed memory layout, such that 8–13% programs do not run correctly without substantial code changes. In addition, we had to apply (non-intrusive) manual fixes to 18% programs.

Though the first three issues can be fixed in future versions, the last two can be considered fundamental design limits. We project that adding support for multithreading would inevitably hamper performance, and relaxing restrictions on memory layout would go against Intel MPX philosophy.

7.2 Background

All spatial and temporal bugs, as well as memory attacks built on such vulnerabilities, are caused by an access to a prohibited memory region. To prevent such bugs, *memory safety* must be imposed on the program, i.e., the following invariant must be enforced: memory accesses must always stay within the originally intended (referent) objects.

Memory safety can be achieved by various methods, including pure static analysis [66, 246],

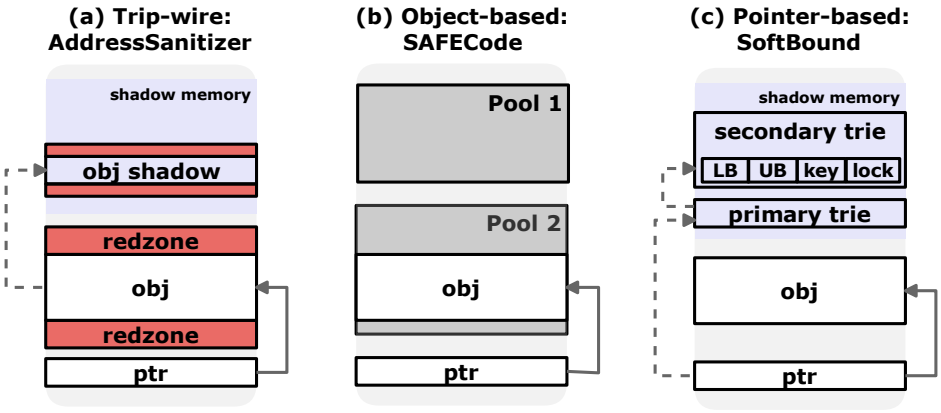

Figure 7.1 – Designs of three memory-safety classes: trip-wire (AddressSanitizer), object-based (SAFECode), and pointer-based (SoftBound).

hardware-based checking [131, 156, 234, 245], probabilistic methods [27, 143, 167], and extensions of the C/C++ languages [116, 152, 161]. In this work, we concentrate on deterministic runtime bounds-checking techniques that transparently instrument legacy programs (Intel MPX is but one of them). These techniques provide the highest security guarantees while requiring little to no manual effort to adapt the program. For a broader discussion, please refer to [225].

Existing runtime techniques can be broadly classified as trip-wire, object-based, and pointer-based [155]. In a nutshell, all three classes create, track, and check against some bounds metadata kept alongside original data of the program. Trip-wire approaches create "shadow memory" metadata for the whole available program memory, pointer-based approaches create bounds metadata per each pointer, and object-based approaches create bounds metadata per each object.

For comparison with Intel MPX, we chose a prominent example from each of the aforementioned classes: AddressSanitizer, SAFECode, and SoftBound. Figure 7.1 highlights the differences between them.

Trip-wire approach: AddressSanitizer [207]. This class surrounds all objects with regions of marked (poisoned) memory called *redzones*, so that any overflow will change values in this—otherwise invariable—region and will be consequently detected. In particular, AddressSanitizer reserves 1/8 of all virtual memory for the *shadow memory* which is accessed only by the instrumentation and not the original program. AddressSanitizer updates data in shadow memory whenever a new object is created and freed, and inserts checks on shadow memory before memory accesses to objects. The check itself looks like this:

```
shadowAddr = MemToShadow(ptr)
if (ShadowIsPoisoned(shadowAddr))
    ReportError()
```

In addition, AddressSanitizer provides means to detect temporal errors via a *quarantine zone*: if a memory region has been freed, it is kept in the quarantine for some time before it becomes allowed for reuse.

AddressSanitizer was built for debugging purposes and is not targeted for security. It is sometimes used in in this context for lack of a better alternatives [34, 155]) but such use is discouraged [236] (e.g., because attackers may abuse the debugging features in AddressSanitizer's run-time library). For example, it may not detect non-contiguous out-of-bounds violations. Nevertheless, it detects many spatial bugs and significantly raises the bar for the attacker. It is also the most widely-used technique in its class, comparing favorably to other trip-wire techniques such as Light-weight Bounds Checking [166], Purify [96], and Valgrind [162].

Object-based approach: SAFECode [63, 64]. This class's main idea is enforcing the intended referent, i.e., making sure that pointer manipulations do not change the pointer's referent object. In SAFECode, this rule is relaxed: each object is allocated in one of several fine-grained partitions—*pools*—determined at compile-time using pointer analysis; the pointer must always land into the predefined pool. This technique allows powerful optimizations and simple runtime checks against the pool bounds:

```
poolAddr = MaskLowBits(ptr)
if (poolAddr not in predefinedPoolAddrs)
    ReportError()
```

On the downside, SAFECode provides worse guarantees than AddressSanitizer—buffer overflow to an object in the same pool will go undetected.

We also inspected and discarded other object-based approaches. CRED [196] has huge performance overheads, mudflap [71] is deprecated in newer versions of GCC, and Baggy Bounds Checking [6] is not open sourced.

Pointer-based approach: SoftBound [157, 158]. Such approaches keep track of pointer bounds (the lowest and the highest address the pointer is allowed to access) and check each memory write and read against them. Note how SoftBound associates metadata *not* with an object but rather with a pointer to the object. This allows pointer-based techniques to detect intra-object overflows (one field overflowing into another field of the same struct) by *narrowing bounds* associated with the particular pointer.

Intel MPX closely resembles SoftBound; indeed, a hardware-assisted enhancement of SoftBound called WatchdogLite shares many similarities with Intel MPX [156]. For our comparison, we used the SoftBound+CETS version which keeps pointer metadata in a two-level trie—similar to MPX's bounds tables—and introduces a scheme to protect against temporal errors [157]. The checks in this version are as follows:

```
LoBound, UpBound, key, lock = TrieLookup(ptr)
if (ptr < LoBound or ptr > UpBound or key != *lock)
    ReportError()
```

As for other pointer-based approaches, MemSafe [217] is not open sourced, and CCured [161] and Cyclone [116] require manual changes in programs.

7.3 Intel Memory Protection Extensions

Intel Memory Protection Extensions (Intel MPX) was first announced in 2013 [109] and introduced as part of the Skylake microarchitecture in late 2015 [106]. The sole purpose of Intel MPX is to transparently add bounds checking to legacy C/C++ programs. Consider a code snippet in Figure 7.2a. The original program allocates an array `a[10]` with 10 pointers to some buffer objects of type `obj` (Line 1). Next, it iterates through the first `M` items of the array to calculate the sum of objects' length values (Lines 3–8). In C, this loop would look like this:

```
for (i=0; i<M; i++) total += a[i]->len;
```

Since M is a variable, a bug or a malicious activity may set M to a value that is larger than `obj` size and an overflow will happen. Also, note how the array item access `a[i]` decays into a pointer `ai` on Line 4, and how the subfield access decays to `lenptr` on Line 6.

Figure 7.2b shows the resulting code with Intel MPX protection applied. First, the bounds for the array `a[10]` are created on Line 3 (the array contains 10 pointers each 8 bytes wide, hence the upper-bound offset of 79). Then in the loop, before the array item access on Line 8, two MPX bounds checks are inserted to detect if `a[i]` overflows (Lines 6–7). Note that since the

(a) Original code

```
   struct obj { char buf[100]; int len }
1  obj* a[10]                                          ;; Array of pointers to objs
2  total = 0
3  for (i=0; i<M; i++):                                 ;; M may be greater than 10
4      ai = a + i                                        ;; Pointer arithmetic on a
5      objptr = load ai                                   ;; Pointer to obj at a[i]
6      lenptr = objptr + 100                                  ;; Pointer to obj.len
7      len = load lenptr
8      total += len                                      ;; Total length of all objs
```

(b) Intel MPX

```
1  obj* a[10]
2  total = 0
3  a_b = bndmk a, a+79                                    ;; Make bounds [a, a+79]
4  for (i=0; i<M; i++):
5      ai = a + i
6      bndcl a_b, ai                                   ;; Lower-bound check of a[i]
7      bndcu a_b, ai+7                                 ;; Upper-bound check of a[i]
8      objptr = load ai
9      objptr_b = bndldx ai                           ;; Bounds for pointer at a[i]
10     lenptr = objptr + 100
11     bndcl objptr_b, lenptr                             ;; Checks of obj.len ⎤
12     bndcu objptr_b, lenptr+3                                                ⎦
13     len = load lenptr
14     total += len
```

Figure 7.2 – Example of bounds checking using Intel MPX.

protected load reads an 8-byte pointer from memory, it is important to check `ai+7` against the upper bound (Line 7).

Now that the pointer to the object is loaded in `objptr`, the program wants to load the `obj.len` subfield. By design, Intel MPX must protect this second load by checking the bounds of the `objptr` pointer. Where does it get these bounds from? In Intel MPX, every pointer stored in memory has its associated bounds also stored in a special memory region accessed via `bndstx` and `bndldx` MPX instructions (see next subsection for details). Thus, when the `objptr` pointer is retrieved from memory address `ai`, its corresponding bounds are retrieved using `bndldx` from the same address (Line 9). Finally, the two bounds checks are inserted before the load of the length value on Lines 11–12[1].

Intel MPX requires modifications at each level of the hardware-software stack[2]:

- At the *hardware level*, new instructions as well as a set of 128-bit registers are added. Also, a bounds violation exception (`#BR`) thrown by these new instructions is introduced.
- At the *OS level*, a new `#BR` exception handler is added that has two main functions: (1) allocating storage for bounds on-demand and (2) sending a signal to the program whenever a bounds violation is detected.
- At the *compiler level*, new Intel MPX transformation passes are added to insert MPX instructions to create, propagate, store, and check bounds. Additional *runtime libraries* provide initialization/finalization routines, statistics and debug info, and wrappers for functions from C standard library.
- At the *application level*, the MPX-protected program may require manual changes due to

[1] Note that narrowing of bounds is not shown for simplicity, see §7.3.3.
[2] Henceforth, we focus on 64-bit Linux-based support of Intel MPX.

unconventional C coding patterns, multithreading issues, or potential problems with other ISA extensions. (In some cases, it is inadvisable to use Intel MPX at all.)

In the following, we detail how Intel MPX support is implemented at each level of the hardware-software stack.

7.3.1 Hardware

At its core, Intel MPX provides 7 new instructions and a set of 128-bit bounds registers. The current Intel Skylake architecture provides four registers named `bnd0-bnd3`. Each of them stores a lower 64-bit bound in bits 0–63 and an upper 64-bit bound in bits 64–127.

Instruction set. The new MPX instructions are: `bndmk` to create new bounds, `bndcl` and `bndcu`/`bndcn` to compare the pointer value against the lower and upper bounds in `bnd` respectively, `bndmov` to move bounds from one `bnd` register to another and to spill them to stack, and `bndldx` and `bndstx` to load and store pointer bounds in special Bounds Tables respectively. Note that `bndcu` has a one's complement version `bndcn` which has exactly the same characteristics, thus we mention only `bndcu` in the following. The example in Figure 7.2b shows how most of these instructions are used. The instruction not shown is `bndmov` which serves mainly for internal rearrangements in registers and on stack.

Intel MPX additionally changes the x86-64 calling convention. In a nutshell, the bounds for corresponding pointer arguments are put in registers `bnd0-bnd3` before a function call and the bounds for the pointer return value are put in `bnd0` before return from the function.

It is interesting to compare the benefits of hardware implementation of bounds-checking against the software-only counterpart—SoftBound in our case [157, 158]. First, Intel MPX introduces separate bounds registers to lower register pressure on the general-purpose register (GPR) file, something that software-only approaches suffer from. Second, software-based approaches cannot modify the calling convention and resort to function cloning, when a set of function arguments is extended to include pointer bounds. This leads to more cumbersome caller/callee code and problems with interoperability with legacy uninstrumented libraries. Finally, dedicated `bndcl` and `bndcu` instructions substitute the software-based "compare and branch" instruction sequence, saving one cycle and exerting no pressure on branch predictor.

The prominent feature of Intel MPX is its backwards-compatibility and interoperability with legacy code. On the one hand, MPX-instrumented code can run on legacy hardware because Intel MPX instructions are interpreted as NOPs on older architectures. This is done to ease the distribution of binaries—the same MPX-enabled program/library can be distributed to all clients. On the other hand, Intel MPX has a comprehensive support to interoperate with unmodified legacy code: (1) a `BNDPRESERVE` configuration bit allows to pass pointers without bounds information created by legacy code, and (2) when legacy code changes a pointer in memory, the later `bndldx` of this pointer notices the change and assigns always-true (`INIT`) bounds to it. In both cases, the pointer created/altered in legacy code is considered "boundless": this allows for interoperability but also creates holes in Intel MPX defense[3] [5].

Storing bounds in memory. The current version of Intel MPX has only 4 bounds registers, which is clearly not enough for real-world programs—we will run out of registers even if we have only 5 distinct pointers. Accordingly, all additional bounds have to be stored (spilled) in memory,

[3] *x264* from PARSEC highlights this issue: its `x264_malloc` function internally calls `memalign` which has no corresponding wrapper. Thus, the pointer returned by this function is "boundless". Since all dynamic objects are created through this function, the whole program operates on "boundless" pointers, rendering Intel MPX protection utterly useless.

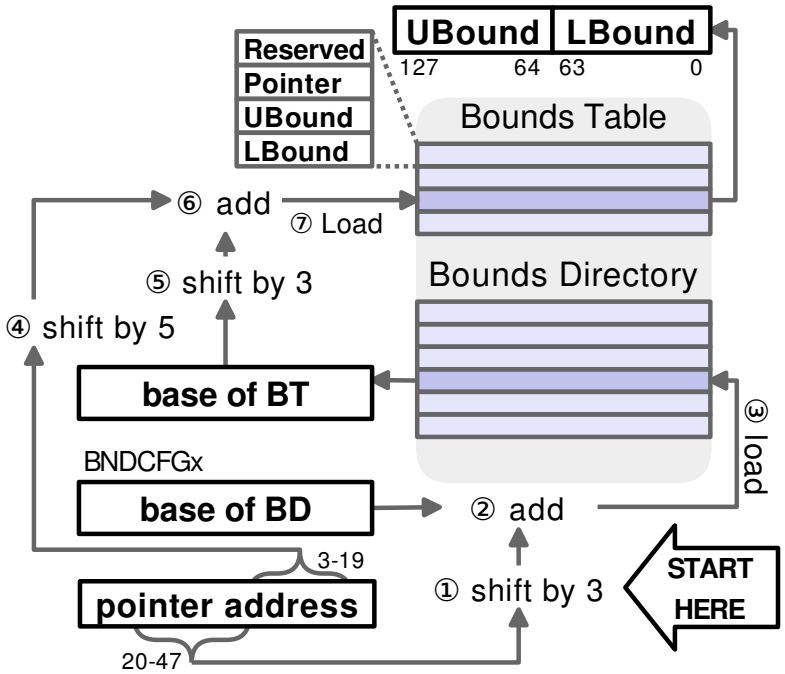

Figure 7.3 – Loading of pointer bounds using two-level address translation.

similar to spilling data out of general-purpose registers. A simple and relatively fast option is to copy them directly into a compiler-defined memory location (on stack) with `bndmov`. However, it works only inside a single stack frame: if a pointer is later reused in another function, its bounds will be lost. To solve this issue, two instructions were introduced—`bndstx` and `bndldx`. They store/load bounds to/from a memory location derived from the address of the pointer itself (see Figure 7.2b, Line 9), thus making it easy to find pointer bounds without any additional information, though at a price of higher complexity.

When `bndstx` and `bndldx` are used, bounds are stored in a memory location calculated with two-level address translation scheme, similar to virtual address translation (paging). In particular, each pointer has an entry in a Bounds Table (BT), which is allocated dynamically and is comparable to a page table. Addresses of BTs are stored in a Bounds Directory (BD), which corresponds to a page directory in our analogy. For a specific pointer, its entries in the BD and the BT are derived from the memory address in which the pointer is stored.

Note that our comparison to paging is only conceptual; the implementation side differs significantly. Firstly, the MMU is not involved in the translation and all operations are performed by the CPU itself. Secondly and most importantly, Intel MPX does not have a dedicated cache (such as a TLB cache), thus it has to share normal caches with application data. In some cases, it may lead to severe performance degradation caused by cache thrashing.

The address translation is a multistage process. Consider loading of pointer bounds (Figure 7.3). In the first stage, the corresponding BD entry has to be loaded. For that, the CPU: ① extracts the offset of BD entry from bits 20–47 of the pointer address and shifts it by 3 bits (since all BD entries are 2^3 bits long), ② loads the base address of BD from the `BNDCFGx`[4] register, and ③ sums the base and the offset and loads the BD entry from the resulting address.

In the second stage, the CPU: ④ extracts the offset of BT entry from bits 3–19 of the pointer address and shifts it by 5 bits (since all BT entries are 2^5 bits long), ⑤ shifts the loaded entry—

[4]In particular, `BNDCFGU` in user space and `BNDCFGS` in kernel mode.

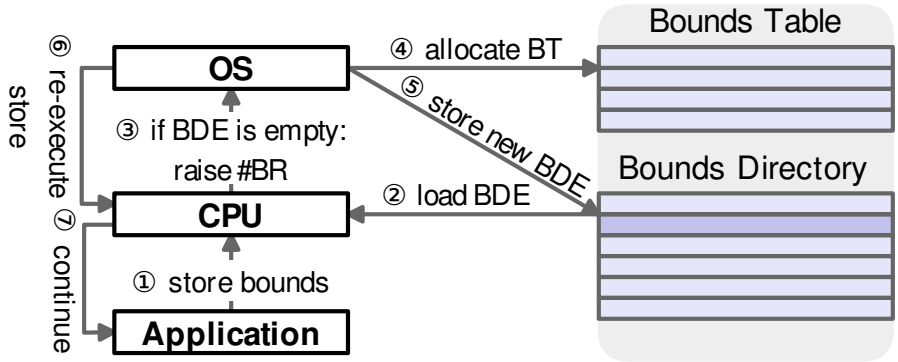

Figure 7.4 – The procedure of Bounds Table creation.

which corresponds to the base of BT—by 3 to remove the metadata contained in the first 3 bits, and ⑥ sums the base and the offset and ⑦ finally loads the BT entry from the resulting address. Note that a BT entry has an additional "pointer" field—if the actual pointer value and the value in this field mismatch, Intel MPX will mark the bounds as always-true (`INIT`). This is required for interoperability with legacy code and only happens when this code modifies the pointer.

This operation is expensive—it requires approximately 3 register-to-register moves, 3 shifts, and 2 memory loads. On top of it, since these 2 loads are non-contiguous, the protected application has worse cache locality.

Interaction with other ISA extensions. Intel MPX can cause issues when used together with other ISA extensions, e.g., Intel TSX and Intel SGX. Intel MPX may cause transactional aborts in some corner cases when used inside an Intel TSX hardware transaction (see [108] for the details). Also, since Bounds Tables and `#BR` exceptions are managed by the OS, Intel MPX cannot be used as-is in an Intel SGX enclave environment. Indeed, the malicious OS could tamper with these structures and subvert correct MPX execution. To prevent such scenarios, Intel MPX allows to move this functionality into the SGX enclave and verify every OS action [128]. Finally, we are not aware of any side-channel attacks that could utilize Intel MPX inside the enclave.

Microbenchmark. As a first step in our evaluation, we analyzed latency and throughput of MPX instructions. For this, we extended the scripts used to build Agner Fog's instruction tables—a de-facto standard for evaluating CPU instructions [79]. For each run, we initialize all `bnd` registers with dummy values to avoid interrupts caused by failed bound checks.

Table 7.1 shows the latency-throughput results, and Figure 7.5 depicts which execution ports can MPX instructions use. As expected, most operations have latencies of one cycle, e.g., the most frequently used `bndcl` and `bndcu` instructions. The serious bottleneck is storing/loading the bounds with `bndstx` and `bndldx` since they undergo a complex algorithm of accessing bounds tables, as explained in the previous section.

In our experiments, we observed that Intel MPX protection does not increase the IPC (instructions/cycle) of programs, which is usually the case for memory-safety techniques (see Figure 7.11). This was surprising: we expected that Intel MPX would take advantage of underutilized CPU resources for programs with low original IPC. To understand what causes this bottleneck, we measured the throughput of typical MPX check sequences. (We originally blamed an unjustified data dependency between `bndcl/u` and the protected memory access but this proved wrong.)

Our measurements pointed to a bottleneck of `bndcl/u b,m` instructions due to contention on port 1. Without checks (Figure 7.6a), our original benchmark could execute two loads in parallel,

Instruction	Description	Lat	Tput
`bndmk b,m`	create pointer bounds	1	2
`bndcl b,m`	check mem-operand addr against lower	1	1
`bndcl b,r`	check reg-operand addr against lower	1	2
`bndcu b,m`	check mem-operand addr against upper	1	1
`bndcu b,r`	check reg-operand addr against upper	1	2
`bndmov b,m`	move pointer bounds from mem	1	1
`bndmov b,b`	move pointer bounds to other reg	1	2
`bndmov m,b`	move pointer bounds to mem	2	0.5
`bndldx b,m`	load pointer bounds from BT	4-6	0.4
`bndstx m,b`	store pointer bounds in BT	4-6	0.3

Note: `bndcu` has a one's complement version `bndcn`, we skip it for clarity

Table 7.1 – Latency (cycles/instr) and Tput (instr/cycle) of Intel MPX instructions; `b`—MPX bounds register; `m`—memory operand; `r`—general-purpose register operand.

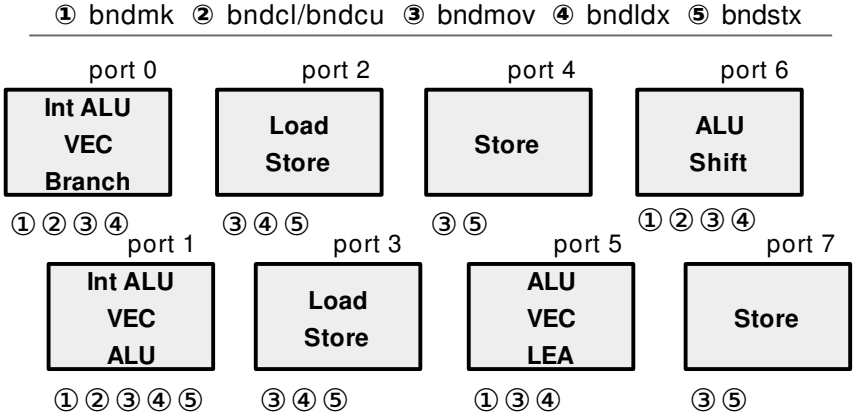

Figure 7.5 – Distribution of Intel MPX instructions among execution ports (Intel Skylake).

achieving a throughput of 2 IPC (note that the loaded data is always in a Memory Ordering Buffer). After adding `bndcl/u b,r` checks (Figure 7.6b), IPC increased to three instructions per cycle (3 IPC): one load, one lower-, and one upper-bound check per cycle. For "bndcl/u b,m" checks (Figure 7.6c), however, IPC became *less* than original: two loads and four checks were scheduled in four cycles, thus IPC of 1.5. In summary, the final IPC was ~1.5–3 (compare to original IPC of 2), proving that the MPX-protected program typically has *approximately the same IPC as the original*.

As Figures 7.9 and 7.10 show, it causes major performance degradation. It can be fixed, however; if the next generations of CPUs will provide the relative memory address calculation on other ports, the checks could be parallelized and performance will improve. We speculate that GCC-MPX could perform on par with AddressSanitizer in this case, because the instruction overheads are similar. Accordingly, ICC version would be even better and the slowdowns might drop lower than 20%. But we must note that we do not have any hard proof for this speculation.

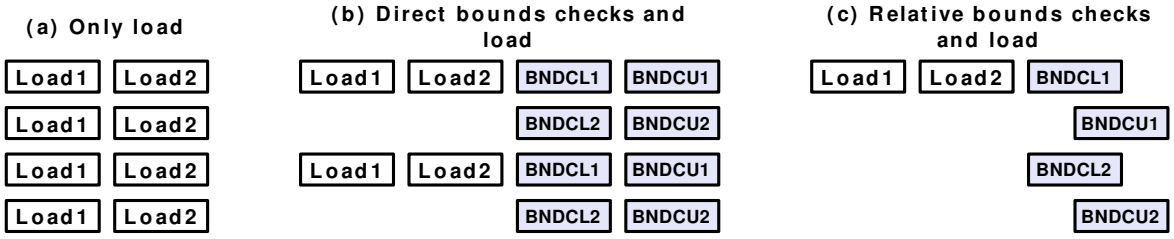

Figure 7.6 – Bottleneck of bounds checking illustrated: since relative memory addresses can be calculated only by port 1, a contention appears and bounds checks are executed sequentially.

7.3.2 Operating System

The operating system has two main responsibilities in the context of Intel MPX: it handles bounds violations and manages BTs, i.e., creates and deletes them. Both these actions are hooked to a new class of exceptions, `#BR`, which has been introduced solely for Intel MPX and is similar to a page fault, although with extended functionality.

Bounds exception handling. If an MPX-enabled CPU detects a bounds violation, i.e., if a referenced pointer appears to be outside of the checked bounds, `#BR` is raised and the processor traps into the kernel (in case of Linux). The kernel decodes the instruction to get the violating address and the violated bounds, and stores them in the `siginfo` structure. Afterwards, it delivers the SIGSEGV signal to the application together with information about the violation in `siginfo`. At this point the application developer has a choice: she can either provide an ad-hoc signal handler to recover or choose one of the default policies: crash, print an error and continue, or silently ignore it.

Bounds tables management. Two levels of bounds address translation are managed differently: BD is allocated only once by a runtime library (at application startup) and BTs have to be created dynamically on-demand. The later is a task of OS. The procedure is presented in Figure 7.4. Each time an application tries to store pointer bounds ①, the CPU loads the corresponding entry from the BD and checks if it is a valid entry ②. If the check fails, the CPU raises `#BR` and traps into the kernel ③. The kernel allocates a new BT ④, stores its address in the BD entry ⑤ and returns in the user space ⑥. Then, the CPU stores bounds in the newly created BT and continues executing the application in the normal mode of operation ⑦.

Since the application is oblivious of BT allocation, the OS also has to free these tables. In Linux, this "garbage collection" is performed whenever a memory object is freed or, more precisely, unmapped. OS goes through the elements of the object and removes all the corresponding BT entries. If one of the tables becomes completely unused, OS will free the BT and remove its entry in the BD.

Microbenchmark. To illustrate the additional overhead of allocating and de-allocating BTs, we manufactured two microbenchmarks that showcase the worst case scenarios. The first one stores a large set of pointers in such memory locations that each of them will have a separate BT, i.e., this benchmark indirectly creates a large number of bounds tables. The second one does the same, but in addition, it frees all the memory right after it has been assigned, thus triggering BT de-allocation. Our measurement results are shown in Table 7.2 (note that we disabled all compiler optimizations to showcase the influence of OS alone). In both cases, most of the runtime parameters (cache locality, branch misses, etc.) of the MPX-protected version are equivalent to the native one. However, the slowdown is noticeable—more than 2 times. It is caused by a single parameter that varies—the number of instructions executed in the kernel

Type	Slowdown	Increase in # of instructions (%)	
		User space	Kernel space
allocation	2.33×	7.5	160
+ de-allocation	2.25×	10	139

Table 7.2 – Worst-case OS impact on performance of MPX.

space. It means that the overhead is caused purely by the BT management in the kernel. From this, we can conclude that OS itself can make an MPX-protected application up to 2.3× slower, although this scenario is quite rare.

In this section, we discussed only Linux implementation. However, all the same mechanisms can also be found in Windows. The only significant difference is that Intel MPX support on Windows is done by a daemon, while on Linux the functionality is implemented in the kernel itself [113].

7.3.3 Compiler and Runtime Library

Hardware Intel MPX support in the form of new instructions and registers significantly lowers performance overhead of each *separate* bounds-checking operation. However, the main burden of efficient, correct, and complete bounds checking of whole programs lies on the compiler and its associated runtime.

Compiler support. As of the date of this writing, only GCC 5.0+ and ICC 15.0+ compilers have support for Intel MPX [72, 113]. To enable Intel MPX protection of applications, both GCC and ICC introduce the new compiler pass called Pointer(s) Checker. Enabling Intel MPX is intentionally as simple as adding a couple of flags to the usual compilation process:

```
>> gcc -fcheck-pointer-bounds -mmpx test.c
>> icc -check-pointers-mpx=rw test.c
```

In a glance, the Pointer Checker pass instruments the original program as follows. (1) It allocates static bounds for global variables and inserts `bndmk` instructions for stack-allocated ones. (2) It inserts `bndcl` and `bndcu` bounds-check instructions before each load or store from a pointer. (3) It moves bounds from one `bnd` register to another using `bndmov` whenever a new pointer is created from an old one. (4) It spills least used bounds to stack via `bndmov` if running out of available `bnd` registers. (5) It loads and stores the associated bounds via `bndldx` and `bndstx` respectively whenever a pointer is loaded/stored from/to memory.

One of the advantages of Intel MPX—in comparison to AddressSanitizer and SAFECode—is that it supports *narrowing of struct bounds* by design. Consider struct `obj` from Figure 7.2. It contains two fields: a 100B buffer `buf` and an integer `len` right after it. It is easy to see that an off-by-one overflow in `obj.buf` will spillover and corrupt the adjacent `obj.len`. AddressSanitizer and SAFECode by design cannot detect such intra-object overflows (though AddressSanitizer can be used to detect a subset of such errors [183]). In contrast, Intel MPX can be instructed to narrow bounds when code accesses a specific field of a struct, e.g., on Line 10 in Figure 7.2b. Here, instead of checking against the bounds of the full object, the compiler would shrink `objptr_b` to only four bytes and compare against these narrowed bounds on Lines 11–12. Narrowing of bounds may require (sometimes intrusive) changes in the source code, and is enabled by default.

By default, the MPX pass instruments both memory writes and reads: this ensures protection from buffer overwrites and buffer overreads. The user can instruct the MPX pass to instrument only writes. The motivation is twofold. First, instrumenting only writes significantly reduces

Compiler & runtime issues	GCC	ICC
– Poor MPX pass optimizations *	22/38	3/38
– Bugs in MPX compiler pass:		
– incorrect bounds during function calls	–	2/38
– conflicts with auto-vectorization passes	–	3/38
– corrupted stack due to C99 VLA arrays	–	3/38
– unknown internal compiler error	1/38	–
– Bugs and issues in runtime libraries:		
– Missing wrappers for libc functions	all	all
– Nullified bounds in `memcpy` wrapper	all	–
– Performance bug in `memcpy` wrapper	–	all

*One compiler has > 10% worse results than the other

Table 7.3 – Issues in the compiler pass and runtime libraries of Intel MPX. Columns 2 and 3 show number of affected programs (out of total 38).[5]

performance overhead of Intel MPX (from 2.5× to 1.3× for GCC). Second, the most dangerous bugs are those that overwrite memory (classic overflows to gain privileged access to the remote machine), and the only-writes protection can already provide sufficiently high security guarantees.

At least in GCC implementation, the pass can be fine-tuned via additional compilation flags. In our experience, these flags provide no additional benefit in terms of performance, security, or usability. For a full list of supported flags, refer to the official documentation of Intel MPX [113].

For performance, compilers must try their best to optimize away redundant MPX code. There are two common optimizations used by GCC and ICC (also used, for example, in Baggy Bounds [6]). (1) Removing bounds-checks when the compiler can statically prove safety of memory access, e.g., access inside an array with a known offset. (2) Moving (hoisting) bounds-checks out of simple loops. Consider Figure 7.2b. If it is known that `M<=10`, then optimization (1) can remove always-true checks on Lines 6–7. Otherwise, optimization (2) can kick in and move these checks before the loop body, saving two instructions on each iteration.

Runtime library. As a final step of the MPX-enabled build process, the application must be linked against two MPX-specific libraries: `libmpx` and `libmpxwrappers` (`libchkp` for ICC).

The `libmpx` library is responsible for MPX initialization at program startup: it enables hardware and OS support and configures MPX runtime options (passed through environment variables). Most of these options concern debugging and logging, but two of them define security guarantees. First, `CHKP_RT_MODE` must be set to "stop" in production use to stop the program immediately when a bounds violation is detected; set it to "count" only for debugging purposes. Second, `CHKP_RT_BNDPRESERVE` defines whether application can call legacy, uninstrumented functions in external libraries; it must be enabled if the whole program is MPX-protected.

By default, `libmpx` registers a signal handler that either halts execution or writes a debug message (depending on runtime options). However, this default handler can be overwritten by the user's custom handler. This can be useful if the program must shutdown gracefully or checkpoint its state.

Another interesting feature is that the user can instruct `libmpx` to disallow creation of BTs by the OS (see §7.3.2). In this case, the `#BR` exception will be forwarded directly to the program which can allocate BTs itself. One scenario where this can come handy is when the user completely

[5]All bugs were acknowledged by developers.

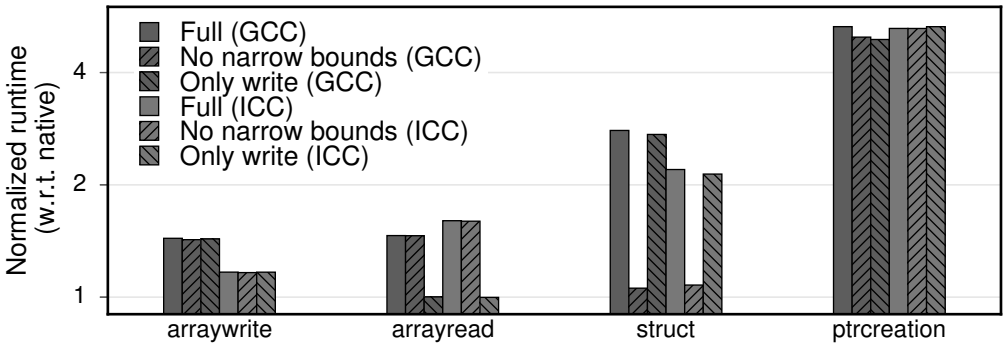

Figure 7.7 – Intel MPX overheads in 3 possible scenarios: application is dominated by bounds-checking (*arraywrite* and *arrayread*), by bounds creation and narrowing (*struct*), and by bounds propagation (*ptrcreation*).

distrusts the OS, e.g., when using SGX enclaves [128].

The `libmpxwrappers` library in GCC (and its analogue `libchkp` in ICC) contain wrappers for functions from C standard library (libc). Similar to AddressSanitizer, Intel MPX implementations do not instrument libc and instead wrap all its functions with a bounds-checking counterparts.

Issues. For both GCC and ICC, the compiler and runtime support have a number of issues summarized in Table 7.3.

Concerning performance, current implementations of GCC and ICC take different stances when it comes to optimizing MPX code. GCC is conservative and prefers stability of original programs over performance gains. On many occasions, we noticed that the GCC MPX pass *disables* other optimizations, e.g., loop unrolling and autovectorization. It also hoists bounds-checks out of loops less often than ICC does. ICC, on the other hand, is more aggressive in its MPX-related optimizations and does *not* prevent other aggressive optimizations from being applied. Unfortunately, this intrusive behavior renders ICC's pass less stable: we detected three kinds of compiler bugs due to incorrect optimizations.

We also observed issues with the runtime wrapper libraries. First, only a handful of most widely-used libc functions are covered, e.g., `malloc`, `memcpy`, `strlen`, etc. This leads to undetected bugs when other functions are called, e.g., the bug with `recv` in §7.5.2. For use in production, these libraries must be expanded to cover *all* of libc. Second, while most wrappers follow a simple pattern of "check bounds and call real function", there exist more complicated cases. For example, `memcpy` must be implemented so that it copies not only the contents of one memory area to another, but also all associated pointer bounds in BTs. GCC library uses a fast algorithm to achieve this, but ICC's `libchkp` has a performance bottleneck (see also §7.4).

Microbenchmarks. To understand the impact of different compiler flags and optimizations, we wrote four microbenchmarks, each highlighting a separate MPX feature. Two benchmarks—*arraywrite* and *arrayread*—perform writes to/reads from memory and stress `bndcl` and `bndcu` accordingly. The *struct* benchmark writes in an inner array inside a struct and stresses the bounds-narrowing feature via `bndmk` and `bndmov`. Finally, the *ptrcreation* benchmark constantly assigns new values to pointers and stresses bounds propagation via `bndstx`. Figure 7.7 shows the performance overheads over native versions.

We can notice several interesting details. First, *arraywrite* and *arrayread* represent bare overhead of bounds-checking instructions (all in registers), 50% in this case. *struct* has a higher overhead of $2.1 - 2.8\times$ due to the more expensive making and moving of bounds to and from the stack. The $5\times$ overhead of *ptrcreation* is due to storing of bounds—the most expensive MPX

Application-level issues	GCC	ICC
– Flexible or variable-sized array (`arr[1]` / `arr[]`)	7/38	7/38
– Accessing struct through struct field *	1/38	3/38
– Custom memory management	2/38	2/38

* GCC affects less programs due to milder rules w.r.t. first field of struct

Table 7.4 – Applications may violate memory-model assumptions of Intel MPX. Columns 2 and 3 show number of misbehaving programs (out of total 38).

operation (see §7.3.1). Such high overhead is alarming because pointer-intensive applications require many loads and stores of bounds.

Second, there is a 25% difference between GCC and ICC in *arraywrite*. This is the effect of optimizations: GCC's MPX pass blocks loop unrolling while ICC's implementation takes advantage of it. (Interestingly, the same happened in case of *arrayread* but the native ICC version was optimized even better, which led to a relatively poor performance of ICC's MPX.)

Third, the overhead of *arrayread* becomes negligible with the only-writes MPX version: the only memory accesses in this benchmark are reads which are left uninstrumented. Finally, the same logic applies to *struct*—disabling narrowing of bounds effectively removes expensive `bndmk` and `bndmov` instructions and lowers performance overhead to a bare minimum.

7.3.4 Application

At the application level, we observed two main issues of Intel MPX. First, Intel MPX cannot support several widely-used C programming idioms (some by design, some due to implementation choices) and thus can break programs. Second and more importantly, there is no support for multithreaded programs.

Not supported C idioms. As discussed previously, one of the main features of Intel MPX—narrowing of bounds—can increase security because the code that explicitly works with one field of a complex object will not be able corrupt other fields. Unfortunately, our evaluation reveals that narrowing of bounds breaks many programs (see Table 7.4). The general problem is that C/C++ programs frequently deviate from the standard memory model [52, 150].

A common C idiom (before C99) is flexible array fields with array size of one, e.g., `arr[1]`. In practice, objects with such array fields have a dynamic size of *more* than one item, but there is no way of MPX knowing this at compile-time. Thus, Intel MPX attempts to narrow bounds to one-item size whenever `arr` is accessed, which leads to false positives. A similar idiom is variable-sized arrays also not supported by Intel MPX, e.g., `arr[]`. These idioms are frequently seen in modern programs, see Table 7.4, row 1. Note that the C99-standard `arr[0]` is acceptable and does not break programs.

Another common idiom is using a struct field (usually the first field of struct) to access other fields of the struct. Again, this breaks the assumptions of Intel MPX and leads to runtime `#BR` exceptions (see Table 7.4, row 2). GCC makes an exception for this case since it is such a popular practice, but ICC is strict and does not have this special rule.

Finally, some programs introduce "memory hacks" for performance, ignoring restrictions of the C memory model completely. The SPEC CPU2006 suite has two such examples: *gcc* has its own complicated memory management with arbitrary type casts and in-pointer bit twiddling, and *soplex* features a scheme that moves objects from one memory region to another by adding an offset to each affected pointer (Table 7.4, row 3). Both these cases lead to false positives.

```
   char* arr[1000]                                          ;; Array with MPX data race
   char obj1                                                  ;; Two adjacent objects ⎤
   char obj2                                                                          ⎦
17 while (true):                                            ;; Background thread
18     for (i=0; i<1000; i++) arr[i] = &obj1
19     for (i=0; i<1000; i++) arr[i] = &obj2
20 while (true):                                            ;; Main thread
21     for (i=0; i<1000; i++) *(arr[i] + offset)
```

Figure 7.8 – This program breaks Intel MPX. If `offset=0` then MPX has false alarms, else — undetected bugs.

Ultimately, all such non-compliant cases must be fixed (indeed, we patched flexible/variable-length array issues to work under Intel MPX). However, sometimes the user may have strong incentives against modifying the original code. In this case, she can opt for slightly worse security guarantees and disable narrowing of bounds via a `fno-chkp-narrow-bounds` flag. Another non-intrusive alternative is to mark objects that must *not* be narrowed (e.g., flexible arrays) with a special compiler attribute.

Multithreading issues. Current Intel MPX implementations may introduce false positives and negatives in multithreaded programs [52]. The problem arises because of the way Intel MPX loads and stores pointer bounds via its `bndldx` and `bndstx` instructions. Recall from §7.3 that whenever a pointer is loaded from main memory, its bounds must also be loaded from the corresponding bounds table (Figure 7.2b, Lines 8-9).

Ideally, the load of the pointer and its bounds must be performed *atomically* (same for stores). However, nor the current hardware implementation neither GCC/ICC compilers enforce this atomicity. This lack of proper multithreading support in Intel MPX can lead to (1) correct programs crashing due to false alarms, or (2) buggy programs being exploited *even if* protected by Intel MPX.

Consider an example in Figure 7.8. A "pointer bounds" data race happens on the `arr` array of pointers. The background thread fills this array with all pointers to the first or to the second object alternately. Meanwhile, the main thread accesses a whatever object is currently pointed-to by the array items. Note that depending on the value of the constant `offset`, the original program is either always-correct or always-buggy: if `offset` is zero, then the main thread always accesses the correct object, otherwise it accesses an incorrect, adjacent object. The second case, if found in a real code, introduces a vulnerability which could be exploited by an adversary.

With Intel MPX, additional `bndstx` instruction is inserted in Line 2 to store the bound corresponding to the first object (same for Line 3 and second object). Also, a `bndldx` instruction is inserted in Line 5 to retrieve the bound for an object referenced by `arr[i]`. Bound checks `bndcl` and `bndcu` are also added at Line 5, before the actual access to the object. Now, the following race can occur. The main thread loads the pointer-to-first-object from the array and—right before loading the corresponding bound—is preempted by the background thread. The background thread overwrites all array items such that they point to the second object, and also overwrites the corresponding bounds. Finally, the main thread is scheduled back and loads the bound, however, the bound now corresponds to the second object. The main thread is left with the pointer to the first object but with the bounds of the second one.

We implemented this test case in C and compiled it with both GCC and ICC. As expected, the MPX-enabled program had both false positives and false negatives.

In case of a correct original program (i.e., with `offset=0`), such discrepancy leads to a *false positive* when actually accessing the object at Line 5. Indeed, the pointer to the object is

correct but the bounds were overwritten by the background thread, so MPX triggers a false-alarm exception. Debugging the root cause of such non-deterministic pseudo-bugs would be a frustrating experience for end users.

The case of an originally buggy program (with `offset=1`) is more disconcerting. After all, Intel MPX is supposed to detect all out-of-bounds accesses, but in this example Intel MPX introduces *false negatives*! Here, the pointer to the first object plus offset incorrectly lends into the second object. But since the main thread checks against the bounds of the second object, this bug is not caught by Intel MPX. We believe that this implementation flaw—that out-of-bounds bugs can *sometimes* go unnoticed—can scare off users of multithreaded applications. We also believe that a resourceful hacker would be able to construct an exploit that, based on these findings, could overcome Intel MPX defense with a high probability [252].

We must note however that we did not observe incorrect behavior in Phoenix and PARSEC multithreaded benchmark suites—we were lucky not to encounter programs that break Intel MPX.

For safe use in multithreaded programs, MPX instrumentation must enforce atomicity of loading/storing pointers and their bounds. At the software (compiler) level, this dictates the use of some synchronization primitive around each pair of `mov-bndldx/bndstx`, being it fine-grained locks, hardware transactional memory, or atomics. Whatever primitive is chosen, we conjecture a significant drop in performance of Intel MPX.

A solution at a microarchitectural level would be to merge the pairs `mov-bndldx/bndstx` and assure their atomic execution. The instruction decoder could detect a `bndldx`, find the corresponding pointer `mov` in the instruction queue, and instruct the rest of execution to handle these instructions atomically. However, we believe this solution could require intrusive changes to the CPU front-end. Moreover, it would significantly limit compiler optimization capabilities.

7.4 Measurement Study

In this section we answer the following questions:

- What is the performance penalty of Intel MPX?
  - How much slower does a program become?
  - How does memory consumption change?
  - How does protection affect scalability of multithreaded programs?
- What level of security does Intel MPX provide?
- What usability issues arise when Intel MPX is applied?

7.4.1 Experimental Setup

All the experimental infrastructure was build using Fex [172] benchmarking framework with corresponding changes for the required build types, measurement tools, and for certain experimental procedures.

Testbed. The machines we used are equipped with an Intel Skylake 3.40GHz CPU with 4 physical cores (8 hyper-threads), 32KB L1, 256KB L2, and 8MB shared L3 caches, 64 GB of RAM, and run a Docker container on top of Ubuntu 16.04 (Linux 4.4.0). The compilers we used are GCC 6.1.0, ICC 17.0.0, and Clang/LLVM 3.8.0. For experiments on case studies, we used two machines with the network bandwidth between them equal to 938 Mbits/sec as measured by iperf.

Measurement tools. We used perf stat to measure all CPU-related parameters: number of cycles and instructions in user-space and kernel-space, L1 and L3 load/store misses, and TLB misses. Not to introduce additional measurement error, we measured these parameters in parts, 8 parameters at a time. Since perf does not provide capabilities for measuring physical memory consumption of a process, we used time `-verbose` and collected maximum resident set size. To gather Intel MPX instruction statistics, we developed a Pin tool.

Benchmarks. We used three benchmark suits in our evaluation: PARSEC 3.0 [32], Phoenix 2.0 [187], and SPEC CPU 2006 [99]. To remove some of the previously found bugs, we applied a patch to SPEC suite. Also, during our work, we found and fixed a set of bugs in them.

All the benchmarks were compiled together with the libraries they depend upon (except raytrace from PARSEC which requires X11 libraries).

Experiments. Each program was executed 10 times, and the results were averaged using arithmetic mean (note, we made sure that variance is very low and it is safe to use arithmetic mean). The mean across different programs in the benchmark suite was calculated using geometric mean. Geometric mean was also used to calculate the "final" mean across three benchmark suites.

7.4.2 Performance

To evaluate overheads incurred by Intel MPX, we tested it on three benchmark suites: Phoenix 2.0 [187], PARSEC 3.0 [32], and SPEC CPU2006 [99]. To put the results into context, we measured not only the ICC and GCC implementations of Intel MPX, but also AddressSanitizer, SAFECode, and SoftBound (see §7.2 for details).

Runtime overhead. We start with the single most important parameter: runtime overhead (see Figure 7.9).

First, we note that ICC-MPX performs significantly better than GCC-MPX. At the same time, ICC is less usable: only 30 programs out of total 38 (79%) build and run correctly, whereas 33 programs out of 38 (87%) work under GCC (see also §7.4.4).

AddressSanitizer, despite being a software-only approach, performs on par with ICC-MPX and better than GCC-MPX. This unexpected result testifies that the hardware-assisted performance improvements of Intel MPX are offset by its complicated design and suboptimal instructions. Although, AddressSanitizer provides worse security guarantees than Intel MPX (§7.4.3).

SAFECode and SoftBound show good results on Phoenix programs, but behave much worse—both in terms of performance and usability—on PARSEC and SPEC. First, consider SAFECode on Phoenix: due to the almost-pointerless design and simplicity of Phoenix programs, SAFECode achieves a low overhead of 5%. However, it could run only 18 programs out of 31 (58%) on PARSEC and SPEC and exhibited the highest overall overheads. SoftBound executed only 7 programs on PARSEC and SPEC (23%). Moreover, both SAFECode and SoftBound showed unstable behavior: some programs had overheads of more than 20×.

Instruction overhead. In most cases, performance overheads are dominated by a single factor: the increase in number of instructions executed in a protected application. It can be seen if we compare Figures 7.9 and 7.10; there is a strong correlation between them.

As expected, the optimized MPX (i.e., ICC version) has low instruction overhead due to its HW assistance (~70% lower than AddressSanitizer). Thus, one could expect sufficiently low performance overheads of Intel MPX once the throughput and latencies of Intel MPX instructions improve (see §7.3.1).

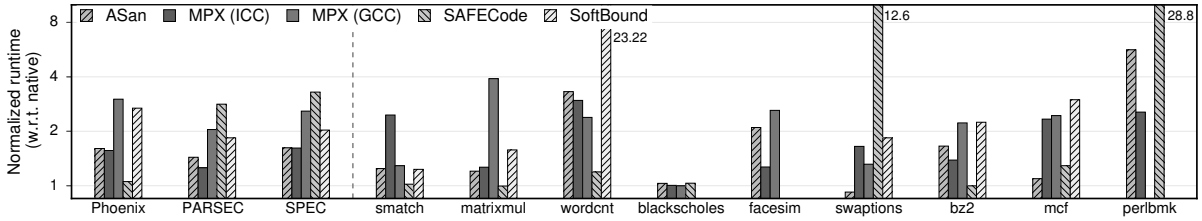

Figure 7.9 – Performance (runtime) overhead with respect to native version.

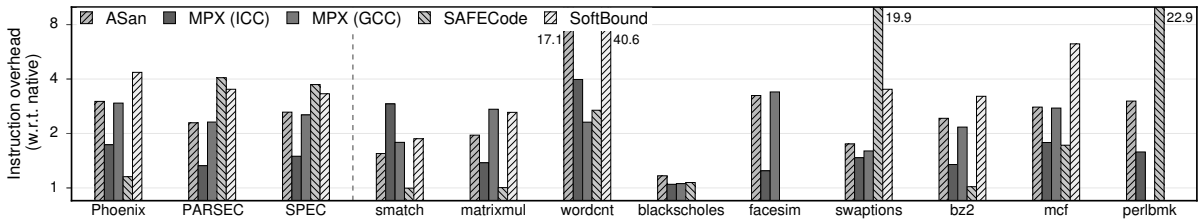

Figure 7.10 – Increase in number of instructions with respect to native version.

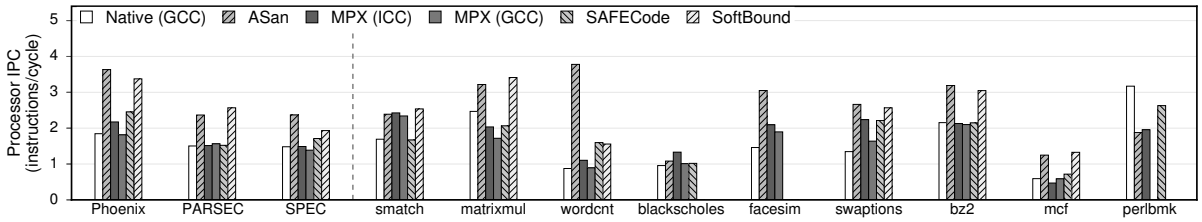

Figure 7.11 – IPC (instructions/cycle) numbers for native and protected versions.

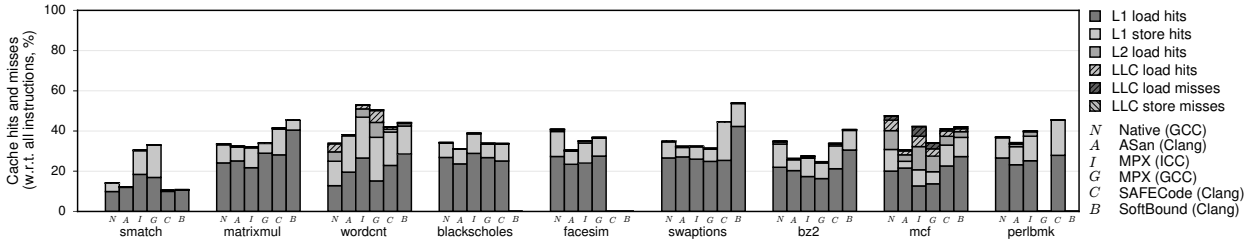

Figure 7.12 – CPU cache behavior of native and protected versions.

Instruction overhead of Intel MPX may also come from the management of Bounds Tables (see §7.3.2). Our microbenchmarks show that it can cause a slowdown of more than 100% in the worst case. However, we did not observe a noticeable impact in real-world applications. Even those applications that create hundreds of BTs exhibit a minor slowdown in comparison to other factors.

IPC. Many programs do not utilize the CPU execution-unit resources fully. For example, the theoretical IPC (instructions/cycle) of our machine is ~5, but many programs achieve only 1–2 IPC in native executions (see Figure 7.11). Thus, memory-safety techniques benefit from underutilized CPU and partially mask their performance overhead.

The most important observation here is that Intel MPX does not increase IPC. Our microbenchmarks (§7.3.1) indicate that this is caused by contention of MPX bounds-checking instructions on one execution port. If this functionality would be available on more ports, Intel MPX would be able to use instruction parallelism to a higher extent and the overheads would be lower.

At the same time, software-only approaches—especially AddressSanitizer and SoftBound—

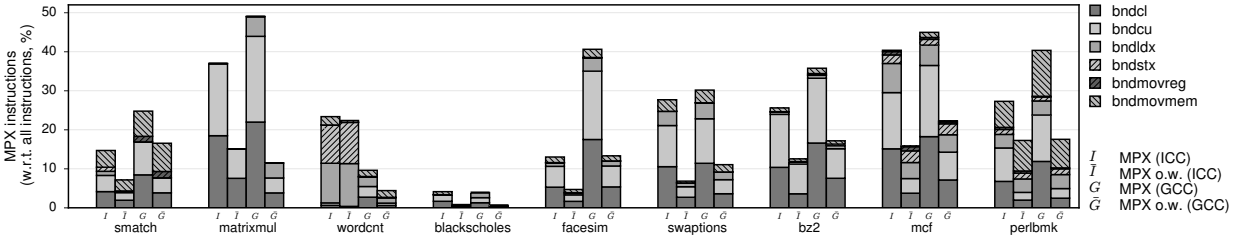

Figure 7.13 – Shares of Intel MPX instructions with respect to all executed instructions.

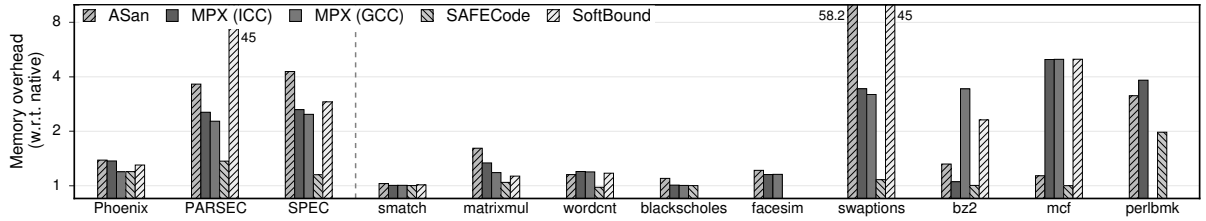

Figure 7.14 – Memory overhead with respect to native version.

significantly increase IPC, partially hiding their performance overheads.

Cache utilization. Some programs are memory-intensive and stress the CPU cache system. If a native program has many L1 or LLC cache misses, then the memory subsystem becomes the bottleneck. In these cases, memory-safety techniques can partially hide their performance overhead.

It can be illustrated with the *wordcnt* example compiled with ICC-MPX (Figure 7.12). It has a huge instruction overhead of 4×, IPC close to native, and (as we will see next) many expensive `bndldx` and `bndstx` operations. And still its performance overhead is only 3×. Why? It appears the native version of *wordcnt* has a significant number of cache misses. They have high performance cost and therefore can partially mask the overhead of ICC-MPX.

Intel MPX instructions. In the case of Intel MPX, one of the most important performance factors is the type of instructions that are used in instrumentation. In particular, storing (`bndstx`) and loading (`bndldx`) bounds require two-level address translation—a very expensive operation that can break cache locality (see §7.3.1). To prove it, we measured the shares of MPX instructions in the total number of instructions of each program (Figure 7.13).

As expected, a lion share of all MPX instructions are bounds-checking `bndcl` and `bndcu`. Additionally, many programs need `bndmov` to move bounds from one register to another (`bndmovreg`) or spill bounds on stack (`bndmovmem`). Finally, pointer-intensive programs require the use of expensive `bndstx` and `bndldx` to store/load bounds in Bounds Tables.

There is a strong correlation between the share of `bndstx` and `bndldx` instructions and performance overheads. For example, *matrixmul* under ICC-MPX almost exclusively contains bounds checks: accordingly, there is a direct mapping between instruction and performance overheads. However, the GCC-MPX version is less optimized and inserts many `bndldx`s, which leads to a significantly higher performance overhead.

The ICC-MPX version of *wordcnt* has a ridiculous share of `bndldx`/`bndstx` instructions. This is due to a performance bug in libchkp library of ICC that uses a naive algorithm for the `memcpy` wrapper (see §7.3.3).

Memory consumption. In some scenarios, memory overheads (more specifically, resident set size overheads) can be a limiting factor, e.g., for servers in data centers which co-locate programs and perform frequent migrations. Thus, Figure 7.14 shows memory overhead measurements.

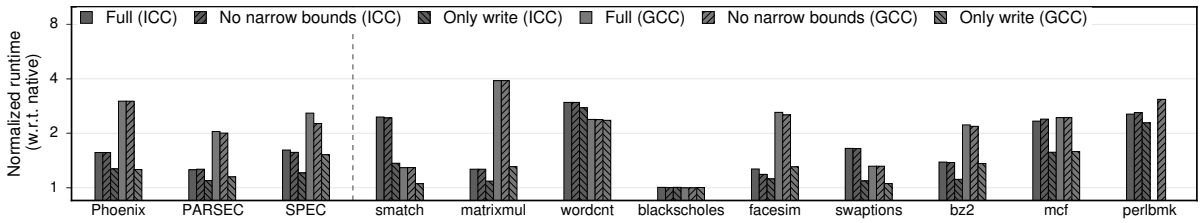

Figure 7.15 – Impact of MPX features—narrowing and only-write protection—on performance.

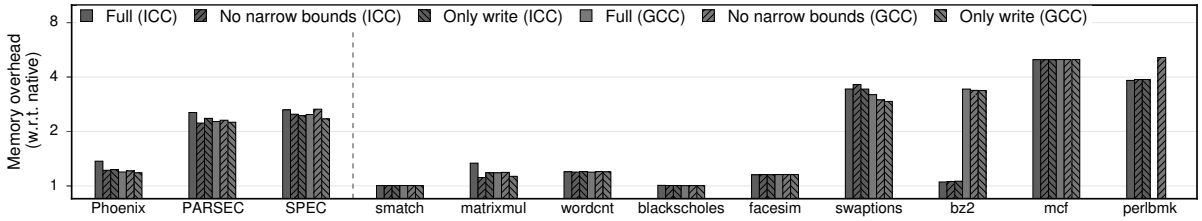

Figure 7.16 – Impact of MPX features—narrowing and only-write protection—on memory.

On average, Intel MPX has a 2.1× memory overhead under ICC version and 1.9× under GCC. It is a significant improvement over AddressSanitizer (2.8×). There are three main reasons for that. First, AddressSanitizer changes memory layout of allocated objects by adding "redzones" around each object. Second, it maintains a "shadow zone" that is directly mapped to main memory and grows linearly with the program's working set size. Third, AddressSanitizer has a "quarantine" feature that restricts the reuse of freed memory.[6] On the contrary, Intel MPX allocates space only for pointer-bounds metadata and has an intermediary Bounds Directory that trades lower memory consumption for longer access time. Interestingly, SAFECode exhibits even lower memory overheads because of its pool-allocation technique. Unfortunately, low memory consumption does not imply good performance.

Influence of additional Intel MPX features. Intel MPX has two main features that influence both performance and security guaranties (§7.3.3). *Bounds narrowing* increases security level but may harm performance. *Only-write protection*, on the other side, improves performance by disabling checks on memory reads.

The comparison of these features is presented in Figures 7.15 and 7.16. As we can see, bounds narrowing has a low impact on performance because it does not change the number of checks. At the same time, it may slightly increase memory consumption because it has to keep more bounds. Only-write checking has the opposite effect—having to instrument less code reduces the slowdown but barely has any impact on memory consumption.

Multithreading. To evaluate the influence of multithreading, we measured execution times of all benchmarks on 2 and 8 threads (see Figure 7.17). Note that only Phoenix and PARSEC are multithreaded (SPEC is not). Also, both SoftBound and SAFECode are not thread-safe and therefore were excluded from measurements.

As we can see from Figure 7.17, the difference in scalability is minimal. For Intel MPX, it is caused by the absence of multithreading support, which means that no additional code is executed

[6] Quarantine is a temporal-protection feature and, in principle, it gives an unfair advantage to Intel MPX which lacks this kind of protection. Indeed, if quarantine zone is disabled, AddressSanitizer's memory overhead drops on average to ~1.5× for both PARSEC and SPEC, although the performance overhead is not influenced. We did not include this number into our main results because the goal of our study was to compare the solutions in their *default* configuration, without any tweaks from the side of end user.

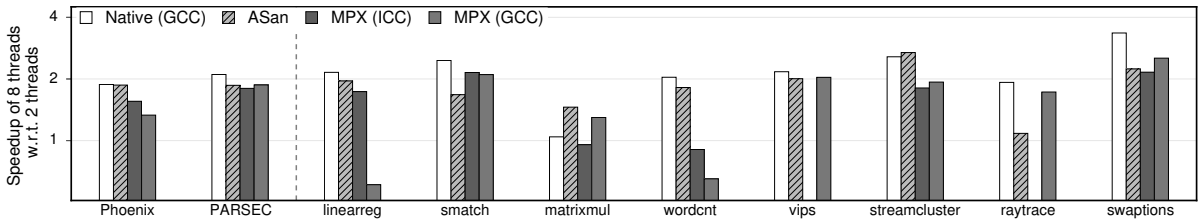

Figure 7.17 – Relative speedup (scalability) with 8 threads compared to 2 threads.

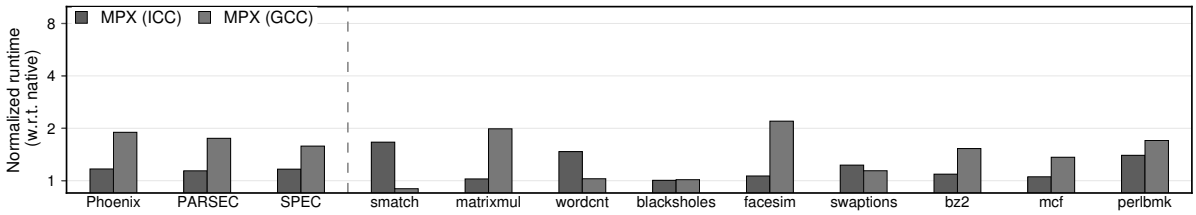

Figure 7.18 – Performance (runtime) overhead with respect to native version on a Haswell CPU that does not support Intel MPX. All MPX instructions are executed as NOPs.

in multithreaded versions. For AddressSanitizer, there is no need for explicit synchronization—the approach is thread-safe by design.

Peculiarly, GCC-MPX experiences not speedups but slowdowns on *linearreg* and *wordcnt*. Upon examining these cases, we found out that this anomaly is due to detrimental cache line sharing of BT entries.

For *swaptions*, AddressSanitizer and Intel MPX scale significantly worse than native. It turns out that these techniques do not have enough spare IPC resources to fully utilize 8 threads in comparison to the native version (the problem of hyperthreading). Similarly, for *streamcluster*, Intel MPX performs worse than AddressSanitizer and native versions. This is again an issue with hyperthreading: Intel MPX instructions saturate IPC resources on 8 threads and thus cannot scale as good as native.

Varying inputs sizes. Different input sizes (working sets) may cause different cache behaviors, which in turn causes changes in overheads. To investigate the extent of such effects, we ran several benchmarks with three inputs—small, medium, and large. The results do not provide any unexpected insights and thus omitted from here. The general trend is that the input size has very little impact on performance overhead.

Runtime overhead on older CPU architectures. As we mention in §7.3.1, MPX-protected applications can be executed even on older Intel CPUs that do not support Intel MPX. In this case, MPX instructions will be executed as NOPs and consequently, no protection will be provided. Yet, NOPs are not free—each NOP takes 1 cycle to execute, they take space in caches, in the instruction pipeline, etc. It means that in such a scenario the application will be slowed down but will not get any additional security guaranties. To evaluate this effect, we run the same set of benchmarks on a Haswell machine. The results are presented in Figure 7.18.

7.4.3 Security

RIPE testbed. We evaluated all approaches against the RIPE security testbed [244]. RIPE is a synthesized C program that tries to attack itself in a number of ways, by overflowing a buffer allocated on the stack, heap, or in data or BSS segments. RIPE can imitate up to 850 attacks, including shellcode, return-into-libc, and return-oriented programming. In our evaluation, even

Approach	Working attacks
MPX (GCC) default *	**41/64** (all memcpy and intra-object overflows)
MPX (GCC)	**0/64** (none)
MPX (GCC) no narrow	**14/64** (all intra-object overflows)
MPX (ICC)	**0/34** (none)
MPX (ICC) no narrow	**14/34** (all intra-object overflows)
AddressSanitizer (GCC)	**12/64** (all intra-object overflows)
SoftBound (Clang)	**14/38** (all intra-object overflows)
SAFECode (Clang)	**14/38** (all intra-object overflows)

*Without `-fchkp-first-field-has-own-bounds` and with `BNDPRESERVE=0`

Table 7.5 – Results of RIPE security benchmark. In Col. 2, "41/64" means that 64 attacks were successful in native GCC version, and 41 attacks remained in MPX version.

under relaxed security flags—we disabled Linux ASLR, stack canaries, and fortify-source and enabled executable stack—modern compilers were susceptible only to a small number of attacks. Under native GCC, only 64 attacks survived, under ICC—34, and under Clang—38. RIPE is specifically tailored to GCC, thus more attacks are possible under this compiler.

The results for all approaches are presented in Table 7.5. Surprisingly, a default GCC-MPX version showed very poor results, with 41 attacks (or 64% of all possible attacks) succeeding. As it turned out, the default GCC-MPX flags are sub-optimal. First, we found a bug in the `memcpy` wrapper which forced bounds registers to be nullified, so the bounds checks on `memcpy` were rendered useless (see Table 7.3). This bug disappears if the `BNDPRESERVE` environment variable is manually set to one. Second, the MPX pass in GCC does *not* narrow bounds for the first field of a struct by default, in contrast to ICC which is more strict. To catch intra-object overflows happening in the first field of structs–the case of RIPE code—one needs to pass the `-fchkp-first-field-has-own-bounds` flag to GCC. When we enabled these two flags, all attacks were prevented; all next rows in the table were tested with these flags.

Other results are expected. Intel MPX versions without narrowing of bounds overlook 14 intra-object overflow attacks, where a vulnerable buffer and a victim object live in the same struct. The same attacks are overlooked by AddressSanitizer, SoftBound, and SAFECode. Interestingly, AddressSanitizer has 12 working attacks, i.e., two attacks less than other approaches. Though we did not inspect this in detail, AddressSanitizer was able to prevent two shellcode intra-object attacks on the heap.

We performed the same experiment with only-writes versions of these approaches, and the results were exactly the same. This is explained by the fact that RIPE constructs only control-flow hijacking attacks and not information leaks (which could escape only-writes protection).

Other detected bugs. During our experiments, we found 6 real out-of-bounds bugs (true positives). Five of these bugs were already known, and one was detected by GCC-MPX and was not previously reported.

The bugs found are: (1) incorrect black-and-white input pictures leading to classic buffer overflow in `ferret`, (2) wrong preincrement statement leading to classic off-by-one bug in `h264ref`, (3) out-of-bounds write in `perlbench`, (4) benign intra-object buffer overwrite in `x264`, (5) benign intra-object buffer overread in `h264ref`, and (6) intra-object buffer overwrite in `perlbench`.

All of these bugs were detected by GCC-MPX with narrowing of bounds. Predictably, three

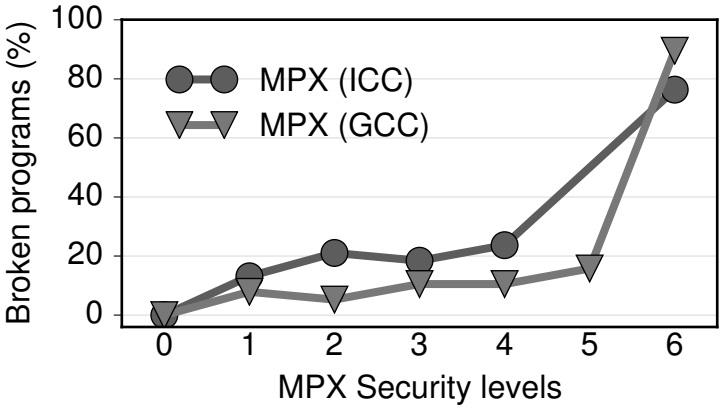

Figure 7.19 – Number of MPX-broken programs rises with stricter Intel MPX protection rules (higher security levels). Level 4 is default.

intra-object bugs and one read-only bug could not be detected by the no-narrowing and only-writes versions of Intel MPX respectively. ICC-MPX detected only three bugs in total: in other cases programs failed due to MPX-related issues (see §7.3.3 and §7.3.4). An interesting correlation emerged: the programs that contain real bugs are also the ones that break most often under Intel MPX.

As expected, AddressSanitizer found only three of these bugs—it checks bounds at the level of whole objects and cannot detect intra-object overflows. SAFECode found bugs (2) and (3), the others either could not be detected due to coarse-grained granularity of bounds-checking or SAFECode could not compile the programs. Unfortunately, SoftBound left bug (2) undetected and broke on other three programs with bugs: `ferret` and `x264` are multithreaded and thus not supported by SoftBound, and `perlbench` would not run correctly.

7.4.4 Usability

As we showed in §7.3.4, some programs break under Intel MPX because they use unsupported C idioms or outright violate the C standard. Moreover, as shown in §7.3.3, other programs even fail to compile or run due to internal bugs in the compiler MPX passes (one case for GCC and 8 for ICC).

Figure 7.19 highlights the *usability* of Intel MPX, i.e., the number of MPX-protected programs that fail to compile correctly and/or need significant code modifications. Note that many programs can be easily fixed (see Table 7.4); we do not count them as broken. MPX *security levels* are based on our own classification and correspond to the stricter protection rules, where level 0 means unprotected native version and 6—the most secure MPX configuration (see §7.6). In total, our evaluation covers 38 programs from the Phoenix, PARSEC, and SPEC benchmark suites.

As can be seen, around 10% of programs break already at the weakest level 1 of Intel MPX protection (without narrowing of bounds and protecting only writes). At the highest security level 6 (with enabled `BNDPRESERVE`), most of the programs fail.

As for other approaches, *no* programs broke under AddressSanitizer. For SAFECode, around 70% programs executed correctly (all Phoenix, half of PARSEC, and 3/4 of SPEC). SoftBound—being a prototype implementation–showed poor results, with only simple programs surviving (all Phoenix, one PARSEC, and 6 SPEC). These results roughly correspond to the ones in the original papers [64, 158].

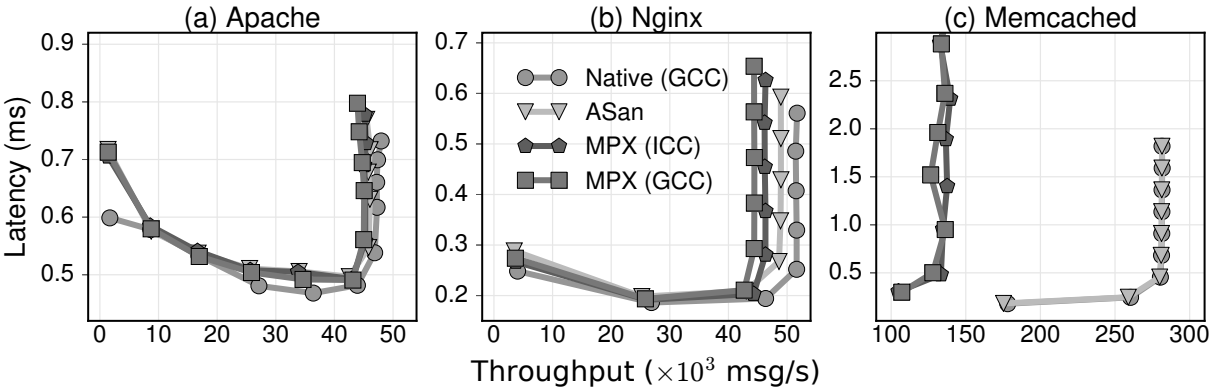

Figure 7.20 – Throughput-latency for (a) Apache web server, (b) Nginx web server, and (c) Memcached caching system.

7.5 Case Studies

To understand how Intel MPX affects complex real-world applications, we experimented with three case studies: Apache and Nginx web servers and Memcached memory caching system. Similar to the previous section, we evaluated these programs along three dimensions: performance and memory overheads, security guarantees, and usability.

We compare default Intel MPX implementations of both GCC and ICC against the native version, as well as AddressSanitizer. We were not able to compile any of the case studies under SoftBound and SAFECode: in most cases, the Configure scripts complained about an "unsupported compiler", and in one case (Apache under SoftBound) the compilation crashed due to an internal compiler error. The native version we chose to show is GCC: native ICC and Clang versions have almost identical results, with an exception of Nginx explained later. For the same reasons, we show only the GCC implementation of AddressSanitizer.

All experiments were performed on the same machines as in the previous section (§7.4). One machine served as a server and a second one as clients, connected with a 1GB Ethernet cable and an actual bandwidth of 938 Mbits/sec. We configured all case studies to utilize all 8 cores of the server (details below). For other configuration parameters, we kept their default values.

All three programs were linked against their dependent libraries statically. We opted for static linking to investigate the complete overhead of all components constituting each program.

7.5.1 Apache Web Server

For evaluation, we used Apache version 2.4.18 linked against OpenSSL 1.0.1f [14]. This OpenSSL version is vulnerable to the infamous Heartbleed bug which allows the attacker to leak confidential information such as secret keys and user passwords in plain-text [227]. Since both AddressSanitizer and Intel MPX do not support inline assembly, we disabled it for all builds of Apache. To fully utilize the server, we used the default configuration of Apache's MPM event model.

The classic ab benchmark was run on a client machine to generate workload, constantly fetching a static 2.3K web-page via HTTP, with a KeepAlive feature enabled. To adapt the load, we increased the number of simultaneous requests at a time.

Unfortunately, while testing against Heartbleed, we discovered that ICC-MPX suffers from a run-time Intel compiler bug[7] in the `x509_cb` OpenSSL function, leading to a crash of Apache.

[7] `https://software.intel.com/en-us/forums/intel-c-compiler/topic/700550`

	Apache	Nginx	Memcached
Native	9.4	4.3	73
MPX	**120**	18	**352**
ASan	33	**380**	95

Table 7.6 – Memory usage (MB) for peak throughput. (GCC-MPX and ICC-MPX showed identical results.)

This bug triggered only on HTTPS connections, thus allowing us to still run performance experiments on ICC-MPX.

Performance. As Figure 7.20a shows, GCC-MPX, ICC-MPX, and AddressSanitizer all show minimal overheads, achieving 95.3%, 95.7%, and 97.5% of native throughput. Overhead in latency did not exceed 5%. Such good performance is explained by the fact that our experiment was limited by the network and not CPU or memory. (We observed around $480 - 520\%$ CPU utilization in all cases.)

In terms of memory usage (Table 7.6), AddressSanitizer exhibits an expected 3.5× overhead. In contrast, Intel MPX variants have dramatic 12.8× increase in memory consumption. This is explained by the fact that Apache allocates an additional 1MB of pointer-heavy data per each client, which in turn leads to the allocation of many Bounds Tables.

Security. For security evaluation, we exploited the infamous Heartbleed bug [12, 227]. In a nutshell, Heartbleed is triggered when a maliciously crafted TLS heartbeat message is received by the server. The server does not sanity-check the length-of-payload parameter in the message header, thus allowing `memcpy` to copy the process memory's contents in the reply message. In this way, the attacker can read confidential memory contents.

AddressSanitizer and GCC-MPX detect Heartbleed[8].

7.5.2 Nginx Web Server

We tested Nginx version 1.4.0—the last version with a stack buffer overflow vulnerability [164]. Nginx was configured with the "autodetected" number of worker processes to load all cores and was benchmarked against the same ab benchmark as Apache. ab was also used as a client.

To successfully run Nginx under GCC-MPX with narrowing of bounds, we had to manually fix a variable-length array `name[1]` in the `ngx_hash_elt_t` struct to `name[0]`. However, ICC-MPX with narrowing of bounds still refused to run correctly, crashing with a false positive in `ngx_http_merge_locations` function. In a nutshell, the reason for this bug was a cast from a smaller type, which rendered the bounds too narrow for the new, larger type. Note that GCC-MPX did *not* experience the same problem because it enforces the first struct's field to inherit the bounds of the whole object by default—in contrast to ICC-MPX which takes a more rigorous stance. For the following evaluation, we used the version of ICC-MPX with narrowing of bounds disabled.

Performance. With regards to performance (Figure 7.20b), Nginx has a similar behavior to Apache. AddressSanitizer reaches 95% of native throughput, while GCC-MPX and ICC-MPX lag behind with 86% and 89.5% respectively. Similar to Apache, this experiment was network-bound,

[8]The actual situation with Heartbleed is more contrived. OpenSSL uses its own memory manager which partially bypasses the wrappers around `malloc` and `mmap`. Thus, in reality memory-safety approaches find Heartbleed only if the length parameter is greater than 32KB (the granularity at which OpenSSL allocates chunks of memory for its internal allocator) [97].

with CPU usage of 225% for native, 265% for Intel MPX, and 300% for AddressSanitizer. (CPU usage numbers prove that HW-assisted approaches impose less CPU overheads.)

As a side note, Nginx has predictable behavior only under GCC. Native ICC version reaches only 85% of the GCC's throughput, and native Clang only 90%. Even more surprising, the ICC-MPX version performed 5% *better* than native ICC; similarly, the AddressSanitizer-Clang version was 10% *better* than native Clang. We are still investigating the reasons for this unexpected behavior.

As for memory consumption (Table 7.6), the situation is opposite as with Apache: Intel MPX variants have a reasonable 4.2× memory overhead, but AddressSanitizer eats up 88× more memory (it also has 625× more page faults and 13% more LLC cache misses). But then why Intel MPX is slower than AddressSanitizer if their memory characteristics indicate otherwise? The reason for the horrifying AddressSanitizer numbers is its "quarantine" feature—AddressSanitizer employs a special memory management system which avoids re-allocating the same memory region for new objects, thus decreasing the probability of temporal bugs such as use-after-free. Instead, AddressSanitizer marks the used memory as "poisoned" and requests new memory chunks from the OS (this explains huge number of page faults). Since native Nginx recycles the same memory over and over again for the incoming requests, AddressSanitizer experiences huge memory blow-up. When we disabled the quarantine feature, AddressSanitizer used only 24MB of memory.

Note that this quarantine problem does not affect performance. Firstly, Nginx is network-bound and has enough spare resources to hide this issue. Secondly, the rather large overhead of allocating new memory hides the overhead of requesting new chunks from the OS.

Security. To evaluate security, the bug under test was a stack buffer overflow CVE-2013-2028 that can be used to launch a ROP attack [11]. Here, a maliciously crafted HTTP request forces Nginx to erroneously recognize a signed integer as unsigned. Later, a `recv` function is called with the overflown size argument and the bug is triggered.

Perhaps surprisingly, AddressSanitizer detects this bug, but both versions of Intel MPX *do not*. The root cause is the run-time wrapper library: AddressSanitizer wraps *all* C library functions including `recv`, and the wrapper—not the Nginx instrumented code—detects the overflow. In case of both GCC-MPX and ICC-MPX, only the most widely used functions, such as `memcpy` and `strlen`, are wrapped and bounds-checked. That is why when `recv` is called, the overflow happens in the unprotected C library function and goes undetected by Intel MPX.

This highlights the importance of full protection—not only protecting the program's own code, but also writing wrappers around all unprotected libraries used by the program. Another interesting aspect is that this overflow bug is read-only and cannot be caught by write-only protection. No matter how tempting it may sound to protect only writes, one must remember that buffer-overread vulnerabilities will slip away from such low-overhead bounds checking.

7.5.3 Memcached Caching System

Lastly, we experimented with Memcached version 1.4.15 [78]. This is the last version susceptible to a simple DDoS attack [151]. In all experiments, Memcached was run with 8 threads to fully utilize the server. For the client we used a memaslap benchmark from libmemcached with a default configuration (90% reads of average size 1700B, 10% writes of average size 400B). We increased the load by adapting the concurrency number.

After some vexing debugging experiences with Nginx and Apache, we were pleased to experience no issues instrumenting Memcached with GCC-MPX and ICC-MPX.

L	Description	RIPE GCC	RIPE ICC	Unfound GCC	Unfound ICC	Broken GCC	Broken ICC	Perf (×) GCC	Perf (×) ICC
0	native program (no protection)	64	34	6	3	0	0	1.00	1.00
1	MPX only-writes and no narrowing of bounds	14	14	3	0	3	5	1.29	1.18
2	MPX no narrowing of bounds	14	14	3	0	2	8	2.39	1.46
3	MPX only-writes and narrowing of bounds	14	0	2	0	4	7	1.30	1.19
4	MPX narrowing of bounds (default)	14	0	0	0	4	9	2.52	1.47
5	+ `fchkp-first-field-has-own-bounds`*	0	–	0	–	6	–	2.52	–
6	+ `BNDPRESERVE=1` (protect all code)	0	0	0	0	34	29	–	–
	AddressSanitizer [207]	12	–	3	–	0	–	1.55	–

* except intra-object overflows through the first field of struct; L5 removes this limitation (relevant for GCC)

Table 7.7 – The summary table with our classification of Intel MPX security levels—from lowest L1 to highest L6—highlights the trade-off between security (number of unprevented *RIPE* attacks and other *Unfound* bugs in benchmarks), usability (number of MPX-*Broken* programs), and performance overhead (average *Perf* overhead w.r.t. native executions). AddressSanitizer is shown for comparison in the last row.

Performance. Performance-wise, Memcached turned out to be the worst case for Intel MPX (see Figure 7.20c). While AddressSanitizer performs on par with the native version, both GCC-MPX and ICC-MPX achieved only $48 - 50\%$ of maximum native throughput.

In case of native and AddressSanitizer, performance of Memcached was limited by network. But it was not the case for Intel MPX: Memcached exercised only 70% of the network bandwidth. The memory usage numbers in Table 7.6 help understand the bottleneck of Intel MPX. While AddressSanitizer imposed only 30% memory overhead, both Intel MPX variants used 350MB of memory (4.8× more than native). This huge memory overhead broke cache locality and resulted in 5.4× more page faults and $10 - 15\%$ LLC misses, making Intel MPX versions essentially memory-bound. (Indeed, the CPU utilization never exceeded 320%.)

Security. For security evaluation, we used a CVE-2011-4971 vulnerability [151]. In this denial-of-service attack, a specially crafted packet is received by the server and passed to the handler (`conn_nread`) which tries to copy all packet's contents into another buffer via the `memmove` function. However, due to the integer signedness error in the size argument, `memmove` tries to copy gigabytes of data and quickly segfaults. All approaches—AddressSanitizer, GCC-MPX, and ICC-MPX—detected buffer overflow in the affected function's arguments and stopped the execution.

7.6 Lessons Learned

Table 7.7 summarizes the results of our work. For convenience, we introduce six *Intel MPX security levels* to highlight the trade-offs between security, usability, and performance.

In general, Intel MPX is a promising technology: it provides the strongest possible security guarantees against spatial errors, it instruments most programs transparently and correctly, its ICC incarnation has moderate overheads of ~50%, it can interoperate with unprotected legacy libraries, and its protection level is easily configurable. However, our evaluation indicates that it is not yet ready for widespread use because of the following issues:

Lesson 1: New instructions are not as fast as expected. First, current Skylake processors perform bounds checking mostly sequentially. Our microbenchmarks indicate this is caused by contention of bounds-checking instructions on one execution port. We project that, if this

functionality would be available on more ports, Intel MPX would be able to use instruction parallelism to a higher extent and the overheads would be lower. Secondly, loading/storing bounds registers from/to memory involves costly two-level address translation, which can contribute a significant share to the overhead. Together, these two issues lead to tangible runtime overheads of ~50% even with all optimizations applied (in the ICC case).

Lesson 2: The supporting infrastructure is not mature enough. Intel MPX support is available for GCC and ICC compilers. At the compiler level, GCC-MPX has severe performance issues (~150%) whereas ICC-MPX has a number of compiler bugs (such that 10% of programs broke in our evaluation). At the runtime-support level, both GCC and ICC provide only a small subset of function wrappers for the C standard library, thus not detecting bugs in many libc functions.

Lesson 3: Intel MPX provides no temporal protection. Currently, Intel MPX protects only against spatial (out-of-bounds accesses) but not temporal (dangling pointers) errors. All other tested approaches—AddressSanitizer, SoftBound, and SAFECode—guarantee some form of temporal safety. We believe Intel MPX can be enhanced for temporal safety without harming performance, similar to SoftBound.

Lesson 4: Intel MPX does not support multithreading. An MPX-protected multithreaded program can have both false positives (false alarms) and false negatives (missed bugs and undetected attacks). Until this issue is fixed—either at the software or at the hardware level—Intel MPX cannot be considered safe in multithreaded environments. Unfortunately, we do not see a simple fix to this problem that would not affect performance adversely.

Lesson 5: Intel MPX is not compatible with some C idioms. Intel MPX imposes restrictions on allowed memory layout which conflict with several widespread C programming practices, such as intra-structure memory accesses and custom implementation of memory management. This can result in unexpected program crashes and is hard to fix; we were not able to run correctly 8–13% programs (this would require substantial code changes).

In conclusion, we believe that Intel MPX has a potential for becoming the memory protection tool of choice, but currently, AddressSanitizer is the only production-ready option. Even though it provides weaker security guarantees than the other techniques, its current implementation is better in terms of performance and usability. SoftBound and SAFECode are research prototypes and they have issues that restrict their usage in real-world applications (although SoftBound provides higher level of security).

We expect that most identified issues with Intel MPX will be fixed in future versions. Still, support for multithreading and restrictions on memory layout are inherent design limitations of Intel MPX which would require sophisticated solutions, which would in turn negatively affect performance. We hope our work will help practitioners to better understand the benefits and caveats of Intel MPX, and researchers—to concentrate their efforts on those issues still waiting to be solved.

8 Conclusion

This thesis introduced five hardware-assisted techniques for dependability. Each of these techniques chooses its own trade-off between performance and level of dependability. In the realm of fault tolerance, Δ-encoding provides very high hardware-fault coverage at the cost of high performance overhead, whereas Elzar and HAFT are more light-weight techniques, trading some fault coverage for better performance. Similarly, in the realm of security, design of SGXBounds substantially decreases performance and memory overheads, whereas the heavy-weight Intel MPX provides better security guarantees. Nonetheless, the distinctive feature of all introduced approaches is hardware assistance, i.e., CPU features that allow to improve performance. In particular, these features include unused IPC of modern CPUs in case of Δ-encoding, Intel AVX technology for Elzar, Intel TSX for HAFT, Intel SGX for SGXBounds, and Intel MPX for efficient pointer-based bounds-checking.

We conclude the thesis with a brief summary of each technique, limitations of CPU features they rely on, impact of the techniques, and directions for future work.

8.1 Summary of techniques

This thesis described five hardware-assisted techniques for dependability: Δ-encoding, Elzar, and HAFT for protection against hardware faults, and SGXBounds and "MPX Explained" for protection against software memory-corruption bugs. In the following, we briefly summarize each of these techniques:

Δ-encoding is a heavy-weight technique to detect all kinds of hardware CPU and RAM faults with a very high probability of 99.997% (Chapter 3). It is able to detect transient, intermittent, and permanent faults in CPU registers, CPU execution units, CPU cache lines, and DRAM memory. We implemented Δ-encoding as a source-to-source compiler that transparently hardens unmodified C programs. Δ-encoding utilizes unused IPC resources of modern CPUs and relies on CPU features such as superscalar out-of-order execution, branch predictors, and deep pipelines.

Elzar is a light-weight technique that detects a particular kind of CPU faults – transient bit-flips in CPU registers and CPU execution units (Chapter 4). Due to its simplified fault model, Elzar achieves lower performance and memory overheads in comparison to Δ-encoding, though it still exhibits overheads of 4–5× and thus is not practical. We implemented Elzar as an LLVM compiler framework to transparently instrument unmodified C/C++ programs. Elzar abuses Intel AVX technology to introduce triple modular redundancy using Single Instruction Multiple Data (SIMD) vectors.

HAFT is another light-weight technique to detect transient CPU faults, i.e., bit-flips in CPU registers and execution units (Chapter 5). HAFT tackles the same problem as Elzar, but exhibits better performance overhead of only 2× due to its superior design. Similar to Elzar, we implemented HAFT as an LLVM compiler framework that hardens unmodified C/C++ programs. For performance, HAFT re-uses Intel TSX technology to allow efficient roll-backs to the previous correct state.

SGXBounds is a memory-safety technique that detects and tolerates memory corruption bugs such as buffer overflows, out-of-bounds accesses, and off-by-one errors (Chapter 6). SGXBounds introduces a novel, simple design to store lower and upper bounds of referent objects for each pointer in the program and insert efficient bounds checks before original memory accesses. SGXBounds' design is influenced by limitations and peculiarities of Intel SGX technology, which allows it to significantly outperform other state-of-the-art approaches to memory safety.

"MPX Explained" is a deep investigation of the recent Intel MPX technology (Chapter 7). Based on our analysis, Intel MPX proved to be sub-optimal in terms of runtime performance and memory overheads, as well as dangerous to use in multithreaded environments. Similarly to SGXBounds, Intel MPX detects memory corruption bugs such as buffer overflows. It is assumed to be faster than software-only competitors since it performs heavy-weight bounds-checking completely in hardware, with the help of new instructions and CPU registers. However, as our evaluation shows, MPX does not live up to its promise and needs a redesign.

8.2 Limitations of CPU features and our proposals

Each of the five techniques detailed in this thesis shows promising results, achieving lower performance overheads and detecting more faults that other state-of-the-art solutions. However, as we have seen with each technique, even though utilizing specific CPU features vastly improves performance, these features come with their own limitations. Below is a short list of our findings.

x86-64 ISA. For Δ-encoding, we rely heavily on the instruction-level parallelism provided by modern Intel CPUs. However, as we discussed in Chapter 3, Δ-encoding could significantly benefit from two CPU modifications. First, accumulations and checks of Δ-encoding could be moved out from the critical path of Δ-encoded operations of a program; these accumulations and checks could be done in parallel either by a specialized CPU coprocessor (watchdog) or even programmed in a paired FPGA [46]. Second, Δ-encoding operations could be greatly sped up if x86-64 ISA would introduce a one-cycle "add–shift" instruction.

Intel AVX. For Elzar, we re-purpose Intel AVX vector instructions (Chapter 4). Unfortunately, Intel AVX misses some instructions that would be beneficial for our fault-tolerance purposes, as we argued in §4.7.1. In particular, we propose AVX load/store instructions which use an address operand from a YMM register rather than from a general-purpose one. We also propose a `cmp`-like family of AVX instructions to speed up the execution of comparisons. Finally, similar to Δ-encoding, we suggest offloading checks to an FPGA. All these enhancements to current CPU designs would lead to an improvement of 150% over the current Elzar version.

Intel TSX. In case of HAFT, we use Intel TSX extension to wrap the whole program execution in hardware transactions (Chapter 5). In our experience, TSX transactions have a rather short timespan of no more than 5,000 instructions. In addition, TSX transactions spuriously abort without any reason. Thus, if a future implementation of TSX would allow for longer and more stable transactions, this would increase fault coverage of HAFT and decrease its performance overhead. In addition, TSX could benefit from rollback-only suspendable transactions ideal for the HAFT scenario, where stores are buffered without aborting on data conflicts and a transaction could be frozen on interrupts (as done in IBM POWER8 [43]).

Intel SGX. SGXBounds builds on an observation of small address space and restricted memory capabilities inside Intel SGX enclaves (Chapter 6). Our findings are twofold. On the one hand, if future implementations of Intel SGX increase the size of EPC, the appeal of our approach

may diminish. On the other hand, SGXBounds crams metadata in the 32 upper bits of a 64-bit register, limiting addressable memory to only 4 GB. If Intel CPUs would introduce 96- or 128-bit registers, SGXBounds could permit larger address spaces and put more metadata in registers themselves.

Intel MPX. As our analysis of Intel MPX shows (Chapter 7), there are several opportunities to improve the design and performance of MPX. For instance, adding an additional execution port would decrease contention of bounds-checking instructions on CPU resources and thus boost performance. Another example is loads/stores of bounds: the current design involves costly two-level address translation and allocates too many bounds tables. If these bottlenecks of Intel MPX are fixed in future versions, we expect around 50% decrease in performance overhead.

8.3 Impact on academia and industry

In the past two years, from 2015 when Δ-encoding was first published and until 2017 when this thesis was finalized, our work gained recognition in academia as well as in industry. In the following, we describe how our techniques and papers influenced other researchers and companies.

Δ-encoding. Δ-encoding was evaluated in a BMW controller safety concept that argues to move redundant computations from specialized hardware DMR/TMR components to software-implemented fault tolerance [82]. The authors prove by means of stochastic model checking that relying on a primary controller running Δ-encoded software provides better performance with a comparable level of fault coverage. Δ-encoding was also mentioned in a recent survey of fault tolerance approaches published at ARCS'2017 [178]. Finally, Δ-encoding was recognized by the scientific community via the Best Student Paper award at DSN'2015.

Elzar. Concurrently and independently, a group from University of California, Irvine developed a technique for SIMD-based detection of CPU faults that is very similar to Elzar. First the group published a small feasibility study (discussed in Chapter 4) [51] and later a full-fledged LLVM-based compiler framework [50]. This work has many striking similarities with our project, including the use of the LLVM IR level to insert vector instructions, the same motivation of improving performance of trivial instruction duplication, and the insight that the developed technique works especially well for floating-point applications. The main differences between Elzar and their work are as follows. (1) Elzar uses Intel AVX for Triple Modular Redundancy (TMR) while [50] relies on Intel SSE for Dual Modular Redundancy (DMR); thus, Elzar can transparently tolerate single faults, while the approach in [50] requires separate error correction. (2) Elzar works on both integer and floating-point data and targets a broad range of applications, while [50] concentrates only on floating-point data; this explains drastically different evaluation results between two works. (3) Elzar targets multithreaded environments and includes extensive evaluation of multithreaded applications, while [50] works only on single-threaded programs but additionally includes evaluation of energy overhead. (4) Finally, we propose tweaks and improvements for the future implementations of Intel AVX, while [50] lacks such a discussion. We found it fascinating how two independent groups of researchers tackled the same problem in very similar ways, coming to comparable conclusions.

HAFT. Similar to HAFT, Haas et al. proposed to use Intel TSX for transaction roll-back to tolerate CPU faults [93]. Unlike HAFT, their work does not use duplicated instructions for fault detection but rather process-level redundancy (PLR). Due to the more complicated design of PLR, the average performance overhead is slightly higher than of HAFT, 2.4× vs 2×. In contrast

to HAFT, the approach by Haas et al. does not support multithreaded applications. Meanwhile, other researchers propose to use HAFT as a building block for safe and secure microservices [77]. Finally, Baier et al. examine the fault injection probabilistic model presented in HAFT in more detail [21].

SGXBounds. SGXBounds was presented at Eurosys'2017 were it received the Best Paper Award. Since then, the paper was covered in the famous "The Morning Paper" blog by Adrian Colyer [209] and was chosen as one of the influential papers on SGX at the technical seminar in Korea Advanced Institute of Science and Technology (KAIST) [216]. In the meantime, novel approaches to memory safety appeared, proposing new ways to mitigate memory corruption bugs: HardScope [168], MPXK for the Linux kernel [192], and Meds [95]. These approaches do not run inside SGX enclaves and thus do not supersede SGXBounds, being orthogonal to our work.

"MPX Explained". Even though our "Intel MPX Explained" project was published only as a technical report [173], it gained significant attention from both academia and industry. For example, we received flattering reviews from the maintainers of Google AddressSanitizer, developers of GCC, and Intel itself (private correspondence). To the best of our knowledge, "Intel MPX Explained" was the first work to empirically prove the drawbacks of the MPX technology and was helpful to numerous researchers in the security field, e.g., profs. Herbert Bos and Don Porter hold our work in high regard (private correspondence). New research papers that use or discuss Intel MPX frequently cite our technical report instead of the official documentation from Intel [36, 69, 205].

8.4 Future work

Even though this thesis shows effective approaches to fault tolerance and security, they are applied separately and do not necessarily augment each other. Thus, an intriguing question arises naturally: is there a way to effectively combine protection against hardware faults *and* software bugs in one synergic approach?

In a first approximation, both hardware faults and software bugs are essentially errors in execution and manifest in the same way: either by crashing the application/node or causing it to produce incorrect results. Thus, it may seem that a single approach treating both these kinds of errors in a unified way would be our "silver bullet". Unfortunately, the only solution that follows this path is Byzantine Fault Tolerance (BFT), notorious for its high performance overheads and impracticality [222].

However, it turns out that hardware faults and software bugs differ in one crucial detail. Hardware CPU faults are random and transient by nature: they can occur at any point of program execution and they usually do not reappear in the same component. Therefore, the frequently executed parts of the program are the most vulnerable, and comparison of two copies of the same data is the best strategy to ensure correctness. On the contrary, software memory-corruption bugs are not random and not transient: they lurk in the "cold" code of a program and they always manifest themselves if hit multiple times. The best strategy to ensure correctness in this case is to check that specific invariants still hold, e.g., that a memory address still points inside a valid object. Note that comparing two copies of the same corrupted address will always succeed and thus useless for detecting memory bugs.

Hence, the question of whether one can combine fault tolerance and security in one practical solution remains open. Are CPU faults and memory corruption bugs fundamentally different?

Or is there a property common to them both that we overlook? We hope that this thesis outlines several approaches to these issues and leave the development of a “silver bullet” solution for the next generation of researchers.

Bibliography

[1] *2011 CWE/SANS Top 25 Most Dangerous Software Errors*. http://cwe.mitre.org/top25/. Accessed: Nov, 2017 (cit. on p. 10).

[2] A. Gupta et al. "Mesa: Geo-Replicated, Near Real-Time, Scalable Data Warehousing". In: *VLDB*. 2014 (cit. on pp. 2, 10, 46, 69).

[3] *ab—Apache HTTP server benchmarking tool*. https://httpd.apache.org/docs/2.4/programs/ab.html. Accessed: Nov, 2017 (cit. on p. 115).

[4] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. "Control-flow Integrity Principles, Implementations, and Applications". In: *ACM Trans. Inf. Syst. Secur.* (2009) (cit. on p. 21).

[5] *AddressSanitizerIntelMemoryProtectionExtensions*. https://github.com/google/sanitizers/wiki/AddressSanitizerIntelMemoryProtectionExtensions. Accessed: Nov, 2017 (cit. on p. 122).

[6] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. "Baggy Bounds Checking: An Efficient and Backwards-compatible Defense Against Out-of-bounds Errors". In: *Proceedings of the 18th Conference on USENIX Security Symposium (Sec)*. 2009 (cit. on pp. 10, 19 sq., 94, 97–100, 103, 105, 113, 117, 120, 128).

[7] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. "Preventing Memory Error Exploits with WIT". In: *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. 2008 (cit. on pp. 22, 97).

[8] *Amazon S3 Availability Event*. http://status.aws.amazon.com/s3-20080720.html. Accessed: Nov, 2017 (cit. on pp. 2, 26, 45, 69, 88).

[9] AMD Corporation. *AMD64 Architecture Programmer's Manual*. Vol. System Programming. 248966-030. 2012 (cit. on p. 36).

[10] Paul E. Ammann and John C. Knight. "Data Diversity: An Approach to Software Fault Tolerance". In: *IEEE Transactions on Computers* 37.4 (1988), pp. 418–425. DOI: 10.1109/12.2185 (cit. on p. 30).

[11] *Analysis of nginx 1.3.9/1.4.0 stack buffer overflow and x64 exploitation (CVE-2013-2028)*. http://www.vnsecurity.net/research/2013/05/21/analysis-of-nginx-cve-2013-2028.html. Accessed: Nov, 2017 (cit. on pp. 116, 142).

[12] *Anatomy of OpenSSL's Heartbleed: Just four bytes trigger horror bug*. http://www.theregister.co.uk/2014/04/09/heartbleed_explained/. Accessed: Nov, 2017 (cit. on pp. 11, 115, 141).

[13] Anselm R Garbe. *Static Linux*. http://sta.li/faq. Accessed: Nov, 2017 (cit. on p. 116).

[14] *Apache HTTP Server Project*. http://httpd.apache.org/. Accessed: Nov, 2017 (cit. on pp. 90, 115, 140).

[15] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. "SCONE: Secure Linux Containers with Intel SGX". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016 (cit. on pp. 94, 96, 100, 115).

[16] Todd Austin. "DIVA: A Dynamic Approach to Microprocessor Verification". In: *Journal of Instruction-Level Parallelism* 2 (2000) (cit. on p. 43).

[17] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. "Efficient detection of all pointer and array access errors". In: *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation (PLDI)*. 1994 (cit. on pp. 10, 94, 97).

[18] A. Avizienis, G. C. Gilley, F. P. Mathur, D. A. Rennels, J. A. Rohr, and D. K. Rubin. "The STAR (Self-Testing And Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design". In: *IEEE Transactions on Computers* 20.11 (1971), pp. 1312–1321. DOI: 10.1109/T-C.1971.223133 (cit. on pp. 11, 43).

[19] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. "Basic Concepts and Taxonomy of Dependable and Secure Computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (Jan. 2004) (cit. on pp. 11, 14 sq.).

[20] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. "You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2014 (cit. on p. 97).

[21] Christel Baier, Joachim Klein, Linda Leuschner, David Parker, and Sascha Wunderlich. "Ensuring the reliability of your model checker: Interval iteration for Markov Decision Processes". In: *International Conference on Computer Aided Verification (CAV)*. 2017 (cit. on p. 148).

[22] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. "Megastore: Providing Scalable, Highly Available Storage for Interactive Services". In: *Proceedings of the Conference on Innovative Data System Research (CIDR)*. 2011 (cit. on pp. 2, 16, 71).

[23] W. Bartlett and L. Spainhower. "Commercial fault tolerance: a tale of two systems". In: *Dependable and Secure Computing, IEEE Transactions on* 1.1 (2004), pp. 87–96 (cit. on pp. 2, 26, 43).

[24] Andrew Baumann, Marcus Peinado, and Galen Hunt. "Shielding applications from an untrusted cloud with Haven". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014 (cit. on p. 94).

[25] Diogo Behrens, Dmitrii Kuvaiskii, and Christof Fetzer. "HardPaxos: Replication Hardened Against Hardware Errors". In: *Proceedings of the 33rd IEEE Symposium on Reliable Distributed Systems (SRDS)*. IEEE. 2014 (cit. on pp. 39, 41).

[26] Diogo Behrens, Marco Serafini, Sergei Arnautov, Flavio P. Junqueira, and Christof Fetzer. "Scalable Error Isolation for Distributed Systems". In: *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. 2015 (cit. on pp. 70 sq., 81, 90).

[27] Emery D. Berger and Benjamin G. Zorn. "DieHard: Probabilistic Memory Safety for Unsafe Languages". In: *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2006 (cit. on pp. 19, 100, 117, 119).

[28] D. Bernick, B. Bruckert, P. Del Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. "NonStop Advanced Architecture". In: *proceedings of the International Conference on Dependable Systems and Networks (DSN)*. 2005 (cit. on pp. 16, 72).

[29] Major Bhadauria, Vincent M. Weaver, and Sally A. McKee. "Understanding PARSEC Performance on Contemporary CMPs". In: *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. 2009 (cit. on p. 59).

[30] Sandeep Bhatkar and R. Sekar. "Data Space Randomization". In: *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 2008 (cit. on pp. 22, 97).

[31] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Flavio Junqueira, and Benjamin Reed. "Reliable Data-center Scale Computations". In: *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*. 2010 (cit. on pp. 2, 70).

[32] Christian Bienia and Kai Li. "PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors". In: *MoBS*. 2009 (cit. on pp. 46, 57, 61, 77, 82, 108, 133).

[33] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. "The PARSEC Benchmark Suite: Characterization and Architectural Implications". In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2008 (cit. on p. 77).

[34] The Tor Blog. *Tor Browser 5.5a4-hardened is released*. `https://blog.torproject.org/blog/tor-browser-55a4-hardened-released`. Accessed: Nov, 2017 (cit. on p. 119).

[35] S. Borkar. "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation". In: *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (Micro)* 25.6 (2005), pp. 10–16 (cit. on pp. 25, 45, 69, 74).

[36] Ajay Brahmakshatriya, Piyus Kedia, Derrick Paul McKee, Pratik Bhatu, Deepak Garg, Akash Lal, and Aseem Rastogi. "An Instrumenting Compiler for Enforcing Confidentiality in Low-Level Code". In: *arXiv preprint arXiv:1711.11396* (2017) (cit. on p. 148).

[37] David T. Brown. "Error Detecting and Correcting Binary Codes for Arithmetic Operations". In: *Electronic Computers, IRE Transactions* EC-9.3 (1960). DOI: `10.1109/TEC.1960.5219855` (cit. on p. 27).

[38] David Brumley, Tzi-cker Chiueh, Robert Johnson, Huijia Lin, and Dawn Song. "RICH: Automatically protecting against integer-based vulnerabilities". In: *Symposium on Network and Distributed Systems Security (NDSS)*. 2007 (cit. on p. 36).

[39] Marc Brunink, Martin Susskraut, and Christof Fetzer. "Boundless Memory Allocations for Memory Safety and High Availability". In: *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. 2011 (cit. on pp. 94, 97, 100, 103).

[40] Nathan Burow, Scott Carr, Stefan Brunthaler, Mathias Payer, Joseph Nash, Per Larsen, and Michael Franz. "Control-Flow Integrity: Precision, Security, and Performance". In: *arXiv preprint arXiv:1602.04056* (2016) (cit. on pp. 21, 97).

[41] Mike Burrows. “The Chubby Lock Service for Loosely-coupled Distributed Systems”. In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*. 2006 (cit. on pp. 2, 16, 71).

[42] Cristian Cadar, Periklis Akritidis, Manuel Costa, Jean-Philippe Martin, and Miguel Castro. *Data Randomization*. Tech. rep. MSR-TR-2008-120. Microsoft Research, 2008 (cit. on pp. 22 sq., 97).

[43] Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. “Robust Architectural Support for Transactional Memory in the Power Architecture”. In: *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*. 2013 (cit. on pp. 92, 146).

[44] Miguel Castro, Manuel Costa, and Tim Harris. “Securing Software by Enforcing Data-flow Integrity”. In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*. 2006 (cit. on pp. 21, 97).

[45] Miguel Castro and Barbara Liskov. “Practical Byzantine Fault Tolerance”. In: *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*. 1999 (cit. on pp. 17, 71).

[46] *CERN openlab Explores New CPU/FPGA Processing Solutions*. `https://www.hpcwire.com/2017/04/14/xeon-fpga-processor-tested-at-cern/`. Accessed: Nov, 2017 (cit. on p. 146).

[47] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. “Paxos Made Live: An Engineering Perspective”. In: *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC)*. 2007 (cit. on pp. 2, 10, 69).

[48] Stephen Checkoway and Hovav Shacham. “Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface”. In: *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2013 (cit. on p. 97).

[49] Xi Chen, Herbert Bos, and Cristiano Giuffrida. “CodeArmor: Virtualizing the Code Space to Counter Disclosure Attacks”. In: *Proceedings of the European Symposium on Security and Privacy (EuroS&P)*. 2017 (cit. on p. 97).

[50] Zhi Chen, Alexandru Nicolau, and Alexander V. Veidenbaum. “SIMD-based Soft Error Detection”. In: *Proceedings of the ACM International Conference on Computing Frontiers (CF)*. 2016 (cit. on p. 147).

[51] Zhi Chen, Ryoichi Inagaki, Alexandru Nicolau, and Alexander Veidenbaum. “Software Fault Tolerance for FPUs via Vectorization”. In: *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. 2015 (cit. on pp. 48, 61, 147).

[52] David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. “Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine”. In: *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2015 (cit. on pp. 95, 100, 103, 130 sq.).

[53] *Cloud Computing - SME Survey*. `https://www.enisa.europa.eu/publications/cloud-computing-sme-survey`. Accessed: Nov, 2017 (cit. on p. 93).

[54] *CloudCamp: Five key concerns raised about cloud computing*. http://www.itnews.com.au/news/cloudcamp-five-key-concerns-raised-about-cloud-computing-223980. Accessed: Nov, 2017 (cit. on p. 93).

[55] *Codebases: Millions of lines of code*. http://www.informationisbeautiful.net/visualizations/million-lines-of-code/. Accessed: Nov, 2017 (cit. on p. 1).

[56] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. "Benchmarking Cloud Serving Systems with YCSB". In: *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. 2010 (cit. on pp. 62, 89).

[57] Miguel Correia, Daniel Gómez Ferro, Flavio P. Junqueira, and Marco Serafini. "Practical Hardening of Crash-tolerant Systems". In: *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*. 2012 (cit. on pp. 69, 71).

[58] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. Cryptology ePrint Archive, Report 2016/086. http://eprint.iacr.org/2016/086. 2016 (cit. on p. 96).

[59] *Coverity Scan: Open Source Report 2014*. http://go.coverity.com/rs/157-LQW-289/images/2014-Coverity-Scan-Report.pdf. Accessed: Nov, 2017 (cit. on p. 1).

[60] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. "Readactor: Practical code randomization resilient to memory disclosure". In: *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. 2015 (cit. on p. 97).

[61] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chenx, and Junfeng Yang. "Paxos Made Transparent". In: *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. 2015 (cit. on pp. 17, 71).

[62] *Data corruption with Opteron CPUs and NVidia chipsets*. https://bugzilla.kernel.org/show_bug.cgi?id=7768. Accessed: Nov, 2017 (cit. on p. 69).

[63] Dinakar Dhurjati and Vikram Adve. "Backwards-compatible array bounds checking for C with very low overhead". In: *Proceeding of the 28th international conference on Software engineering (ICSE)*. 2006 (cit. on pp. 10, 94, 97, 120).

[64] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. "SAFECode: enforcing alias analysis for weakly typed languages". In: *Proceedings of the 27th Conference on Programming Language Design and Implementation (PLDI)*. 2006 (cit. on pp. 103, 105, 117 sq., 120, 139).

[65] Björn Döbel and Hermann Härtig. "Can We Put Concurrency Back into Redundant Multithreading?" In: *Proceedings of the International Conference on Embedded Software (EMSOFT)*. 2014 (cit. on pp. 17, 48 sq., 70 sq.).

[66] Nurit Dor, Michael Rodeh, and Mooly Sagiv. "CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C". In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*. 2003 (cit. on p. 118).

[67] Gregory J. Duck and Roland H. C. Yap. "Heap bounds protection with Low Fat Pointers". In: *Proceedings of the 25th International Conference on Compiler Construction (CC)*. 2016 (cit. on pp. 99 sq.).

[68] Gregory J. Duck, Roland H. C. Yap, and Lorenzo Cavallaro. "Stack Bounds Protection with Low Fat Pointers". In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2017 (cit. on pp. 98–100).

[69] Gregory J Duck and Roland HC Yap. "EffectiveSan: Type and Memory Error Detection using Dynamically Typed C/C++". In: *arXiv preprint arXiv:1710.06125* (2017) (cit. on p. 148).

[70] EGAS Workgroup. *Standardized E-Gas Monitoring Concept for Gasoline and Diesel Engine Control Units*. Version 5.5. 2013 (cit. on p. 38).

[71] Frank Eigler. *Mudflap: pointer use checking for C/C++*. `https://gcc.gnu.org/wiki/Mudflap_Pointer_Debugging`. Accessed: Nov, 2017 (cit. on p. 120).

[72] Ilya Enkovich. *Intel(R) Memory Protection Extensions (Intel MPX) support in the GCC compiler*. `https://gcc.gnu.org/wiki/IntelMPXsupportintheGCCcompiler`. Accessed: Nov, 2017 (cit. on p. 127).

[73] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. "XFI: Software Guards for System Address Spaces". In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*. 2006 (cit. on p. 97).

[74] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. "Missing the Point(Er): On the Effectiveness of Code Pointer Integrity". In: *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP)*. 2015 (cit. on p. 21).

[75] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. "Shoestring: Probabilistic Soft Error Reliability on the Cheap". In: *Proceeding of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2010 (cit. on pp. 48, 56, 74, 81).

[76] C. Fetzer and P. Felber. "Transactional memory for dependable embedded systems". In: *Proceedings of the 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 2011 (cit. on p. 72).

[77] Christof Fetzer. "Building critical applications using microservices". In: *IEEE Security & Privacy* 14.6 (2016), pp. 86–89 (cit. on p. 148).

[78] Brad Fitzpatrick. "Distributed Caching with Memcached". In: *Linux Journal* 2004.124 (2004) (cit. on pp. 89, 114, 142).

[79] Agner Fog. "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs". In: *Copenhagen University College of Engineering* (2011) (cit. on p. 124).

[80] P. Forin. "Vital Coded Microprocessor Principles and Application for Various Transit Systems". In: *IFAC/IFIP/IFORS Symposium* (1989) (cit. on pp. 26, 43).

[81] *Funny statistics for the Linux kernel*. `https://www.linuxcounter.net/statistics/kernel`. Accessed: Nov, 2017 (cit. on p. 1).

[82] Majdi Ghadhab, Matthias Kuntz, Dmitrii Kuvaiskii, and Christof Fetzer. "A controller safety concept based on software-implemented fault tolerance for fail-operational automotive applications". In: *International Workshop on Formal Techniques for Safety-Critical Systems*. Springer. 2015, pp. 189–205 (cit. on p. 147).

[83] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. "Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors". In: *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA)*. 1990 (cit. on p. 74).

[84] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. "Enhanced operating system security through efficient and fine-grained address space randomization". In: *Proceeding of the 21st USENIX Security Symposium (Sec)*. 2012 (cit. on p. 97).

[85] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante. *Software-Implemented Hardware Fault Tolerance*. Springer, 2006 (cit. on pp. 9 sq., 26).

[86] *Google Compute Engine Incident 16007*. `https://status.cloud.google.com/incident/compute/16007`. Accessed: Nov, 2017 (cit. on p. 2).

[87] John Graham-Cumming. *Incident report on memory leak caused by Cloudflare parser bug*. `https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/`. Accessed: February, 2017 (cit. on p. 94).

[88] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. "What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems". In: *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. 2014 (cit. on pp. 45, 69).

[89] Zhenyu Guo, Chuntao Hong, Mao Yang, Lidong Zhou, Li Zhuang, and Dong Zhou. "Rex: Replication at the Speed of Multi-core". In: *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*. 2014 (cit. on pp. 17, 71).

[90] PK Gupta. *Xeon+FPGA Platform for the Data Center*. `http://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf`. Accessed: Nov, 2017 (cit. on p. 65).

[91] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. "MiBench: A Free, Commercially Representative Embedded Benchmark Suite". In: *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*. WWC '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14. DOI: `10.1109/WWC.2001.15` (cit. on p. 40).

[92] F. Haas, S. Weis, S. Metzlaff, and T. Ungerer. "Exploiting Intel TSX for fault-tolerant execution in safety-critical systems". In: *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. 2014 (cit. on p. 72).

[93] Florian Haas, Sebastian Weis, Theo Ungerer, Gilles Pokam, and Youfeng Wu. "Fault-Tolerant Execution on COTS Multi-core Processors with Hardware Transactional Memory Support". In: *International Conference on Architecture of Computing Systems (ARCS)*. 2017 (cit. on p. 147).

[94] *Hackers Are the Real Obstacle for Self-Driving Vehicles*. `https://www.technologyreview.com/s/608618/hackers-are-the-real-obstacle-for-self-driving-vehicles/`. Accessed: Nov, 2017 (cit. on p. 2).

[95] Wookhyun Han, Byunggill Joe, Byoungyoung Lee, Chengyu Song, and Insik Shin. "Enhancing Memory Error Detection for Large-Scale Applications and Fuzz Testing". In: *Symposium on Network and Distributed Systems Security (NDSS)*. 2018 (cit. on p. 148).

[96] Reed Hastings and Bob Joyce. "Purify: Fast detection of memory leaks and access errors". In: *Proceedings of the Winter 1992 USENIX Conference*. 1991 (cit. on p. 119).

[97] *Heartbleed vs malloc.conf*. `http://www.tedunangst.com/flak/post/heartbleed-vs-mallocconf`. Accessed: Nov, 2017 (cit. on p. 141).

[98] Jörg Henkel, Lars Bauer, Nikil Dutt, Puneet Gupta, Sani Nassif, Muhammad Shafique, Mehdi Tahoori, and Norbert Wehn. "Reliable On-chip Systems in the Nano-era: Lessons Learnt and Future Trends". In: *Proceedings of the Design Automation Conference (DAC)*. 2013 (cit. on p. 45).

[99] John L. Henning. "SPEC CPU2006 Benchmark Descriptions". In: *SIGARCH Computer Architecture News* (2006) (cit. on pp. 108, 133).

[100] Maurice Herlihy and J. Eliot B. Moss. "Transactional Memory: Architectural Support for Lock-free Data Structures". In: *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*. 1993 (cit. on pp. 8, 72).

[101] Johannes Hofmann, Jan Treibig, Georg Hager, and Gerhard Wellein. "Comparing the Performance of Different x86 SIMD Instruction Sets for a Medical Imaging Application on Modern Multi- and Manycore Chips". In: *Proceedings of the Workshop on Programming Models for SIMD/Vector Processing (WPMVP)*. 2014 (cit. on p. 64).

[102] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. "ZooKeeper: Wait-free Coordination for Internet-scale Systems". In: *Proceedings of the Conference on USENIX Annual Technical Conference (USENIX ATC)*. 2010 (cit. on pp. 2, 10, 16, 71).

[103] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. "Cosmic Rays Don'T Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design". In: *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2012 (cit. on pp. 1, 10).

[104] Intel. *Technology and Computing Requirements for Self-Driving Cars*. `http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/automotive-autonomous-driving-vision-paper.pdf`. Accessed: Nov, 2017 (cit. on p. 26).

[105] *Intel 64 and IA-32 Architectures Software Developer Manuals*. `https://software.intel.com/en-us/articles/intel-sdm`. Accessed: Nov, 2017 (cit. on p. 1).

[106] Intel Corporation. *Chip Shot: Intel Unleashes Next-Gen Enthusiast Desktop PC Platform at Gamescom*. `https://newsroom.intel.com/chip-shots/chip-shot-intel-unleashes-next-gen-enthusiast-desktop-pc-platform-at-gamescom/`. Accessed: Nov, 2017 (cit. on p. 120).

[107] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 248966-030. 2014 (cit. on p. 31).

[108] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 2016 (cit. on pp. 101, 124).

[109] Intel Corporation. *Introduction to Intel(R) Memory Protection Extensions*. `https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions`. Accessed: Nov, 2017 (cit. on pp. 19, 120).

[110] *Intel Memory Protection Extensions Enabling Guide (rev. 1.01)*. `https://software.intel.com/sites/default/files/managed/9d/f6/Intel_MPX_EnablingGuide.pdf`. Accessed: Nov, 2017 (cit. on pp. 10, 94, 97–100, 103, 107).

[111] *Intel Software Development Emulator (Intel SDE)*. `https://software.intel.com/en-us/articles/intel-software-development-emulator`. Accessed: Nov, 2017 (cit. on p. 81).

[112] *Intel wants to make the Atom processor the brains of the connected cars*. https://gigaom.com/2014/05/29/intel-wants-to-make-the-atom-processor-the-brains-of-the-connected-car/. Accessed: Nov, 2017 (cit. on p. 1).

[113] *Intel(R) Memory Protection Extensions Enabling Guide*. https://software.intel.com/en-us/articles/intel-memory-protection-extensions-enabling-guide. Accessed: Nov, 2017 (cit. on pp. 127 sq.).

[114] *Introduction to SPARC M7 and Silicon Secured Memory (SSM)*. https://swisdev.oracle.com/_files/What-Is-SSM.html. Accessed: Nov, 2017 (cit. on pp. 98, 117).

[115] Michael Isard. "Autopilot: Automatic Data Center Management". In: *SIGOPS Operating Systems Review* 41.2 (2007), pp. 60–67 (cit. on pp. 10, 69).

[116] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. "Cyclone: A safe dialect of C". In: *Proceedings of the USENIX 2002 Annual Technical Conference (ATC)*. 2002 (cit. on pp. 119 sq.).

[117] Richard W M Jones and Paul H J Kelly. "Backwards-compatible bounds checking for arrays and pointers in C programs". In: *Proceedings of the 3rd International Workshop on Automatic Debugging (AADEBUG)*. 1997 (cit. on p. 103).

[118] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. "CheapBFT: Resource-efficient Byzantine Fault Tolerance". In: *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*. 2012 (cit. on pp. 17, 71).

[119] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. "All About Eve: Execute-verify Replication for Multi-core Servers". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 2012 (cit. on pp. 17, 71).

[120] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. "Countering Code-injection Attacks with Instruction-set Randomization". In: *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*. 2003 (cit. on p. 23).

[121] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. "Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software". In: *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*. 2006 (cit. on p. 97).

[122] Sangman Kim, Michael Z. Lee, Alan M. Dunn, Owen S. Hofmann, Xuan Wang, Emmett Witchel, and Donald E. Porter. "Improving Server Applications with System Transactions". In: *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*. 2012 (cit. on pp. 17, 71).

[123] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors". In: *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 2014 (cit. on pp. 26, 30).

[124] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. "Zyzzyva: Speculative Byzantine Fault Tolerance". In: *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. 2007 (cit. on pp. 17, 71).

[125] Dmitrii Kuvaiskii and Christof Fetzer. "Delta-encoding: Practical Encoded Processing". In: *Proceedings of The 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2015 (cit. on p. 25).

[126] Dmitrii Kuvaiskii, Oleksii Oleksenko, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. "Elzar: Triple Modular Redundancy using Intel AVX". In: *Proceedings of the 46nd International Conference on Dependable Systems and Networks (DSN)*. 2016 (cit. on pp. 45, 70, 74).

[127] Dmitrii Kuvaiskii, Rasha Faqeh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. "HAFT: Hardware-Assisted Fault Tolerance". In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. 2016 (cit. on pp. 46, 69).

[128] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. "SGXBounds: Memory Safety for Shielded Execution". In: *Proceedings of the 2017 ACM European Conference on Computer Systems (EuroSys)*. 2017 (cit. on pp. 93, 117, 124, 129).

[129] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. "Code-pointer Integrity". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 2014 (cit. on pp. 20, 97).

[130] Marta Kwiatkowska, Gethin Norman, and David Parker. "PRISM: Probabilistic Model Checking for Performance and Reliability Analysis". In: *SIGMETRICS Performance Evaluation Review* 36.4 (2009), pp. 40–45 (cit. on p. 82).

[131] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight Jr., and Andre DeHon. "Low-fat Pointers: Compact Encoding and Efficient Gate-level Implementation of Fat Pointers for Spatial Safety and Capability-based Security". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2013 (cit. on pp. 98, 113, 117, 119).

[132] Leslie Lamport. "How to make a correct multiprocess program execute correctly on a multiprocessor". In: *IEEE Transactions on Computers* C-28.9 (1997), pp. 690–691 (cit. on p. 74).

[133] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. "SoK: Automated Software Diversity". In: *Proceedings of the 35th IEEE Symposium on Security and Privacy (SP)*. 2014 (cit. on pp. 20, 97).

[134] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation". In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. 2004 (cit. on pp. 54, 79 sq., 106).

[135] Viktor Leis, Alfons Kemper, and Thomas Neumann. "Exploiting hardware transactional memory in main-memory databases". In: *Proceedings of the 30th International Conference on Data Engineering (ICDE)*. 2014 (cit. on pp. 72 sq.).

[136] *LevelDB key-value storage library*. `https://github.com/google/leveldb`. Accessed: Nov, 2017 (cit. on p. 90).

[137] Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. "Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design". In: *SIGOPS Operating Systems Review* 42.2 (2008), pp. 265–276. DOI: `10.1145/1353535.1346315` (cit. on p. 26).

[138] *libMemcached*. http://libmemcached.org/. Accessed: Nov, 2017 (cit. on p. 114).

[139] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. "Dthreads: Efficient Deterministic Multithreading". In: *Proceedings of the 23d ACM Symposium on Operating Systems Principles (SOSP)*. 2011 (cit. on pp. 17, 71).

[140] *LLVM Atomic Instructions and Concurrency Guide*. http://llvm.org/docs/Atomics.html. Accessed: Nov, 2017 (cit. on p. 74).

[141] *LogCabin Distributed Storage System*. https://github.com/logcabin/logcabin. Accessed: Nov, 2017 (cit. on p. 90).

[142] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P Chung, Taesoo Kim, and Wenke Lee. "ASLR-Guard: Stopping address space leakage for code reuse attacks". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2015 (cit. on p. 97).

[143] Vitaliy B. Lvin, Gene Novark, Emery D. Berger, and Benjamin G. Zorn. "Archipelago: Trading Address Space for Reliability and Security". In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2008 (cit. on pp. 100, 119).

[144] R.E. Lyons and W. Vanderkulk. "The Use of Triple-Modular Redundancy to Improve Computer Reliability". In: *IBM Journal of Research and Development*. 1962 (cit. on pp. 16, 48).

[145] Thomas M Chen and Jean-Marc Robert. "The Evolution of Viruses and Worms". In: (Dec. 2004) (cit. on p. 11).

[146] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. "CCFI: Cryptographically Enforced Control Flow Integrity". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2015 (cit. on pp. 21, 97).

[147] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. "Innovative Instructions and Software Model for Isolated Execution". In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. 2013 (cit. on pp. 94, 96).

[148] Paul E. McKenney, Maged M. Michael, Josh Triplett, and Jonathan Walpole. "Why the Grass May Not Be Greener on the Other Side: A Comparison of Locking vs. Transactional Memory". In: *SIGOPS Operating Systems Review* 44.3 (2010), pp. 93–101 (cit. on p. 72).

[149] A. Meixner, M.E. Bauer, and D.J. Sorin. "Argus: Low-Cost, Comprehensive Error Detection in Simple Cores". In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (Micro)*. 2007, pp. 210–222 (cit. on p. 43).

[150] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. "Into the Depths of C: Elaborating the De Facto Standards". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2016 (cit. on pp. 97, 130).

[151] *Memcached bug: CVE-2011-4971*. http://www.cvedetails.com/cve/cve-2011-4971. Accessed: Nov, 2017 (cit. on pp. 94, 114, 142 sq.).

[152] Microsoft Research. *Checked C*. https://www.microsoft.com/en-us/research/project/checked-c/. Accessed: Nov, 2017 (cit. on pp. 117, 119).

[153] S.S. Mukherjee, M. Kontz, and S.K. Reinhardt. “Detailed design and evaluation of redundant multi-threading alternatives”. In: *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 2002 (cit. on pp. 17, 43, 48, 71).

[154] *musl libc*. `http://www.musl-libc.org/`. Accessed: Nov, 2017 (cit. on p. 81).

[155] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. “Everything You Want to Know About Pointer-Based Checking”. In: *Proceedings of the 1st Summit on Advances in Programming Languages (SNAPL)*. 2015 (cit. on pp. 94, 97, 118 sq.).

[156] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. “WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking”. In: *Proceedings of the 2014 International Symposium on Code Generation and Optimization (CGO)*. 2014 (cit. on pp. 19 sq., 119 sq.).

[157] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. “CETS: Compiler Enforced Temporal Safety for C”. In: *Proceedings of the 2010 International Symposium on Memory Management (ISMM)*. 2010 (cit. on pp. 19, 103, 120, 122).

[158] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. “SoftBound: Highly Compatible and Complete Spatial Memory Safety for C”. In: *Proceedings of the 30th Conference on Programming Language Design and Implementation (PLDI)*. 2009 (cit. on pp. 19, 97, 100, 103, 117 sq., 120, 122, 139).

[159] R. Nathan and D.J. Sorin. “Nostradamus: Low-cost hardware-only error detection for processor cores”. In: *Design, Automation and Test in Europe Conference and Exhibition (DATE)*. 2014, pp. 1–6 (cit. on p. 43).

[160] National Institute of Standards and Technology. *National Vulnerability Database*. `https://web.nvd.nist.gov`. Accessed: Nov, 2017 (cit. on p. 117).

[161] George C. Necula, Scott McPeak, Westley Weimer, George C. Necula, Scott McPeak, and Westley Weimer. “CCured”. In: *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*. 2002 (cit. on pp. 117, 119 sq.).

[162] Nicholas Nethercote and Julian Seward. “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation”. In: *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*. 2007 (cit. on pp. 10, 94, 97 sq., 117, 119).

[163] *New defective S3 load balancer corrupts relayed messages*. `https://forums.aws.amazon.com/thread.jspa?threadID=22709`. Accessed: Nov, 2017 (cit. on pp. 2, 45, 69).

[164] *nginx: The Architecture of Open Source Applications*. `http://www.aosabook.org/en/nginx.html`. Accessed: Nov, 2017 (cit. on pp. 94, 115, 141).

[165] Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. “Cycles, Cells and Platters: An Empirical Analysis of Hardware Failures on a Million Consumer PCs”. In: *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*. 2011 (cit. on pp. 1, 26, 45, 69).

[166] Ashish Misra Niranjan Hasabnis and R. Sekar. “Light-weight Bounds Checking”. In: *Proceedings of the 2012 ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*. 2012 (cit. on pp. 97 sq., 117, 119).

[167] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. "Exterminator: Automatically Correcting Memory Errors with High Probability". In: *Communications of ACM* (2008) (cit. on p. 119).

[168] Thomas Nyman, Ghada Dessouky, Shaza Zeitouni, Aaro Lehikoinen, Andrew Paverd, N Asokan, and Ahmad-Reza Sadeghi. "Hardscope: Thwarting DOP with hardware-assisted run-time scope enforcement". In: *arXiv preprint arXiv:1705.10295* (2017) (cit. on p. 148).

[169] N. Oh, S. Mitra, and E.J. McCluskey. "ED4I: error detection by diverse data and duplicated instructions". In: *IEEE Transactions on Computers* 51.2 (2002), pp. 180–199. DOI: 10.1109/12.980007 (cit. on p. 43).

[170] N. Oh, P.P. Shirvani, and E.J. McCluskey. "Error detection by duplicated instructions in super-scalar processors". In: *IEEE Transactions on Reliability* 51.1 (2002), pp. 63–75. DOI: 10.1109/24.994913 (cit. on pp. 17 sq., 29, 42 sq., 46, 48, 51, 74).

[171] Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. "Observing and Preventing Leakage in MapReduce". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2015 (cit. on p. 97).

[172] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, and Christof Fetzer. "Fex: A Software Systems Evaluator". In: *Proceedings of the 47st International Conference on Dependable Systems & Networks (DSN)*. 2017 (cit. on pp. 108, 132).

[173] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. "Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches". In: *arXiv:1702.00719* (2017) (cit. on pp. 103, 108, 113 sq., 117, 148).

[174] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. "Pig Latin: A Not-so-foreign Language for Data Processing". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2008 (cit. on p. 70).

[175] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. "Kendo: Efficient Deterministic Multithreading in Software". In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2009 (cit. on pp. 17, 71).

[176] Diego Ongaro and John Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *Proceedings of the USENIX Conference on USENIX Annual Technical Conference (USENIX ATC)*. 2014 (cit. on pp. 10, 90).

[177] H. Orman. "The Morris worm: a fifteen-year perspective". In: *IEEE Security Privacy* 1.5 (2003), pp. 35–43 (cit. on p. 11).

[178] Lukas Osinski, Tobias Langer, and Juergen Mottok. "A survey of fault tolerance approaches on different architecture levels". In: *ARCS 2017; 30th International Conference on Architecture of Computing Systems; Proceedings of*. VDE. 2017, pp. 1–9 (cit. on p. 147).

[179] Mike Owens and Grant Allen. *SQLite*. Springer, 2010 (cit. on p. 91).

[180] David A Patterson, Garth Gibson, and Randy H Katz. *A case for redundant arrays of inexpensive disks (RAID)*. Proceedings of the ACM SIGMOD international conference on Management of data (SIGMOD), 1988 (cit. on p. 10).

[181] Daniel Porto, João Leitão, Cheng Li, Allen Clement, Aniket Kate, Flavio Junqueira, and Rodrigo Rodrigues. "Visigoth Fault Tolerance". In: *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*. 2015 (cit. on pp. 17, 71).

[182] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. "Designing Distributed Systems Using Approximate Synchrony in Data Center Networks". In: *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2015 (cit. on pp. 17, 71).

[183] Alexander Potapenko. *AddressSanitizerIntraObjectOverflow*. `https://github.com/google/sanitizers/wiki/AddressSanitizerIntraObjectOverflow`. Accessed: Nov, 2017 (cit. on p. 127).

[184] Milos Prvulovic, Zheng Zhang, and Josep Torrellas. "ReVive: Cost-effective Architectural Support for Rollback Recovery in Shared-memory Multiprocessors". In: *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*. 2002 (cit. on p. 72).

[185] *Quantifying the Impact of Cloudbleed*. `https://blog.cloudflare.com/quantifying-the-impact-of-cloudbleed/`. Accessed: Nov, 2017 (cit. on pp. 11, 18).

[186] Ravi Rajwar and James R. Goodman. "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution". In: *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (Micro)*. 2001 (cit. on p. 79).

[187] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. "Evaluating MapReduce for Multi-core and Multiprocessor Systems". In: *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*. 2007 (cit. on pp. 46, 57, 82, 108, 133).

[188] M. Rebaudengo, M.S. Reorda, M. Violante, and Marco Torchiano. "A source-to-source compiler for generating dependable software". In: *First IEEE International Workshop on Source Code Analysis and Manipulation*. 2001, pp. 33–42 (cit. on p. 35).

[189] Steven K. Reinhardt and Shubhendu S. Mukherjee. "Transient Fault Detection via Simultaneous Multithreading". In: *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*. 2000 (cit. on p. 30).

[190] George A. Reis, Jonathan Chang, and David I. August. "Automatic Instruction-Level Software-Only Recovery". In: *In proceedings of Annual IEEE/ACM International Symposium on Microarchitecture (Micro)*. 2007 (cit. on pp. 46 sq., 49, 51, 56, 61, 72, 81).

[191] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. "SWIFT: Software Implemented Fault Tolerance". In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. 2005 (cit. on pp. 17 sq., 29, 43, 46, 48–51, 56, 70, 74 sq., 77, 81).

[192] Elena Reshetova, Hans Liljestrand, Andrew Paverd, and N Asokan. "Towards Linux Kernel Memory Safety". In: *arXiv preprint arXiv:1710.06175* (2017) (cit. on p. 148).

[193] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, and T. Leu. "A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors)". In: *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC)*. 2004 (cit. on pp. 95, 104).

[194] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee Jr. "Enhancing Server Availability and Security Through Failure-oblivious Computing". In: *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation (OSDI)*. 2004 (cit. on pp. 95, 104).

[195] Roman Shaposhnik. *What does dynamic linking and communism have got in common?* `https://blogs.oracle.com/rvs/entry/what_does_dynamic_linking_and`. Accessed: Nov, 2017 (cit. on p. 116).

[196] Olatunji Ruwase and Monica S Lam. "A Practical Dynamic Buffer Overflow Detector." In: *Proceeding of the Network and Distributed System Security Symposium (NDSS)*. 2004 (cit. on pp. 97, 120).

[197] G.P. Saggese, N.J. Wang, Z.T. Kalbarczyk, S.J. Patel, and R.K. Iyer. "An experimental study of soft errors in microprocessors". In: *In proceedings of Annual IEEE/ACM International Symposium on Microarchitecture (Micro)*. 2005 (cit. on pp. 30, 45, 78).

[198] Ute Schiffel. "Hardware Error Detection Using AN-Codes". PhD thesis. Technische Universität Dresden, 2011 (cit. on pp. 26 sq., 29, 34, 42 sq.).

[199] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. "ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software". In: *Computer Safety, Reliability, and Security*. Ed. by Erwin Schoitsch. Vol. 6351. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, pp. 169–182. DOI: `10.1007/978-3-642-15651-9_13` (cit. on p. 28).

[200] H. Schirmeier, M. Hoffmann, R. Kapitza, D. Lohmann, and O. Spinczyk. "FailStar: Towards a versatile fault-injection experiment framework". In: *Proceedings of the Architecture of Computing Systems (ARCS)*. 2012 (cit. on p. 81).

[201] Fred B. Schneider. "Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial". In: *ACM Computing Surveys* 22.4 (1990), pp. 299–319 (cit. on pp. 2, 16, 71).

[202] Bianca Schroeder, Garth Gibson, et al. "A large-scale study of failures in high-performance computing systems". In: *IEEE Transactions on Dependable and Secure Computing (TDSC)*. 2010 (cit. on pp. 45, 69).

[203] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. "DRAM Errors in the Wild: A Large-scale Field Study". In: *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. 2009 (cit. on pp. 1, 25, 30).

[204] *Securing the Future of Authentication with ARM TrustZone-based Trusted Execution Environment and Fast Identity Online (FIDO)*. `https://www.arm.com/files/pdf/TrustZone-and-FIDO-white-paper.pdf`. Accessed: Nov, 2017 (cit. on p. 94).

[205] Alexander Senier, Martin Beck, and Thorsten Strufe. "PrettyCat: Adaptive guarantee-controlled software partitioning of security protocols". In: *arXiv preprint arXiv:1706.04759* (2017) (cit. on p. 148).

[206] Jaebaek Seo, Byoungyoung Lee, Sungmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. "SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs". In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2017 (cit. on p. 97).

[207] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. "AddressSanitizer: A Fast Address Sanity Checker". In: *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*. 2012 (cit. on pp. 10, 19, 94, 97 sq., 100, 102 sq., 105, 109, 117–119, 143).

[208] *Serious Linux kernel security bug fixed*. `http://www.zdnet.com/article/serious-linux-kernel-security-bug-fixed/`. Accessed: Nov, 2017 (cit. on p. 2).

[209] *SGXBounds: memory safety for shielded execution (blog post)*. `https://blog.acolyer.org/2017/06/06/sgxbounds-memory-safety-for-shielded-execution/`. Accessed: Nov, 2017 (cit. on p. 148).

[210] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. "On the Effectiveness of Address-space Randomization". In: *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*. 2004 (cit. on p. 97).

[211] Muhammad Shafique, Siddharth Garg, Jörg Henkel, and Diana Marculescu. "The EDA Challenges in the Dark Silicon Era: Temperature, Reliability, and Variability Perspectives". In: *Proceedings of the Design Automation Conference (DAC)*. 2014 (cit. on pp. 1, 45, 69).

[212] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. "Preventing Page Faults from Telling Your Secrets". In: *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security (AsiaCCS)*. 2016 (cit. on p. 97).

[213] P. Shivakumar, M. Kistler, S.W. Keckler, D. Burger, and L. Alvisi. "Modeling the effect of technology trends on the soft error rate of combinational logic". In: *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. 2002 (cit. on p. 69).

[214] A. Shye, T. Moseley, V.J. Reddi, J. Blomstedt, and D.A. Connors. "Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance". In: *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. 2007 (cit. on pp. 17, 48, 70 sq.).

[215] Daniel P. Siewiorek, G. Bell, and A. C. Newell. *Computer Structures: Principles and Examples*. New York, NY, USA: McGraw-Hill, Inc., 1982 (cit. on p. 11).

[216] *SigOPS: Computer Architecture Reading Group at KAIST*. `http://sigops.kaist.ac.kr/`. Accessed: Nov, 2017 (cit. on p. 148).

[217] Matthew S. Simpson and Rajeev K. Barua. "MemSafe: Ensuring the Spatial and Temporal Memory Safety of C at Runtime". In: *Software — Practice and Experience* (2013) (cit. on pp. 19, 120).

[218] Jared C. Smolens, Brian T. Gold, Babak Falsafi, and James C. Hoe. "Reunion: Complexity-Effective Multicore Redundancy". In: *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (Micro)*. 2006 (cit. on p. 72).

[219] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. "Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization". In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP)*. 2013 (cit. on p. 97).

[220] *Software Bug, Cascading Failures Caused Amazon Outage*. `http://www.datacenterknowledge.com/archives/2012/10/27/cascading-failures-caused-amazon-outage`. Accessed: Nov, 2017 (cit. on p. 2).

[221] *Software optimization resources*. http://www.agner.org/optimize. Accessed: Nov, 2017 (cit. on p. 1).

[222] Y. J. Song, Flavio P. Junqueira, and Benjamin Reed. "BFT for the skeptics". In: *BFTW3*. 2009 (cit. on pp. 2, 46, 71, 148).

[223] Daniel J. Sorin, Milo M. K. Martin, Mark D. Hill, and David A. Wood. "SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery". In: *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*. 2002 (cit. on p. 72).

[224] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. "Breaking the Memory Secrecy Assumption". In: *Proceedings of the Second European Workshop on System Security (EUROSEC)*. 2009 (cit. on p. 97).

[225] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "SoK: Eternal War in Memory". In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP)*. 2013 (cit. on pp. 10 sq., 18–21, 23, 94, 97, 100, 118 sq.).

[226] Jun Tang, Yong Cui, Qi Li, Kui Ren, Jiangchuan Liu, and Rajkumar Buyya. "Ensuring Security and Privacy Preservation for Cloud Data Services". In: *ACM Computing Surveys* (2016) (cit. on p. 93).

[227] *The Heartbleed Bug*. http://heartbleed.com/. Accessed: Nov, 2017 (cit. on pp. 11, 14, 18, 94, 115, 140 sq.).

[228] *This Car Runs on Code*. https://spectrum.ieee.org/transportation/systems/this-car-runs-on-code. Accessed: Nov, 2017 (cit. on p. 1).

[229] Ulrich Drepper. *Static Linking Considered Harmful*. https://www.akkadia.org/drepper/no_static_linking.html. Accessed: Nov, 2017 (cit. on p. 116).

[230] *US aviation authority: Boeing 787 bug could cause loss of control*. https://www.theguardian.com/business/2015/may/01/us-aviation-authority-boeing-787-dreamliner-bug-could-cause-loss-of-control. Accessed: Nov, 2017 (cit. on p. 2).

[231] *uthash: Hash Table for C Structures*. https://troydhanson.github.io/uthash/. Accessed: Nov, 2017 (cit. on p. 106).

[232] Various authors. *Dynamic Linking*. http://harmful.cat-v.org/software/dynamic-linking/. Accessed: Nov, 2017 (cit. on p. 116).

[233] Victor van der Veen, Nitish Dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. "Memory Errors: The Past, the Present, and the Future". In: *Proceedings of the 15th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 2012 (cit. on pp. 10, 97, 117).

[234] Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. "MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging". In: *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA)*. 2007 (cit. on p. 119).

[235] G.S. Veronese, M. Correia, A.N. Bessani, Lau Cheuk Lung, and P. Verissimo. "Efficient Byzantine Fault-Tolerance". In: *IEEE Transactions on Computers* 62.1 (2013), pp. 16–30 (cit. on pp. 17, 71).

[236] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. "High System-Code Security with Low Overhead". In: *Proceedings of the 2015 Symposium on Security and Privacy (SP)*. 2015 (cit. on p. 119).

[237] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. "Efficient Software-based Fault Isolation". In: *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP)*. 1993 (cit. on p. 97).

[238] Cheng Wang, H. Kim, Y. Wu, and V. Ying. "Compiler-Managed Software-based Redundant Multi-Threading for Transient Fault Detection". In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. 2007 (cit. on p. 71).

[239] N.J. Wang and S.J. Patel. "ReStore: symptom based soft error detection in microprocessors". In: *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. 2005 (cit. on p. 43).

[240] Zhaoguo Wang, Hao Qian, Jinyang Li, and Haibo Chen. "Using restricted transactional memory to build a scalable in-memory database". In: *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*. 2014 (cit. on pp. 72 sq.).

[241] Jiesheng Wei, A. Thomas, Guanpeng Li, and K. Pattabiraman. "Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults". In: *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2014 (cit. on p. 81).

[242] *What is the function of the 10,000 sensors of a Boeing 787?* `https://good-planes.com/civil-aircraft/boeing-787-dreamliner/what-is-the-function-of-the-10000-sensors-of-a-boeing-787/7839/`. Accessed: Nov, 2017 (cit. on p. 1).

[243] Wikipedia. *2009-11 Toyota vehicle recalls*. `http://en.wikipedia.org/wiki/2009-11_Toyota_vehicle_recalls`. Accessed: Nov, 2017 (cit. on p. 26).

[244] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. "RIPE: Runtime Intrusion Prevention Evaluator". In: *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*. 2011 (cit. on pp. 94, 112, 137).

[245] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. "The CHERI Capability Model: Revisiting RISC in an Age of Risk". In: *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA)*. 2014 (cit. on pp. 98, 119).

[246] Yichen Xie, Andy Chou, and Dawson Engler. "ARCHER : Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors". In: *ACM SIGSOFT Software Engineering Notes* (2003) (cit. on pp. 117 sq.).

[247] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. "Ad Hoc Synchronization Considered Harmful". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 2010 (cit. on p. 74).

[248] Wei Xu, Daniel C. DuVarney, and R Sekar. "An efficient and backwards-compatible transformation to ensure memory safety of C programs". In: *ACM SIGSOFT Software Engineering Notes* (2004) (cit. on p. 97).

[249] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems". In: *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP)*. 2015 (cit. on p. 97).

[250] G. Yalcin, A. Cristal, O. Unsal, A. Sobe, D. Harmanci, P. Felber, A. Voronin, J.-T. Wamhoff, and C. Fetzer. "Combining Error Detection and Transactional Memory for Energy-Efficient Computing below Safe Operation Margins". In: *Proceedings of the 22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. 2014 (cit. on p. 72).

[251] Gulay Yalcin, Osman Sabri Unsal, and Adrian Cristal. "Fault Tolerance for Multi-threaded Applications by Leveraging Hardware Transactional Memory". In: *Proceedings of the ACM International Conference on Computing Frontiers (CF)*. 2013 (cit. on p. 72).

[252] Junfeng Yang, Ang Cui, Sal Stolfo, and Simha Sethumadhavan. "Concurrency Attacks". In: *Proceedings of the 4th Conference on Hot Topics in Parallelism (HotPar)*. 2012 (cit. on p. 132).

[253] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. "How Do Fixes Become Bugs?" In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE)*. 2011 (cit. on p. 2).

[254] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. "Performance Evaluation of Intel Transactional Synchronization Extensions for High-performance Computing". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. 2013 (cit. on pp. 70, 72 sq.).

[255] Jing Yu, Maria Jesus Garzaran, and Marc Snir. "ESoftCheck: Removal of Non-vital Checks for Fault Tolerance". In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. 2009 (cit. on pp. 18, 48, 56).

[256] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. "Practical Control Flow Integrity and Randomization for Binary Executables". In: *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. 2013 (cit. on p. 97).

[257] Yun Zhang, Jae W. Lee, Nick P. Johnson, and David I. August. "DAFT: Decoupled Acyclic Fault Tolerance". In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2010 (cit. on pp. 17, 48, 70 sq.).

[258] Yun Zhang, Soumyadeep Ghosh, Jialu Huang, Jae W. Lee, Scott A. Mahlke, and David I. August. "Runtime Asynchronous Fault Tolerance via Speculation". In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. 2012 (cit. on pp. 17, 48, 70 sq.).