Hardware-Software Co-design for Practical Memory Safety

Mohamed Tarek Bnziad Mohamed Hassan

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
under the Executive Committee
of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2022

© 2022

Mohamed Tarek Bnziad Mohamed Hassan

All Rights Reserved

Abstract

Hardware-Software Co-design for Practical Memory Safety

Mohamed Tarek Bnziad Mohamed Hassan

A vast amount of software, from low-level systems code to high-performance applications, is written in memory-unsafe languages such as C and C++. The lack of memory safety in C/C++ can lead to severe consequences; a simple buffer overflow can result in code or data corruption anywhere in the program memory [1]. The problem is even worse in systems that constantly operate on inputs of unknown trustworthiness. For example, in 2021 a memory safety vulnerability was discovered in `sudo`, a near-ubiquitous utility available on major Unix-like operating systems [2]. The vulnerability, which remained silent for over 10 years, allows any unprivileged user to gain root privileges on a victim machine using a default `sudo` configuration. As memory-safe languages are unlikely to displace C/C++ in the near future, efficient memory safety mechanisms for both existing and future C/C++ code are needed.

Both industry and academia have proposed various techniques to address the C/C++ memory safety problem over the last three decades, either by software-only or hardware-assisted solutions. Software-only techniques such as Google's AddressSanitizer [3] are used to detect memory errors during the testing phase before products are shipped. While sanitizers have been shown to be effective at detecting memory errors with little effort, they typically suffer from high runtime overheads and increased memory footprint. Hardware-assisted solutions such as Oracle's Application Data Integrity (ADI) [4] and ARM's Memory Tagging Extension (MTE) [5] have much lower performance overheads, but they do not offer complete protection. Academic proposals manage to

minimize the performance costs of memory safety defenses while maintaining fine-grained security protection. Unfortunately, state-of-the-art solutions require complex metadata that increases the program memory footprint, complicates the hardware design, and breaks compatibility with the rest of the system (e.g., unprotected libraries).

To address these problems, the research within this thesis innovates in the realm of compiler transformations and hardware extensions to improve the state of the art in memory safety solutions. Specifically, this thesis shows that ***leveraging common software trends and rethinking computer microarchitectures can efficiently circumvent the problems of traditional memory safety solutions for C and C++***. First, I present a novel cache line formatting technique, dubbed Califorms [6]. Califorms builds on a concept called memory blocklisting, which prohibits a program from accessing certain memory regions based on program semantics. State-of-the-art hardware-assisted memory blocklisting, while much faster than software blocklisting, creates memory fragmentation for each use of the blocklisted location. To prevent this issue, Califorms encodes the metadata, which is used to identify the blocklisted locations, in the blocklisted (i.e., dead) locations themselves. This inlined metadata can be then integrated into the microarchitecture by changing the cache line format. As a result, both the metadata and data are fetched together, eliminating the need for extra memory accesses. Hence, Califorms reduces the performance overheads of memory safety while providing byte-granular protection and maintaining very low hardware overheads.

Secondly, I explore how leveraging common software trends can reduce the performance and memory costs of memory permitlisting (also known as base & bounds). Thus, I present No-FAT [7], a novel technique for enforcing spatial and temporal memory safety. The key observation that enables No-FAT is the increasing adoption of binning allocators. No-FAT, when used with a binning allocator, is able to *implicitly* derive an allocation's bounds information (i.e., the base address and size) from the pointer itself without relying on expensive metadata. Moreover, as No-FAT's memory instructions are aware of allocation bounds information, No-FAT effectively mitigates certain speculative attacks (e.g., Spectre-V1, which is also known as bounds checking bypass) with no additional cost. While No-FAT successfully detects memory safety violations, it falls

short against physical attacks. Hence, I propose C-5 [8], an architecture that complements No-FAT with strong data encryption. C-5 strictly uses access control in the L1 cache and encrypts program data at the L1-L2 cache interface. As a result, C-5 mitigates both in-process and physical attacks without burdening system performance.

In addition to memory blocklisting and permitlisting, a cost-effective way to alleviate the memory safety threats is by deploying exploit mitigation techniques (e.g., Intel's CET [9] and ARM's PAC [10]). Unfortunately, current exploit mitigations offer incomplete security protection in order to save on performance. This thesis investigates potential opportunities to boost the security guarantees of exploit mitigations while maintaining their low overheads. Thus, I present ZeRØ [11], a hardware primitive that preserves pointer integrity at no performance cost, effectively mitigating pointer manipulation attacks such as ROP, COP, JOP, COOP, and DOP. ZeRØ proposes unique memory instructions and a novel metadata encoding scheme to protect code and data pointers from memory safety violations. The combination of instructions and metadata allows ZeRØ to avoid explicitly tagging every word in memory. On 64-bit systems, ZeRØ encodes the pointer type and location in the currently unused upper pointer bits. This way ZeRØ reduces the performance overheads of enforcing pointer integrity to zero while requiring simple hardware modifications.

Finally, although current mitigation techniques excel at providing efficient protection for high-end devices, they typically suffer from significant performance and energy overheads when ported to the embedded domain. As a result, there is a need for developing new defenses that (1) have low overheads, (2) provide high security coverage, and (3) are especially designed for embedded devices. To achieve these goals I present EPI, an efficient pointer integrity mechanism that is tailored to microcontrollers and embedded devices [12]. Similar to ZeRØ, EPI assigns unique tags to different program assets and uses unique memory instructions for accessing them. However, EPI uses a 32-bit friendly encoding scheme to inline the tags within the program data. EPI introduces runtime overheads of less than 1%, making it viable for embedded and low-resource systems.

Table of Contents

List of Figures viii

List of Tables ix

Abbreviations x

Publications xi

Acknowledgments xiii

Dedication xv

Preface xvi

I Introduction **1**

Chapter 1: Introduction 2

1.1 What Is Memory Safety? 2

1.2 Why Is Memory Safety Still A Concern? 3

1.2.1 Memory-Safe Languages 4

1.2.2 Testing & Verification 5

1.2.3 Hardware-Based Mitigations 5

1.3 Why Are Current Hardware-Based Solutions Impractical? 7

1.4 Thesis Statement & Contributions 8

1.4.1 Califorms 8

1.4.2 No-FAT 9

1.4.3 C-5 10

1.4.4 ZeRØ 10

1.4.5 EPI 11

1.5 Thesis Organization 12

Chapter 2: Background 13

2.1 Program Components . . . 13
2.1.1 Code . . . 14
2.1.2 Globals . . . 14
2.1.3 Stack . . . 14
2.1.4 Heap . . . 14
2.2 Memory Safety Definition . . . 15
2.2.1 Spatial Memory Safety . . . 16
2.2.2 Temporal Memory Safety . . . 16
2.2.3 Type Memory Safety . . . 16
2.3 Memory Safety Attacks . . . 17
2.3.1 Code Corruption . . . 17
2.3.2 Control-Flow Hijacking . . . 17
2.3.3 Data-Flow Hijacking . . . 18
2.3.4 Data Corruption . . . 18

II Rethinking Microarchitectures For Efficient Memory Blocklisting 19

Chapter 3: Cache Line Formats . . . 20
3.1 Motivation . . . 22
3.2 System Overview . . . 24
3.3 Architecture Support . . . 26
3.3.1 BLOC Instruction . . . 26
3.3.2 Privileged Exceptions . . . 27
3.4 Microarchitecture Design . . . 27
3.4.1 L1 Cache: Bit Vector Approach . . . 27
3.4.2 L2 Cache and Beyond: Sentinel Approach . . . 29
3.4.3 L1 to/from L2 Califorms Conversion . . . 32
3.4.4 Load/Store Queue Modifications . . . 33
3.5 Software Design . . . 34
3.5.1 Dynamic Memory Management . . . 34
3.5.2 Compiler Support . . . 34
3.5.3 Operating System Support . . . 35
3.6 Security Analysis . . . 36
3.6.1 Threat Model . . . 36
3.6.2 Hardware Attacks and Mitigations . . . 36
3.6.3 Software Attacks and Mitigations . . . 37

3.7 Evaluation . . . 39
3.7.1 Hardware Overheads . . . 39
3.7.2 Software Performance Overheads . . . 41
3.8 Summary . . . 44

III Leveraging Current Software Trends For Efficient Memory Permitlisting 46

Chapter 4: Architectural Support for Low Overhead Memory Safety Checks . . . 47
4.1 Motivation . . . 48
4.2 System Overview . . . 49
4.2.1 Preliminaries . . . 49
4.2.2 How does No-FAT provide Inter-allocation Spatial Memory Safety? . . . 50
4.2.3 How does No-FAT provide Intra-allocation Spatial Memory Safety? . . . 52
4.2.4 How does No-FAT provide Temporal Memory Safety? . . . 53
4.2.5 Handling Procedure Calls and Nested Pointers . . . 54
4.3 Architecture Support . . . 56
4.4 Microarchitecture Design . . . 57
4.4.1 `MAST` . . . 57
4.4.2 Bounds Checking Module . . . 57
4.4.3 Base Computing Module . . . 58
4.4.4 Dedicated Register File . . . 59
4.5 Software Design . . . 59
4.5.1 Dynamic Memory Management . . . 59
4.5.2 Compiler Support . . . 60
4.5.3 Operating System Support . . . 61
4.6 Security Analysis . . . 61
4.6.1 Threat Model . . . 62
4.6.2 Security Discussion . . . 62
4.6.3 Spectre-V1 Resiliency . . . 64
4.6.4 Deployment Considerations . . . 66
4.7 Evaluation . . . 68
4.7.1 Hardware Overheads . . . 68
4.7.2 Software Performance Overheads . . . 69
4.7.3 Software Memory Overheads . . . 71
4.7.4 Temporal Memory Safety Analysis . . . 72

4.7.5 Buf2Ptr Analysis . . . 73
4.8 Summary . . . 74

Chapter 5: A Counter C-4 Architecture . . . 75
5.1 Motivation . . . 76
5.2 How Does Cryptographic Capability Computing (C^3) work? . . . 76
5.2.1 The Object Life-cycle in C^3 . . . 77
5.2.2 C^3's Design Choices . . . 79
5.2.3 Threat Model . . . 79
5.3 C-4: Assessing the Security Guarantees of Cryptographic Capability Computing . . . 80
5.3.1 Attack #1: Exploiting the XOR-based Data Encryption . . . 80
5.3.2 Attack #2: Targeting Spatial Memory Safety . . . 82
5.3.3 Attack #3: Undermining Temporal Memory Safety . . . 84
5.3.4 Attack #4: Understanding the Security Coverage . . . 86
5.4 End-to-End Case Study . . . 87
5.4.1 Vulnerability Description . . . 87
5.4.2 Baseline Exploitation Methodology . . . 88
5.4.3 C^3 Exploitation Challenges & C-4 Countermeasures . . . 91
5.4.4 Additional Real-World Exploits . . . 93
5.5 C-5: A Counter C-4 Architecture . . . 93
5.5.1 Security Enhancements . . . 94
5.5.2 Performance Optimizations . . . 97
5.5.3 C-5 Implementation . . . 99
5.6 Security Analysis . . . 100
5.6.1 Addressing Traditional Memory Safety Violations . . . 100
5.6.2 Mitigating Physical Attacks . . . 101
5.6.3 Mitigating The C-4 Attacks . . . 102
5.6.4 Preventing the end-to-end exploit . . . 102
5.6.5 Limitations . . . 103
5.7 Evaluation . . . 103
5.7.1 Evaluation Methodology . . . 103
5.7.2 SPEC CPU2017 Performance Evaluation . . . 104
5.7.3 Real-world Case Studies . . . 108
5.7.4 Security Evaluation . . . 109
5.8 Summary . . . 110

IV Efficient Exploit Mitigations For Servers & Embedded Systems 112

Chapter 6: Zero-Overhead Resilient Operation Under Pointer Integrity Attacks 113
6.1 Motivation 114
6.2 System Overview 114
6.2.1 How Does ZeRØ Work? 115
6.2.2 Main Components 116
6.3 Architecture Support 117
6.4 Microarchitecture Design 119
6.4.1 L1 Data Cache Modifications 119
6.4.2 Exception Handling Circuitry 121
6.4.3 L2/L3 Cache Modifications 122
6.4.4 L1 to/from L2 Transformation Module 123
6.4.5 Load/Store Queue Modifications 124
6.5 Software Design 125
6.5.1 Memory Management 125
6.5.2 Compiler Support 126
6.5.3 Operating System Support 126
6.6 Security Analysis 128
6.6.1 Threat Model 128
6.6.2 Security Discussion 129
6.6.3 Limitations 131
6.7 Evaluation 133
6.7.1 Hardware Measurements 133
6.7.2 Software Performance 134
6.7.3 Comparison with Prior Work. 135
6.8 Summary 136

Chapter 7: Efficient Pointer Integrity For Securing Embedded Systems 138
7.1 Motivation 139
7.2 System Overview 140
7.2.1 Function Pointer Integrity 140
7.2.2 Data Pointer Integrity 142
7.2.3 Return Address Integrity 142
7.2.4 Metadata Management 143
7.2.5 Putting It All Together 144
7.3 Microarchitecture Design 145

7.3.1 Processor Modifications . . . 145
7.3.2 Memory Hierarchy Modifications . . . 145
7.4 Software Design . . . 149
7.4.1 Software Properties . . . 149
7.4.2 Compiler Support . . . 152
7.4.3 Operating System Support . . . 152
7.5 Security Analysis . . . 153
7.5.1 Threat Model . . . 154
7.5.2 Security Discussion . . . 154
7.5.3 Limitations . . . 156
7.6 Evaluation . . . 156
7.6.1 Experimental Setup . . . 157
7.6.2 Performance Results . . . 157
7.6.3 Comparison with ARM's PAC . . . 158
7.6.4 Hardware Overheads . . . 159
7.7 Summary . . . 160

V Comparison With Prior Work On Architectural Support For Memory Safety 161

Chapter 8: Related Work . . . 162
8.1 Memory Safety Error Detection . . . 162
8.1.1 Memory Blocklisting . . . 162
8.1.2 Memory Tagging . . . 165
8.1.3 Memory Permitlisting . . . 165
8.2 Memory Safety Exploit Mitigation . . . 168
8.2.1 Shadow Stack-Based Techniques . . . 168
8.2.2 Encryption-Based Techniques . . . 170

VI Conclusion 172

Chapter 9: Conclusion . . . 173
9.0.1 Advancing Memory Blocklisting . . . 173
9.0.2 Advancing Memory Permitlisting . . . 174
9.0.3 Advancing Exploit Mitigation . . . 174

References . . . 176

List of Figures

1.1 Timeline for memory safety exploitation techniques and mitigations 3

2.1 The main components of a program's address space. 13
2.2 Memory corruption root causes, targets, and end results. 15

3.1 A high level overview of how Califorms works. 21
3.2 Califorms Insertion Polices. 22
3.3 Struct density histogram of SPEC CPU2006 benchmarks and the V8 engine. 23
3.4 Performance overhead with additional paddings for every field within structs. . . . 23
3.5 Califorms-bitvector Format. 28
3.6 Pipeline diagram for the L1 cache hit operation with Califorms. 28
3.7 Califorms-sentinel Format. 29
3.8 Logic diagram for Califorms conversion from the L1 cache to L2 cache. 29
3.9 Logic diagram of Califorms conversion from the L2 cache to L1 cache. 31
3.10 Califorms slowdown with additional one-cycle access latency for L2 and L3 caches. 41
3.11 Slowdown of the Califorms opportunistic policy and full insertion policy. 43
3.12 Slowdown of the Califorms intelligent insert policy. 43

4.1 A high level overview of how No-FAT enforces spatial memory safety. 52
4.2 Pipeline diagram for the L1 cache hit operation with No-FAT. 58
4.3 No-FAT performance overheads on the SPEC CPU2017 benchmarks. 71
4.4 Memory usage for different memory allocators. 71

5.1 The C^3 pointer format as proposed in the original paper. 77
5.2 A high level overview of how C^3 works. 78

5.3 Targeting C^3's spatial memory safety. 82
5.4 Undermining C^3's temporal memory safety. 84
5.5 A high-level overview of CVE-2018-4192. 88
5.6 A visualization of how the `addrof` primitive works. 90
5.7 A high level overview of how C-5 works. 95
5.8 C-5's temporal memory safety extension. 96
5.9 C-5's implementation overview. 99
5.10 Performance overheads for different No-FAT and C-5 variants. 105
5.11 Slowdowns with additional 6-cycles access latency for the L2 cache for C-5. 107
5.12 Nginx performance evaluation with C-5. 108
5.13 Duktape performance evaluation with C-5. 109

6.1 A high level overview of how ZeRØ's pointer integrity mechanism works. 115
6.2 ZeRØ's metadata encoding in the L1 data cache. 120
6.3 Pipeline diagram for the L1 cache hit operation with ZeRØ. 120
6.4 ZeRØ's metadata encoding in L2/L3 data cache and main memory. 121
6.5 Block diagram of the ZeRØ's L1-to-L2 transformation module. 124
6.6 Block diagram of the ZeRØ's L2-to-L1 transformation module. 124
6.7 Performance overheads of ZeRØ and three different ARM's PAC configurations. . . 136

7.1 Embedded systems market trend from 2013 to 2019. 140
7.2 A sample C application highlighting how EPI protects function pointers. 141
7.3 Finite state machine of the different EPI metadata and instructions. 144
7.4 EPI's metadata encoding in the L1 data cache on 32-bit architectures. 146
7.5 EPI's metadata encoding in the L2 cache and main memory. 147
7.6 Different pointers layout on 32-bit architectures after applying EPI's optimizations. 151
7.7 Performance overheads of the SPEC CPU2017 workloads for EPI and ARM's PAC. 158

List of Tables

3.1 `BLOC` instruction K-map. 26
3.2 Area, delay and power overheads of Califorms. 39
3.3 Hardware configuration of the simulated system for Califorms. 41

4.1 Area, delay and power overheads of No-FAT. 69
4.2 Memory usage for SPEC CPU2017 benchmarks. 72
4.3 Configuration sizes per each bin in the No-FAT binning memory allocator. 72
4.4 Number of heap allocations that require extra padding bytes with No-FAT. 73

5.1 C-5's ISA extensions. 99
5.2 C-5's base address cache sensitivity analysis. 106
5.3 Simulation parameters for C-5 data encryption evaluation. 107
5.4 RIPE Results Summary 110
5.5 Security Microbenchmarks Summary. 110

6.1 Actions taken on various instructions with ZeRØ. 119
6.2 Area, delay and power overheads of ZeRØ. 133
6.3 Number of unique LLVM function and data pointer types. 134

8.1 Categorization of prior work on spatial memory safety. 163
8.2 Comparison with prior work on memory safety error detection. 164
8.3 Comparison with prior work on memory safety exploit mitigation. 169

Abbreviations

ARM PAC	**ARM P**ointer **A**uthenti**c**ation
ARM MTE	**ARM M**emory **T**agging **E**xtension
ASan	**A**ddress**San**itizer
BLOC	**B**locklisting **Loc**ation
Buf2Ptr	**B**uffer **to P**ointer
C^3	**C**ryptographic **C**apability **C**omputing
C-4	**C**ompromising **C**ryptographic **C**apability **C**omputing
C-5	**C**ounter **C-4** Architecture
Califorms	**Ca**che **Li**ne **Form**ats
Intel CET	**Intel C**ontrol-flow **E**nforcement **T**echnology
CFI	**C**ontrol **F**low **I**ntegrity
CFG	**C**ontrol **F**low **G**raph
COP	**C**all **O**riented **P**rogramming
COOP	**C**ounterfeit **O**bject **O**riented **P**rogramming
CHERI	**C**apability **H**ardware **E**nhanced **RI**SC **I**nstructions
CPtrST	**C**ode **Poin**t**er St**ore
CPtrLD	**C**ode **Poin**t**er Loa**d
CRA	**C**ode **R**euse **A**ttack
CPI	**C**ode **P**ointer **I**ntegrity
DFI	**D**ata **F**low **I**ntegrity
DOP	**D**ata **O**riented **P**rogramming
DPtrST	**D**ata **Poin**t**er St**ore
DPtrLD	**D**ata **Poin**t**er Loa**d
EPI	**E**fficient **P**ointer **I**ntegrity
GE	**G**ate **E**quivalent
Intel MPX	**Intel M**emory **P**rotection e**X**tension
JOP	**J**ump **O**riented **P**rogramming
MAST	**M**emory **A**llocation **S**ize **T**able
REST	**R**andom **E**mbedded **S**ecret **T**okens
ROP	**R**eturn **O**riented **P**rogramming
SPARC ADI	**SPARC A**pplication **D**ata **I**ntegrity
UAF	**U**se-**A**fter-**F**ree
VLSI	**V**ery **L**arge- **S**cale **I**ntegration
ZeRØ	**Ze**ro-Overhead **R**esilient **O**peration Under Pointer Integrity Attacks

Publications

Parts of this thesis have been published earlier. The text in this thesis differs from the published versions in minor editorial changes that were made to improve readability. The following publications form the core of this thesis.

- Hiroshi Sasaki, Miguel A. Arroyo, **Mohamed Tarek Ibn Ziad**, Koustubha Bhat, Kanad Sinha, and Simha Sethumadhavan. “Practical Byte-Granular Memory Blacklisting using Califorms", *In the Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture (MICRO’52)*, Columbus, Ohio, USA, October 2019.

- **Mohamed Tarek Ibn Ziad**, Miguel A. Arroyo, Evgeny Manzhosov, Ryan Piersma, and Simha Sethumadhavan. “No-FAT: Architectural Support for Low Overhead Memory Safety Checks", *In the Proceedings of the 48th International Symposium on Computer Architecture (ISCA-48)*, Worldwide Event, June 2021.

- **Mohamed Tarek Ibn Ziad**, Miguel A. Arroyo, Evgeny Manzhosov, and Simha Sethumadhavan. “ZeRØ: Zero-Overhead Resilient Operation Under Pointer Integrity Attacks", *In the Proceedings of the 48th International Symposium on Computer Architecture (ISCA-48)*, Worldwide Event, June 2021.

- **Mohamed Tarek Ibn Ziad**, Miguel A. Arroyo, Evgeny Manzhosov, Vasileios P. Kemerlis, and Simha Sethumadhavan. ‘EPI: Efficient Pointer Integrity For Securing Embedded Systems", *In the Proceedings of the 2021 IEEE International Symposium on Secure and Private Execution Environment Design (SEED ’21)*, Worldwide Event, September 2021.

- **Mohamed Tarek Ibn Ziad**, Evgeny Manzhosov, and Simha Sethumadhavan. “C-4: Compromising Cryptographic Capability Computing", *Currently under review*.

The following publications are not included in this thesis.

- Miguel A. Arroyo, **Mohamed Tarek Ibn Ziad**, Hidenori Kobayashi, Junfeng Yang, and Simha Sethumadhavan. “YOLO: Frequently Resetting Cyber-Physical Systems for Security", *In the Proceedings of the SPIE Defense and Commercial Sensing*, Baltimore, Maryland, USA, April 2019.

- **Mohamed Tarek Ibn Ziad**, Miguel A. Arroyo, Evgeny Manzhosov, Vasileios P. Kemerlis, and Simha Sethumadhavan. “Using Name Confusion to Enhance Security", *Technical Report*, arXiv: 1911.02038, August 2020.

- **Mohamed Tarek Ibn Ziad**, Miguel A. Arroyo, and Simha Sethumadhavan. “SPAM: Stateless Permutation of Application Memory", *Technical Report*, arXiv: 2007.13808, September 2020.

- Evgeny Manzhosov, Adam Hastings, Meghna Pancholi, Ryan Piersma, **Mohamed Tarek Ibn Ziad** and Simha Sethumadhavan. “MUSE: Multi-Use Error Correction Codes", *Technical Report*, arXiv: 2107.09245, July 2021.

Acknowledgements

All praise belongs to Allah, Lord of all the worlds, the Gracious, the Merciful. I would like to thank God Almighty for bestowing upon me the chance, strength, and ability to complete this work. I want to take this opportunity to express my deep gratitude to the people who guided, helped, and encouraged me throughout my doctorate journey.

First, I am sincerely thankful to my advisor, Simha Sethumadhavan, for his kindness, guidance, and support over the past five years. Despite his many responsibilities, he was always available to answer my questions and dispel my concerns. I am grateful for all the technical and non-technical discussions we had together. I learned many lessons from him on both a professional and personal level. Thank you for teaching me how to be a true researcher.

During my time at Columbia University, I was fortunate to work with great colleagues, without whom none of my projects would have been completed. I would like to thank Miguel Arroyo for spending numerous days and nights working with me on building software infrastructures, debugging complex programs, and brainstorming new research ideas. I also thank Evgeny Manzhosov for helping me run microarchitectural simulations and create hardware prototypes. Additional thanks to my research collaborators—Hiroshi Sasaki, Kanad Sinha, Koustubha Bhat, Ryan Piersma, and Vasileios Kemerlis for contributing to my thesis publications. I also thank the past and the present members of the Computer Architecture and Security Technologies Lab (CASTL)—Adrian Tang, Yipeng Huang, Adam Hastings, Abhishek Shah, Meghna Pancholi, and Andreas Kellas for the interesting thoughts and rich discussions.

Over the past two summers, I was fortunate to have two (virtual) internships, in which I had the pleasure to interact with awesome folks in industry, present my work, receive thoughtful feedback, and get a taste of real-world challenges. I wish to thank my internship mentors from Qualcomm—Can Acar, Arvind Krishnaswamy, and Victor van der Veen for their help and support during Summer 2020. I also thank my NVIDIA mentors—Aamer Jaleel, Mark Stephenson, Michael Sullivan, and Steve Keckler for giving me the opportunity to apply my research work to GPUs and providing me with ultimate support during Summer 2021.

I would like to thank my committee members—Steven Bellovin, Martha Kim, Vasileios Kemerlis, and Santosh Nagarakatte for taking time from their busy schedule to participate in this dissertation. I would also like to thank Michael Sikorski, Suman Jana, Eleni Drinea, and Janet Kayfetz for teaching me great classes at Columbia University. Many thanks to Maria Joanta, Jessica Rosa, Cynthia Meekins, and the rest of the departmental staff at the CS department for helping me navigate the administrative matters smoothly. Thanks to Mohamed Watheq El-Kharashi, my first mentor at Ain Shams University, Egypt, for providing me with endless support during my undergraduate studies and inspiring me to pursue this career path.

Before all of that, I will always be grateful to my father Tarek Mohamed Hassan and my mother Nesrin Ahmed Radwan. Despite being more than 5,000 miles away from me, they have always been by my side throughput this PhD journey, sharing my successes and failures, and praying for me to shine. I would have never been in this place today without their love, support, and motivation. I am grateful for them forever. Finally, I am thankful to my soul mate and wife, Dina Abdulwahid, who traveled across the world to be with me. Thank you for caring so much for me. Your unconditional love is the fuel that keeps me going through all the struggles in my life.

To Mom & Dad

and

my wife, Dina.

Preface

This document is submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Graduate School of Arts and Sciences, Columbia University.

In this dissertation, I use “we” almost exclusively throughout as this work was accomplished in collaboration with many others, without whose help the work would not have been possible. In some cases, I use “I” whenever the text is derived from my own conclusions and explanations, which I am fully responsible for the content.

Part I

Introduction

Chapter 1: Introduction

The C and C++ programming languages are the gold standard for implementing a wide range of software systems such as safety critical firmware, operating system kernels, and network protocol stacks for performance and flexibility reasons. As these languages do not guarantee the validity of memory accesses (i.e., enforce memory safety), seemingly benign program bugs can lead to silent memory corruption, difficult-to-diagnose crashes, and, most importantly, security exploitation. Attackers can compromise the security of the whole computing ecosystem by exploiting a memory safety error with a suitably crafted input. Since the first documented overflow attack in 1972 [13], there has been a long-lasting arms race between attackers and defenders, as shown in Figure 1.1. New attacks are followed by potential mitigations that aim at preventing the attackers' malicious effects while current mitigations are followed by novel attacks that propose new approaches for bypassing the deployed mitigations. Despite massive advances in exploit mitigations, memory safety errors have risen to be the most exploited vulnerabilities because current mitigations do not address the root cause of the problem, which is the lack of memory safety.

1.1 What Is Memory Safety?

Memory safety is a program property that guarantees memory objects can only be accessed between their intended bounds, during their lifetime, and given their original (or compatible) type. Violating any of these requirements results in a memory corruption. For example, accessing objects beyond their intended bounds is called *spatial memory safety violation* (e.g., buffer overflow). On the other hand, accessing objects beyond their lifetime is called *temporal memory safety violation* (e.g., use-after-free and uninitialized reads). Finally, accessing objects with an incompatible type is referred to as *type confusion*, which can lead to both spatial and temporal violations.

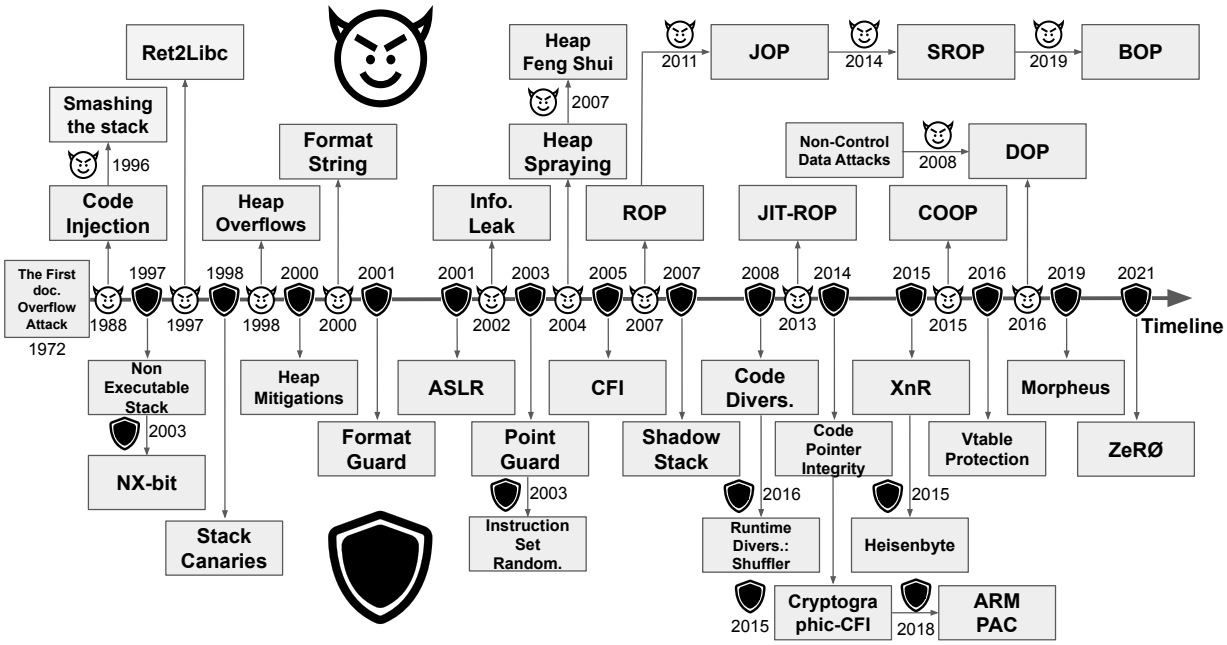


Figure 1.1: Timeline for memory safety exploitation techniques (marked with demons) and mitigations (marked with shields).

Unfortunately, memory safety vulnerabilities are common. They are easy for programmers to introduce unintentionally, especially in large code bases and systems with complex logic. For example, Microsoft recently revealed that the root cause of around 70% of all exploits targeting their products is software memory safety violations [14]. Similarly, roughly 70% of all serious security bugs that have been fixed in the Google Chrome stable branch since 2015 are memory safety bugs, with half of them being use-after-free vulnerabilities [15]. Moreover, the Project Zero team at Google reports that memory corruption issues are the root-cause of 68% of listed CVEs for zero-day vulnerabilities between 2014 and 2019 [16]. With the increasing number of memory safety vulnerabilities discovered each year, a natural question to ask is: *why does the memory safety problem persist despite the large efforts done by software vendors?*

1.2 Why Is Memory Safety Still A Concern?

To better understand why memory safety threats have not been completely resolved till now, we need to have a look on the available solutions so far.

1.2.1 Memory-Safe Languages

One possible solution to the memory (un)safety problem is to refrain from using languages that rely on programmers to manually manage memory (i.e., C/C++). Instead, developers can use languages that provide automatic memory management (i.e., memory safe languages), such as Java, Python, and Rust. While those languages are widely-adopted, they are not going to replace C/C++ anytime soon for the following reasons.

- **Performance.** C offers high performance as it allows the programmer to directly interact with the underlying hardware. Alternative safe languages use automatic memory management, which adds extra overheads that are not acceptable, especially for real time systems.

- **Communication.** C is more suitable for system level code as it allows communication between different entities via memory. For example, a program can write to a memory-mapped location that will be consumed externally by another device in the system. Memory-safe languages do not guarantee this unless the entire system is written with them and compiled as a whole program.

- **Completeness.** Even for safe languages, the automatic memory management runtimes themselves are written in C and C++. The runtimes may have bugs that can be exposed through programs written in memory safe languages.

- **Maturity.** Existing developer skills and the huge ecosystem of tools and libraries around C and C++ means that shifting to new languages would be a slow and long-term process. We may have to wait at least as long as it took for C/C++ to become mainstream (i.e., decades).

- **Legacy Code.** There is a vast amount of legacy C code deployed throughout the world in operating systems, web browsers, shared libraries, and embedded software. Such systems consist of millions of lines of C code, preventing the complete transition away from C or its variants anytime soon.

The above reasons necessitate the development of memory safety solutions for C and C++ applications.

1.2.2 Testing & Verification

To address the threat of memory safety violations, software testing tools (e.g., Google's AddressSanitizer [3]) and fuzz testing are widely deployed. In software fuzz testing binaries are instrumented with a tool like AddressSanitizer in order to detect memory safety vulnerabilities, then run with inputs mutated from a set of exemplary inputs in the hopes of detecting bugs before deployment. Google has reported that it has been performing fuzzing on about $25,000$ machines continuously since 2016, which has resulted in the identification of many critical bugs in software such as Google Chrome and several open source projects [17]. Assuming 15 cents per CPU hour for large memory machines—a requirement for reasonable performance on fuzz testing—the investment in software fuzzing for detecting memory errors could be close to a billion dollars at just one company.

Despite a Herculean effort by software vendors, memory safety vulnerabilities continue to slip through, ending up in deployed systems. Recognizing that pre-deployment fuzz tests can never be complete, companies have also proposed post-deployment crowdsourced fuzz testing [18, 19, 20, 21]. For instance, Google recently proposed fuzzing Android software in the field on user phones when they are plugged into power. Assuming that these tests run when phones are being charged, and assuming most users leave their phone charged overnight, on a global scale, the amount of energy invested in producing reliable software may be even higher than the amount of time running the software with crowdsourced testing. Thus, developing more efficient memory error detectors can have significant green benefits in addition to improving security and reliability.

1.2.3 Hardware-Based Mitigations

Both academia [22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32] and industry [4, 33, 5, 9, 10] have proposed various hardware-based techniques to address the C/C++ memory safety problem over

the last three decades. Prior work can be broadly categorized into the following four classes:

- **Memory Blocklisting.** This class of memory safety techniques (also known as tripwires) aims to detect overflows by marking the memory regions on either side of an allocation, and flagging accesses to them. For example, REST [30] stores a predetermined 8–64B random number, dubbed a token, in the memory to be invalidated. Spatial memory safety violations are detected by comparing cache lines with the token when they are fetched.

- **Memory Permitlisting.** This class of memory safety solutions (also known as base & bounds) attaches bounds metadata to every pointer or allocation. The metadata can be stored in a shadow (also known as disjoint) memory region (e.g., Hardbound [26], Intel's MPX [33], CHEx86 [31], and AOS [32]) or be marshaled with the pointer by extending its size (e.g., CHERI [29]).

- **Memory Tagging.** This class of memory safety techniques associates a "color" with newly allocated memory, and stores the same color in the upper bits of the data pointer that is used to access the allocated memory. At runtime, the hardware enforces spatial memory safety by comparing the colors of the pointer and accessed memory. For example, SPARC's ADI [4] assigns 4-bit colors to every 64B of memory (i.e., limiting the minimum allocation size to 64B), while ARM's MTE [5] uses 4-bit colors per every 16B of memory [34]. Since metadata bits are acquired along with the corresponding data, no extra memory operations are needed.

- **Exploit Mitigation.** This class of defenses neither detects nor prevents memory safety violations. Instead, exploit mitigations focus on preventing the attackers from manipulating main program assets (e.g., return addresses and function pointers) to protect the victim machine. Examples include Intel's CET [9], which uses a shadow stack to protect return addresses and ARM's PAC [10], which uses cryptographic message authentication codes to protect the integrity of pointers.

The main advantage of hardware-assisted solutions is its low performance overheads compared to software-based techniques. Thus, hardware solutions can be used as always-on mitigations to catch memory safety bugs in the field, if those bugs escape testing. Unfortunately, state-of-the-art hardware proposals suffer from other limitations.

1.3 Why Are Current Hardware-Based Solutions Impractical?

Current hardware-based memory safety proposals suffer from one or more of the following limitations, making them impractical to use in deployed systems:

- **Complex metadata management.** Techniques that store the metadata (e.g., per pointer base and bounds information) in disjoint shadow memory [26, 33, 32, 31] require extra memory accesses per pointer load/store to fetch and update the metadata. Besides the performance cost, frequently accessing the disjoint metadata may introduce atomicity problems for multithreading applications. *A practical memory safety solution needs to efficiently manage and access its own metadata*.

- **Breaking binary compatibility.** To avoid disjointly storing the metadata, some techniques (also known as FAT pointers [29, 35]) increase the pointer width to include the metadata. Using FAT pointers changes objects layouts in memory and breaks compatibility with the rest of the system (e.g., unprotected libraries). *A practical memory safety solution needs to be compatible with the other system entities without introducing (de)serialization penalties*.

- **Incomplete protection.** Memory tagging techniques associate a tag with newly allocated memory, and store the same tag in the upper bits of the data pointer that is used to access the allocated memory. Due to the tag's limited size, memory tagging solutions offer low entropy for temporal protection and non-adjacent spatial violations. *A practical memory safety solution needs to provide strong guarantees against different memory safety threats*.

- **Side-channel resiliency.** Speculative side-channel attacks (also known as Spectre) represent a major concern to current memory safety techniques, as Spectre can be used to undermine

their security guarantees. For example, attackers can infer the memory tag value of some memory region without triggering a memory tagging violation and use that to bypass memory tagging solutions [36]. *A practical memory safety solution needs to consider speculative side-channel attacks and ensure they cannot be used to bypass the solution itself.*

1.4 Thesis Statement & Contributions

In light of the above discussion, this dissertation makes the following thesis statement:

Leveraging common software trends and rethinking computer microarchitectures can efficiently circumvent the problems of traditional memory safety solutions for C and C++.

To support the above thesis statement, this dissertation makes the following contributions:

1.4.1 Califorms

The first contribution of this thesis is novel Cache Line Formats (Califorms) [6]. Califorms uses a two-fold approach for reducing memory safety overheads. First, instead of checking access bounds for each pointer access, Califorms blocklists all memory locations that should never be accessed. This reduces the additional work required to enforce memory safety, such as comparing bounds. Second, Califorms uses a novel metadata storage scheme for storing blocklisted information. The key observation is that by using dead memory spaces in the program, we can store necessary memory safety metadata for free for nearly half of the program's objects. These dead spaces can occur for several reasons, including language alignment requirements. When we cannot find naturally occurring dead spaces, we automatically insert them using compiler transformation. In order to distinguish the dead bytes from normal bytes in memory, Califorms uses a compressed encoding that requires one bit per each 64B cache line. If an attacker accesses these dead (i.e., blocklisted) regions, we detect this rogue access without any additional metadata accesses as our metadata resides inline.

Experimental results from the SPEC CPU2006 benchmark suite indicate that the overheads of Califorms are quite low: software overheads range from 2 to 14% slowdown (or alternatively, 1.02x to 1.14x performance overhead) depending on the amount and location of padding bytes used. The hardware induced overheads are also negligible, on average less than 1%. These overheads are substantially lower compared to the state-of-the-art software or hardware supported schemes (viz., 2.2x performance and 1.1x memory overheads for EffectiveSan [37], and 1.7x performance and 2.1x memory overheads for Intel's MPX [38]).

1.4.2 No-FAT

The second contribution of this thesis is a novel technique, dubbed No-FAT [7], for enforcing spatial and temporal memory safety by implicitly deriving the metadata information from the pointer itself. The key observation that enables No-FAT is the increasing adoption of binning allocators. We observe that current memory allocators use bags of pages (called bins), where each bin allocates objects of the same size. Using bins enables the allocator to quickly serve allocation requests and increases performance by maintaining allocation locality [39, 40, 41, 42]. No-FAT, when used with a binning allocator, is able to *implicitly* derive allocations bounds information (i.e., the base address and size) from the pointer itself without relying on expensive metadata. The hardware/software contract has to be tweaked slightly to facilitate the cooperation of No-FAT with binning allocators: the standard allocation sizes used by a binning allocator need to be supplied to the hardware and special load and store instructions are created to access the allocation sizes. In other words, the memory allocation size (e.g., `malloc` size) becomes an architectural feature.

No-FAT introduces an average of 8% performance overheads on the SPEC CPU2017 benchmark suite. No-FAT also provides resilience against the growing threat of speculative execution attacks. For example, Spectre-V1 [43] attacks exploit speculative execution to access out-of-bounds memory, effectively bypassing software based bounds checks. No-FAT's memory instructions are aware of allocation bounds information. Thus, allocation bounds information can be used to verify if memory accesses are within valid bounds even for speculatively executed instructions.

1.4.3 C-5

During program execution, user data can be manipulated from within the victim program (e.g., via a memory safety vulnerability) or from outside the program (e.g., via an inter-process side-channel attack). A recent work from Intel Labs, called Cryptographic Capability Computing (C^3), claims to provide resiliency against in-process and physical attacks [44]. In this thesis, I conduct a detailed assessment of the C^3's security claims. I uncover four different attack vectors against C^3, dubbed C-4, and show how these attacks can bypass the C^3's spatial and temporal memory safety guarantees, in addition to breaking its data confidentiality. My attacks exploit C^3's fundamental design choices, such as (1) using a fixed one-time pad for per-object data encryption, (2) lacking bounds checking on pointer arithmetic and usages, (3) providing low entropy against temporal safety violations, and (4) leaving the application's stack, global, and intra-allocation objects unprotected. Naively addressing the proposed attacks will require redesigning C^3 to use bounds checking and a stronger cipher, which will result in high performance overheads and negate C^3's stateless and compatibility claims.

Thus, the third contribution of this thesis is C-5, a Counter C-4 Architecture [8]. C-5 integrates strong data encryption with No-FAT to mitigate both in-process and physical attacks without burdening the system performance. Our evaluation results using the SPEC CPU2017 benchmark suite show that Counter C-4 Architecture introduces no runtime cost. Furthermore, we evaluate C-5 on the Nginx web server [45] and observe negligible overheads on the transfer rate and throughput for various file sizes.

1.4.4 ZeRØ

The aforementioned three contributions of this thesis focus on detecting memory safety errors to secure software written in C and C++. Another prominent way of thwarting memory safety attackers is by using exploit mitigation techniques. Due to their low overheads, hardware vendors have invested in deploying exploit mitigations such as Intel's CET [9] and ARM's PAC [10]. So, *can leveraging software properties and rethinking microarchitectures help to enhance the security*

guarantees of exploit mitigations while maintaining their low overheads? The answer is "yes". The fourth contribution of this thesis is ZeRØ, a hardware primitive that provides zero-overhead resilient operation under pointer integrity attacks [11]. ZeRØ leverages the currently unused upper bits in 64-bit pointers to store metadata that identifies different pointer types and distinguish them from regular non-pointer data. The hardware then uses special memory access instructions to access pointers based on their encoded types and hence prevents any malicious memory accesses (via buffer overflows for example) from corrupting pointers.

ZeRØ uses a novel metadata encoding scheme that allows it to precisely store all the required metadata to identify different program assets with a single bit per every cacheline in L2 and main memory (less than 0.2% memory overheads). Additionally, ZeRØ avoids crashing the victim program upon detecting an attack. Instead, ZeRØ raises an advisory exception to the operating system and continues program execution after skipping the violating memory access. This prevents the attacker from abusing ZeRØ to launch a denial-of-service attack. Our experimental results on the SPEC CPU2017 benchmark suite indicate that the software overheads of ZeRØ are 0% compared to a baseline. Additionally, our VLSI implementation results show that ZeRØ can be efficiently added to modern processors with negligible performance, area, and power overheads. Unlike other pointer authentication solutions, ZeRØ does not need to dedicate an energy budget to cryptographic co-processors [46, 10, 47] or standalone shadow stacks [9].

1.4.5 EPI

Embedded systems interact with many aspects of our daily lives, ranging from cell phones and life saving medical devices to aircraft and satellite systems. Due to their resource-constrained nature, embedded applications and firmwares are typically written in C to take advantage of its direct memory management and high performance, making them vulnerable to memory corruption attacks. Unfortunately, state-of-the-art exploit mitigation techniques are mainly designed for 64-bit processors and thus perform poorly when deployed on non 64-bit processors, the common choice for embedded systems [48, 49]. As a result, there is a need for solutions to the problem of securing

embedded 32-bit systems with minimal performance, power, and area overheads. Hence, my fifth contribution is Efficient Pointer Integrity (EPI), a hardware-based technique that mitigates memory safety-based attacks by ensuring the integrity of valuable application assets (i.e., pointers) [12].

Similarly to ZeRØ, EPI assigns unique tags to different program assets and uses unique memory instructions for accessing them. Unlike ZeRØ, which relies on the currently unused upper bits in 64-bit pointers to inline its metadata, EPI implements a novel metadata encoding scheme that is tailored for 32-bit architectures. The key observation that enables our EPI encoding is that leveraging common software properties allows for harvesting extra bits from pointers on 32-bit architectures. Examples for such properties include aligning stack frames and program functions, compacting code space, and inserting padding bytes. Our experimental results show that EPI's software introduces an average of 0.88% runtime overheads on the SPEC CPU2017 benchmark suite while having negligible latency and energy overheads.

1.5 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 provides an overview on program segments and memory safety in the C and C++ programming languages. Chapter 3 presents Califorms as an efficient memory blocklisting technique. Chapter 4 summarizes the key components of my permitlisting solution, No-FAT. Next, Chapter 5 discusses the C-5 architecture for mitigating both in-process and physical attacks. Afterwards, I illustrate my exploit mitigation technique, ZeRØ in Chapter 6 and its 32-bit variant, EPI in Chapter 7. Chapter 8 outlines how the techniques described in this thesis are different compared to prior work in the area of memory safety error detection and exploit mitigation. Finally, Chapter 9 concludes the thesis.

Chapter 2: Background

This chapter provides the necessary background information on the different concepts that will be needed over the course of this thesis. First, I summarize the main components of a computer program . Then, I introduce the memory safety problem in C and C++. Afterwards, I provide an overview of memory corruption attacks to further motivate the need for stronger defenses.

2.1 Program Components

As memory safety vulnerabilities can affect any segment of a program's memory, this section summarizes the main components of a program's address space. Figure 2.1 shows the memory layout of a typical C program. The top of the address space (i.e., addresses closer to `0x0`), is reserved for the operating system. The remaining parts are usable by the user-land process and include the program's code and data sections in addition to the dynamically changing regions of memory—the heap and the stack.

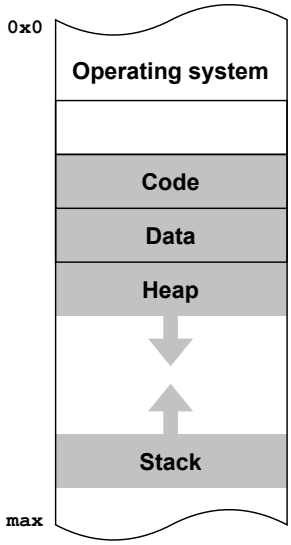


Figure 2.1: The main components of a program's address space.

2.1.1 Code

The code segment (also known as the `.text` section) is used to store the program instructions that need to be executed at runtime. This segment is typically marked as read-only in order to prevent attackers (or benign program errors) from modifying the instructions.

2.1.2 Globals

Global variables are seen by the entire program and are typically stored on the `.data` and `.bss` memory segments. Specifically, the `.data` segment stores global variables and static variables that are initialized by the programmer. On the other hand, the `.bss` segment contains uninitialized static data, i.e., global variables and local static variables that are initialized to zero or do not have explicit initialization in the program source code. For example, a global variable declared as `int k = 7;` would be stored in the `.data` segment, whereas a global variable declared `int m;` would be stored in the `.bss` segment.

2.1.3 Stack

Local variables, which are used by individual program functions, are stored in the stack. When a function is called, the return address (and the caller saved registers) are pushed to the stack. The newly called function then allocates room on the stack for storing its own local variables. This way the stack grows (and shrinks) dynamically as the program runs. The *stack pointer* is used to track the top of the stack during program execution.

2.1.4 Heap

The heap part of a program's address space is used to store dynamically allocated memory. It is managed by memory management functions such as `malloc`, `calloc`, `realloc`, and `free` which are provided by a memory allocator. A memory allocator is responsible for partitioning the heap memory space and serving the requested allocation sizes in a timely manner while minimizing internal and external fragmentation.

Memory allocators can be categorized into two main categories: binning and coalescing. Binning memory allocators divide the available space into fixed-size regions, where each region is used to allocate objects of a pre-determined size. Thus, the memory size returned to a program is rounded up to one of the standard sizes offered by the allocator. Examples of binning memory allocators include Jemalloc [40], Microsoft's Mimalloc [42], and Google's TCmalloc [39]. On the other hand, coalescing memory allocators dynamically join and split memory regions depending on the requested chunk size. Thus, they can provide the exact amount of memory requested by the program at the cost of an additional allocation header to store its size. An example for coalescing memory allocators is Dlmalloc [50]. In general, memory allocators use system calls, such as `brk`, `sbrk`, and `mmap` for expanding the heap size.

2.2 Memory Safety Definition

For a C or C++ application to be memory safe, all memory objects should only be accessed: (1) between their intended bounds, (2) during their lifetime, and (3) given their original (or compatible) type. Violating any of the above requirements can lead to silent memory corruption, difficult-to-diagnose crashes, and, most importantly, security exploitation [1].

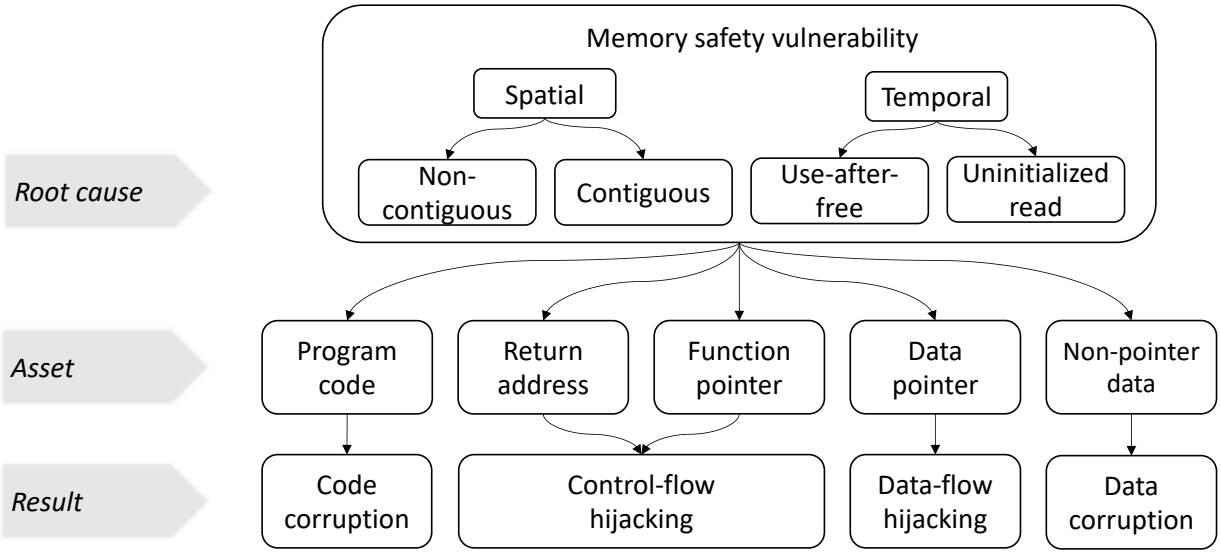


Figure 2.2: Memory corruption root causes, targets, and end results.

2.2.1 Spatial Memory Safety

This class of memory safety vulnerabilities occurs when a pointer is used to access an object beyond its intended bounds (i.e., base address and size) [51]. Examples include buffer under-/overflows, in which the application writes beyond the buffer's original bounds, causing a memory corruption in a different memory object. As shown in Figure 2.2, spatial violations can be further categorized into contiguous violations (in which the attacker overwrites an adjacent buffer) or a non-contiguous violation (in which the attacker overwrites arbitrary locations in memory). Finally, if the victim buffer is a field within a C struct or a member within a C++ class, such a violation is referred to as an intra-object spatial memory safety violation.

2.2.2 Temporal Memory Safety

This type of memory safety violations occurs when a pointer is used to access an object beyond its lifetime (e.g., use-after-free or uninitialized read). In use-after-free vulnerabilities the application uses a dangling pointer to access a heap object after it is deleted, or a stack object after its stack frame has been destroyed. For uninitialized reads, the programmer allocates an object and starts reading from it before writing anything to it. As a result, previous contents of the object might be leaked, violating memory safety.

2.2.3 Type Memory Safety

This class of memory safety violations (also known as type confusion) occurs when a memory location is accessed with an incompatible type. For example, the program first allocates (or initializes) a memory object using one type, and then accesses the same object later using a type that is incompatible with the original type due to an unsafe typecasting. Type confusion can trigger a spatial memory safety violation if the sizes of the two confused objects do not match. It can also trigger a temporal memory safety violation if the same memory region was allocated to one object and then assigned to an object of a different type after the original object was deleted.

2.3 Memory Safety Attacks

No matter what memory safety vulnerability a program has, an attacker can exploit it to manipulate one of the program assets, as shown in Figure 2.2. Valuable assets include program code, return addresses on the stack, function and data pointers on the heap, and non-pointer data (i.e., local variables and struct fields). Based on what asset is manipulated, an attacker can achieve different end results.

2.3.1 Code Corruption

Traditional approaches for exploiting memory vulnerabilities aimed at either (i) overwriting program instructions in memory with an attacker's payload or (ii) dumping the attacker's code discretely to the program stack and executing it. Nowadays, code corruption attacks are ineffective due to the widespread deployment of W^X [52]. In other words, an attacker cannot overwrite program data (i.e., code is marked as readable and executable but not writable) and cannot write and execute their own code (i.e., data is marked as readable and writable but not executable).

2.3.2 Control-Flow Hijacking

This line of attacks, which are also known as code reuse attacks (CRAs), exploits memory vulnerabilities to overwrite code pointers stored in memory. Corrupting a code pointer can cause a control-flow transfer to anywhere in executable memory. Code pointers include return addresses on the stack and function pointers anywhere in memory. As code pointers are stored in program memory (stack and heap), they are a common target for attackers. For example, return oriented programming (ROP) [53, 54] corrupts return addresses, whereas call- and jump-oriented programming [55, 56] corrupt function pointers (or indirect code addresses in general).

To mount a CRA, the attacker has to first analyze the code to identify the attack gadgets, or sequences of instructions in the victim program that end with a return or jump instruction. Second, the attacker uses a memory corruption vulnerability to inject a sequence of target addresses corre-

sponding to a sequence of gadgets. When the function returns (or a code pointer is dereferenced), it moves to the location of the first gadget. As that gadget terminates with a control flow instruction (e.g., return), it transfers program execution to the next gadget, and so on. As CRAs execute existing instructions belonging to the program, they are not prevented by W^X.

2.3.3 Data-Flow Hijacking

In contrast to control-flow hijacking attacks, data-oriented programming (DOP) attacks can cause malicious end results without changing the control flow of the program. Prior works show that manipulating data pointers in memory is sufficient for the attacker to achieve arbitrary computations on program input [57, 58, 59]. As DOP attacks do not alter the program control flow, they can easily bypass all control-flow integrity solutions. Thus, DOP is an appealing attack technique for future run-time exploitation defenses.

2.3.4 Data Corruption

This last class of attacks targets non-pointer data items while stored in memory. Examples include manipulating program flags to bypass selective checks and changing configuration parameters [60]. Mitigating non-pointer data corruption attacks requires full memory safety solutions, which traditionally come with high performance overheads.

Part II

Rethinking Microarchitectures For Efficient Memory Blocklisting

Chapter 3: Cache Line Formats

Historically, program memory safety violations have provided a significant opportunity for exploitation by attackers. To address this threat, software checking tools [3] and commercial hardware support for memory safety [4, 33] have enabled programmers to detect and fix memory safety violations before deploying software. Current software and hardware-supported solutions excel at providing coarse-grained inter-object memory safety, which involves detecting memory access beyond arrays and heap allocated regions (`malloc`'d struct and class instances). However, these solutions are not suitable for fine-grained memory safety (i.e., intra-object memory safety or detecting overflows within objects, such as fields within a struct, or members within a class) due to the high performance overheads and/or need for making intrusive changes to the source code [51]. For instance, a recent work that aims to provide intra-object overflow protection functionality incurs a 2.2x performance overhead [37]. These overheads are problematic because they not only reduce the number of pre-deployment tests that can be performed, but also impede post-deployment continuous monitoring, which researchers have pointed out is necessary for detecting benign and malicious memory safety violations [34]. Thus, a low overhead memory safety solution that can enable continuous monitoring and provide complete program safety has been elusive.

In this chapter I propose Califorms, a novel hardware primitive that allows blocklisting of a memory location (i.e., if accessed due to programming errors or malicious attempts, it reports a privileged exception) at *byte granularity* with low area and performance overheads. The main obstacle to blocklisting a memory region at a fine granularity (e.g., to prevent intra-object overflows) is the overhead of maintaining metadata. We solve this problem based on the following key observation: *a blocklisted region need not store its metadata (indicating it is blocklisted) separately, but can rather store it within itself (since it contains no useful data!).* With this principle, we utilize existing or added bytes between object elements to blocklist a region. This in-place compact data

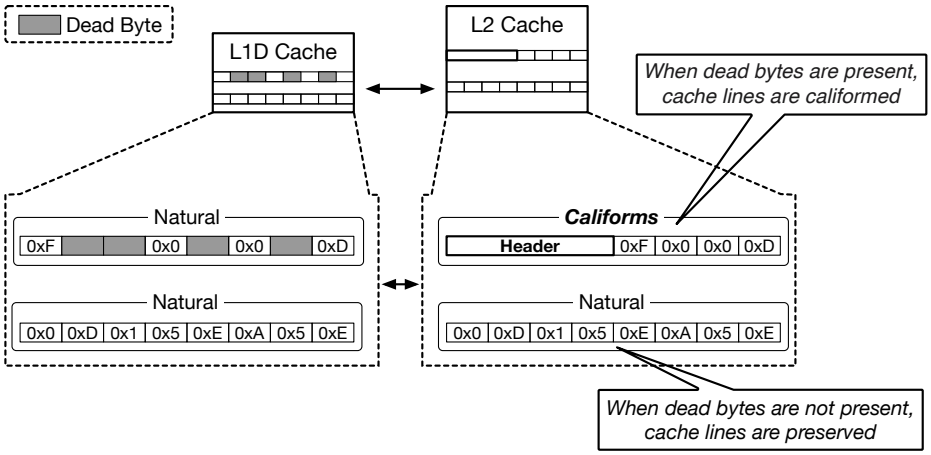


Figure 3.1: A high level overview of how Califorms works. Califorms offers memory safety by detecting accesses to dead bytes in memory. Dead bytes are not stored beyond the L1 data cache and identified using a special header in the L2 cache (and beyond) resulting in very low overhead. The conversion between these formats happens when lines are filled or spilled between the L1 and L2 caches. The absence of dead bytes results in the cache lines stored in the same natural format across the memory system.

structure avoids additional operations for accessing the metadata, making it very performant in comparison.

The challenge lies in how to reduce the additional hardware overhead required to identify normal data versus metadata. A naive implementation requires additional one bit (to specify normal data or metadata) per byte, which results in 12.5% area overhead. We manage to reduce the overhead substantially, to one bit per cache line (typically 64 bytes, thus area overhead of 0.2%), by changing how data is stored within a cache line. For cache lines which contain metadata (within blocklisted bytes), the actual data is stored following the "header", which indicates the location of blocklisted bytes, as shown in Figure 3.1.

The remainder of this chapter is organized as follows. Section 3.1 provides further motivation for Califorms. Section 3.2 explains the full system overview of Califorms. Section 3.3 discusses the Califorms architectural support. Section 3.4 details the microarchitectural design of Califorms whereas Section 3.5 specifies its software design. Section 3.6 analyzes the Califorms security guarantees. Section 3.7 evaluates the hardware and software overheads of Califorms. Section 3.8 summarizes the chapter.

3.1 Motivation

```
struct A {
 char c;
 int i;
 char buf[64];
 void (*fp)();
}
```

(a) Original.

```
struct A_opportunistic {
 char c;
 /* compiler inserts
  * padding bytes
  * for alignment */
 char padding_bytes[3];
 int i;
 char buf[64];
 void (*fp)();
}
```

(b) Opportunistic.

```
struct A_full {
 /* we protect every field
  * with random
  * security bytes */
 char security_bytes[2];
 char c;
 char security_bytes[1];
 int i;
 char security_bytes[3];
 char buf[64];
 char security_bytes[2];
 void (*fp)();
 char security_bytes[1];
}
```

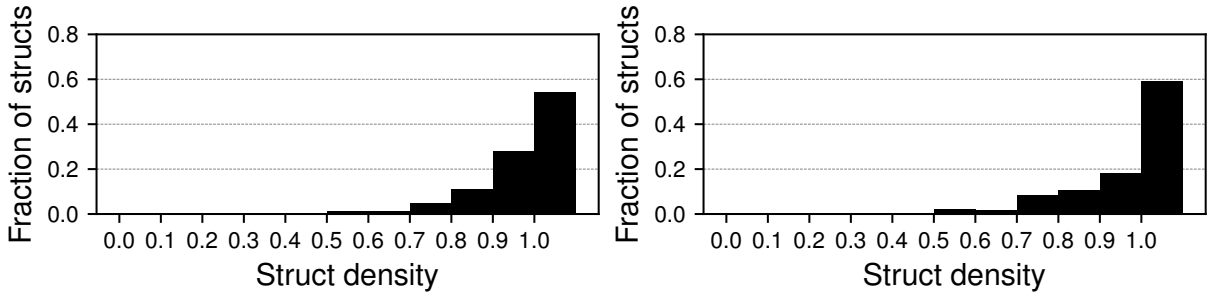
(c) Full.

```
struct A_intelligent {
 char c;
 int i;
 /* we protect boundaries
  * of arrays and pointers
  * with random
  * security bytes */
 char security_bytes[3];
 char buf[64];
 char security_bytes[2];
 void (*fp)();
 char security_bytes[3];
}
```

(d) Intelligent.

Figure 3.2: Califorms Insertion Polices. (a) Original source code and examples of three security bytes harvesting strategies: (b) *opportunistic* uses the existing padding bytes as security bytes, (c) *full* protect every field within the struct with security bytes, and (d) *intelligent* surrounds arrays and pointers with security bytes.

One of the key ways in which we mitigate the overheads for fine-grained memory safety is by opportunistically harvesting padding bytes in programs to store metadata. So how often do these occur in programs? Before we answer the question let us concretely understand padding bytes with an example. Consider the `struct A` defined in Figure 3.2a. Let us say the compiler inserts a three-byte padding in between `char c` and `int i` as in Figure 3.2b because of the C language requirement that integers should be padded to their natural size (which we assume to be four bytes here). These types of paddings are not limited to C/C++ but also required by many other languages and their runtime implementations. To obtain a quantitative estimate on the amount of paddings, we developed a compiler pass to statically collect the padding size information. Figure 3.3 presents the histogram of struct densities for SPEC CPU2006 C and C++ benchmarks and the V8 JavaScript engine. Struct density is defined as the sum of the size of each field divided by the total size of the `struct` including the padding bytes (i.e., the smaller or sparse the struct density the more padding bytes the struct has). The results reveal that 45.7% and 41.0% of structs within SPEC and V8, respectively, have at least one byte of padding. This is encouraging since even without introducing additional padding bytes (no memory overhead), we can offer protection for certain



(a) SPEC CPU2006 C and C++ benchmarks.

(b) V8 JavaScript engine.

Figure 3.3: Struct density histogram of SPEC CPU2006 benchmarks and the V8 JavaScript engine. More than 40% of the structs have at least one padding byte.

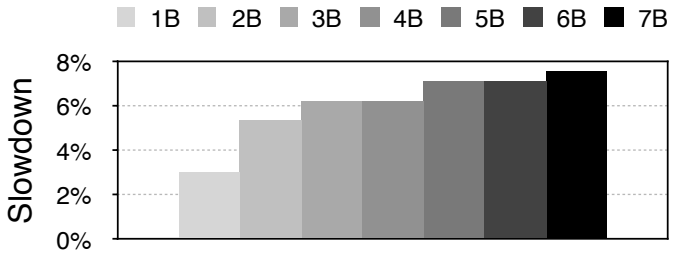


Figure 3.4: Average performance overhead with additional paddings (one byte to seven bytes) inserted for every field within structs (and classes) of SPEC CPU2006 C and C++ benchmarks.

compound data types restricting the remaining attack surface.

Naturally, one might inquire about the safety for the rest of the program. To offer protection for all defined compound data types, we can insert random sized padding bytes, also referred to as security bytes, between every field of a `struct` or member of a `class` as in Figure 3.2c (full strategy). Random sized security bytes are chosen to provide a probabilistic defense as fixed sized security bytes can be jumped over by an attacker once she identifies the actual size (and the exact memory layout). By carefully choosing the minimum and maximum of random sizes, we can keep the average size of security bytes small (few bytes). Intuitively, the higher the unpredictability (or randomness) within the memory layout, the higher the security level we can offer.

While the full strategy provides the widest coverage, not all of the security bytes provide the same security utility. For example, basic data types such as `char` and `int` cannot be easily overflowed past their bounds. The idea behind the intelligent insertion strategy is to prioritize insertion of security bytes into security-critical locations as shown in Figure 3.2d. We choose

data types which are most prone to abuse by an attacker via overflow type accesses: (1) arrays and (2) data and function pointers. In Figure 3.2d, the array `buf[64]` and the function pointer `fp` are protected with random sized security bytes. While it is possible to utilize padding bytes present between other data types without incurring memory overheads, doing so would come at an additional performance overhead.

In comparison to opportunistic harvesting, the other more secure strategies (full and intelligent) come at an additional performance overhead. We analyze the performance trend in order to decide how many security bytes can be reasonably inserted. For this purpose we developed an LLVM pass which pads every field of a `struct` and member of a `class` with fixed size paddings. We measure the performance of SPEC CPU2006 benchmarks by varying the padding size from one byte to seven bytes (since eight bytes is the finest granularity that state-of-the-art technique can offer [30]). The detailed evaluation environment and methodology are described later in Section 3.7.

Figure 3.4 demonstrates the average slowdown when inserting additional bytes for harvesting. As expected, we can see the performance overheads grow as we increase the padding size, mainly due to ineffective cache usage. On average the slowdown is 3.0% for one byte and 7.6% for seven bytes of padding. The figure presents the ideal (lower bound) performance overhead when fully inserting security bytes into compound data types; the hardware and software modifications we introduce add additional overheads on top of these numbers. We strive to provide a mechanism that allows the user to tune the security level at the cost of performance and thus explore several security byte insertion strategies to reduce the performance overhead in this work.

3.2 System Overview

The Califorms framework consists of the following three components:

- **Architecture Support.** We introduce a new instruction called `BLOC`, mnemonic for Blocklist LOCations, that blocklists memory locations at byte granularity and raises a privileged exception upon misuse of blocklisted locations (Section 3.3).

- **Microarchitecture Design.** We invent a new cache line formats, or Califorms, that enable low cost access to the metadata. We propose different Califorms for L1 cache versus L2 cache and beyond (Section 3.4).

- **Software Design.** We explain compiler, memory allocator and operating system extensions which insert the security bytes at compile time and manages them via the `BLOC` instruction at runtime (Section 3.5).

At compile time, each compound data type (`struct` or `class`) is examined and security bytes are added according to a user defined insertion policy viz. opportunistic, full or intelligent, by a source-to-source translation pass. At execution time when compound data type instances are dynamically created in the heap, we use a new version of `malloc` that issues `BLOC` instructions to arrange the security bytes after the space is allocated. When the `BLOC` instruction is executed, the cache line format is transformed at the L1 cache controller (assuming a cache miss) and is inserted into the L1 data cache. Upon an L1 eviction, the L1 cache controller transforms the cache line to meet the Califorms of the L2 cache.

While we add additional metadata storage to the caches, we refrain from doing so for main memory and persistent storage to keep the changes local within the CPU core. When a califormed cache line is evicted from the last-level cache to main memory, we keep the cache line califormed and store the additional one metadata bit into spare ECC bits similar to Oracle's ADI [4, 34].[1] When a page is swapped out from main memory, the page fault handler stores the metadata for all the cache lines within the page into a reserved address space managed by the operating system; the metadata is reclaimed upon swap in. Therefore, our design keeps the cache line format califormed throughout the memory hierarchy. A califormed cache line is un-califormed only when the corresponding bytes cross the boundary where the califormed data cannot be understood by the other end, such as writing to I/O (e.g.,pipe, filesystem or network socket). Finally, when an object is freed, the freed bytes are filled with security bytes and quarantined for offering temporal memory

[1] ADI stores four bits of metadata per cache line for allocation granularity enforcement while Califorms stores one bit for sub-allocation granularity enforcement.

safety. At runtime, when a rogue load or store accesses a security byte the hardware returns a privileged, precise security exception to the next privilege level which can take any appropriate action including terminating the program.

3.3 Architecture Support

3.3.1 BLOC Instruction

The format of the instruction is "`BLOC R1, R2, R3`". The value in register `R1` points to the starting (64B cache line aligned) address in the virtual address space, denoting the start of the 64B chunk which fits in a single cache line. Table 3.1 represents a K-map for the `BLOC` instruction. The value in register `R2` indicates the attributes of said region represented in a bit vector format (`1` to set and `0` to unset the security byte). The value in register `R3` is a mask to the corresponding 64B region, where `1` allows and `0` disallows changing the state of the corresponding byte. The mask is used to perform partial updates of metadata within a cache line. We throw a privileged exception when the `BLOC` instruction tries to set a security byte to an existing security byte, or unset a security byte from a normal byte.

Table 3.1: `BLOC` instruction K-map. X represents "Don't Care".

		R2, R3		
		X, $\overline{\text{Allow}}$	$\overline{\text{Set}}$, Allow	Set, Allow
Initial	Regular Byte	Regular Byte	Exception	Security Byte
	Security Byte	Security Byte	Regular Byte	Exception

The `BLOC` instruction is treated similarly to a store instruction in the processor pipeline since it modifies the architectural state of data bytes in a cache line. It first fetches the corresponding cache line into the L1 data cache upon an L1 miss (assuming a write allocate cache policy). Next, it manipulates the bits in the metadata storage to appropriately set or unset the security bytes.

3.3.2 Privileged Exceptions

When the hardware detects an access violation (i.e., access to a security byte), it throws a privileged exception once the instruction becomes non-speculative. There are some library functions which violate the aforementioned operations on security bytes such as `memcpy` so we need a way to suppress the exceptions. In order to permitlist such functions, we manipulate the exception mask registers and let the exception handler decide whether to suppress the exception or not. Although privileged exception handling is more expensive than handling user-level exceptions (because it requires a context switch to the kernel), we stick with the former to limit the attack surface. We rely on the fact that the exception itself is a rare event and would have negligible effect on performance.

3.4 Microarchitecture Design

The microarchitectural support for our technique aims to keep the common case fast: L1 cache uses the straightforward scheme of having one bit of additional storage per byte. All califormed cache lines are converted to the straightforward scheme at the L1 data cache controller so that typical loads and stores which hit in the L1 cache do not have to perform address calculations to figure out the location of original data (which is required for Califorms of L2 cache and beyond). This design decision guarantees that the common case latencies will not be affected due to security functionality. Beyond the L1, the data is stored in the optimized califormed format, i.e., one bit of additional storage for the entire cache line. The transformation happens when the data is filled in or spilled from the L1 data cache (between the L1 and L2), and adds minimal latency to the L1 miss latency.

3.4.1 L1 Cache: Bit Vector Approach

To satisfy the L1 design goal we consider a naive (but low latency) approach which uses a bit vector to identify which bytes are security bytes in a cache line. Each bit of the bit vector corresponds to each byte of the cache line and represents its state (normal byte or security byte).

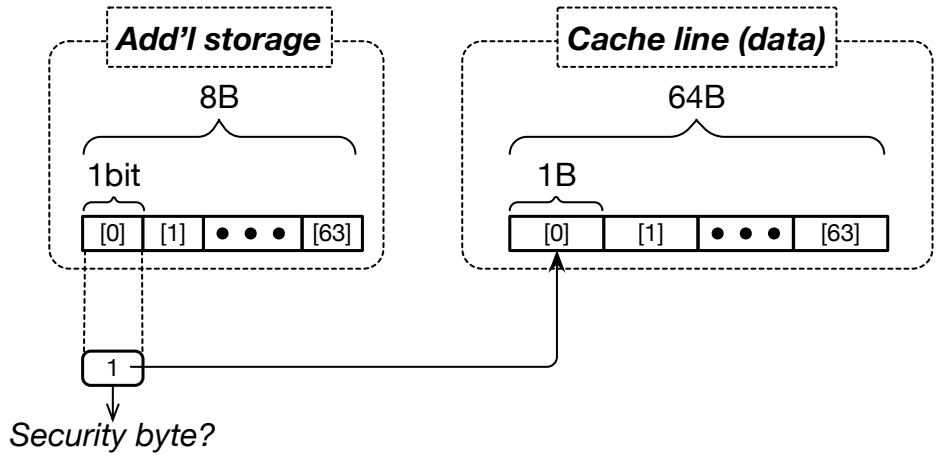


Figure 3.5: Califorms-bitvector Format: L1 Califorms implementation using a bit vector that indicates whether each byte is a security byte. HW overhead of 8B per 64B cache line.

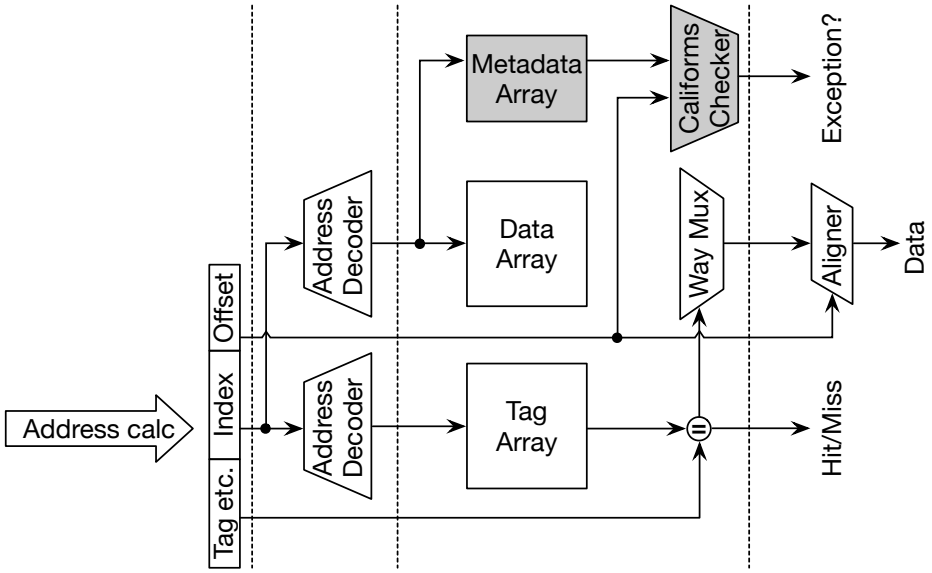


Figure 3.6: Pipeline diagram for the L1 cache hit operation. The shaded blocks correspond to Califorms components.

Figure 3.5 presents a schematic view of this implementation *califorms-bitvector*. The bit vector requires a 64-bit (8B) bit vector per 64B cache line which adds 12.5% storage overhead for the L1 data cache (comparable to ECC overhead for reliability).

Figure 3.6 shows the L1 data cache hit path modifications for Califorms. If a load accesses a security byte (which is determined by reading the bit vector) an exception is recorded to be processed when the load is ready to be committed. Meanwhile, the load returns a pre-determined

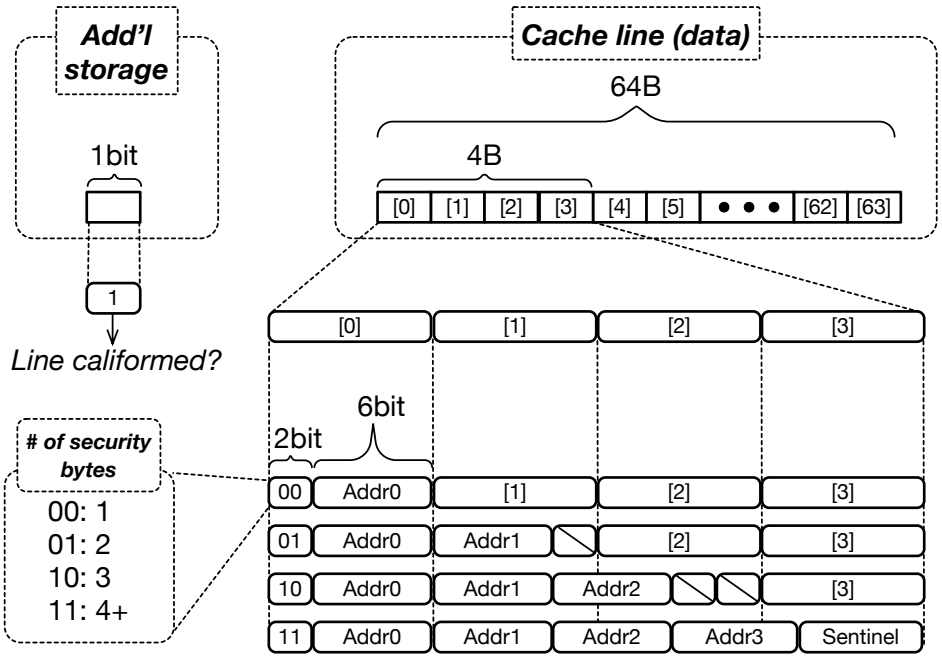


Figure 3.7: Califorms-sentinel that stores a bit vector in security byte locations. HW overhead of 1-bit per 64B cache line.

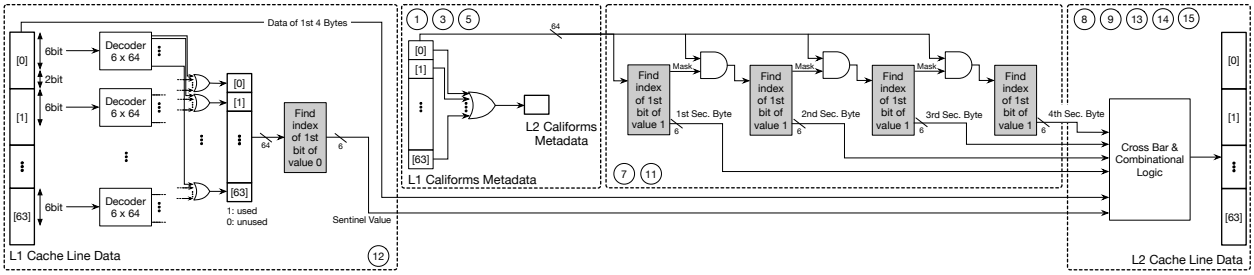


Figure 3.8: Logic diagram for Califorms conversion from the L1 cache (califorms-bitvector) to L2 cache (califorms-sentinel). The shaded blocks are constructed using 64 shift blocks followed by a single comparator. The circled numbers refer to the corresponding steps in Algorithm 1.

value for the security byte (in our design the value zero which is the value that the memory region is initialized to upon deallocation). Returning this fixed value is meant to be a countermeasure against speculative side-channel attacks that seek to identify security byte locations (discussed in greater detail in Section 3.6). On store accesses to security bytes, we report an exception when the store commits.

3.4.2 L2 Cache and Beyond: Sentinel Approach

For L2 and beyond, we take a different approach that allows us to recognize whether each byte

```
 1: Read the Califorms metadata for the evicted line and OR them
 2: if result is 0 then
 3:    Evict the line as is and set Califorms bit to zero
 4: else
 5:    Set Califorms bit to one
 6:    if num security bytes (N) < 4 then
 7:       Get locations of first N security bytes
 8:       Store data of first N bytes in locations obtained in 7
 9:       Fill the first N bytes based on Figure 3.7
10:    else
11:       Get locations of first four security bytes
12:       Scan least 6-bit of every byte to determine sentinel
13:       Store data of first four bytes in locations obtained in 11
14:       Fill the first four bytes based on Figure 3.7
15:       Use the sentinel to mark the remaining security bytes
16:    end
17: end
```

Algorithm 1: Califorms conversion from the L1 cache (califorms-bitvector) to L2 cache (califorms-sentinel).

is a security byte with fewer bits, as using the L1 metadata format throughout the system will increase the cache area overhead by 12.5%, which may not be acceptable. We propose *califorms-sentinel*, which has a 1-bit or 0.2% metadata overhead per 64B cache line. For main memory, we store the additional bit per cache line size in the DRAM ECC spare bits, thus completely removing any cycle time impact on DRAM access or modifications to the DIMM architecture.

The key insight that enables the savings is the following observation: *the number of bits required to address all the bytes in a cache line, which is six bits for a 64 byte cache line, is less than a single byte.* For example, let us assume that there is (at least) one security byte in a 64B cache line. Considering a byte granular protection there are at most 63 unique values (bytes) that non-security bytes can have. Therefore, we are guaranteed to find a six bit pattern that is not present in any of the normal bytes', for instance least significant, six bits. We use this pattern as a sentinel value to represent the security bytes in the cache line. Now if we store this six bit (sentinel value) as additional metadata, the storage overhead will be seven bits (six bits plus one bit to specify if the cache line is califormed) per cache line. In this work, we further propose a new cache line format which stores the sentinel value within a security byte to reduce the metadata overhead down to one bit per cache line.

As presented in Figure 3.7, califorms-sentinel stores the metadata into the first four bytes (at most) of the 64B cache line. Two bits of the first (0th) byte are used to specify the number of

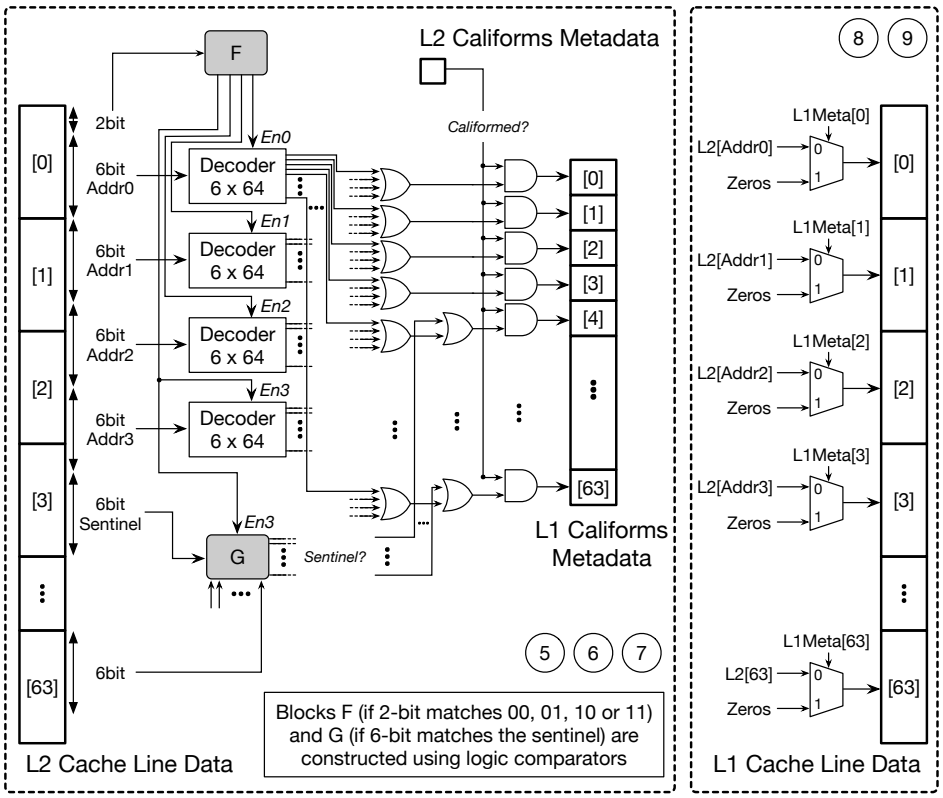


Figure 3.9: Logic diagram of Califorms conversion from the L2 cache (califorms-sentinel) to L1 cache (califorms-bitvector). The shaded block F and G consist of four and 60 comparators, respectively. The circled numbers refer to the corresponding steps in Algorithm 2.

security bytes within the cache line: `00`, `01`, `10` and `11` represent one, two, three, and four or more security bytes, respectively. The sentinel is used only when we have more than four security bytes. If there is only one security byte in the cache line, we use the remaining six bits of the 0th byte to specify the location of the security byte, and the original value of the 0th byte is stored in the security byte. Similarly when there is two or three security bytes in the cache line, we use the bits of the second and third bytes to locate them. The key observation is that, we gain two bits per security byte since we only need six bits to specify a location in the cache line. Therefore when we have four security bytes, we can locate four addresses and have six bits remaining in the first four bytes. This remaining six bits can be used to store a sentinel value, which allows us to have any number of additional security bytes.

Although the sentinel value depends on the actual values within the 64B cache line, it works naturally with a write-allocate L1 cache (which is the most commonly used cache allocation policy

```
 1: Read the Califorms bit for the inserted line
 2: if result is 0 then
 3:    Set the Califorms metadata bit vector to [0]
 4: else
 5:    Check the least significant 2-bit of byte 0
 6:    Set the metadata of byte[Addr[0-3]] to one based on 5
 7:    Set the metadata of byte[Addr[byte==sentinel]] to one
 8:    Set the data of byte[0-3] to byte[Addr[0-3]]
 9:    Set the new locations of byte[Addr[0-3]] to zero
10: end
```

Algorithm 2: Califorms conversion from the L2 cache (califorms-sentinel) to L1 cache (califorms-bitvector).

in modern microprocessors). The cache line format is transformed upon L1 cache eviction and insertion (califorms-bitvector to/from califorms-sentinel), while the sentinel value only needs to be found upon L1 cache eviction (L1 miss). Also, it is important to note that califorms-sentinel supports critical-word first delivery since the security byte locations can be quickly retrieved by scanning only the first 4B of the first 16B flit.

3.4.3 L1 to/from L2 Califorms Conversion

Figure 3.8 and Algorithm 1 show the logic diagram and the high-level process of the spill (L1 to L2 conversion) module, respectively. The circled numbers in the figure refer to the corresponding steps in the algorithm. There are four components presented in the figure. From the left, the first block details the process of determining the sentinel value (line 12). We scan the least 6-bits of every byte, decode them, and `OR` the output to construct a used-values vector. The used-values vector is then processed by a "find-index" block to get the sentinel value. The find-index block takes a 64-bit input vector and searches for the index of the first zero value. It is constructed using 64 shift blocks followed by a single comparator. In the second block, L1 cache (califorms-bitvector) metadata for the evicted line is `OR`ed to construct the L2 cache (califorms-sentinel) metadata. The third block shows the logic for getting the locations of the first four security bytes (lines 7 and 11). It consists of four successive combinational find-index blocks (each detecting one security byte) in our evaluated design. This logic can be easily pipelined into four stages if needed, to completely hide the latency of the spill process in the pipeline. Finally in the last block, we form the L2 cache line based on Figure 3.7.

Figure 3.9 shows the logic diagram for the fill (L2 to L1 conversion) module, as summarized in Algorithm 2. The shaded blocks F and G are constructed using logic comparators. The one bit metadata of L2 Califorms is used to control the value of the L1 cache (califorms-bitvector) metadata. The first two bits of the L2 cache line are used as inputs for the comparators (block F) to detect how many security bytes the cache line contain. Block F outputs four signals (En0 to En3) which enable the four decoders. Only if those two bits are `11`, the sentinel value is read from the fourth byte and fed, with the least 6-bits of each byte, to 60 comparators simultaneously to set the rest of the L1 metadata bits. Such parallelization reduces the latency impact of the fill process.

3.4.4 Load/Store Queue Modifications

Since the `BLOC` instruction updates the architectural state, it is functionally a store instruction and handled as such in the pipeline. However, there is a key difference: unlike a store instruction, the `BLOC` instruction should not forward the value to a younger load instruction whose address matches within the load/store queue (LSQ) but instead return the value zero. This functionality is required to provide tamper-resistance against side-channel attacks. Additionally, upon an address match, both load and store instructions subsequent to an in flight `BLOC` instruction are marked for Califorms exception; exception is thrown when the instruction is committed to avoid any false positives due to misspeculation.

In order to detect an address match in the LSQ with a `BLOC` instruction, first a cache line address should be matched with all the younger instructions. Subsequently upon a match, the value stored in the LSQ for the `BLOC` instruction which contains the mask value (to set/unset security bytes) is used to confirm the final match. To facilitate a match with a `BLOC` instruction, each LSQ entry should be associated with a bit to indicate whether the entry contains a `BLOC` instruction. Detecting a complete match may take multiple cycles, however, a legitimate load/store instruction should never be forwarded a value from a `BLOC` instruction, and thus the store-to-load forwarding from a `BLOC` instruction is not on the critical path of the program (i.e., its latency should not affect performance), and we do not evaluate its effect in our evaluation. Alternately, if LSQ modifications

are to be avoided, the BLOC instructions can be surrounded by memory serializing instructions (i.e., ensure that BLOC instructions are the only in flight memory instructions).

3.5 Software Design

We describe the memory allocator, compiler and the operating system changes to support Califorms in the following section.

3.5.1 Dynamic Memory Management

We can consider two approaches to applying security bytes: (1) **Dirty-before-use.** Unallocated memory has no security bytes. We set security bytes upon allocation and unset them upon deallocation; or (2) **Clean-before-use.** Unallocated memory remains filled with security bytes all the time. We clear the security bytes (in legitimate data locations) upon allocation and set them upon deallocation.

Ensuring temporal memory safety in the heap remains a non-trivial problem [61]. We therefore choose to follow a *clean-before-use* approach in the heap, so that deallocated memory regions remain protected by security bytes.[2] In order to provide temporal memory safety (to mitigate use-after-free exploits), we do not reallocate recently freed regions until the heap is sufficiently consumed (quarantining). Additionally, both ends of the heap allocated regions are protected by security bytes in order to provide inter-object memory safety. Compared to the heap, the security benefits are limited for the stack since temporal attacks on the stack (e.g., use-after-return attacks) are much rarer. Hence, we apply the *dirty-before-use* scheme on the stack.

3.5.2 Compiler Support

Our compiler-based instrumentation infers where to place security bytes within target objects, based on their type layout information. The compiler pass supports three insertion policies: the

[2]It is natural to use a variant of BLOC instruction which bypasses (does not store into) the L1 data cache, just like the non-temporal (or streaming) load/store instructions (e.g., MOVNTI, MOVNTQ, etc) when deallocating a memory region; deallocated region is not meant to be used by the program and thus polluting the L1 data cache with those memory is harmful and should be avoided. However, we do not evaluate the use of such instructions in this thesis.

first *opportunistic* policy supports security bytes insertion into existing padding bytes within the objects, and the other two support modifying object layouts to introduce randomly sized security byte spans that follow the *full* or *intelligent* strategies described in Section 3.1. The first policy aims at retaining interoperability with external code modules (e.g., shared libraries) by avoiding type layout modification. Where this is not a concern, the latter two policies help offer stronger security coverage—exhibiting a tradeoff between security and performance. Finally, Califorms is fully compatible with multi-threaded applications as the metadata is accessed simultaneously with regular program data in the L1 data cache.

3.5.3 Operating System Support

Privileged Exceptions. As the Califorms exception is privileged, the operating system needs to properly handle it as with other privileged exceptions (e.g., page faults). We also assume the faulting address is passed in an existing register so that it can be used for reporting/investigation purposes. Additionally, for the sake of usability and backwards compatibility, we have to accommodate copying operations similar in nature to `memcpy`. For example, a simple `struct` to `struct` assignment could trigger this behavior, thus leading to a potential breakdown of software with Califorms support. Hence, in order to maintain usability, we allow permitlisting functionality to suppress the exceptions. This can either be done with a privileged store (requiring a syscall) or an unprivileged store. Both options represent different design points in the performance-security tradeoff spectrum.

Page Swaps. As we have discussed in Section 3.2, data with security bytes is stored in main memory in a califormed format. When a page with califormed data is swapped out from main memory, the page fault handler needs to store the metadata for the entire page into a reserved address space managed by the operating system; the metadata is reclaimed upon swap in. The kernel has enough address space in practice (kernel's virtual address space is 128TB for a 64-bit Linux with 48-bit virtual address space) to store the metadata for all the processes on the system since the size of the metadata is minimal (8B for a 4KB page or 0.2%).

3.6 Security Analysis

This section defines the threat model and discusses the security claims of Califorms.

3.6.1 Threat Model

We assume a threat model comparable to that used in contemporary related works [30, 35, 29]. We assume the victim program to have one or more vulnerabilities that an attacker can exploit to gain arbitrary read and write capabilities in the memory; our goal is to prevent both spatial and temporal memory violations. Furthermore, we assume that the adversary has access to the source code of the program, therefore she is able to glean all source-level information. However, she does not have access to the host binary (e.g., server-side applications). Finally, we assume that all hardware is trusted—it does not contain and/or is not subject to bugs arising from exploits such as physical or glitching attacks. Due to its recent rise in relevance however, we maintain side-channel attacks in our design of Califorms within the purview of our threats. Specifically, we accommodate attack vectors seeking to leak the location and value of security bytes.

3.6.2 Hardware Attacks and Mitigations

Metadata Tampering Attacks. A key feature of Califorms is the absence of metadata that is accessible by the program via regular load-stores. This makes our technique immune to attacks that explicitly aim to leak or tamper metadata to bypass the defense. This, in turn, implies a smaller attack surface as far as software maintenance/isolation of metadata is concerned.

Bit-Granular Attacks. Califorms's capability of fine-grained memory protection is the key enabler for intra-object overflow detection. However, our byte granular mechanism is not enough for protecting bit-fields without turning them into `char` bytes functionally. This should not be a major detraction since security bytes can still be added around composites of bit-fields.

Side-Channel Attacks. Our design takes multiple steps to be resilient to side-channel attacks. Firstly, we purposefully avoid having our hardware modifications introduce timing variances to

avoid timing based side-channel attacks. Additionally, to avoid speculative execution side channels ala Spectre [43], our design returns zero on a load to camouflage security byte with normal data, thus preventing speculative disclosure of metadata. We augment this further by requiring that deallocated objects (heap or stack) be zeroed out in software [62]. This is to reduce the chances of the following attack scenario: consider a case if the attacker somehow knows that the padding locations should contain a non-zero value (for instance, because she knows the object allocated at the same location prior to the current object had non-zero values). However, while speculatively disclosing memory contents of the object, she discovers that the padding location contains a zero instead. As such, she can infer that the padding there contains a security byte. If deallocations were accompanied with zeroing, however, this assumption can be made with a lower likelihood. Hence, making Califorms return a fixed value (zero), complemented by software actively zeroing out unused locations, reduces the attacker's probability of speculatively predicting security byte locations, as well as leaking its exact value.

3.6.3 Software Attacks and Mitigations

Coverage-Based Attacks. For emitting `BLOC` instructions to work on the padding bytes (in an object), we need to know the precise type information of the allocated object. This is not always possible in C-style programs where `void*` allocations may be used. In these cases, the compiler may not be able to infer the correct type, in which case intra-object support may be skipped for such allocations. Similarly, our metadata insertion policies (viz., intelligent and full) require changes to the type layouts. This means that interactions with external modules that have not been compiled with Califorms support may need (de)serialization to remain compatible. For an attacker, such points in execution may appear lucrative because of inserted security bytes getting stripped away in those short periods. We note however that the opportunistic policy can still remain in place to offer some protection. On the other hand, for those interactions that remain oblivious to type layout modifications (e.g., passing a pointer to an object that shall remain opaque within the external module), our hardware-based implicit checks have the benefit of persistent tampering protection,

even across binary module boundaries.

Permitlisting Attacks. Our concession of allowing permitlisting of certain functions was necessary to make Califorms more usable in common environments without requiring significant source modifications. However, this also creates a vulnerability window wherein an adversary can piggy back on these functions in the source to bypass our protection. To confine this vector, we keep the number of permitlisted functions as minimal as possible.

Derandomization Attacks. Since Califorms can be bypassed if an attacker can guess the security bytes location, it is crucial that it be placed unpredictably. For the attacker to carry out a guessing attack, the virtual address of the target object has to be leaked, in order to overwrite a certain number of bytes within that object. To know the address of the object of interest, she typically has to scan the process's memory: the probability of scanning without touching any of the security bytes is $(1 - P/N)^O$ where O is number of allocated objects, N is the size of each object, and P is number of security bytes within that object. With 10% padding ($P/N = 0.1$), when O reaches 250, the attack success goes to 10^{-20}. If the attacker can somehow reduce O to 1, which represents the ideal case for the attacker, the probability of guessing the element of interest is $1/7^n$ (since we insert 1–7 wide security bytes), compounding as the number of padding spans to be guessed ($= n$) increases.

The randomness is, however, introduced statically akin to `randstruct` plugin introduced in recent Linux kernels which randomizes structure layout of those which are specified (it does not offer detection of rogue accesses unlike Califorms does) [63, 64]. The static nature of the technique may make it prone to brute force attacks like BROP [65] which repeatedly crashes the program until the correct configuration is guessed. This could be prevented by having multiple binaries of the same program with different padding sizes or simply by better logging, when possible. Another mitigating factor is that BROP attacks require specific type of program semantics, namely, automatic restart-after-crash with the same memory layout. Applications with these semantics can be modified to spawn with a different padding layout in our case and yet satisfy application level requirements.

3.7 Evaluation

3.7.1 Hardware Overheads

Cache Access Latency Impact of Califorms. Califorms adds additional state and operations to the L1 data cache and the interface between the L1 and L2 caches. The goal of this section is to evaluate the access latency impact of the additional state and operations described in Section 3.4. Qualitatively, the metadata area overhead of L1 Califorms is 12.5%, and the access latency should not be impacted as the metadata lookup can happen in parallel with the L1 data and tag accesses; the L1 to/from L2 Califorms conversion should also be simple enough so that its latency can be completely hidden. However, the metadata area overhead can increase the L1 access latency and the conversions might add little latency. Without loss of generality, we measure the access latency impact of adding califorms-bitvector on a 32KB direct mapped L1 cache in the context of a typical energy optimized tag and data, formatting L1 pipeline with multicycle fill/spill handling. For the implementation we use the 65nm TSMC core library, and generate the SRAM arrays with the ARM Artisan memory compiler.

Table 3.2: Area, delay and power overheads of Califorms (GE represents gate equivalent). L1 Califorms (califorms-bitvector) adds negligible delay and power overheads to the L1 cache access.

L1 Califorms	**Area (GE)**	**Delay** (ns)	**Power** (mW)
L1 Overheads	[+18.69%] 412,263.87	[+1.85%] 1.65	[+2.12%] 16.17
Fill Module	8,957.16	1.43	0.18
Spill Module	34,561.80	5.50	0.52

Table 3.2 summarizes the results for the L1 Califorms (califorms-bitvector). As expected, the overheads associated with the califorms-bitvector are minor in terms of delay (1.85%) and power consumption (2.12%). We found the SRAM area to be the dominant component in the total cache area (around 98%) where the overhead is 18.69% (higher than 12.5%).

The results of fill/spill modules are reported separately in the bottom half of Table 3.2. The latency impact of the fill operation is within the access period of the L1 design. Thus, the transformation can be folded completely within the pipeline stages that are responsible for bringing cache

lines from L2 to L1. The timing delay of the less performance sensitive spill operation is larger than that of the fill operation (5.5ns vs. 1.4ns) as we use pure combinational logic to construct the califorms-sentinel format in one cycle, as shown in Figure 3.8. This cycle period can be reduced by dividing the operations of Algorithm 1 into two or more pipeline stages. For instance, getting the locations of the first four security bytes (lines 7 and 11) consists of four successive combinational blocks (each detecting one security byte) in our evaluated design. This logic can be easily pipelined into four stages. Therefore we believe that the latency of both the fill and spill operations can be minimal (or completely hidden) in the pipeline.

Performance with Additional Cache Access Latency. Our VLSI implementation results imply that there will be no additional L2/L3 latency imposed by implementing Califorms. However, this might not be the case depending on several implementation details (e.g., target clock frequency) so we pessimistically assume that the L2/L3 access latency incurs additional one cycle latency overhead. In order to evaluate the performance of the additional latency posed by Califorms, we perform detailed microarchitectural simulations.

We run SPEC CPU2006 benchmarks with ZSim [66] processor simulator for evaluation. All the benchmarks are compiled with Clang version 6.0.0 with "`-O3 -fno-strict-aliasing`" flags. We use the `ref` input sets and representative simulation regions are selected with PinPoints [67]. We do not warmup the simulator upon executing each SimPoint region, but instead use a relatively large interval length of 500M instructions to avoid any warmup issues. MaxK used in SimPoint region selection is set to 30.[3] Table 3.3 shows the parameters of the processor, an Intel Westmere-like out-of-order core which has been validated against a real system whose performance and microarchitectural events to be commonly within 10% [66]. We evaluate the performance when both L2 and L3 caches incur additional latency of one cycle.

As shown in Figure 3.10 slowdowns range from 0.24% (`hmmer`) to 1.37% (`xalancbmk`).

[3]For some benchmark-input pairs we saw discrepancies in the number of instructions measured by PinPoints vs. ZSim and thus the appropriate SimPoint regions might not be simulated. Those inputs are: `foreman_ref_encoder_main` for `h264ref` and `pds-50` for `soplex`. Also, due to time constraints, we could not complete executing SimPoint for `h264ref` with `sss_encoder_main` input and excluded it from the evaluation.

Table 3.3: Hardware configuration of the simulated system for Califorms.

Core	x86-64 Intel Westmere-like OoO core at 2.27GHz
L1 inst. cache	32KB, 4-way, 3-cycle latency
L1 data cache	32KB, 8-way, 4-cycle latency
L2 cache	256KB, 8-way, 7-cycle latency
L3 cache	2MB, 16-way, 27-cycle latency
DRAM	8GB, DDR3-1333

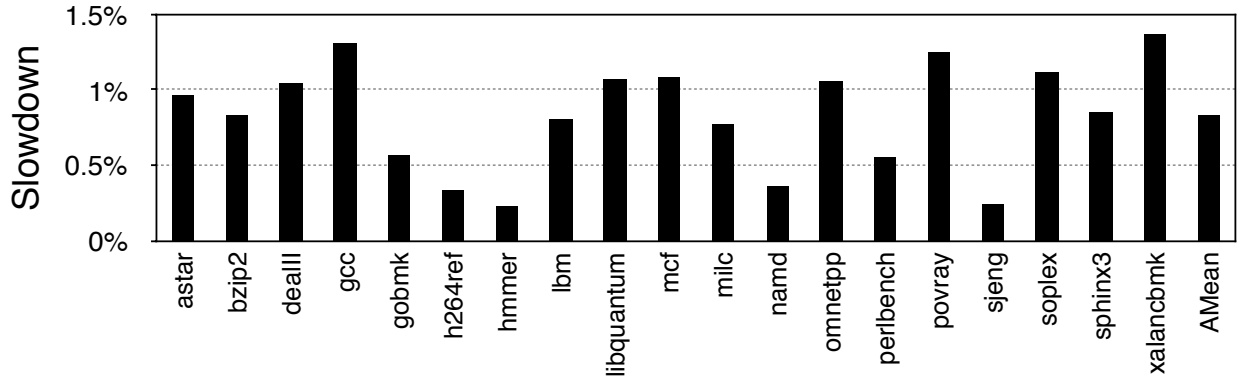


Figure 3.10: Califorms slowdown with additional one-cycle access latency for L2 and L3 caches.

The average performance slowdown is 0.83% which is well in the range of error when executed on real systems.

3.7.2 Software Performance Overheads

Our evaluations so far revealed that the hardware modifications required to implement Califorms add little or no performance overhead. Here, we evaluate the overheads incurred by the software based changes required to enable inter-/intra-object and temporal memory safety with Califorms: the effect of underutilized memory structures (e.g., caches) due to additional security bytes, the additional work necessary to issue `BLOC` instructions (and the overhead of executing the instructions themselves), and the quarantining to support temporal memory safety.

Evaluation Setup. We run the experiments on an Intel Skylake-based Xeon Gold 6126 processor running at 2.6GHz with RHEL Linux 7.5 (kernel 3.10). We omit `dealII` and `omnetpp` due to library compatibility issues in our evaluation environment, and `gcc` since it fails when executed with the memory allocator with inter-object spatial and temporal memory safety support. The re-

maining 16 SPEC CPU2006 C/C++ benchmarks are compiled with our modified Clang version 6.0.0 with "`-O3 -fno-strict-aliasing`" flags. We use the `ref` inputs and run to completion. We run each benchmark-input pair five times and use the shortest execution time as its performance. For benchmarks with multiple `ref` inputs, the sum of the execution time of all the inputs are used as their execution times. We use the arithmetic mean to represent the average slowdown.[4]

We estimate the performance impact of executing a `BLOC` instruction by emulating it with a dummy store instruction that writes some value to the corresponding cache line's padding byte. Since a single `BLOC` instruction is able to caliform the entire cache line, issuing one dummy store instruction per to-be-califormed cache line suffices. In order to issue the dummy stores, we implement a LLVM pass to instrument the code to hook into memory allocations and deallocations. We then retrieve the type information to locate the padding bytes, calculate the number of dummy stores and the address they access, and finally emit them. Therefore, all the software overheads we need to pay to enable Califorms are accounted for in our evaluation.

For the random sized security bytes, we evaluate three variants: we fix the minimum size to one byte while varying the maximum size to three, five and seven bytes (i.e., on average the amount of security bytes inserted are two, three and four bytes, respectively). In addition, in order to account for the randomness introduced by the compiler, we generate three different versions of binaries for the same setup (e.g., three versions of `astar` with random sized paddings of minimum one byte and maximum three bytes). The error bars in the figure represent the minimum and the maximum execution times among 15 executions (three binaries $\times$ five runs) and the average of the execution times is represented as the bar.

Performance of the Opportunistic and Full Insertion Policies with `BLOC` Instructions. Figure 3.11 presents the slowdown incurred by three set of strategies: full insertion policy (with random sized security bytes) *without* `BLOC` instructions, the opportunistic policy *with* `BLOC` in-

[4]The use of arithmetic mean of the speedup (execution time of the original system divided by that of the system with additional latency) means that we are interested in a condition where the workloads are not fixed and all types of workloads are equally probable on the target system [68, 69].

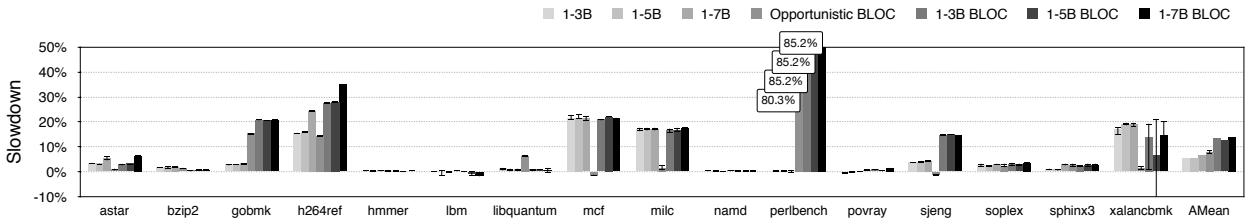


Figure 3.11: Slowdown of the opportunistic policy, and full insertion policy with random sized security bytes (with and without `BLOC` instructions). The average slowdowns of opportunistic and full insertion policies are 6.2% and 14.2%, respectively.

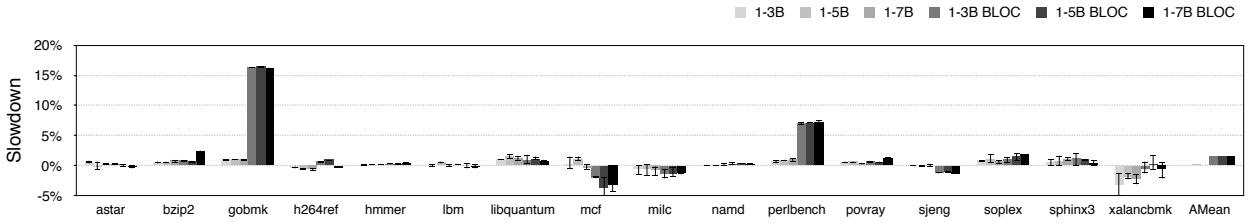


Figure 3.12: Slowdown of the intelligent insert policy with random sized security bytes (with and without `BLOC` instructions). The average slowdown is 2.0%.

structions, and the full insertion policy *with* `BLOC` instructions. Since the first strategy does not execute `BLOC` instructions it does not offer any security coverage, but is shown as a reference to showcase the performance breakdown of the third strategy (cache underutilization vs. executing `BLOC` instructions).

First, we focus on the three variants of the first strategy, which are shown in the three left most bars. We can see that different sizes of (random sized) security bytes does not make a large difference in terms of performance. The average slowdown of the three variants are 5.5%, 5.6% and 6.5%, respectively. This can be backed up by our results shown in Figure 3.4, where the average slowdowns of additional padding of two, three and four bytes ranges from 5.4% to 6.2%. Therefore in order to achieve higher security coverage without losing performance, using a random sized bytes of, minimum of one byte and maximum of seven bytes, is promising. When we focus on individual benchmarks, we can see that a few benchmarks including `h264ref`, `mcf`, `milc` and `omnetpp` incur noticeable slowdowns (ranging from 15.4% to 24.3%).

Next, we examine the opportunistic policy *with* `BLOC` instructions, which is shown in the middle (fourth) bar. Since this strategy does not add any additional security bytes, the overheads are

purely due to the work required to setup and execute BLOC instructions. The average slowdown of this policy is 7.9%. There are benchmarks which encounter a slowdown of more than 10%, namely gobmk, h264ref and perlbench. The overheads are due to frequent allocations and deallocations made during program execution, where we have to calculate and execute BLOC instructions upon every event (since every compound data type requires security bytes management). For instance perlbench is notorious for being malloc-intensive, and reported as such elsewhere [3].

Lastly the third policy, the full insertion policy *with* BLOC instructions, offers the highest security coverage in Califorms based system with the highest average slowdown of 14.0% (with the random sized security bytes of maximum seven bytes). Nearly half (seven out of 16) the benchmarks encounter a slowdown of more than 10%, which might not be suitable for performance-critical environments, and thus the user might want to consider the use of the following intelligent insertion policy.

Performance of the Intelligent Insertion Policy with BLOC Instructions. Figure 3.12 shows the slowdowns of the intelligent insertion policy with random sized security bytes (*with* and *without* BLOC instructions, in the same spirit as Figure 3.11). First we focus on the strategy without executing BLOC instructions (the three bars on the left). The performance trend is similar such that the three variants with different random sizes have little performance difference, where the average slowdown is 0.2% with the random sized security bytes of maximum seven bytes. We can see that none of the programs incurs a slowdown of greater than 5%. Finally with BLOC instructions (three bars on the right), gobmk and perlbench have slowdowns of greater than 5% (16.1% for gobmk and 7.2% for perlbench). The average slowdown is 1.5%, where considering its security coverage and performance overheads the intelligent policy might be the most practical option for many environments.

3.8 Summary

In this chapter I presented Califorms, a hardware primitive which allows blocklisting a memory location at byte granularity with low area and performance overhead. A key observation behind

Califorms is that a blocklisted region need not store its metadata separately but can rather store them within itself. Califorms utilizes byte-granular existing or added space between object elements to blocklist a region. This in-place compact data structure avoids additional operations for fetching the metadata making it very performant in comparison. Further, by changing how data is stored within a cache line, Califorms reduces the hardware area overheads substantially. Subsequently, if the processor accesses a blocklisted byte or a security byte, due to programming errors or malicious attempts, it reports a privileged exception.

To provide memory safety, Califorms inserts security bytes between and within data structures (e.g.,between fields of a `struct`) upon memory allocation and clear them on deallocation. Notably, by doing so, Califorms can even detect intra-object overflows in a practical manner, thus addressing one of the prominent open problems in area of memory safety and security.

Part III

Leveraging Current Software Trends For Efficient Memory Permitlisting

Chapter 4: Architectural Support for Low Overhead Memory Safety Checks

As we approach the end of Moore's Law, there is a need to develop better hardware that can run software programs faster without compromising security. One promising approach is to rethink the current software-hardware interface. In current architectural abstractions, lots of software properties are lost during compilation and thus are not transferred to hardware. Such a narrow interface limits the hardware's ability to accelerate software execution. If software properties are provided to hardware, we can significantly enhance the overall performance. However, care must be taken to ensure that the software-hardware interface is not too wide, since providing more software information to the hardware may badly impact portability. Thus, we need innovative abstractions that strike the right balance not only to improve security but also to serve broader needs.

This chapter presents No-FAT[1], which is an example of a balanced abstraction that efficiently exposes memory allocation sizes to the architecture to enhance security while maintaining portability. The key observation that enables No-FAT is that *if memory allocation sizes (e.g., malloc sizes) are made an architectural feature, then it is possible to implicitly derive the allocation bounds information (i.e., the base address and size) from the pointer itself without relying on explicit metadata.* Our No-FAT abstraction is inline with the software development trend towards using binning memory allocators, which makes them attractive for software such as Chrome, Android, and Windows applications. Binning allocators place similarly-sized objects in collections of pages (called bins). Using bins enables the allocator to quickly serve allocation requests and increases performance by maintaining allocation locality [39, 40, 41, 42]. This technological change facilitates the idea of standardizing memory allocation sizes in No-FAT.

[1]The name is an allusion to No-Fat Milk, which has fewer calories. Also, closely related work in this area refers to their schemes as Fat and Low Fat pointers.

This new abstraction enables multiple security and performance benefits, such as:

- **Improving fuzz-testing time.** Currently companies spend hundreds of millions of dollars testing software programs for bugs. A majority of these bugs tend to be intricate memory safety bugs. Exposing allocation sizes to hardware simplifies the checks for memory safety and improves the fuzz testing bandwidth by over 10x based on state-of-the-art solutions (e.g., AddressSanitizer based fuzzing).

- **Improving run-time security.** Despite the best effort of software engineers to produce bug-free code, some of these bugs do end up in production, and pose a risk to end users. If users wish to protect against remaining residual risk, our solution offers the lowest overhead protection among all published memory safety solutions that thwart data corruption attacks.

- **Improving resilience to Spectre-V1 attacks.** Exposing allocation sizes to the hardware allows the hardware to effectively perform bounds checking even for speculative memory accesses.

The remainder of this chapter is organized as follows. Section 4.1 motivates No-FAT. Section 4.2 provides an overview of how No-FAT works. Section 4.3 enumerates the No-FAT instruction set extensions. Section 4.4 details the microarchitectural design of No-FAT whereas Section 4.5 specifies its software design. Section 4.6 analyzes the security guarantees of No-FAT and discusses its deployment considerations. Section 4.7 evaluates the software and hardware costs of No-FAT. Section 4.8 summarizes the chapter.

4.1 Motivation

To minimize the damage caused by memory safety violations, software vendors have highly invested in static and dynamic checking tools. Due to the limited scope of static analysis techniques, specifically when applied to large code bases, dynamic tools (also known as sanitizers) have gained more popularity. To detect memory safety errors, applications are instrumented with

tools like Google's AddressSanitizer [3] and run with randomly generated inputs. In 2016, Google announced its OSS-Fuzz project for continuously fuzzing open source software [70]. Between 2016 and 2021, over 500 critical open source projects have been integrated into the OSS-Fuzz program, resulting in over 6,500 vulnerabilities and 21,000 functional bugs being fixed [20]. This large investment in software fuzzing with its promising results motivates the need for more efficient memory safety solutions that can accelerate the fuzzing process and uncover more bugs.

Chapter 3 presented one instance of hardware-based techniques that achieves a byte-granularity protection via memory blocklisting, namely Califorms. While blocklisting-based systems come with minimal performance overheads, they could be bypassed by non-adjacent buffer overflows (i.e., when the attacker guesses the size of the tripwired region and jumps over them), which represents 27% of Microsoft's memory safety CVEs [36]. Thus, there is a need to develop low overhead base & bounds techniques that can catch different instances of memory safety violations with deterministic security guarantees.

4.2 System Overview

4.2.1 Preliminaries

Binning memory allocators have gained prominence in the past decade and are now widely used [39, 40, 41, 42]. In a binning alloctor, the heap is divided into regions where each region is used to allocate objects of a pre-determined size. Thus, the memory size returned to a program is rounded up to one of the standard sizes offered by the allocator. For example, allocation requests that are less than 16 bytes come from the first region, allocation requests for 16 to 32 bytes come from the second region and so on. In contrast, non-binning allocators can provide the exact amount of memory requested by the program at the cost of an additional allocation header to store its size [50]. Binning allocators trade off a little memory fragmentation for faster allocation and deallocation times, and practically speaking, the fragmentation overheads tend to be negligible for most programs. In this work, we expose the pre-determined sizes offered by a binning memory

allocator to the hardware to provide memory safety.[2]

4.2.2 How does No-FAT provide Inter-allocation Spatial Memory Safety?

The goal of inter-allocation spatial memory safety is to be able to identify pointer-based accesses that access addresses outside the region of memory allocated to that pointer. To perform this check we need three pieces of information: (1) the starting address of the allocation, (2) the size of the space allocated to the pointer, and (3) the address of the pointer-based access. The benefit of binning allocators is that (1) and (2) can be computed from (3) using simple arithmetic and concurrently with the data access.

Given a pointer address, we determine the region that the pointer is from. Say each region is S GiB, and the heap starts at address H. Then the region of the pointer is $(ptr - H) >> log_2(S)$. Once the region is known, we can know the size of the allocation because all allocations from the region are of the same size. The base address of the allocation can be computed by $\lfloor (ptr/size) \rfloor * size$, whereas the combination of integer division and multiplication has the effect of rounding ptr down to the nearest $size(ptr)$-aligned boundary, which is the base address.

For example, let us assume that the heap starting address is H = `0x380000000000` and the memory allocator uses 64 bins (i.e., regions) each of size 32 GiB, where the third region is used to store allocations of size 32B. When the program executes `char* A = malloc(32)`, the memory allocator might return the following base address: `0x381000000040`. Now, given an arbitrary pointer ptr = `0x381000000045`, the hardware computes the region number by subtracting the heap starting address (i.e., `0x001000000045`) and ignoring the 35 LSBs (i.e., `0x002`, which is the third region). Then, the hardware retrieves the allocation size from the hardware table. Finally, the base address can be computed as $\lfloor$ `(0x381000000045/32)` $\rfloor * 32$ = `0x381000000040`.

How does this information help protect against attacks? Let us say an attacker has the ability to control the index variable of a dynamically allocated array. With this ability, the attacker can cause the pointer to go out-of-bounds and subvert the memory instruction in order to read/write from a

[2] A recent study [71] proposes passing semantic information from software to hardware to achieve better resource utilization and enhance performance. However, neither allocation size nor fine-grained security were included.

different allocation. If we simply calculated the base address from the attacker modified address we would not be able to catch the attack since we do not have an expectation of what the base address ought to have been. To avoid this case, No-FAT extends memory access instructions with an extra operand that carries a trusted base address. The trusted base address is simply the base address returned by `malloc`. This way we can verify the correctness of the access by computing the base address of the input pointer and matching it against the trusted base address, which is part of the instruction.

Computing the base address of a pointer for every memory access instruction is a costly operation as it includes a 64-bit division operation followed by a 64-bit multiplication. Division is a relatively expensive operation even on modern CPUs. To simplify the bounds checking operation, No-FAT uses the following check $isValid(ptr, base) = ptr - base < size(base)$. The idea is simple. As the instruction holds the trusted base address, we first compute its corresponding size by extracting the region number as explained before. Then, we compare this size to the difference between the input pointer and the trusted (i.e., instruction-based) base address. If the pointer overflows to an adjacent allocation, the difference will be larger than the computed difference. If the pointer underflows to a previous allocation, $ptr - base$ will be a negative number that will be interpreted as a large positive number that is $\geq size(base)$ as we use unsigned arithmetic.

In order to better understand how No-FAT enforces spatial memory safety, let us consider the example shown in Figure 4.1. The program reserves a 12B allocation on the heap in Line 2 and writes `'A'` to the second byte, `ptr[1]`. To make the allocation an architectural feature, No-FAT maintains the allocation sizes that are used by each bin in a hardware structure, called the memory allocation size table (MAST). It then modifies the compiler to propagate the allocation base address, `ptr`, from malloc output to memory instructions (i.e., `secure_store`). During the execution of the `secure_store` instruction in Line 3, No-FAT's Bounds Checking Module verifies the validity of the memory access by (1) computing the difference between the memory address, `ptr[1]`, and the propagated allocation base address, `ptr`, and (2) comparing this offset to the standard size that is retrieved from the MAST.

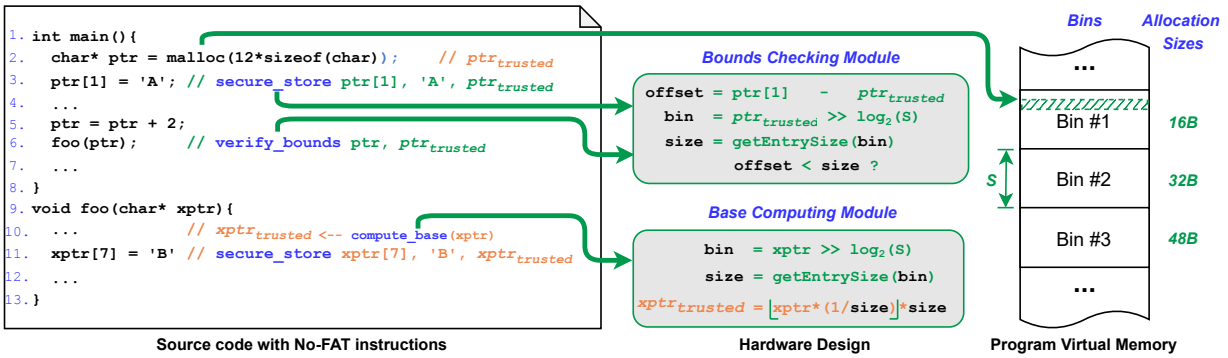


Figure 4.1: A high level overview of how No-FAT enforces spatial memory safety. No-FAT takes advantage of binning memory allocators, which divide the program virtual memory into S-Byte bins where each bin holds objects of a pre-defined size (e.g., $16B, 32B, 48B, \cdots, etc$). No-FAT uses a compiler pass to insert new instructions that (1) perform the bounds checking in parallel to the memory access and (2) implicitly recompute the allocation base address inside new functions.

To make No-FAT compatible with unprotected code, memory instructions that need to perform the check are emitted using special instructions. Specifically No-FAT uses Secure Load (`secure_load`) and Secure Store (`secure_store`) instructions (see Section 4.3) that use the allocation base address as a distinct operand. This operand is propagated in the binary using a compiler pass (see Section 4.5). This way `secure_load` and `secure_store` can verify access boundaries using the *isValid* check, as described above. On machines which do not have hardware support for No-FAT, `secure_load` can be interpreted as a regular load and the third operand will be ignored.

4.2.3 How does No-FAT provide Intra-allocation Spatial Memory Safety?

The goal of intra-allocation spatial memory safety is to prevent overflows from one field to another within the same allocation. The strategy used by No-FAT for intra-allocation safety is to convert the intra-allocation memory safety problem to an inter-allocation problem. No-FAT uses a source-to-source transformation, Buf2Ptr, which has been previously used in the area of data layout optimizations for enhancing performance [72, 73, 74]. Buf2Ptr promotes buffer fields, which exist in C/C++ structs (or classes), into their own allocations. To illustrate Buf2Ptr, consider the example in Listing 4.2. The array field, `buf[10]`, within the struct, `Foo`, is replaced with a promoted

pointer, p_buf, and a new variable for the original array is defined (Foo_buf[10]). As a result of this transformation, allocations, deallocations, and usages of the original field must also be properly promoted. For example, an allocation for a composite data type (e.g., Foo) becomes separate allocations based on the number of fields promoted (e.g., Foo_buf). As the standalone allocations have their own base address, they can be protected with No-FAT, as described above.

```
1
2 struct Foo {
3   char buf[10];
4   int value;
5 };
6
7
8 struct Foo *f = malloc(
9   sizeof(struct Foo));
10
11
12
13 f->buf[7] = 'A';
14
15 free(f);
16
```

(a) Original

```
// Promoted Type
char Foo_buf[10];
struct Foo {
  char *p_buf;
  int value;
};
// Promoted Allocations
struct Foo *f = malloc(
  sizeof(struct Foo));
f->p_buf = malloc(
  sizeof(Foo_buf));
// Promoted Usages
f->p_buf->buf[7] = 'A';
// Promoted Deallocations
free(f->p_buf);
free(f);
```

(b) Transformed

Listing 4.2: An example of Buf2Ptr transformation.

4.2.4 How does No-FAT provide Temporal Memory Safety?

To enforce temporal memory safety, No-FAT tags the upper 16-bits of data pointers on 64-bit systems with a random value upon malloc. This value is propagated in our new instructions (i.e., secure_load and secure_store) as part of the memory address and the allocation base address. This way, comparing the tag of the memory address with the tag of the allocation base address catches temporal safety violation with a probability of $1 - (1/2^{16}) = 99.9984\%$. When a virtual memory region is reallocated to a different object it receives a new random tag, implicitly nullifying all dangling pointers which used to point to the old object, as they are likely to have different tags. The work-flow is as follows:

1. Upon calling malloc or **new**, a 16-bit random tag is generated and inserted in both, the

pointer upper bits and the memory object at `memory [trusted base + size - 2]`.

2. Every `secure_load` or `secure_store` instruction within a function scope compare the tag of the memory address versus the tag of the trusted base address (i.e., the `malloc` output) for temporal correctness.

3. If a pointer is used in a different function scope (or context), its tag will be retrieved from `memory [trusted base + size - 2]` as part of executing the `compute_base` instruction directly after computing the trusted base address.

4. When the object is deallocated, we set the 16-bit tag in the memory object to a unique pattern, **`0xFFFF`**, to prevent any dangling pointer from accessing the deleted object.

Listing 4.3 shows a typical temporal memory safety violation example, in which the program first allocates an object (Line 4), stores the pointer to this object into a global variable (Line 8), and deallocates the object without freeing the reference in the global variable (Line 10). Later on, another function, `Bar`, accidentally accesses the global variable, `q`, which is still pointing to the deallocated object causing a use-after-free violation. With No-FAT, the aforementioned memory access (Line 15) will fail due to a mismatch between the tag of the global pointer (i.e., **`0xCAFE`**) and the tag of the memory object (i.e., **`0xFFFF`**), which is retrieved from memory as part of executing the `compute_base` instruction in the beginning of function `Bar`. This way No-FAT provides temporal protection even if the temporal violation occurs in different function scope. If the use-after-free is delayed until the memory region is allocated to another object, No-FAT can still catch the temporal safety violation with a probability of $1 - (1/2^{16}) = 99.9984\%$ as the new object will likely get a different tag other than **`0xCAFE`**. Finally, as No-FAT enforces spatial memory safety, an attacker cannot manipulate the tags while being stored in memory.

4.2.5 Handling Procedure Calls and Nested Pointers

Consider the following: `q = p + 16; x = Bar(q);`. Here, `p` is a pointer to a 32B allocation, and `q` is a derived pointer to a field within the allocation. In this case, the use of a pointer happens in

```
1 int *q;  // global pointer
2 void Bar();
3 int main() {
4  int *p = (int*) malloc(12); /* generate a random
5       tag, 0xCAFE, store it in the upper bits of p
6       and to memory location[base + size - 2] */
7  ...
8  q = p; // propagate the tag from p to q
9  ...
10 free(p); // set memory[base + size - 2] to 0xFFFF.
11 ...
12 Bar();
13 }
14 void Bar() {
15  *q = 0xABC; // use-after-free violation
16    /* With No-FAT, the compute_base instruction
17       computes the base address of q and retrieves
18       its tag from memory[base + size - 2]. As the
19       pointed-to object was previously deleted, its
20       tag is set to 0xFFFF, which causes the above
21       store instruction to fail */
22 }
```

Listing 4.3: A use-after-free example.

a different function (i.e., context) than the one where it was originally created. Thus, all functions using the base pointer (i.e., `p`) or its derivatives (e.g., `q`) need to be given access to the base address. One way to do this would use a source-to-source transformation to add an extra operand to all functions that use pointer arguments. This way the address would be in the stack of the needed function. This solution requires changing the function signature and breaks compatibility with unprotected code.

Instead, we use a different, simpler abstraction. Whenever a data pointer goes out of context (i.e., passed to another function or spilled to memory), we first verify that it is an in-bounds pointer using a Verify Bounds (`verify_bounds`) instruction (see Section 4.3). When a pointer is loaded from memory, we first compute its base address using a `compute_base` instruction and propagate this base address to all the following memory instructions as a third operand.

With this approach, can the attacker abuse pointers that escape to another function? This is not possible because (1) we verify the bounds of the pointer before spilling it to memory and (2) we protect the memory with No-FAT so we are assured that the pointer stored in memory cannot be

overwritten. This abstraction also permits No-FAT to use only intra-procedural analysis, which simplifies the implementation. Going back to our example, we first verify the bounds of `q` before calling `Bar(q)`. This is done with one `verify_bounds` instruction that takes the base address of `q` as an operand and matches it against the computed base address of `p + 16`. Inside `Bar`, we first call `compute_base` with `q` as an operand to retrieve its base address and propagate this base address to all memory instructions that uses `q` as an address.

4.3 Architecture Support

No-FAT adds the following instructions to the ISA:

• **`secure_store/secure_load <R1>, <R2>, <R3>`**: These instructions use three register operands. The values that are stored in registers `R1` and `R2` point to the store/load address and source/destination register as usual. The value in register `R3` is reserved for the allocation base address and is propagated by the compiler. Upon executing this instruction, the hardware computes the allocation size of `R3` and compares it to the difference between the address stored in `R1` and `R3`. An exception is thrown in case of $\texttt{R1} - \texttt{R3} \geq \mathit{size}(\texttt{R3})$. Additionally, the hardware matches the upper 16 bits of `R1` and `R3` to detect temporal memory safety violations.

• **`verify_bounds <R1>, <R2>`**: This instruction is used to check the bounds of pointers before storing them to memory (or passing them to a different function). It uses two register operands. The value in register `R1` is a pointer whereas the value in register `R2` is reserved for the allocation base address and is propagated by the compiler. Similar to `secure_store` and `secure_load`, upon executing this instruction, the hardware computes the allocation size of `R2` and compares it to the difference between the address stored in `R1` and `R2`. An exception is thrown in case of $\texttt{R1} - \texttt{R2} \geq \mathit{size}(\texttt{R2})$ to indicate that an out-of-bounds pointer is being stored to memory.

• **`compute_base <R1>, <R2>`**: This instruction takes a memory address (i.e., pointer) as input in `R1` and returns the allocation base address of this pointer in `R2`. This instruction is used to retrieve the correct base address of pointers that are passed to different contexts (e.g., through function calls).

4.4 Microarchitecture Design

In this section, we describe the four hardware components that are needed to enable No-FAT.

4.4.1 MAST

The Memory Allocation Size Table is a hardware structure, which is initialized at program startup with a process's allocation size configuration. The table is designed to work with binning allocators. The MAST enables No-FAT to support generic (i.e., non-powers-of-two) allocation sizes for each bin.[3]

In this work, we use a simple binning allocator that divides the heap into N equally sized bins. Based on our experiments, using 64 distinct bins is sufficient to balance performance and memory utilization. Thus, we use a 64-entry MAST with an entry size of 16B resulting in a total size of 1KB. Each entry holds an 8B size field and an 8B inverse size field. The size field of the nth entry is used to hold the allocation size used for the nth allocator bin. The inverse size field is an optimization that is discussed later. As a program's heap is contiguous, we use a single hardware register to store the starting address of the program heap and use it to derive the starting address of all bins. Some binning allocators (e.g., TCmalloc [39] and Jemalloc [40]) may change the allocation size used by one bin at runtime if all objects in the bin are freed. In this case, the allocator can simply update the MAST entry with the new size. We leave the investigation of other memory allocators to future work.

4.4.2 Bounds Checking Module

The bounds checking module takes two 64-bit operands, Ptr and $Base_{Ptr}$. It subtracts the two operands and compares the result with the allocation size of $Base_{Ptr}$. To compute the size of a given base pointer, the bounds checking module first maps the pointer to an allocation bin using simple subtract and shift operations followed by an access to the MAST to retrieve the allocation

[3]Using power-of-two sized objects can eliminate the need for MAST at the cost of additional memory overhead. This is a common optimization that was used in other systems such as Baggy bounds [75].

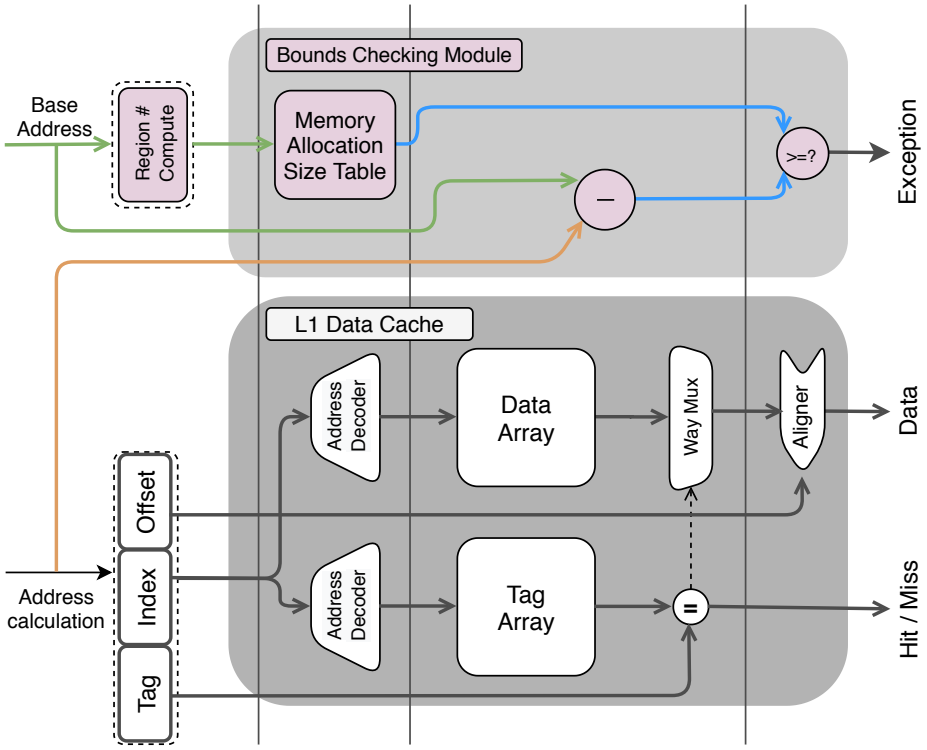


Figure 4.2: Pipeline diagram for the L1 cache hit operation. The bounds checking operations (top) are pipelined to avoid adding any access latency to L1 data.

size. Next, the bounds checking module uses a subtraction operation ($Ptr - Base_{Ptr}$) followed by a 64-bit unsigned comparison with the recently retrieved size (i.e., $size(Base_{Ptr})$). The last step is the temporal check, which is done with a 16-bit comparison operation between the upper 16 bits of Ptr and $Base_{Ptr}$.

The bounds checking module is invoked during the `secure_load` and `secure_store` instructions to prevent out-of-bounds pointer dereference and during the `verify_bounds` instruction to prevent out-of-bounds pointers from escaping to memory. As shown in Figure 4.2, the check operation can be totally hidden within the access latency for the L1 data cache.

4.4.3 Base Computing Module

As discussed in Section 4.2.5, pointers can be passed from one context to another. As No-FAT relies on simple intra-procedural compiler analysis, it needs to recompute the base address every time a pointer is loaded from memory (e.g., double pointers) or used as a function argument.

This feature is currently implemented with `compute_base` instruction that invokes the Base Computing Module.

This module takes a 64-bit *ptr* operand and computes its base address using $\lfloor ptr/size(ptr) \rfloor * size(ptr)$. While $size(ptr)$ requires one `MAST` lookup, the division operation is costly. No-FAT uses a common optimization that replaces the expensive division $(ptr/size(ptr))$ with a cheaper multiplication $(ptr * (1/size(ptr)))$ by using fixed-point arithmetic. This approach is feasible since the set of allocation sizes is constant, and thus the set of allocation size reciprocals can be pre-calculated and stored in the `MAST` along side with allocation size.

4.4.4 Dedicated Register File

As our `secure_load` and `secure_store` instructions use a third register operand, they may introduce register pressure. Thus, No-FAT adds a set of architectural registers that the compiler can exclusively use for holding and propagating allocation base addresses. The new registers are saved in a separate register file that is accessed in parallel to the regular register file. The contents of this register file are never spilled to memory as they can always be recomputed using the `compute_base` instruction, if needed.

4.5 Software Design

In this section, we describe the memory allocator, compiler and operating system changes to support No-FAT.

4.5.1 Dynamic Memory Management

One of No-FAT's key contributions is making the allocation size an architectural feature (i.e., sharing the allocation size information between software and hardware). To enable this feature, No-FAT requires binning memory allocators, in which a memory page is used to allocate objects of the same size. No-FAT does not add any constraints on how the allocator manages its internal metadata (e.g., free lists). No-FAT only intercepts calls to common memory management oper-

ations. For example, No-FAT intercepts all calls to `malloc`/`new` and tags the returned pointer with a random 16-bit value for ensuring temporal memory safety. Upon deallocation, No-FAT intercepts the calls to `free`/`delete` and removes the tag bits before calling the allocator's own `free`/`delete` API. When pointers are passed to uninstrumented code, tags are ignored by the hardware to maintain compatibility.

4.5.2 Compiler Support

Heap Instrumentation. In order to guarantee spatial protection, we implement an instrumentation pass at the LLVM IR level that replaces program loads and stores with our new instructions, `secure_load` and `secure_store`. To prepare the allocation base address register operand, we use simple function-level analysis to propagate the pointers returned by `malloc` or `new` intra-procedurally. To handle out-of-context pointers (e.g., those that are loaded from memory or passed as function arguments), our compiler pass inserts `compute_base` instructions in the corresponding locations to resolve the allocation base address. Our pass inserts `verify_bounds` instructions in places where a pointer is stored to memory. This can happen due to (a) casting pointer (`ptr`) to an integer (i.e., `i = (int)ptr`), (b) storing `ptr` to memory (i.e., `*r = ptr`), (c) passing `ptr` to a function (`f(ptr)`), and (d) returning `ptr` from a function (i.e., `return ptr`).

Source-to-Source Transformation. In order to achieve intra-allocation memory safety, we use a source-to-source transformation (Buf2Ptr), as described in Section 4.2.3. Buf2Ptr is implemented using Clang's rewriter interface. First, we perform an AST traversal over each translation unit to collect a whole program view of composite data types (e.g., structs) and their usages. Then, we perform a second traversal to perform the actual rewriting.

Stack & Global Instrumentation. In order to achieve full memory safety on all memory segments, we extend No-FAT to protect objects that are allocated on the stack and global memory. At compile time, No-FAT instruments all stack and global allocations (e.g., `alloca`) to use the same bins, which are used to satisfy heap allocations. This way No-FAT uses a unified method to enforce memory safety on all program memory segments. To avoid overheads related to allocating

stack objects on the heap, we adopt the same pointer mirroring and memory aliasing techniques used in prior work [76].

4.5.3 Operating System Support

`MAST` Initialization. During program initialization, the memory allocator needs to pass the allocation size information to the hardware. This is a one time task that can be done with a special system call or by writing to a hardware-mapped memory region. The size of the table is fixed, as described in Section 4.4.

Context Switching. Upon a context switch, No-FAT requires the operating system (OS) to store the `MAST` (and the dedicated register file contents) of the interrupted process and update the `MAST` and register file of the new process. Both the `MAST` and the register file contents are of fixed size and can be stored as part of the process control block. This step is likely to add minimal overhead (a few `load` and `store` instructions takes $\leq 0.1\mu$S) to the OS context switch (typically $3-5\mu$S).

Privileged Exceptions. When No-FAT's hardware detects an access violation, it throws a privileged exception once the instruction becomes non-speculative. The operating system needs to properly handle this exception as with other privileged exceptions (e.g., page faults). We also assume the faulting address is passed in an existing register so that it can be used for reporting/investigation purposes.

Finally, as No-FAT uses regular data pointers and does not change an object's memory layout, it naturally supports key OS functionalities such as inter-process data sharing, copy-on-write, and memory-mapped files. As No-FAT uses no per-word metadata, it does not require any changes to the page swapping subsystem.

4.6 Security Analysis

In this section, we first define the threat model. Then, we analyze the security guarantees of No-FAT and discuss its deployment considerations.

4.6.1 Threat Model

Adversarial Capabilities. We assume a threat model comparable to that used in contemporary related work on memory safety defenses [29, 30, 35, 31, 32]. We assume the victim program to have one or more vulnerabilities that an attacker can exploit to gain arbitrary read and write capabilities in the memory; our goal is to mitigate both spatial and temporal memory violations. Furthermore, we assume that the adversary is aware of No-FAT and has access to the source code, or binary image, of the target program. Finally, we assume that the attacker cannot tamper with the per-process size configurations as they are stored in the `MAST` and are kept as read-only in kernel memory upon context switch.

Hardening Assumptions. We assume that all hardware components including the ones proposed in this work are trusted and tamper-resistant, and therefore consider attacks that exploit hardware vulnerabilities, such as rowhammer [77] and side-channel attacks [78], to be out of scope. For speculative execution attacks [43], we include Spectre-V1 (also known as bounds checking bypass) in our threat model as it violates memory safety (speculatively). We do not include Spectre variants that manipulate branch predictor buffers as No-FAT does not affect program branch behavior.

4.6.2 Security Discussion

Buffer Under-/Over-flows. No-FAT defends against the exploitation of buffer overflows (and underflows) by detecting out-of-bounds pointers. No-FAT takes advantage of making the allocation size (per memory page) an architectural feature to enforce spatial memory safety. No-FAT not only protects heap-based allocations, but also stack and global memory regions. To do so, No-FAT reserves alias regions for stack and global objects such that both can use the same allocation size (per memory page) feature. No-FAT's protection applies to both inter- and intra-allocation safety (as Buf2Ptr reduces the intra-allocation problem to inter-allocation).

Use-after-frees. No-FAT provides temporal memory safety by tagging data pointers and validating the tags as part of the spatial bounds checking process. The same allocated virtual/physical mem-

ory region can have up to 2^{16} different tags, increasing the chances of catching dangling pointers (as dangling pointers use the old tags of the same allocated region).

Control-Flow Hijacking Attacks. In many attack scenarios, corrupting code pointers becomes a preferred attack vector. For instance, control-flow hijacking attacks, such as ROP [53] and its variants [79, 56], corrupt the return address of a function (or a function pointer) to hijack the control flow of a program. As all of the aforementioned attacks typically start with a spatial/temporal memory safety violation, No-FAT effectively stops control-flow hijacking attacks by eliminating their root cause.

Data-Oriented Attacks. Given a memory safety vulnerability, attackers can launch a data-only attack [60, 57, 59, 80] without abusing any code pointer. No-FAT mitigates those attacks by ensuring that all loads/stores happen between their legitimate bounds. If attackers move a pointer out of bounds to write to a (non-)adjacent allocation, No-FAT throws an exception as the computed base address of the malicious pointer does not match the base address operand of the `secure_load`/`secure_store` instructions.

Uninitialized Reads. No-FAT does not explicitly mitigate uninitialized read attacks, in which attackers can leak information from stack/heap locations by loading from these locations before doing a store operation. To mitigate this attack vector, No-FAT requires that deallocated objects (heap or stack) be zeroed out. Prior work showed that this process can be done efficiently in software [62].

Third-party Library Attacks. While No-FAT maintains full compatibility with third party libraries that are not instrumented with our compiler pass, we offer no security guarantees about vulnerabilities that exist in such uninstrumented code. To increase the security coverage, we create software wrappers for commonly used memory functions that appear in third-party libraries (e.g., `memcpy`, `memset`, and `memove`) to ensure that they cannot be used by an attacker to undermine No-FAT. For example, Listing 4.4 shows the pseudocode for our `memcpy` wrapper that first computes the base address of the source/destination pointer and ensures it matches the base address of the source/destination plus size before calling the original `memcpy` function.

```
1 void *memcpy_wrap(void *dst, void *src, size_t n){
2  compute_base src, src_base
3  compute_base src+n, src_end_base
4  Assert(src_base == src_end_base)
5  compute_base dst, dst_base
6  compute_base dst+n, dst_end_base
7  Assert(dst_base == dst_end_base)
8  return memcpy(dst, src, n);
9 }
```

Listing 4.4: Example `memcpy` wrapper.

4.6.3 Spectre-V1 Resiliency

A key advantage of No-FAT over prior memory safety defenses is its natural resiliency to certain classes of speculative side-channel attacks, namely Spectre-V1 (bounds checking bypass) [43]. We first summarize how Spectre-V1 works. Then, we show how it can undermine prior memory safety techniques. Finally, we describe how No-FAT mitigates it with no extra cost.

```
1 if (i < a->length) { // mispredicted branch
2   secret = a->data[i];
3   val = b[64 * secret]; // secret is leaked
4 }
```

Listing 4.5: Example speculative execution attack.

Attack Summary. To better understand how Spectre-V1 works, let us consider the example shown in Listing 4.5, in which the attacker controls the index, `i`. The attacker first trains the branch predictor by supplying multiple valid values for `i` (i.e., less than `a->length`). Then, the attacker provides an out-of-bounds index `i > a->length`. While this index violates the software bounds check in Line 1, the hardware will mispredict the condition (i.e., branch is taken) and speculatively executes Lines 2 and 3. As a result, a speculative buffer overread occurs at Line 2 and the read value (`secret`) is used as an index at Line 3. The attacker finally leaks the `secret` value via a covert channel as speculative execution leaves traces in processor structures (e.g, data caches). For example, the address in Line 3 depends on the `secret`, thus flushing and reloading the L1 data cache will allow the attacker to find out which cache line was used and reveals the secret.

Prior Work. Spectre-V1 is a main concern for prior memory safety techniques as it can be used to undermine their security guarantees. For example, attackers can infer the memory tag value

of memory without triggering a memory tagging violation and use that to bypass memory tagging solutions (i.e., SPARC's ADI [4] and ARM's MTE [5]) [36]. To mitigate Spectre-V1, prior work suggested inserting serialization instructions (or fences) at certain program points to prevent the processor from speculatively bypassing bounds checks [81]. This approach can result in up to 10x runtime overheads [82]. Another line of work proposed isolating speculatively accessed data to prevent leakage via covert-channels [83, 84]. While these defenses reduce the performance overheads, they add substantial complexities to the hardware design.

No-FAT vs. Spectre-V1. No-FAT's `secure_load` and `secure_store` instructions are resilient against Spectre-V1 by construction. Even if the processor mispredicted the branch instruction in Line 1 of Listing 4.5, the `secure_load` that is used in Line 2 holds the legitimate base and bounds of `a->data` as a third operand. Thus, it immediately recognizes the speculative access as an out-of-bounds access and does not allow `a->data[i]` to access the cache (to avoid modifying the cache state). Hence, No-FAT is resilient against the recent Spectre attack, namely the μop Disclosure Primitive, which exploits the micro-op cache as a timing channel to transmit out-of-bounds secrets [85]. Additionally, No-FAT prevents the dependent load instruction from executing by unmarking the ready bit on the register that has the load value (then raising an exception when the out-of-bounds access is non-speculative). We delay raising the exception until the commit stage to avoid false alarms (i.e., if the out-of-bounds memory access happens due to a benign branch misprediction).

Other Spectre Variants. As stated in Section 4.6.1, No-FAT does not protect against Spectre-variants other than V1 (bounds checking bypass) as the main focus in this work is memory safety. Examples of other Spectre variants include Spectre-V2 (also known as branch target injection), which can be used by an attacker to pollute the branch target buffer and force the victim program to speculatively jump to an arbitrary sequence of instructions (called a Spectre gadget). If the Spectre gadget has memory access instructions (e.g., `secure_loads`), they will be speculatively executed based on the current contents of register `R1` (memory address) and register `R3` (allocation base address) even if those register contents belong to an incorrect execution context. The same

argument applies if the Spectre gadget includes `compute_base` instructions. Other Spectre-V2 mitigations can be used to address this attack vector [86].

4.6.4 Deployment Considerations

In this section, we discuss the system requirements, strengths, and weaknesses of No-FAT.

System Requirements. No-FAT requires the usage of binning memory allocators. While the internal details of the memory allocator are irrelevant to No-FAT design, the minimum requirement is to have allocations of the same sizes per any memory page. Some allocators (e.g., non-binning or header-based allocators) violate the above requirement by allocating objects of different sizes in the same page. The non-binning allocators rely on an allocation header to keep track of each allocation size. These allocators are not protected by No-FAT as the cost of deriving allocation base addresses will be much higher (requiring a memory access to the allocation header every time any memory instruction is executed).

Additionally, while porting No-FAT's spatial memory protection to non-64-bit systems is possible, the temporal memory aspect strictly requires 64-bit systems in order to store the temporal tags in the upper bits of the data pointers. On non-64-bit systems, temporal memory safety can be achieved with less efficient approaches such as free list randomization and memory quarantining.

Handling Gnarly, Gory C Idioms. Some C programmers have the propensity to exploit undefined behaviors. By undefined behaviors we mean behaviors that are not explicitly disallowed in the standard. These issues are documented in an excellent exposition by the authors of the CHERI system [87]. One of the most common of these idioms is the case of intentionally creating out-of-bounds pointers. Although it is unclear why programmers follow this idiom, it exists, and we strive to identify such cases.

Consider the following: `q = p + 100; x = Bar(q);`. Here, `p` is a pointer to a 32B allocation. After the arithmetic operation, `q` is an out-of-bounds pointer that will be passed to function `Bar`. Inside `Bar`, the program may do `z = q - 100` before using `z` to access memory. Since No-FAT only uses intra-procedural analysis, `Bar` will recompute the base address for `q` at function entry

using `compute_base`. However, as `q` is already an out-of-bounds pointer, the resultant base address will be wrong (i.e., it will not point to the original object pointed to by `p`). We refer to this case as a data pointer escape operation. If the compiler performs only intra-procedural analysis to determine and encode pointers into checking loads and stores, it will result in an insufficient check. To handle this case, No-FAT uses `verify_bounds` to catch out-of-bounds pointers before they escape to memory (or a different function), as described in Section 4.2.5. This functionality should help programmers catch undefined behavior and fix it.

One special case of the benign out-of-bounds pointers is the off-by-one pointer. For a given array, `a[N]`, the C standard permits the programmer to generate pointers to elements `a[0]`, `a[1]`, up to `a[N]`. While the last element does not exist, it is permitted to generate a pointer to it. However, any attempt to dereference such a pointer should result in a memory safety error. In order to support the off-by-one pointers (i.e., avoid generating an alarm with `verify_bounds` whenever they are created), we may add one padding byte for all memory allocation requests before invoking the underlying memory allocator. This way No-FAT can support off-by-one pointers without causing security violations (as the additional byte does not contain useful data) or performance degradation (as the binning memory allocators already round up the requested allocation sizes).

Strengths. Compared to many other systems, No-FAT provides deterministic memory safety guarantees at the finest granularity. No-FAT provides protection against a wide variety of spatial/temporal memory safety violations including control flow hijacking attacks, data oriented attacks, and pure data corruption. No-FAT's protection comes with minimal performance overheads and minor hardware changes. Furthermore, No-FAT naturally mitigates a common speculative execution threat (Spectre-V1) at no additional cost.

Weaknesses. Buf2Ptr requires the precise type information of an allocated object. While this is guaranteed for C++ objects, it is not always possible in C-style programs where `void*` allocations may be used. In these cases, the compiler may not be able to infer the correct type, in which case intra-allocation support may be skipped. This is a common limitation for techniques that rely on source-level transformations for intra-allocation protection [6]. Our evaluation results

in Section 4.7.5 show that the number of intra-allocation buffers is minimal compared to total allocations. Cases with ambiguous types are not common and can be properly handled with program annotations, if needed. We leave this part to future work.

4.7 Evaluation

We evaluate No-FAT across multiple dimensions. First, we measure the hardware overheads of No-FAT. Second, we compare the performance of No-FAT against state-of-the-art pre- and post-deployment memory safety solutions using SPEC CPU2017. Third, we analyze No-FAT's memory overheads. Fourth, we analyze the costs of No-FAT's temporal tags. Fifth, we evaluate Buf2Ptr by estimating its memory and performance for all benchmarks.

4.7.1 Hardware Overheads

No-FAT requires minimal hardware changes. Qualitatively, No-FAT requires a 1KB `MAST` and extra logic to compute the allocation base address (namely, one subtract, one shift, two 64-bit multipliers) and the bounds checking module (namely, one subtract, one shift, one 64-bit comparator, and one 16-bit comparator). As the bounds checking operations happen in parallel to the L1 data and tag accesses, processor clock frequency should not be impacted. We quantified these overheads by adding No-FAT to a typical energy optimized 32KB direct mapped L1 cache. We implement our modules using Verilog and synthesize them with the Synopsys design compiler and the 45nm NangateOpenCell library. We generate the SRAM arrays (for `MAST` and the tag/data arrays) with OpenRAM [88].

Table 4.1 summarizes our VLSI implementation results. The timing delay of the bounds checking module is minimal ($0.81ns$) as it uses a pipelined design that first fetches the allocation size from `MAST` and then does a subtraction followed by comparison operations. This latency can be overlapped with the access latency of L1 cache. The bounds checking module adds 6% additional area compared to the L1 data cache. This area is dominated by the SRAMs of `MAST` and the two comparators. On the other hand, the base computing module (which is invoked by

the compute_base instruction) area is dominated by the 64-bit multiplier. The module latency can be further optimized with a more customized multiplier.

Table 4.1: Area, delay and power overheads of No-FAT (GE represents gate equivalent).

Hardware Structure	**Area (GE)**	**Delay (*ns*)**	**Power (*mW*)**
Baseline L1 data cache	503,914	1.99	29.7
Bounds checking module	32,130	0.81	1.16
Base computing module	27,346	1.50	1.17

4.7.2 Software Performance Overheads

Our VLSI measurements show that No-FAT hardware modifications add no performance overhead. Here, we evaluate the software-based overheads. No-FAT instructions secure_load and secure_store are similar to regular loads and stores. Thus, they do not increase code size. While our instructions use one more register operand compared to regular memory instructions, the extra register pressure is compensated for by adding a No-FAT-specific register file (i.e., similar to Intel MPX). The additional functionality performed by our instructions can be totally hidden within the processor pipeline as shown in Section 4.7.1. However, No-FAT requires a binning memory allocator and invokes additional instructions (verify_bounds to verify pointer bounds before storing them to memory and compute_base to compute the allocation base address of arbitrary pointers when they are loaded from memory).

Without loss of generality, we implement No-FAT on top of a binning allocator (Binning-Malloc [41]) that divides the virtual memory into 64 regions, each of size 32GB. Each region is used to satisfy heap-allocation requests of a unique size. Stack and global memory allocations are satisfied using special carved out sections of the same 32GB regions. To estimate the compute_base instruction overheads, we implement an IR pass using the LLVM/Clang compiler [89] to instrument the code and insert two mul instructions followed by a store instead of compute_base instructions in the corresponding locations. Similarly, we insert dummy store instructions in place of verify_bounds instructions. We use a store to make sure

the instruction is not omitted by compiler optimizations.

Evaluation Setup. We run our experiments on a bare-metal Intel Skylake-based Xeon Gold 6126 processor running at 2.6GHz with RHEL Linux 7.5 (kernel 3.10). We implement No-FAT using `Clang 4.0.0` and compare it against AddressSanitizer (ASan) and Intel MPX, as representatives of pre- and post-deployment memory safety solutions, respectively. Each tool is run using its best recommended settings [38]. We run each tool such that it suppresses its warnings or errors so that benchmarks run to completion. Additionally, we disable any reporting to minimize the performance impact this functionality may have. Given the difference in compiler versions and optimization levels that each tool supports, we normalize each against their respective baselines for proper comparison.[4] To get better insight on No-FAT overheads, we also run a software-only version of No-FAT that explicitly checks pointer bounds in software with no hardware support (Software-EBB) and a malloc-only version that only uses the binning memory allocator with no bounds checking (Binning-Malloc). We use the SPEC CPU2017 benchmark suite with `ref` inputs and run to completion. To minimize variability, each benchmark is executed 5 times and the average of the execution times is reported.

Performance Results. Figure 4.3 summarizes the performance overheads of SPEC CPU2017 for different tools normalized to their corresponding baseline. The geometric mean of each tool is as follows: ASan (2.07x), MPX (2.06x)[5], Software-EBB (2.0x), Binning-Malloc (1.04x) and No-FAT (1.08x). The main reason for No-FAT overheads comes from the underlying memory allocator, which introduces 1.04x overheads. For example, `gcc` allocates many small objects that are padded to the nearest Binning-Malloc size. As a result it introduces 62% extra runtime with No-FAT, whereas its Binning-Malloc version has a 55% slowdown. Configuring the allocation sizes at program initialization should reduce the padding and the overheads.

[4] We use `Clang 7.0` for ASan and `GCC 7.3.1` for Intel MPX.

[5] `gcc`, `perlbench`, `namd`, and `blender` failed to run with MPX due to unrecoverable errors. Thus, we exclude them from MPX averages.

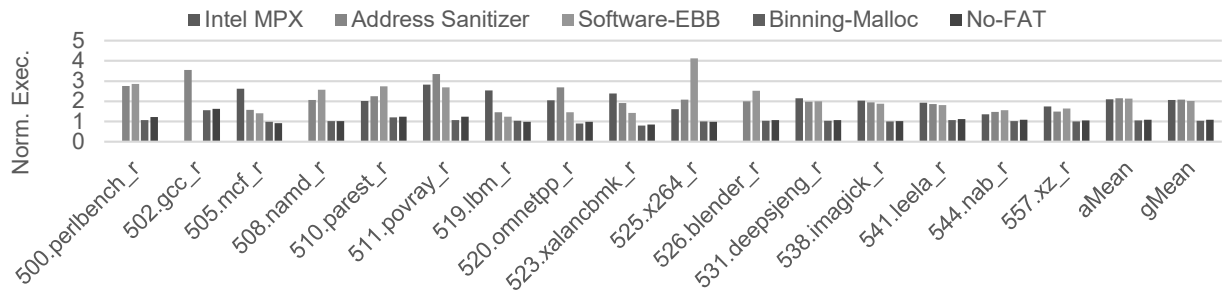


Figure 4.3: Performance overheads of the SPEC CPU2017 benchmarks for different tools normalized to their corresponding baseline.

4.7.3 Software Memory Overheads

To accurately measure the memory usage of No-FAT, we use a Linux-based utility, Syrupy, that regularly takes snapshots of the memory of a running process [90]. We measure the peak resident set size (RSS) to get the actually used memory rather than virtual address space which is reserved. Table 4.2 shows that the No-FAT's binning allocator only adds 6.66% memory overheads on average compared to the `stdlib` allocator with `gcc`, `parest`, `povray` as outliers. We inspect those allocation intensive benchmarks by running them with six different memory allocators (including No-FAT). Figure 4.4 shows that No-FAT memory overheads are comparable to other binning (i.e., Jemalloc [40], TCmalloc [39], and Scudo [91][6]) and non-binning allocators (i.e., Dlmalloc [50]).

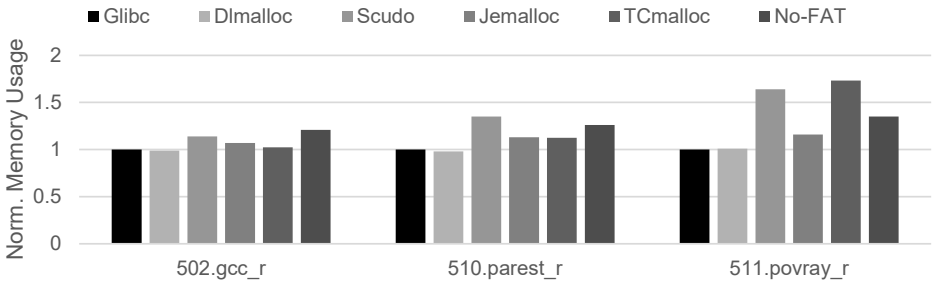


Figure 4.4: Memory usage for the three allocation-intensive benchmarks with different memory allocators.

[6]Scudo is a hybrid allocator that allocates similar-sized objects using bins and uses a per-object header for storing metadata as well.

Table 4.2: Memory usage for SPEC CPU2017 benchmarks.

Bench.	**Memory usage (MB)** No-FAT	**# of Heap allocations** Buf2Ptr
perlbench	[+3.27%] 160.80	[+48.7E0] 54.2E6
gcc	[+20.96%] 1,555.57	[+199.3E3] 2.7E6
mcf	[+0.07%] 610.64	[0.0E0] 495.4E3
namd	[-3.71%] 156.53	[0.0E0] 20.2E3
parest	[+26.05%] 527.07	[+107.4E6] 265.1E6
povray	[+35.04%] 8.75	[+10E0] 63.4E3
lbm	[+0.04%] 411.66	[0.0E0] 2.0E0
omnetpp	[+3.79%] 251.36	[+1.7E6] 454E6
xalancbmk	[+6.95%] 512.83	[0.0E0] 138.4E6
x264	[+1.50%] 159.40	[+1.5E0] 2.2E3
blender	[+12.42%] 710.61	[+3.4E6] 9.1E6
deepsjeng	[+0.25%] 702.54	[+15.0E6] 15.0E6
imagick	[-0.91%] 285.05	[+1.0E0] 9.3E6
leela	[-7.01%] 23.56	[+1.2E3] 53.8E6
nab	[+7.74%] 159.03	[+38.1E3] 374.5E3
xz	[+0.04%] 727.30	[+0.0E0] 41.0E0

4.7.4 Temporal Memory Safety Analysis

Storing a 16-bit tag as part of an object does not cause additional memory overheads as the binning allocator already rounds allocation-sizes up to the nearest bin size. Using bin sizes from Table 4.3, we collected heap allocations statistics for the SPEC CPU2017 benchmarks at runtime with the reference input set.

Table 4.3: Configuration sizes per each bin in the No-FAT binning memory allocator.

Sizes
16, 32, 48, 64, 80, 96, 112, 128, 144, 160, 192, 224, 256, 272, 320, 384, 448, 512, 528, 640, 768, 896, 1024, 1040, 1280, 1536, 1792, 2048, 2064, 2560, 3072, 3584, 4096, 4112, 5120, 6144, 7168, 8192, 8208, 10240, 12288, 16KB, 32KB, 64KB, 128KB, 256KB, 512KB, 1MB, 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, 128MB, 256MB, 512MB, 1GB, 2GB, 4GB, 8GB, 16GB, 32GB

Table 4.4 shows the total number of heap allocations and the fraction of allocations that have fewer than two padding bytes after being rounded up to the nearest bin size. For the majority of benchmarks, allocations already have more than two extra bytes that can be used for temporal tag

storage. The only exceptions are `omnetpp` and `imagick`, which allocate objects of irregular sizes. For example, 50% of the allocations made by `imagick` are of size 79, which will have only one byte after being rounded up to the nearest allocation size, 80. In this case, No-FAT satisfies the allocation request by using the adjacent bin to guarantee that the last two bytes of the object are unused. With this approach, for `imagick`, we noticed no additional runtime overheads over a bin size of 80. Alternately as the bin sizes are configurable for each process, we can adjust them to take the extra two bytes into account.

Table 4.4: Number of heap allocations that require extra padding bytes with No-FAT in SPEC CPU2017 benchmarks.

Bench.	**Total number of heap allocations**	**Allocations that have less than two padding bytes**	
		(#)	**(%)**
mcf	495,305	0	0%
namd	20,227	4	0.02%
parest	157,697,392	24	≈ 0%
povray	62,002	169	0.27%
lbm	2	0	0%
omnetpp	452,336,434	84,193	0.02%
xalancbmk	138,365,251	4	≈ 0%
x264	3,431	5	0.15%
deepsjeng	1	0	0%
imagick	37,234,132	18,616,657	50.0%
leela	53,759,984	0	0%
nab	336,412.00	0	0%
xz	41	0	0%

4.7.5 Buf2Ptr Analysis

Memory. The memory overheads of Buf2Ptr are reported separately. As Buf2Ptr promotes intra-allocation buffers to standalone allocations, it adds additional heap allocations as reported between brackets in the last column of Table 4.2. The majority of benchmarks add few extra allocations with the exception of `parest`, `blender`, and `deepsjeng`. The latter is interesting as it performs a single malloc call to 15 million structs, each with one intra-buffer. So, even though the extra

allocations look large, the actual source code transformation is minimal as it affects one struct.

Performance. The performance overheads of promoting intra-allocation fields to standalone allocations are amortized over program execution. For example, a buffer field of size 64B typically requires eight 8B loads in regular execution. With Buf2Ptr, one can argue that 16 8B loads will be needed as every load now passes through one level of indirection. However, as we implement Buf2Ptr as a source level transformation, we take advantage of the compiler to optimize the access to only 9 8B loads (i.e., one load to get the new base address followed by the original 8 loads with their address adjusted to the new base). We verify this hypothesis by measuring the overhead of implementing Buf2Ptr for the C programs in SPEC CPU2017 benchmarks. The overheads are less than 1% compared to a baseline that uses the same memory allocator (without Buf2Ptr).

4.8 Summary

In this chapter I discussed No-FAT, a secure architecture for implicitly deriving allocation bounds. No-FAT enforces spatial memory safety and a degree of temporal safety without increasing program memory footprint, while maintaining full compatibility with unprotected code. Overall, No-FAT incurs 8% performance degradation compared to a 100% slowdown for its software version, while providing extra security guarantees. This has the benefits of reducing fuzz-testing overheads to improve pre-deployment software testing. Furthermore, if end users are willing to pay 8% performance degradation for memory safety protection, then No-FAT is an excellent solution. The benefits of No-FAT go well beyond memory safety: for instance, having the allocation size as an architectural feature can help accelerate garbage collectors for memory safe languages; it also provides an opportunity for enhancing the predictability of memory prefetchers and DRAM controllers.

Chapter 5: A Counter C-4 Architecture

There is a lot of interest in hardware support for mitigating memory safety errors, as evidenced by SPARC's ADI [4], Intel's MPX [38], and ARM's MTE [5]. This interest has been motivated by memory safety attacks on critical systems over the course of several years [16]. The holy grail of research in this area is realizing a technique that provides strong security guarantees, has low runtime and memory overheads, and requires minimal hardware changes. A recent work from Intel Labs, called Cryptographic Capability Computing (C^3) [44], claims to provide all of the above features with negligible overheads.

In this chapter I analyze the security claims of C^3 and describe four attacks to compromise it, C-4. The attacks exploit C^3's fundamental design choices, such as (1) using a fixed one-time pad for per-object data encryption, (2) lacking bounds checking on pointer arithmetic and usages, (3) providing low entropy against temporal safety violations, and (4) leaving the application's stack, global, and intra-allocation objects unprotected. Thus, those attacks can bypass the C^3 spatial and temporal memory safety guarantees, in addition to breaking its data confidentiality.

Naively addressing the proposed attacks will require redesigning C^3 to use bounds checking and a stronger cipher, which will result in high performance overheads and negate C^3's stateless and compatibility claims. Thus, I propose an alternate implementation for the cryptographic isolation without incurring significant overheads. This new implementation counters the attacks in C-4, and hence we call it the Counter C-4 Architecture (or C-5 [8]). C-5 provides strong memory safety guarantees without suffering from the shortcomings of C^3. C-5 relies on two main principles to counter C-4: (1) using strong access-control rules to prevent memory safety violation while data is processed in the L1 data cache and (2) applying a strong cipher to protect the application data confidentiality in L2 cache and beyond. C-5 uses No-FAT as a baseline and implements multiple optimizations to reduce its performance cost and boost its security guarantees.

The remainder of this chapter is organized as follows. Section 5.1 provides further motivation. Section 5.2 describes how the C^3 architecture works. Section 5.3 details the security assessment of the C^3 claims and describes multiple attacks to compromise it, C-4. Next, Section 5.4 presents an end-to-end attack against C^3. Afterwards, Section 5.5 specifies the details of the proposed solution, C-5 and shows how it addresses all the shortcomings of C^3 and mitigates C-4. Section 5.6 discusses the security benefits of C-5 and its limitations. Section 5.7 evaluates the performance cost and security guarantees of C-5. Section 5.8 summarizes the chapter.

5.1 Motivation

While traditional memory safety techniques are good at detecting spatial and temporal software memory safety violations, the recent development of hardware vulnerabilities that can leak secrets (e.g., hardware side-channels [92, 93, 94], ColdBoot [95]) or corrupt memory (e.g., RowHammer [77]) need different specialized approaches. The use of multiple security countermeasures complicates the deployment of hardened software especially when operational resources and budgets for security are limited. As pointed out by Saltzer and Schroeder [96], an economy of mechanism is valuable to handle multiple software and hardware memory security issues. Thus, there is a need for developing solutions that can cohesively mitigate both memory safety violations, and hardware-based vulnerabilities with minimal performance, memory, and complexity overheads.

5.2 How Does Cryptographic Capability Computing (C^3) work?

C^3 enforces memory safety by strongly associating each pointer with its legitimated pointed-to memory allocation so that pointers cannot reliably access unrelated allocations [44]. C^3 achieves this goal by encrypting portions of the pointer itself to prevent pointer forgery and encrypting the object contents before storing them in memory using the pointer as a key. In this section, we explain how C^3 interacts with the allocation life-cycle, Then, we summarize C^3's main design choices and define the threat model.

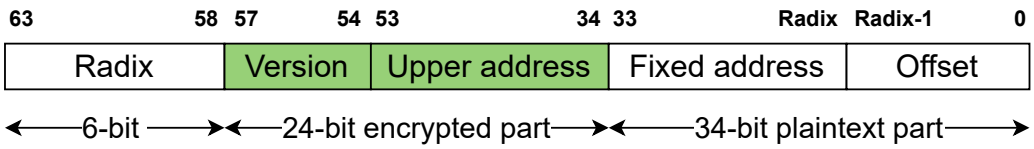


Figure 5.1: The C^3 pointer format as proposed in the original paper [44].

5.2.1 The Object Life-cycle in C^3

(1) Object Allocation. When an object is allocated on the heap, a memory management function such as `malloc` or `new` returns a plaintext pointer to the application to use. C^3 changes the conventional pointer format, as shown in Figure 5.1, to create the following fields.

- **Radix:** A 6-bit field that identifies which portion of the pointer is constant throughout the entire allocation.
- **Version:** A 4-bit value that is changed every time the same memory region is assigned to a new object for mitigating temporal safety violations.
- **Encrypted address:** A 24-bit encrypted range that includes the original upper pointer bits and the “Version" field.
- **Fixed address & Offset:** A 34-bit plaintext range that is left for the application to update without cryptographic operations.

While C^3 is agnostic to the memory allocator used by the applications (i.e., it does not change how objects are laid out in memory), it uses a best-fit algorithm to assign each allocation suitable “Radix" and “Fixed address" values and generates a random “Version" field. Then, C^3 encrypts the pointer using a tweakable block cipher, called the K-cipher [97], as shown on top side of Figure 5.2. The inputs to the encryption module are the plaintext pointer, a per-process encryption key, a per-object tweak, and the version. Only the middle portion of the pointer is encrypted, namely bits[34:57]. Those bits are first concatenated with a 4-bit “version" value, which can be assigned randomly or in sequence for a given memory slot for temporal memory safety. The tweak is

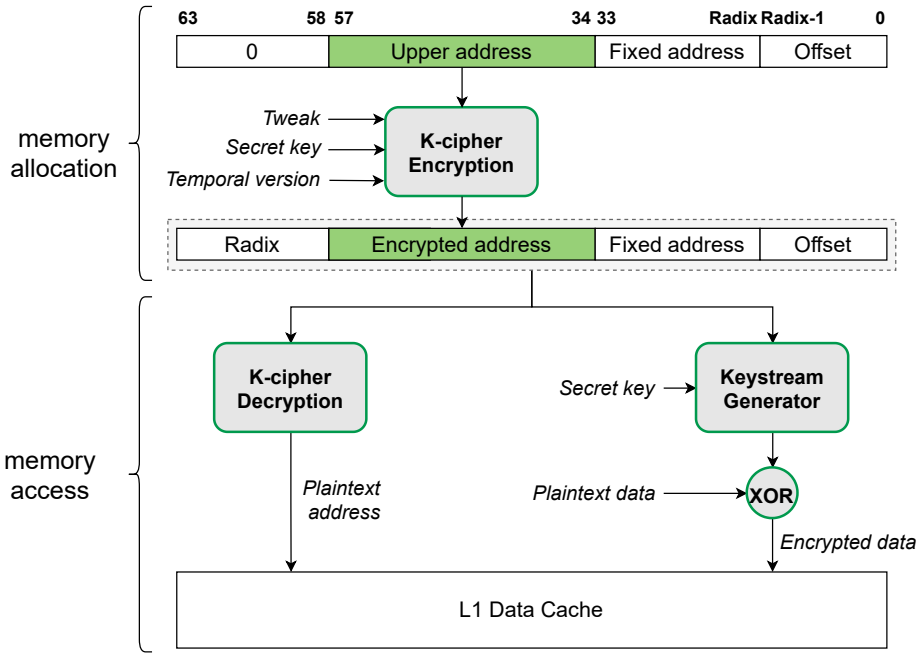


Figure 5.2: A high level overview of how C^3 [44] creates formatted pointers upon `malloc` and uses them during memory access operations.

generated by concatenating the 6-bit “Radix" field with a 34-bit padded “Fixed address" that is obtained from the plaintext pointer. The reason why the fixed address bits are included in the tweak is to ensure the application cannot reliably tamper with them.

The 34-bit lower portion of the pointer is left as plaintext to allow regular program instructions (such as pointer arithmetic operations) to update them without any further changes. Thus, the maximum allocation size that can be protected with the current C^3 format is 16GB. Finally, the pointer formatting operation can either be done purely in software or using a new `allocate <rb>, <rc>` instruction, where `rb` holds the input/output pointer and `rc` holds the per-object “Radix" and “Version" bits.

(2) Object Access. When a pointer is used to access a memory object as part of a regular load or store instruction, C^3 first detects whether the pointer is formatted (i.e., has the encrypted portion) or not by examining the “Radix" bits. Those upper 6-bits are set to all zeroes for unformatted pointers and to non-zero values for C^3 formatted pointers. In the case of having a formatted pointer, it is first used along with a per-process key to seed a keystream generator. The output of this generator will

then be used as an XOR mask to encrypt the data before being stored in the L1 data cache, as shown in the bottom side of Figure 5.2. Additionally, the formatted pointer will be decrypted by using the K-Cipher to generate the plaintext memory address before accessing the cache. The original C^3 paper discusses microarchitecture optimizations to hide the latency of pointer decryption during memory access [44].

(3) Object De-allocation When the object is deleted by calling `free` or `delete` on its base pointer, C^3 decrypts the pointer before passing it to the memory allocator de-allocation function. C^3 mitigates the threat of dangling pointers by assigning a different "Version" field to new objects that are assigned the same virtual memory region of the deleted object in the future.

5.2.2 C^3's Design Choices

C^3 made a set of design choices in order to provide an efficient stateless memory safety solution [44]. First, a pre-computed keystream is used to encrypt the allocation data in order to minimize the cryptographic load and store latency and hence reduce the overall performance overheads of C^3. Second, C^3 opts to encrypt parts of the pointers for spatial memory safety instead of using bounds checking with additional metadata. Third, C^3's new pointer format only assigns 4 bits for temporal memory safety protection. Fourth, C^3 prioritizes legacy application protection over complete memory safety by only protecting the heap-allocated objects and leaving the stack, global, and intra-allocation heap objects unprotected. In Section 5.3, *we show how the above design decisions lead to C-4: compromising the security guarantees of C^3*.

5.2.3 Threat Model

We consider a threat model that is consistent with state-of-the-art memory safety defenses, including C^3 [44]. Specifically, we assume that the attacker is aware of the applied defenses and has access to the binary image or source code of the target application. Additionally, the target application is written in a memory unsafe language, such as C/C++, and suffers from one or more memory safety vulnerabilities that allow the attacker to read from and write to arbitrary memory

addresses. The attacker's goal is to (ab)use the memory safety bugs to leak or overwrite memory contents and/or achieve privilege escalation. Finally, we assume that all code sections are non-writable (i.e., immutable code). Thus, attacks that might modify program code at runtime, such as rowhammer [77] and CLKSCREW [98], are out of scope.

5.3 C-4: Assessing the Security Guarantees of Cryptographic Capability Computing

In this section, we describe multiple software-only attack vectors against the C^3 architecture.

5.3.1 Attack #1: Exploiting the XOR-based Data Encryption

In order to uniquely encrypt each object in memory, C^3 XORs the contents of the object with a pre-computed keystream before writing them to memory and after reading them from memory. The keystream itself is generated using the formatted pointer (and a per-process key) as inputs. This way, the keystream generation is triggered as soon as the address generation unit prepares the formatted address for the load/store instruction, and thus data encryption introduces no delays to the critical load path. We show that the aforementioned design decision trades off security for performance as the object-level encryption can be bypassed with a simple out-of-bounds heap vulnerability.

Attack Description. Our key insight is that most memory objects are initialized with known values. These values can be leaked to an attacker by simply inspecting the source code or the binary image of the victim application. For example, consider an attacker who wants to use an out-of-bounds read vulnerability in allocation X to read the contents of an adjacent heap allocation Y. As per the C^3 construction, the two allocations will have two distinct keystreams, K_X and K_Y, respectively. Hence, the over-read data (using a pointer from X) will be garbled as following:

$$\begin{aligned} Y_{garbled} &= Y_{encrypted} \oplus K_X \\ &= \{Y \oplus K_Y\} \oplus K_X \end{aligned} \tag{5.1}$$

The attacker can learn the plaintext of `Y` by triggering the vulnerability twice. First, for a known

value, Y_1, the attacker reads the garbled data:

$$Y_{garbled\_1} = \{Y_1 \oplus K_Y\} \oplus K_X \tag{5.2}$$

As both Y_1 and $Y_{garbled\_1}$ are known, the attacker computes $A = K_Y \oplus K_X$. Second, the attacker uses A to encrypt/decrypt any later usage of Y. For example, after Y is updated by the victim, the attacker triggers the read vulnerability again to get:

$$\begin{aligned} Y_{garbled\_2} &= \{Y_2 \oplus K_Y\} \oplus K_X \\ &= Y_2 \oplus A \end{aligned} \tag{5.3}$$

As both A and $Y_{garbled\_2}$ are known, the attacker recovers the unknown secret, Y_2.

Similarly, an attacker with an out-of-bounds write capability can write arbitrary values V to the allocation Y using the computed term A because $Y_{malicious} = V \oplus A$ will be written to memory as $V \oplus A \oplus K_X = V \oplus K_Y$, which matches the format of legitimately encrypted data at allocation Y.

Potential Mitigation. Mitigating the above attacks requires encrypting the data itself instead of XORing it with a pre-generated keystream. This solution, however, will introduce significant performance overheads for C^3 as data encryption cannot be performed in parallel to regular data access any more. While using the same keystream for XOR-based encryption is an issue that has been discussed before in different contexts (e.g., bypassing instruction set randomization [99, 100]), it is important to point it out to avoid making the same mistakes again while mitigating memory safety.

Takeaway. Using XOR-based encryption to protect an object's contents in memory breaks data confidentiality if the object contents are set to known values during program execution (e.g., during initialization).

5.3.2 Attack #2: Targeting Spatial Memory Safety

One main design choice in C^3 is to use pointer encryption instead of authentication for mitigating pointer corruption and forgery. This way, C^3 avoids using dedicated pointer bits for storing authentication codes as in ARM's PAC [10] or allocation colors as in ARM's MTE [5]. The original C^3 work stated that "*Pointers do not require special protections or tags beyond the cryptographic encoding itself.*" [44]. However, C^3 only encrypts the middle 24 bits of the pointer and leaves the rest unencrypted to be compatible with traditional pointer arithmetic instructions. This design choice allows an attacker to violate spatial memory safety without having to break the encryption scheme.

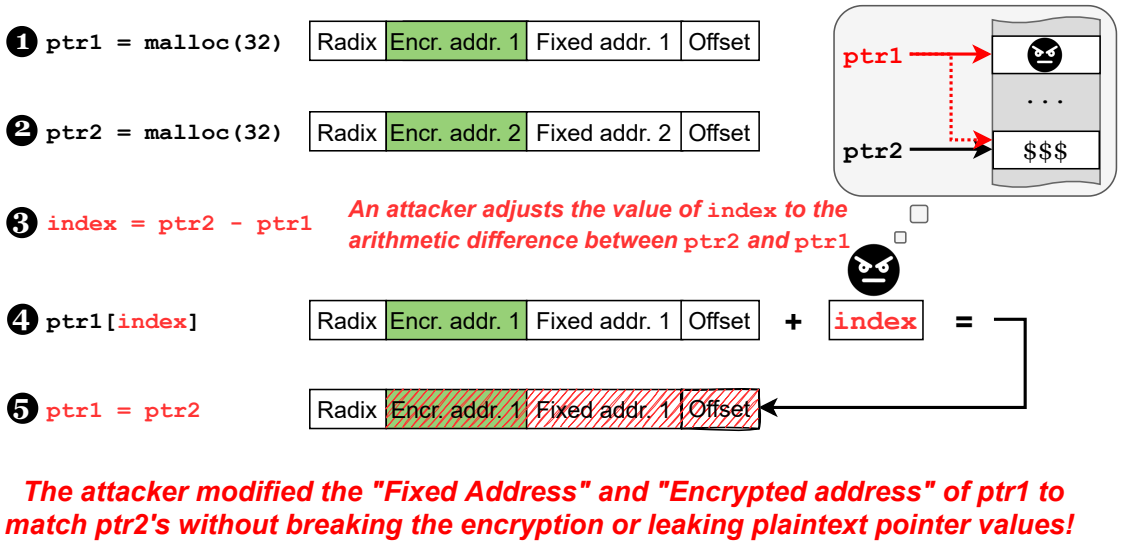


Figure 5.3: Targeting C^3's spatial memory safety.

Attack Description. To better understand how our attack works, let us consider the code snippet from Figure 5.3, in which the program creates pointers to two different objects, `ptr1` and `ptr2`, and uses an `index` to access the contents of the first object that is pointed to by `ptr1`. As `ptr1` and `ptr2` point to similarly-sized objects, they will have the same "Radix" value but different "Fixed address" fields. As the "Fixed address" is used as a tweak during pointer encryption, the "Encrypted address" fields in `ptr1` and `ptr2` are different. Given a memory safety vulnerability, an attacker can control the value, "index". The attacker's goal is to overwrite the contents of the

second object, which is pointed to by `ptr2`. As pointer arithmetic operations can be performed on the formatted pointer without restrictions, an attacker can simply adjust the value of “index" to match the arithmetic difference between `ptr1` and `ptr2` (i.e., `index = ptr2 - ptr1`). This way, adding “index" to `ptr1` will create a forged pointer that matches `ptr2`’s “Fixed address" and “Encrypted address" fields without having to break the encryption scheme or revealing the plaintext pointer value.

But how can an attacker gain access to the formatted values of `ptr1` and `ptr2` in order to compute their difference? There are several options for doing so. The first option is to leak the pointers while they are stored in global variables [101, 102, 103], as C^3 is currently applied for heap objects only (to avoid software recompilation). The second option is to use our XOR-based attack to leak the contents of arbitrary objects, including any pointers that are stored within those leaked objects. The third option is to use an arbitrary-read memory safety vulnerability to leak the formatted pointer values (e.g., using a format string attack [104, 105, 106, 107, 108, 109]).

Potential Mitigation. The root cause of our spatial memory safety attack is the lack of bounds checking. Unlike other memory safety solutions, C^3 has no metadata that can be used to validate the pointer (e.g., the per-instruction trusted base address in No-FAT [7] or the per-pointer base and bounds in CHERI [29]). If a pointer’s bounds information exists, manipulating the pointer bits can be easily detected when the pointer is used to access memory as part of a load/store instruction (as the manipulated pointer will not match the legitimate base and bounds).

One way to fix this problem in C^3 is to instrument pointer arithmetic operations to ensure that they do not overflow to the “Encrypted address" field. This solution, however, has its own challenges as it (1) requires program recompilation to identify pointer arithmetic operations and replace them with special instructions or (2) adds per-word tags in memory to distinguish pointers from regular data in order to allow traditional arithmetic instructions to operate differently on both. In either case an extra latency will be added to the simple pointer addition operation, which will increase performance overheads. Alternatively, this attack can be mitigated if the formatted pointers are never leaked to the attacker. Such perfect pointer confidentiality requires (1) enforcing

memory safety on the stack and global segments and (2) encrypting the application data itself in memory.

Takeaway. Applying bounds checking using a per-instruction, per pointer, or per-object metadata is necessary to achieve spatial memory safety.

5.3.3 Attack #3: Undermining Temporal Memory Safety

The original C^3 paper claims that its temporal memory safety guarantees are 1M times stronger than ARM's MTE due to its having a 24-bit encrypted slice in the pointer as opposed to a 4-bit tag in ARM's MTE. Next, we show that the aforementioned claim does not always hold.

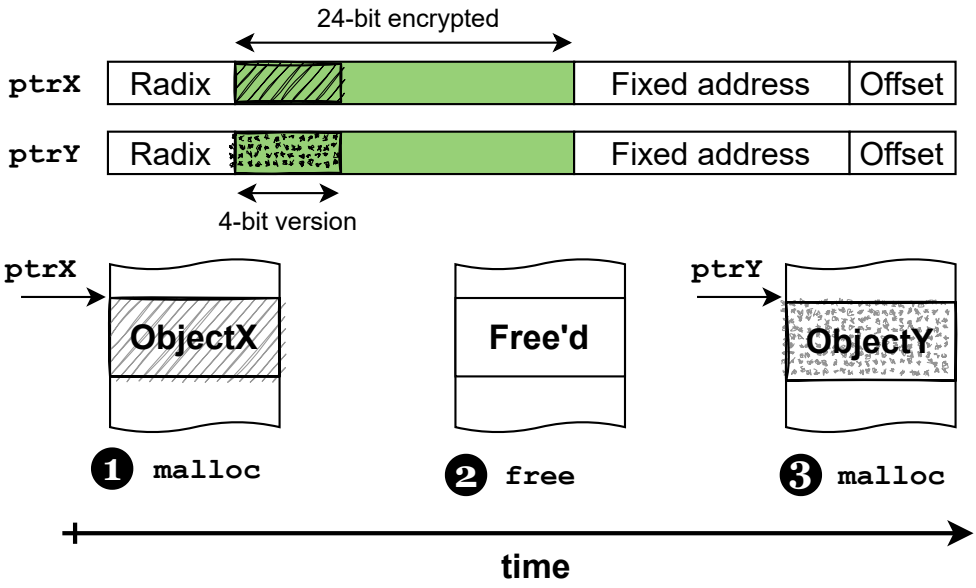


Figure 5.4: Undermining C^3's temporal memory safety.

Attack Description. In typical use-after-free vulnerabilities, the attacker triggers the bug to free a victim's object while leaving a dangling pointer that is still pointing to the deleted memory location. Next, the attacker uses heap spraying to force the allocation of an attacker-controlled object on top of the same memory location that is pointed to by the dangling pointer. Any future usage of the dangling pointer will access the attacker-controlled contents, resulting in security exploitation. We have seen instances of such vulnerabilities being exploited in major real-world applications [110,

111, 112, 113]. As the 48-bit virtual address of the memory region is reused in UAF, the “Offset" and “Fixed address" parts of the new pointer (e.g., `ptrX`) and the dangling pointer (e.g., `ptrY`) will be identical, as shown in Figure 5.4. However, the “Version" and “Radix" fields might be different based on the underlying memory allocator.

For binning memory allocators, objects of similar sizes are allocated near each other in memory regions (also known as bins or arenas [40, 39, 42]). In this case, the “Radix" field of `ptrX` and `ptrY` will be identical. Thus, the only source of diversity inside the “Encrypted address" field in `ptrX` and `ptrY` is the 4-bit “Version" field. In other words, given a memory region that is being frequently deleted and allocated, the 24-bit encrypted address field will change $2^4 = 16$ times on average. Hence, the attacker needs to trigger the vulnerability at most 16 times in order to get a 24-bit encrypted slice in the attacker-controlled object that matches the 24-bit slice that is part of the dangling pointer. The situation will be even worse on future processors with 57-bit virtual address spaces, in which the number of unused upper pointer bits will become seven bits instead of 16. As C^3 requires six bits for radixes, only one bit will be available for temporal memory safety.

For non-binning (aka coalescing) memory allocators, the 6-bit “Radix" field of `ptrX` and `ptrY` might be different. As noted in the original C^3 paper [44], intentionally using a different “Radix" for the recently freed memory locations increases the temporal safety entropy to 4+6 = 10 bits. While 10 bits only requires 1024 trials, which are fairly easy to exhaust by an active attacker, C^3’s encryption further facilities the attack as follows. Unlike ARM’s MTE, which generates an exception upon tag mismatch, C^3 uses the mismatched tag to generate a keystream that will be used for accessing the memory object, resulting in a silent data corruption. First, the lack of synchronous exceptions allows the attacker to keep triggering the vulnerability without notifying the operating system. Second, if the victim object contains elements of limited outcomes (e.g., boolean fields), the attacker does not need to completely exhaust the 10-bit (or even 24-bit) entropy for successfully controlling the old memory region contents. Instead, the attacker needs to keep trying until they get an encrypted slice that results in a keystream, which causes the victim field

values to flip (from one to zero or zero to one) upon decryption.

Potential Mitigation. Addressing the above attack in C^3 requires either (1) increasing the size of the “Version" field, which might not be feasible due to the compact C^3 pointer formatting, or (2) redesigning how C^3 works in order to keep track of each object and pointer temporal tag such that an exception could be immediately generated upon a use-after-free violation.

Takeaway. The entropy of a temporal safety scheme is determined by the smallest source of variance in the scheme, i.e., the version field. Moreover, the lack of explicit temporal safety checks increases the attack success chances.

5.3.4 Attack #4: Understanding the Security Coverage

Attack Description. The security of a system is determined by the security of its weakest components. The same concept applies to programs as well. The original C^3 paper chose to protect heap allocations only and did not include stack and global segments in order to avoid program recompilation. While this design choice enables the benefits of protecting legacy binaries, leaving two major memory components unprotected can simply undermine the security guarantees of the whole system, including the heap allocated objects. For example, pointers to heap allocations can be stored on the stack and global segments, and thus they can be leaked to attackers, facilitating other attacks such as the spatial memory safety attack described in Section 5.3.2. In other words, not protecting the stack and global memory can cause harm beyond the stack and global objects themselves.

Moreover, C^3's heap protection only prevents inter-allocation violations (i.e., overflows from one allocation to another). C^3 does not prevent intra-allocation spatial memory safety violations (i.e., overflows from one field to another within the same allocation). The lack of intra-allocation protection is due to the fact that the “Radix" field (and hence the pointer and memory encryption) is determined based on the base pointer, which is returned by `malloc`, and not the derived pointers that might point to any sub-allocation. Using intra-allocation overflows, an attacker can read or write arbitrary pointers within the same allocation and use the leaked/corrupted pointers to trigger

further attacks. Sub-allocation vulnerabilities do exist [114] and can achieve high levels of reliable exploitation [115].

Potential Mitigation. Extending C^3 to protect the stack and global segments is possible with the help of compiler support. Protecting the stack buffers might introduce non-zero performance cost as the memory slot picking routine and the pointer formatting instructions will need to be executed for every stack buffer allocation compared to simply adjusting the stack pointer under baseline (i.e., unprotected) execution. Preventing the intra-allocation overflows could be achieved by using bounds-narrowing, as in Intel's MPX [38], or by using Buf2Ptr, as in No-FAT [7].

Takeaway. In order to provide complete memory safety, all memory segments of a program should be properly protected.

5.4 End-to-End Case Study

This section presents an end-to-end attack against C^3 using a real-world vulnerability in a JavaScript engine. We first describe the vulnerability and summarize an exploitation methodology against a baseline (insecure) system. Then, we explain how C^3 makes it challenging to abuse this vulnerability and how our proposed C-4 attacks overcome these challenges. Finally, we discuss other real-world vulnerabilities that could be used as starting points for our end-to-end attack.

5.4.1 Vulnerability Description

Our attack uses the CVE-2018-4192 vulnerability [116] from JavaScriptCore (JSC), the engine inside WebKit [117] and the Safari web browser. The vulnerability occurs due to a race condition between the JSC garbage collector and the `array.reverse()` method. The JSC garbage collector adopts a mark-and-sweep algorithm. It first uses standalone threads to scan the heap memory and mark live objects as used, as shown in Figure 5.5-❷. Afterwards, all unmarked objects are deleted during the sweeping phase. If the `array.reverse()` function is called on any JavaScript array during the marking phase (Figure 5.5-❸), some elements of the array (i.e., `Array[3-5]`) could skip being marked (Figure 5.5-❹) and hence will be deleted later during

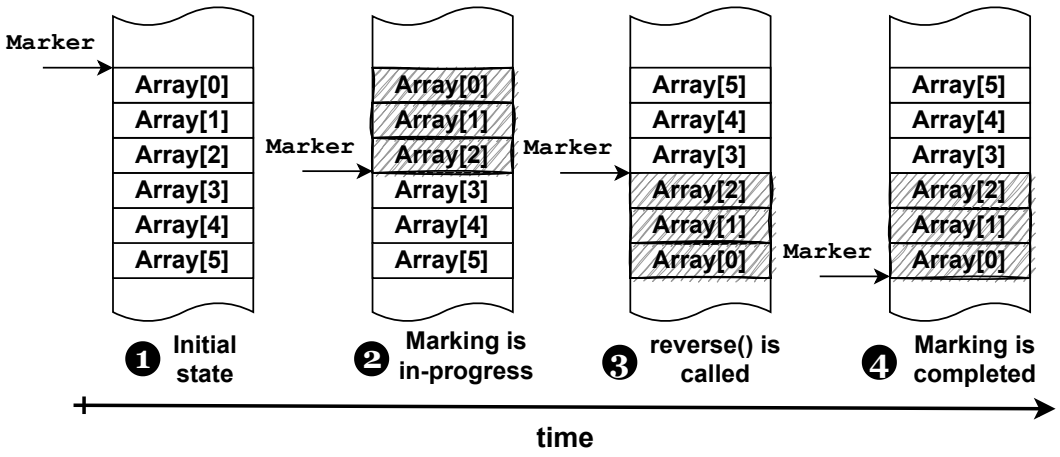


Figure 5.5: A high-level overview of CVE-2018-4192. A JavaScript array is being marked as live during an in-progress garbage collection marking phase. The shaded elements (`Array[0-2]`) are successfully marked. Due to a simultaneous call to `array.reverse()`, parts of the array (i.e., `Array[3-5]`) are not marked and hence will be deleted later during the sweeping phase, resulting in a UAF.

the sweeping phase. The incorrectly deleted memory regions will be available for new allocation requests, causing a UAF scenario, in which an attacker abuses the dangling pointers to the deleted elements for controlling the contents of the new allocations. A detailed root cause analysis of CVE-2018-4192 is available online [118] in addition to the implemented fix [119].

5.4.2 Baseline Exploitation Methodology

At Pwn2Own 2018, Burnett et al. exploited the CVE-2018-4192 vulnerability to achieve remote code execution in the context of the Safari Web Browser [120]. For completeness, we first summarize the original exploitation methodology before explaining our C-4 modifications.

(1) Triggering the vulnerability to cause a UAF. In this step, the attacker first creates multiple JavaScript arrays, `free_targets` where each entry of the array is basically an object. As JSC stores the object's properties and data elements in a separate memory location called the butterfly[1] (and maintains a pointer to that location in the object itself), the butterflies will be allocated similarly to Figure 5.5-❶. Next, the attacker initializes the arrays with float values and repeat-

[1]The name refers to the JSC object's layout, in which the butterfly pointer points to the middle of the storage region whereas object's properties and data elements are stored to the left and right, similar to the wings of a butterfly.

```
1 // Get the address of a given JavaScript object
2 function addrof(object) {
3  array_target[0] = object;
4  return Int64.fromDouble(oob_capability[idx]);
5 }
6
7 // Return a JavaScript object at a given address
8 function fakeobj(addr) {
9  oob_capability[idx] = addr.asDouble();
10  return array_target[0];
11 }
```

Listing 5.1: The implementation of the `addrof` and `fakeobj` primitives for leaking the address of and creating fake JavaScript objects, respectively.

edly invokes `array.reverse()` on them. As the attacker uses multiple arrays, the chances of triggering the race condition between the `array.reverse()` function and the JSC garbage collection threads is high. This race condition causes the incorrect deletion of one or more of the `free_target`'s butterflies, leaving a set of dangling pointers. From a security standpoint, manipulating the contents of a butterfly is dangerous as it maintains a 32-bit "length" field that controls the size of the main object's elements. Corrupting this field causes an out-of-bounds access beyond the JavaScript object's legitimate size.

(2) Creating a relative read/write primitive using type confusion. Here the attacker creates one JavaScript array, `vuln`, and dynamically changes its sizes while triggering the race condition. The goal is to force the allocation of the `vuln`'s butterfly in one or more of the deleted memory regions. This way a type confusion occurs between the data elements of `vuln`, which are written by the attacker, and the contents of the `free_target`'s butterflies that include the "length" field. The type confusion allows the attacker to write arbitrary values to the 32-bit "length" field, leading to a relative out-of-bounds read/write violation. Checking the success of this step is done by reading the `free_target.length` for all `free_target` objects. If any abnormal values are found, that means the butterfly of this array entry has been deleted and is now a dangling pointer into a memory region, which is owned by the `vuln`'s butterfly.

(3) Building an arbitrary read/write primitive. The two aforementioned steps provide the attacker with a relative read/write capability as the "length" field is a 32-bit signed integer. To achieve

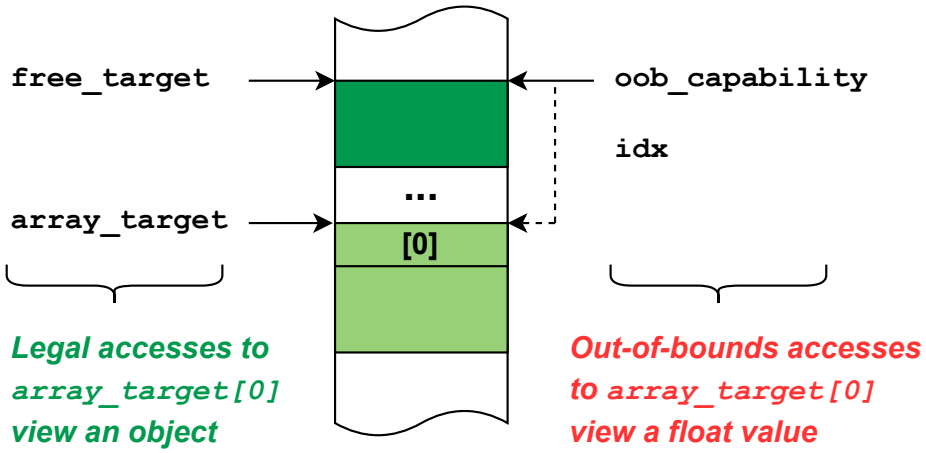


Figure 5.6: A visualization of how the `addrof` primitive works. Legally storing an object into an array and then maliciously reading it back as a float leaks the object's address.

an arbitrary read/write capability for accessing any location in memory, we need more generic primitives. Two well-known examples of these primitives are called `addrof` and `fakeobj` [121]. These primitives can be used for leaking the address of and creating fake JavaScript objects.

The key insight of the `addrof` primitive is that JSC uses a special 64-bit encoding to store integers, floats, and pointers as JavaScript values. Thus, storing an object into an array and maliciously reading it back as a float leaks the object's address (i.e., pointer). Hence, `addrof` works by first creating a JavaScript array, `array_target`, whose butterfly is located near (i.e., within a 2^{32}-byte offset) a `free_target`'s butterfly that has an abnormal "length" value. Then, the attacker uses normal JavaScript code to place any object into the first index of `array_target`, as shown in Listing 5.1 (Line 3) and the left-hand-side of Figure 5.6. Finally, the attacker abuses the relative read/write capability in the `free_target`'s butterfly (aka `oob_capability[idx]`) to read the address of the stored object (as a float) out of `array_target`, as shown in Listing 5.1 (Line 4) and the right-hand-side of Figure 5.6.

On the other hand, `fakeobj` works by first abusing the relative read/write capability to write a given address into the `array_target`'s butterfly as a float and then reading the same array index normally from JavaScript to return a JavaScript object for the stored pointer. Finally, the attacker uses the `addrof` and `fakeobj` primitives for crafting a fake JavaScript object to control

its metadata. For example, controlling the backing store pointer of JavaScript `TypedArrays` allows reading and writing arbitrary memory locations.

(4) Achieving arbitrary code execution. Given an arbitrary memory access capability, there are several ways to achieve code execution. One approach is to locate a stack address and then overwrite a return address to hijack the program control flow using return oriented programming. Alternatively, the attacker may use the read/write capability to (a) leak a pointer into the Just-In-Time (JIT) compiled code for a JavaScript function, (b) write a shellcode to this location as JIT memory pages are writable and executable, and (c) call the function, resulting in the shellcode being executed. Getting a JIT compiled code only requires creating a JavaScript function object and calling it multiple times to trigger JIT compilation.

5.4.3 C^3 Exploitation Challenges & C-4 Countermeasures

We analyze the potential challenges that C^3's pointer formatting and data encryption create against exploiting the above vulnerability and how the C-4 attacks overcome them.

Step (1). Since the CVE-2018-4192 vulnerability occurs due to how the JSC garbage collector and `array.reverse()` interact, an attacker can trigger the bug on a C^3-protected system, generating dangling pointers to one or more of the `free_target`'s butterflies.

Step (2). As C^3 assigns random 4-bit "Version" (and potentially 6-bit "Radix") values to reused memory regions, the probability of getting the same formatted address for the attacker-controlled `vuln`'s butterfly and the `free_target`'s butterfly is one in 2^{10}. However, C^3 does not keep track of the per-pointer and per-object "Version" and "Radix" fields, and thus no exception is generated upon mismatch. As a result, an exact address match is not required for achieving a type confusion because an unmatched formatted address can still produce a random number in the "length" field of the `free_target`'s butterfly upon decrypting the garbled data, as explained in Attack #3 (Section 5.3.3). Hence, our C-4 countermeasure is to repeat the step multiple times and use JavaScript to check for any abnormal (i.e., large enough) outcome in `free_target.length`. This way we can achieve a relative out-of-bounds read/write capability even if the exact value of the abnormal

length is not deterministically controlled by the attacker.

Step (3). The `addrof` and `fakeobj` exploitation primitives access out-of-bounds memory, `oob_capability[idx]`, relative to the base address of the `free_target`'s butterfly, as shown in Listing 5.1 Lines 4 and 9, respectively. Thus, garbled data will be written/read to/from the attacker-owned `array_target[0]`, preventing the exploit. To overcome this challenge, we use the C-4 Attack #1. Let us assume that the base address of the `free_target`'s butterfly is X, the nearby object, `array_target`, is Y, and the out-of-bounds access from X to Y is `oob_capability[idx]`. As explained in Section 5.3.1, we first compute $A = K_Y \oplus K_X$ given our prior knowledge of any plaintext initial value at the desired offset within Y. This requirement is straightforward to satisfy as `array_target` is owned by the attacker, who can initialize its elements to zero and its butterfly pointers to null. Once A is computed, the C^3's XOR-based data encryption is bypassed and we can reliably read/write data from/to `oob_capability[idx])`. While the accessed out-of-bounds items are essentially object addresses, they will be formatted as per the C^3 pointer format (See Figure 5.1). This is not an issue as we are not trying to reveal the plaintext addresses of the objects. Instead, we use the formatted address as is to interact with the process (e.g., we read the formatted address of an arbitrary object using `addrof` and create our fake object at this formatted address using `fakeobj`).

After crafting the fake object, we cannot write arbitrary memory addresses to its backing store pointer as all addresses need to be properly formatted as per C^3 in order to access memory contents. To overcome this challenge, we use our read/write primitives to read formatted addresses (e.g., pointers) from within known objects and use the leaked addresses to reach more live objects and disclose their contents if needed, inspired by how traditional JIT ROP attacks disclose randomized code pages on the fly.

Step (4). While C^3 restricts the arbitrary read/write primitive to traverse live objects, achieving remote code execution is still feasible. First, we retrieve the formatted address of any JIT-compiled function using our modified `addrof` primitive. Then, we use our C-4 Attack #1 to compute $A = K_Y \oplus K_X$ given our prior knowledge of the plaintext instructions of the JIT-compiled function as

the original JIT function and the JSC JIT compiler are both known. Finally, we use Attack #1 again to overwrite the JIT-compiled function memory with the shellcode XORed with A, resulting in $shellcode \oplus A \oplus K_X = shellcode \oplus K_Y$ to be written to memory. This way calling the function will retrieve and execute $shellcode \oplus K_Y \oplus K_Y = shellcode$ under the C^3 execution model.

5.4.4 Additional Real-World Exploits

Our C-4 can enhance other real-world exploits, which were discovered in recent years, allowing them to bypass C^3-protected systems. UAF-based examples include the CVE-2017-2491 vulnerability that was exploited in Pwn2Own 2017 [122] and the CVE-2017-2491 vulnerability that was used in the first stage of the Pegasus exploit [123]. Both exploits start with a UAF vulnerability and escalate to remote code execution. On the other hand, integer-overflow related exploits (e.g., the CVE-2019-8601 exploit [124]), exploits due to unexpected callbacks (e.g., the CVE-2016-4622 exploit [121]), and JIT compiler incorrect optimization bugs (e.g., the CVE-2018-4233 exploit [125]) similarly lead to remote code execution using a spatial memory violation as a starting point.

A common theme among the aforementioned exploits is that they abuse the initial temporal or spatial memory safety bug in the JavaScript engine to gain relative read/write primitives that are then escalated to arbitrary read/write capabilities with `addrof` and `fakeobj`, leading to remote code execution. As described in Section 5.4.3, our proposed C-4 attacks upgrade those exploitation steps to bypass C^3 defenses. For example, Attack #1 allows `addrof` and `fakeobj` to leak the C^3 formatted addresses of and create fake JavaScript objects whereas Attack #3 facilities the exploitation of UAF vulnerabilities. Finally, while Attack #2 and Attack #4 are not directly adopted in the above end-to-end case study, they can be used in other exploitation scenarios, as discussed in Section 5.3.

5.5 C-5: A Counter C-4 Architecture

We present C-5, which counters the attacks in C-4, and hence we call it the Counter C-4 Architecture (or C-5). C-5 provides strong memory safety guarantees without suffering from the

shortcomings of C^3. The two main principles that we will use to counter C-4 are (1) using strong access-control rules to prevent memory safety violation while data is processed in the L1 data cache and (2) applying a strong cipher to protect the application data confidentiality in L2 cache and beyond. While enforcing strong access control rules might increase the overall performance overheads, our No-FAT work has demonstrated that memory safety rules can be checked in hardware with low overheads. In this section, we illustrate multiple optimizations that reduce the performance cost of No-FAT and boost its security guarantees. Finally, we describe the C-5 implementation.

5.5.1 Security Enhancements

C-5 enhances the security guarantees of No-FAT by (1) supplementing it with data encryption to ensure application data confidentiality and (2) implementing a temporal memory safety extension to increase the per-object temporal entropy against UAF attacks from 16 bits to 64 bits. Here, we describe the two enhancements.

Applying Memory Encryption. C-5 uses the QARMA block cipher [126], which was introduced for ARM's Pointer Authentication technology (ARM's PAC) [10]. Unlike ARM's PAC, which uses QARMA to authenticate pointers and store the authentication code in the pointer upper bits, C-5 uses QARMA for encrypting cache lines as they transfer from the L1 data cache to L2 cache and beyond. As shown in Figure 5.7, keeping the application data encrypted beyond the L1 data cache protects the data from physical attackers, while avoiding the high cost of encryption/decryption at the core to L1 data cache interface.

While the data is unencrypted in the L1 data cache (to allow for fast access), the L1 data is still protected by the memory safety checks. This hybrid protection ensures strong security guarantees at low performance cost for C-5. On the other hand, C^3 cannot keep data unencrypted in the L1 data cache as C^3 relies on encryption for providing memory safety. Additionally, moving the encryption to the L1-L2 interface allows C-5 to use strong encryption schemes (as opposed to the vulnerable XOR-based encryption used in C^3) without affecting the overall system performance,

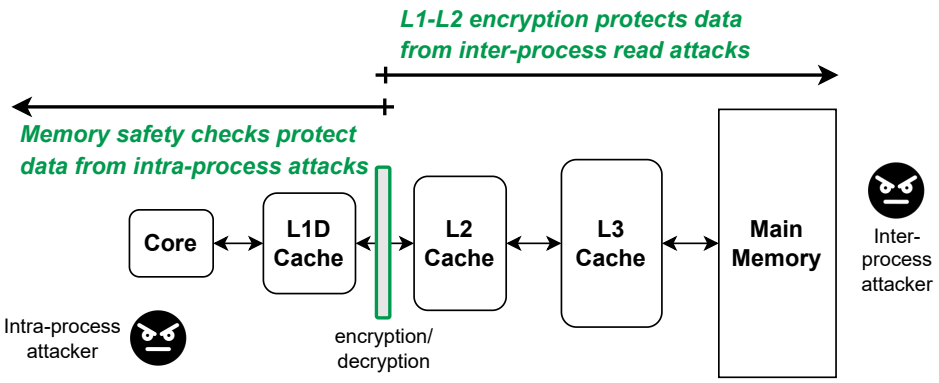


Figure 5.7: A high level overview of how C-5 works. We use memory safety checks to protect the application data from intra-process attackers while being stored in the L1 data cache. We use a per-cache line encryption to protect data confidentiality from side-channel leakage attacks while being stored in L2, L3, and main memory.

because the encryption latency will be masked by the L2 access latency.

Boosting Temporal Memory Safety. C-5 uses an explicit lock and key mechanism for enhancing the temporal memory safety guarantees of No-FAT. Each pointer holds a key, whereas each object has a lock. If an object is deleted, its lock will be changed to a new value, implicitly nullifying all dangling pointers that used to point to this object. Unlike prior work (e.g., CETS [127]) that maintains the lock and keys in disjoint metadata tables in program memory, C-5 embeds the lock and keys within the allocation itself, as shown in Figure 5.8. Thus, C-5 achieves better utilization of the application memory as the allocation padding bytes are typically unused under the binning allocator model—if the allocation size is less than the bin size.

Figure 5.8 shows how the lock and keys are stored using an example of a parent struct with two children. When a new allocation is created with `malloc` or `new`, additional space is added to the end of the allocation to store the following metadata:

- **META:** A 16-bit field that stores the size of the per-allocation metadata at the end of the object.
- **LOCK:** A 64-bit value that is randomly generated upon object allocation, is stored towards the end of the object, and is propagated alongside the pointer in an extended 128-bit register in hardware.

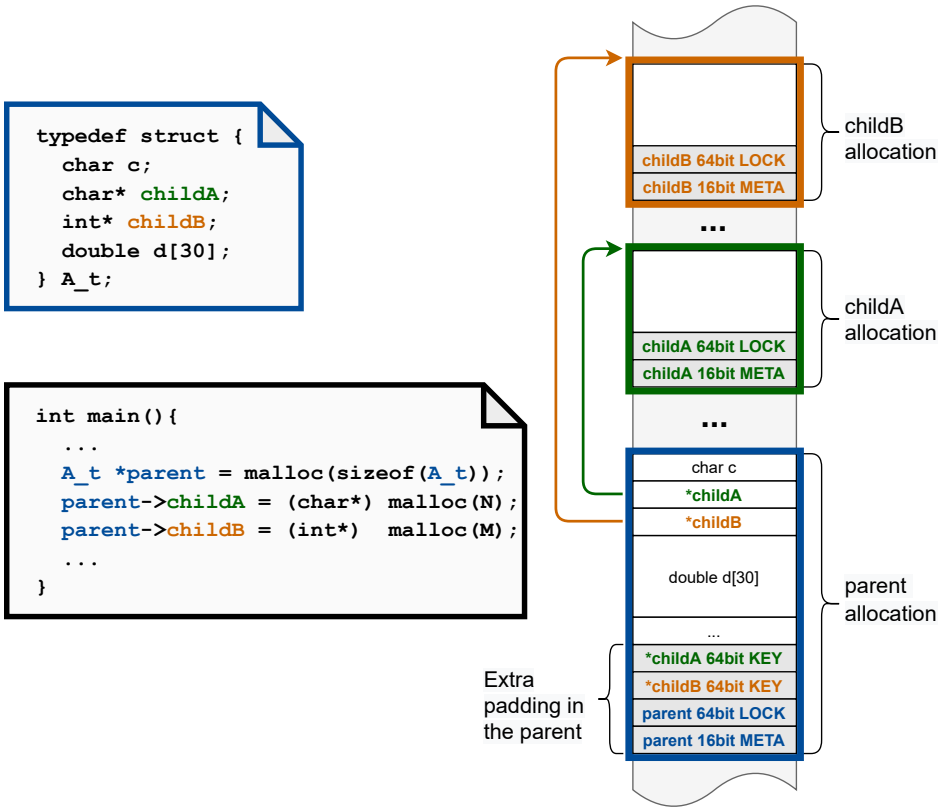


Figure 5.8: C-5's temporal memory safety extension. Each allocation is expanded by the following padding bytes, which are only accessible by the hardware. `META` stores the number of padding bytes per object. `LOCK` stores a random 64-bit temporal tag that is propagated in extended 128-bit registers in hardware. When a pointer is spilled to memory (e.g., storing `*childA` in `parent`), the upper 64 bits of the extended register are stored to the corresponding `KEY` field. Each allocation can have one `LOCK` and as many `KEY`s as the number of its internal pointers.

- **KEY:** When a pointer is spilled to memory (e.g., storing `*childA` within `parent` in Figure 5.8), the upper 64 bits of the extended register are stored to the corresponding `KEY` field. Each allocation can have as many `KEY` fields as the total number of its internal pointers.

A natural question to ask is: how can C-5 efficiently retrieve the per-allocation metadata and ensure its integrity? First, as C-5 leverages binning memory allocators, the C-5 hardware can quickly find the `META` location (i.e., `Memory[base address + size - 2]`). Second, given the `META` value, the hardware can ensure that no intra-allocation overflow can corrupt the other metadata fields stored in the padding bytes, making the per-allocation metadata architecturally inaccessible. This feature not only allows C-5 to protect its metadata, but also permits precise spatial

memory safety checks compared to the original No-FAT work. In No-FAT, off-by-one overflows might go undetected, as they will be part of the binning allocator padding.

When a pointer is loaded from memory (e.g., loading `parent->childA`), C-5 loads the pointer temporal tag from the corresponding `KEY` field in the parent allocation to the upper 64 bits of the 128-bit hardware register. Then, our modified hardware computes the allocation base address (i.e., `childA`'s base), fetches the pointer `LOCK` from `Memory[childA's base + size - 10]` with its `META` from `Memory[childA's base + size - 2]`, and passes both to the trusted base register, $\texttt{ptr}_{\texttt{trusted}}$, in the new load and store instruction. When the pointer is used as part of a memory access instruction, its spatial memory safety is checked using: `ptr - base <= size - META`. The temporal safety is checked by comparing the tag of $\texttt{ptr}_{\texttt{trusted}}$, which is originally obtained from the `LOCK` field, against the pointer's tag, which is obtained from the `KEY` field. All information is available to hardware at the time of memory access as `base`, `META` and `LOCK` are all part of the 128-bit trusted base register, whereas the `KEY` is part of the 128-bit extended hardware register that holds the 64-bit regular pointer as well.

5.5.2 Performance Optimizations

The original No-FAT work reports that No-FAT incurs an average of 8% performance overheads on the SPEC CPU2017 benchmark suite [7]. C-5 implements two optimizations to eliminate this performance cost by using a generic binning memory allocator and a small in-hardware base address cache.

Relaxing the Memory Allocator Constraints. The original No-FAT work uses a simple binning allocator, dubbed Binning-Malloc [41], which divides the virtual memory into 64 regions, each of size 32GB. Each region is used to satisfy heap-allocation requests of a unique size whereas stack and global memory allocations are satisfied using special carved out sections of the same 32GB regions. While Binning-Malloc simplifies the hardware design, as the bin sizes are all stored in a single 64-entry MAST table, it might cause memory fragmentation due to the limited number of available allocation sizes and the large size of each allocation bin (or arena). On average,

half of the No-FAT overheads (i.e., 4%) are caused by Binning-Malloc [7]. C-5 addresses the aforementioned concerns by showing how other binning memory allocators can be adopted for hardware-assisted memory safety.

The idea is simple. Instead of enforcing the memory allocator's use of a fixed number of bins, the memory allocator can arbitrarily define the number of bins it uses and the allocation size that is used within each bin. The only requirement is that the bin size is no less than the size of a page (i.e., 4KB or 2MB for systems with huge pages support). C-5 maintains 16B of metadata per memory page to store the allocation size (and inverse) of this page. We use this metadata to compute the allocation base address within a page using the steps from Figure 4.1, where `S` equals page size. If a memory page is used to satisfy an allocation that is larger than page size, we simply store the allocation base address in the per-page metadata so that it is easily retrieved when a pointer to this large allocation is loaded from memory. Finally, this new metadata only consumes less than 0.4% storage overheads and can be maintained as an extension to leaf page table entries, offering great flexibility to the binning memory allocator.

Reducing Base Address Computation Overheads. We observe that the second main source of performance overheads in the original No-FAT work (4% on average for SPEC CPU 2017 benchmarks) is the base address computation when a pointer is loaded from memory [7]. As shown in Figure 4.1, the last step of the Base Computing Module involves two successive multiplications. C-5 reduces this performance cost by introducing a simple base address cache that is used to store a derived (i.e., non-base) pointer against its correct base address. As programs tend to store the same derived pointer in memory after operating on it, this pointer will result in a base address cache hit when it is loaded from memory. A high hit rate can reduce the dominant cost of computing a base address from two multiplications (6-8 cycles) to a single cache look-up (1-2 cycles).

Our base address cache is simple in design. It requires no communication with other memory levels since missed accesses can simply be recomputed using the base computing module. Thus, it can be simply flushed during content switching to guarantee non-interference between different processes.

5.5.3 C-5 Implementation

Figure 5.9 provides an overview of our C-5 framework. Next, we discuss the different components.

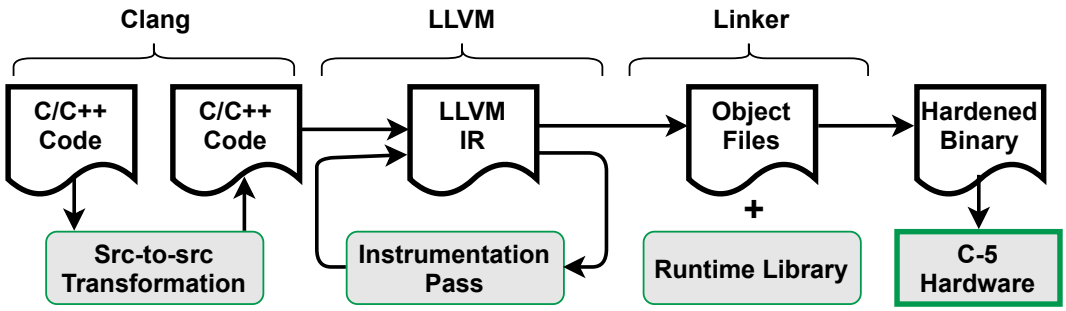


Figure 5.9: C-5's implementation overview.

ISA. C-5 uses the same instruction set architecture (ISA) extensions as in No-FAT [7]. The only difference is the `define_size` instruction, which is used to let the memory allocator control the allocation size per each virtual memory page. The complete ISA extensions are summarized in Table 5.1.

Table 5.1: C-5's ISA extensions. `rd`/`rs` denote `dst`/`src` registers that hold data; `rb` holds the trusted base pointers.

Instruction	Format	Operation
Secure Load	`secure_load rd, rs, rb`	`rd = Mem[EffAddr]`
Secure Store	`secure_store rd, rs, rb`	`Mem[EffAddr] = rs`
Verify Bounds	`verify_bounds rs, rb`	`rs - rb < size(rs)`
`Compute Base`	`compute_base rb, rs`	`rb = Base(rs)`
Define Size	`define_size r1, r2`	`EntrySize(r1) = r2`

Hardware. C-5 uses the main hardware blocks of No-FAT, namely the Bounds Checking unit, the Base Computing module, and the trusted base address register file. Additionally, C-5 extends all registers to 128 bits in order to store the extended 64-bit temporal tags of pointers, if needed. Finally, an N-entry fully-associative cache is used to accelerate the Base Computing module operations.

Compiler. C-5 uses compiler support to leverage the ISA extensions. As shown in Figure 5.9, C-5

first uses a source-to-source transformation to (1) promote intra-allocation buffers into standalone allocations (also known as Buf2Ptr [7]) and (2) expand allocations with extra space for storing the temporal safety metadata. Second, we implement an instrumentation pass at the LLVM IR level to replace program loads and stores with the `secure_load` and `secure_store` instructions. The compiler pass uses simple intra-procedural analysis to insert `verify_bounds` and `compute_base` instructions in addition to spill/fill the temporal tag to/from its corresponding location in the parent object before a pointer is stored to memory and after the pointer is loaded from memory, respectively. For providing complete security coverage, we also protect stack and global allocations by assigning them to special bins, similarly to prior work [76, 7]. Finally, we link our instrumented object files with a C-5 aware runtime library that intercepts calls to memory management operations such as `malloc` and `free`. To maintain compatibility with uninstrumented third-party libraries, the temporal tags are ignored by the hardware when pointers are passed to uninstrumented code (e.g., the operating system or third-party libraries).

5.6 Security Analysis

In this section, we discuss the security guarantees of C-5 and explain its current limitations.

5.6.1 Addressing Traditional Memory Safety Violations

Buffer Under-/Over-flows. C-5 relies on No-FAT for enforcing strict access control rules in the L1 cache. Thus, C-5 detects spatial memory safety violations by detecting out-of-bounds pointers. This protection is applied to all program segments—heap, stack, and global. Similarly to No-FAT, C-5 achieves intra-object spatial memory safety by deploying the Buf2Ptr transformation.

Use-after-frees. As described in Section 5.5.1, C-5 enhances the temporal memory safety guarantees of No-FAT by using 64-bit temporal tags. These tags are marshaled with program pointers in hardware and are stored in the object padding bytes in memory. If a use-after-free violation occurs, the new memory allocation will be detected with a high probability ($1 - (1/2^{64}) \approx 100\%$) as the

deleted and new allocations will likely have different 64-bit tags.

Control- & Data-Flow Hijacking Attacks. As C-5 provides fine-grained memory safety, it mitigates the different forms of control-flow hijacking attacks (e.g., ROP [53] and its variants [79, 56]) by protecting return addresses and code pointers. Similarly, data-flow manipulations attacks (e.g., DOP [57, 59, 80]) are detected by preventing illegal accesses to data pointers.

Data-Only Attacks. Given a memory safety vulnerability, attackers can corrupt non-pointer data items, such as program flags and configuration files [60]. C-5 mitigates those attacks by ensuring that all loads/stores happen between their legitimate bounds and within the valid object lifetime.

Spectre-V1 Attacks C-5 enforces spatial memory safety for committed and transient instructions because the legitimate allocation bounds re part of the memory instruction itself. Thus, our solutions holds the promise of preventing Spectre-V1 (bounds checking bypass) [43]. If a transient instruction accesses an out-of-bounds memory, we mark the instruction as unsafe and raise the exception only when the violating instruction is committed. The reason why we delay raising the exception is to avoid causing false alarms (i.e., if the out-of-bounds memory access occurs due to a benign branch misprediction). C-5 does not protect against other Spectre variants and holistic defenses are likely to be required to completely mitigate such vulnerabilities.

5.6.2 Mitigating Physical Attacks

Physical attacks can compromise the program data while being stored in main memory. Examples include ColdBoot [95] and RowHammer [77]. While memory encryption technologies (e.g., AMD's Secure Memory Encryption) protects data in DRAM by using a page-level encryption, they are vulnerable to other attacks (e.g., NetCAT [128] and CrossTalk [129]), which leak the unencrypted data from the structures that are shared among different cores (e.g., LLC). To mitigate these attacks, C-5 opts to encrypt data in the L2 cache and beyond (not just in DRAM) while leaving the per-core private data in the L1 data cache under the protection of the memory safety rules.

5.6.3 Mitigating The C-4 Attacks

Unlike C^3, which relies on cryptographic isolation for enforcing spatial memory safety [44], C-5 uses strict access control rules. Thus, C-5 does not permit accessing out-of-bounds memory in Attack #1. Similarly, using a pointer arithmetic operation to substitute one pointer with another in Attack #2 will fail against C-5 when the pointer is used to access memory (as the trusted pointer base, $\texttt{ptr}_{\texttt{trusted}}$, and bounds will not match the manipulated pointer bounds).

While C^3 provides 4 bits of temporal entropy, which can be increased to 10 bits if a different "Radix" is used for the recently freed memory locations, C-5 provides 64-bit temporal safety entropy, highly mitigating Attack #3. Finally, C-5 enforces memory safety for all program memory regions: heap, stack, and global. Thus, compromising data items on the stack/global memory in Attack #4 cannot be used to undermine our security guarantees. While this design choice requires program recompilation, we note that recompiling the source code is not an obstacle in modern systems, as evidenced by commercial security techniques [10, 38, 9].

5.6.4 Preventing the end-to-end exploit

The probability of successfully creating a type confusion against C-5 is one in 2^{64} due to the extended temporal tag. Unlike C^3, which permits the attacker to keep trying multiple times and checking for abnormal `free_target.length` values, C-5 generates an exception as soon as a tag mismatch occurs between the dangling pointer tag (i.e., the `free_target`'s butterfly tag) and the $\texttt{ptr}_{\texttt{trusted}}$ tag (i.e., the `vuln`'s butterfly tag), which is retrieved from the newly allocated object.

Even if we assumed that the attacker will be lucky enough to correctly guess the 64-bit tag in the first trial (or the attack starts with a non temporal safety violation), the following step (i.e., building an arbitrary read/write primitive) is immediately blocked by C-5 as we generate a hardware exception upon accessing out-of-bounds memory regardless of the software checks that depend on the corrupted `free_target.length` value. This way C-5 prevents the attacker from achieving arbitrary code execution or even building a read/write capability.

5.6.5 Limitations

Type Information Availability. C-5 inherits its intra-allocation spatial memory safety capabilities from No-FAT [7]. Thus, C-5 similarly requires the availability of the per-allocation type information. Some C applications violate this requirement by using `void*` allocations. Those ambiguous cases could be addressed by intra-procedural backward analysis (to find the correct allocation type based on previous casting) or with programmer-guided annotations.

Replay Attacks. Similarly to C^3 [44], C-5's data encryption only provides confidentiality guarantees (against inter-process data leakage attacks) with no data integrity guarantees. Thus, both techniques are vulnerable to physical replay attacks. In replay attacks, an adversary monitors the memory contents to record the contents of objects at time, `t1`, and blindly replies those contents at later time, `t2`, where `t2` is greater than `t1`. Mitigating physical replay attacks requires freshness to guarantee that the memory always contains the latest written application contents. We note that recent commercial implementations such as AMD SEV and Intel TDX do not include mitigation for replay attacks. Such mitigation could be achieved with orthogonal techniques such as Merkle Trees [130, 131, 132, 133].

5.7 Evaluation

In this section, we evaluate the performance cost and security guarantees of our hardware-assisted mechanism, C-5. We first describe our evaluation methodology and experimental setup. Second, we provide the SPEC CPU2017 results for each of our proposed optimizations. Third, we extend our performance analysis to include two real-world applications. Finally, we perform a quantitative security evaluation of C-5.

5.7.1 Evaluation Methodology

Similarly to prior work [47, 6, 7], we use a real machine to emulate the runtime overheads of our proposed solution instead of microarchitectural simulators. We implement an LLVM pass [89]

to instrument the given applications and insert a chain of dependent instructions to emulate the overheads of the new ISA extensions. For example, we use one `load`, two `mul` instructions, and a `store` to emulate the `compute_base` instruction operation. We insert dummy `store` instructions in place of `verify_bounds` instructions.[2] Regular program `loads` and `stores` are not replaced as the bounds checking operation is performed in parallel to the L1 data cache access. Finally, we link the final executable against a runtime library that implements the required binning memory allocator.

Experimental Setup. Since our C-5 architecture has close parallels with No-FAT [7], we opt to use the same evaluation setup. We instrument and compile the evaluated applications using `Clang 4.0.0` and run the binaries on a bare-metal Intel Skylake-based Xeon Gold 6126 processor running at 2.6GHz with RHEL Linux 7.5 (kernel 3.10). To minimize variability, each application is executed 5 times and the average of the execution times is reported after normalization to baseline (i.e., unprotected) execution.

5.7.2 SPEC CPU2017 Performance Evaluation

Here, we use the SPEC CPU2017 benchmarks with `ref` inputs and run all benchmarks to completion. We evaluate the performance of the following configurations:

- **No-FAT.** This configuration represents the regular No-FAT [7]. It uses a simple binning allocator (Binning-Malloc [41]) that divides the virtual memory into 64 regions, each of size 32GB. No-FAT provides inter- and intra-allocation spatial memory safety with 16-bit temporal memory safety guarantees.

- **No-FAT+.** In this configuration, we add the temporal memory safety extension (as described in Section 5.5.1) on top of No-FAT. Our evaluation includes the cost of adding the per-object metadata (e.g., `LOCK` and `KEYs`) and accessing them when pointers are loaded/stored from/to memory. This configuration provides the strongest access-control security guarantees without having any performance optimizations.

[2] We use a `store` to make sure the inserted instructions are not omitted by compiler optimizations.

- **C-5.** This configuration integrates No-FAT+ with mimalloc, a compact general purpose binning memory allocator [42]. We adopt mimalloc to show that C-5 can gain performance benefits by using a modern binning memory allocator, as explained in Section 5.5.2.

- **C-5 with base$.** In this settings, we assume that C-5 is equipped with an ideal base address cache (base$) that can serve the requests of the `compute_base` instructions without incurring the performance overheads of executing two consecutive multiplications for computing the allocation base address.

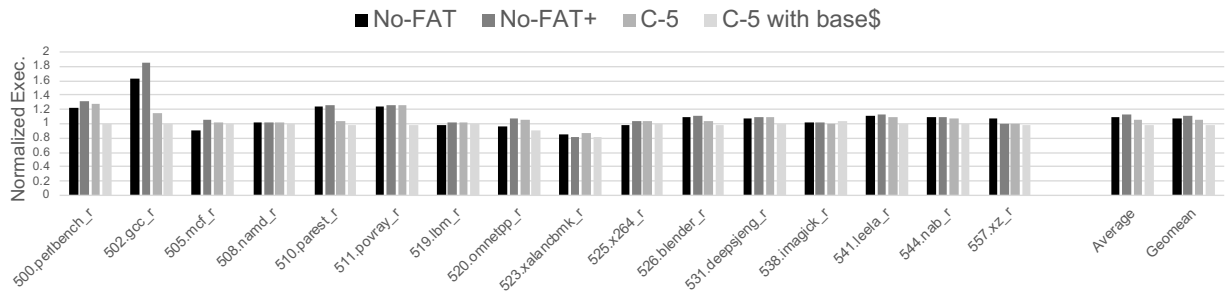


Figure 5.10: Performance overheads of the SPEC CPU2017 benchmarks for different No-FAT and C-5 variants normalized to an unprotected baseline.

Results. Figure 5.10 summarizes the performance overheads of SPEC CPU2017 benchmarks for the different configurations normalized to a baseline execution. The geometric mean of each configuration is as follows: No-FAT (1.08x), No-FAT+ (1.11x), C-5 (1.06x), and C-5 with base $ (0.98x). The results suggest that extending the temporal memory safety entropy to 64 bits introduces low performance overheads (i.e., 3% on top of the original No-FAT work). On the other hand, using mimalloc as the underlying binning memory allocator reduces the overall overheads of our proposed system, as it outperforms the baseline non-binning allocator from glibc. Finally, an ideal base address cache can completely eliminate the performance overheads of C-5. Next, we provide further results to support this claim.

Base Address Cache Analysis. We discussed how our base address cache works in Section 5.5.2. As we cannot emulate the performance gains of a hardware-based cache on a real machine, we opt to analyze the base address cache behavior using a pure software implementation. We write

Table 5.2: C-5's base address cache sensitivity analysis. "# Accesses" represents the total number of `compute_base` instructions. "Hit Rate" is the % of `compute_base` instructions that hit in an N-entry base address cache.

Benchmarks	**# Accesses**	**Hit Rate**		
		N = 32	**N = 64**	**N = 128**
500.perlbench_r	2.44×10^{8}	68.21%	99.49%	99.60%
502.gcc_r	5.46×10^{10}	69.58%	74.10%	78.78%
505.mcf_r	6.05×10^{10}	54.78%	58.09%	61.55%
508.namd_r	1.64×10^{10}	96.13%	96.14%	96.14%
510.parest_r	10.58×10^{10}	89.53%	96.08%	96.82%
511.povray_r	18.15×10^{10}	77.71%	79.35%	83.26%
519.lbm_r	60.56×10^{2}	99.97%	99.97%	99.97%
520.omnetpp_r	9.32×10^{10}	75.74%	86.60%	88.14%
523.xalancbmk_r	7.80×10^{10}	33.61%	36.55%	40.12%
525.x264_r	1.65×10^{10}	71.27%	79.75%	84.48%
526.blender_r	6.24×10^{10}	50.73%	59.21%	80.42%
531.deepsjeng_r	9.12×10^{8}	99.99%	99.99%	99.99%
538.imagick_r	7.21×10^{10}	89.01%	89.01%	89.02%
541.leela_r	1.93×10^{10}	74.19%	75.51%	77.85%
544.nab_r	5.00×10^{10}	95.60%	95.60%	95.60%
557.xz_r	67.17×10^{8}	99.81%	99.81%	99.81%
Average	5.11×10^{10}	77.87%	82.83%	85.72%

an N-entry C++ fully associative cache that uses the least recently used (LRU) replacement policy. At runtime, each entry is used to store an arbitrary input pointer and its corresponding base address. We modify our LLVM compiler pass to access the software cache on each emulated `compute_base` instruction and measure the hit rate.

Table 5.2 shows the results of our analysis on the SPEC CPU2017 benchmarks. The "# Accesses" column shows the total number of executed `compute_base` instructions (i.e., the total number of base address cache accesses). The "Hit Rate" columns show the percentage of `compute_base` instructions that causes a base address cache hit for a number of cache entries, N = {32, 64, 128}. In case of a cache miss, we compute the correct base address and use the LRU policy to replace one entry from the cache with the new entry. The results confirm that a small base address cache (e.g., with 64 entries) is sufficient to eliminate the overheads of 82.83% of the total `compute_base` instructions (i.e., billions of cycles).

Table 5.3: Simulation parameters for C-5 data encryption evaluation.

Parameter	Value
Core	SimpleTimingCPU (In-Order) at 3.4GHz
L1 inst. cache	32KB, 8-way, 1-cycle latency
L1 data cache	32KB, 8-way, 3-cycle latency
L2 cache	256KB, 8-way, 6-cycle latency
L3 cache	8MB, 16-way, 18-cycle latency
DRAM	8GB, DDR4-2400

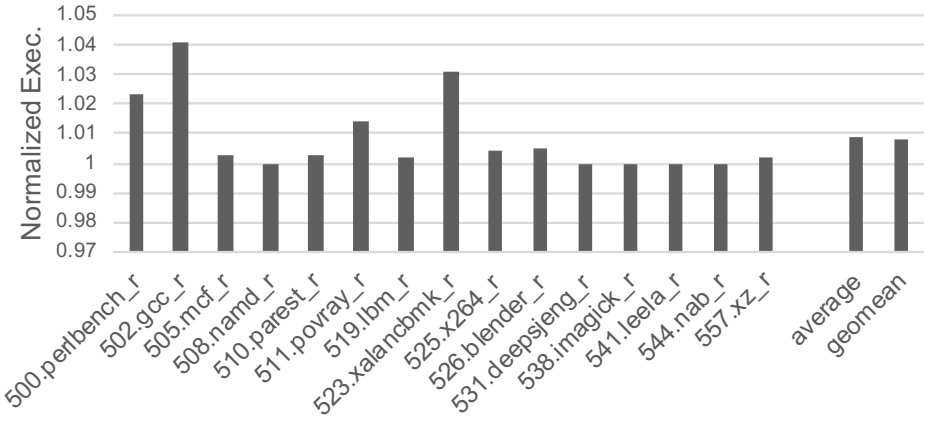


Figure 5.11: Slowdowns with additional 6-cycles access latency for the L2 cache for C-5.

Application Data Encryption. In our performance evaluation results so far, we do not include the cost of encrypting the application data because we cannot emulate this cost on a real machine. While the encryption cost is going to be masked by the L2 cache access latency, we evaluate it separately for completeness. We run the SPEC CPU2017 workloads with `ref` inputs on the Gem5 simulator [134]. To reduce the simulation times without affecting the results accuracy, we opt to run the first 400 Million instructions of each workload.

Table 5.3 shows the simulation parameters for the baseline system in Gem5. We add an additional 6 cycles to the L2 access time to model the overheads of the QARMA encryption/decryption on the L1-L2 interface. As shown in Figure 5.11, C-5's data encryption adds 0.8% overheads on average. This minor performance cost is expected as C-5 keeps the common case fast by leaving the application data in plaintext in L1 under the protection of the strong spatial and temporal

memory safety checks. These memory safety checks introduce no performance penalty as they are overlapped with the regular L1 access path.

5.7.3 Real-world Case Studies

In addition to the standard SPEC benchmarks, we evaluate C-5 (without the data encryption component) on two real-world applications: the Nginx web server [45] and the Duktape Javascript interpreter [135].

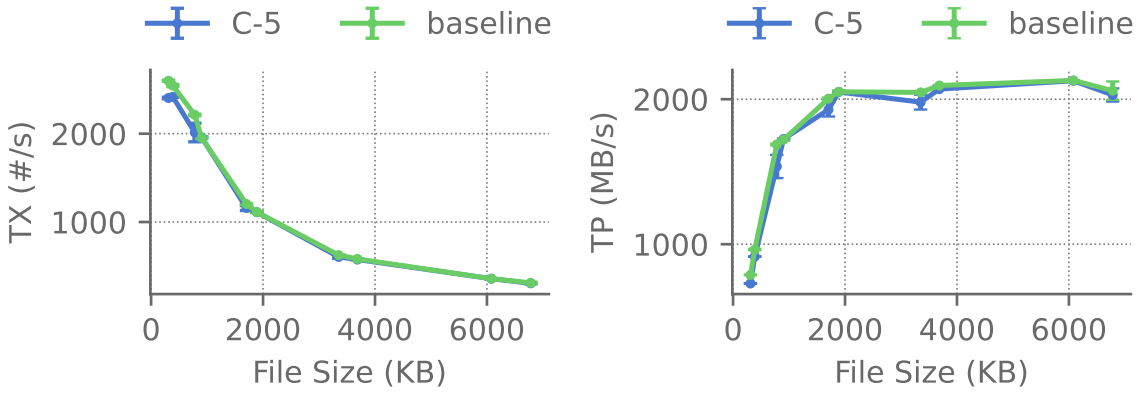


Figure 5.12: Nginx performance evaluation with C-5.

Nginx. In this experiment, we use Nginx [45] (version 1.21.6), In order to simulate typical workload configurations, we used Nginx to serve different-sized files using the page weight (i.e., the amount of data served) of modern websites according to the 2019 HTTP Archive Web Almanac report [136]. To generate the client load, we used the multi-threaded Siege [137] benchmarking tool. We issued 500 requests with 50 concurrent connections for each page weight using the loopback interface to avoid network congestion issues. Figure 5.12 shows the transfer rates (TX) and throughput (TP) of C-5 and baseline execution. C-5 introduces negligible overheads for all the tested file sizes. While we do not evaluate Nginx data encryption in Gem5, it is expected to introduce no further overheads, especially for such I/O bounded workloads.

Duktape. We evaluate the Duktape [135] (version 2.5.0) Javascript interpreter with a default

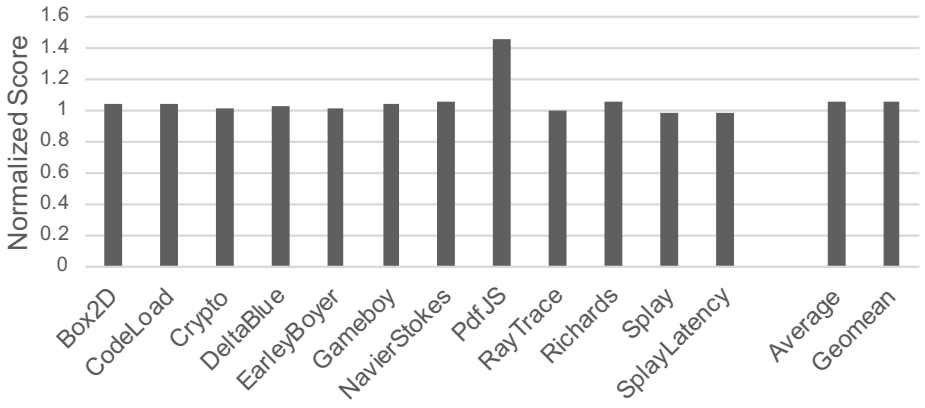


Figure 5.13: Duktape performance evaluation with C-5.

build configuration running the Octane 2 benchmark suite [138][3]. Figure 5.13 summarizes the benchmark scores as reported by the Octane 2 suite after being normalized to the baseline. C-5's spatial and temporal memory safety protections introduce 5% performance slowdowns on average without using the base address cache optimization.

5.7.4 Security Evaluation

C-5 provides deterministic inter- and intra-allocation spatial memory safety and strong probabilistic temporal memory safety (with 64-bit tags). C-5's access-control rules in addition to its memory encryption mitigate all attacks that we proposed against C^3 [44]. In order to quantify the security benefits of C-5, we implement a software-only version of C-5 that explicitly checks pointer bounds and temporal tags in software with no hardware support. We use this version solely for performing the security analysis and not performance evaluation. We compare the security guarantees of this software version against two state-of-the-art techniques: AddressSanitizer (ASan) [3] and Intel's MPX [38], as representatives of pre- and post-deployment memory safety solutions, respectively.[4]

RIPE. First, we use RIPE [139], an open source intrusion prevention benchmark suite. We ported RIPE to 64-bit systems and compiled it with our software-only version of C-5. The total number of

[3]Duktape does not currently support all benchmarks.

[4]We use `Clang 7.0` for building ASan and `GCC 7.3.1` for building Intel's MPX.

attacks that survive with a baseline (i.e., unprotected) GCC and Clang is 50 attacks. Table 5.4 summarizes the results for the three tools. ASan failed to prevent two attacks that use intra-allocation overflows. C-5 and Intel's MPX stopped all the attacks, including the intra-allocation attacks, due to their Buf2Ptr and bounds-narrowing features, respectively. However, C-5 avoids the prohibitive runtime costs of Intel's MPX.

Table 5.4: RIPE Results Summary

	ASan	**Intel's MPX**	**C-5**
Working Attacks	2/50	0/50	0/50

Microbenchmarks. While RIPE provides various tests that hijack the application control-flow given a memory safety vulnerability, it lacks important attack categories, such as type-confusion and information leakage. Thus, we implement a small set of security microbenchmarks to examine those missing attacks. Table 5.5 summarizes our results for the same three tools. C-5 provides complete coverage over the spectrum of these vulnerabilities.

Table 5.5: Security Microbenchmarks Summary.

	ASan	**Intel's MPX**	**C-5**
Intra-Overflow	✗	✓	✓
Inter-Overflow	✓	✓	✓
Use-after-free	✓	✗	✓
Type Confusion	✓	✓	✓
Buffer Over-read	✓	✓	✓

5.8 Summary

The arms race between memory safety attackers and defenders is not going to end any time soon given the sheer volume of C and C++ code in current systems. In this chapter, we made another move in the continuing arms race game by analyzing the security guarantees of a recent hardware-based security architecture, C^3. We show that C^3's claims of achieving strong spatial

and temporal memory safety while introducing negligible overheads can be broken with carefully designed attack vectors. We described four different attacks to counter C^3, called C-4, analyzed their root causes, used them to mount an end-to-end attack against the WebKit's JavaScript engine, and discussed potential solutions to fix them. As applying the potential fixes will completely redesign C^3, we presented an alternate solution that can achieve the same security goals but using simpler techniques. We built our solution, C-5, to counter C-4 by integrating strong data encryption with access control rules. C-5 comes with a set of security enchantments and performance optimizations that makes it a viable hardware-assisted solution for modern systems. Our quantitative performance and security evaluation results confirm that C-5 can provide strong security guarantees at no runtime cost, making it resilient against a wide variety of memory corruption attacks, including the ones we proposed in this chapter.

Part IV

Efficient Exploit Mitigations For Servers & Embedded Systems

Chapter 6: Zero-Overhead Resilient Operation Under Pointer Integrity Attacks

Most end users want security but do not want the inconvenience of having it: they do not want their batteries drained, or apps slowed, or to be bothered with updates and crashes. This is the unfortunate reality that sends novel security techniques with even minor performance overheads to the crypt of great security ideas. Techniques that have been mass deployed in hardware (e.g., W^X and SMEP/SMAP) are the ones that have close to zero overheads. Even techniques like ARM's PAC—which does have significant overhead when applied fully—is applied partially to only protect code pointers, and only to the kernel to keep the overheads small. Thus, low performance overhead and convenience are key to widespread adoption of security techniques.

In this chapter I present ZeRØ, a hardware primitive that preserves pointer integrity at no additional performance cost. In traditional processors, memory instructions can freely access any memory location. There are no restrictions on the type of operands used by a memory instruction. This behavior is fundamental for attackers to craft their exploits. As a result, ZeRØ introduces unique sets of memory instructions for the different categories of pointers that make up a program (i.e., code and data). Having specific memory instructions for code pointers, data pointers, and regular data allows ZeRØ to enforce access control rules that maintain pointer integrity when under attack. Unlike prior work, which tags every word in memory to identify different program assets (e.g., code and data pointers) [140, 141], ZeRØ uses a novel metadata encoding scheme that allows it to precisely store all the required metadata to identify different program assets with just a single bit per every cache line in L2 and main memory (less than 0.2% memory overheads).

The remainder of this chapter is organized as follows. Section 6.1 motivates ZeRØ. Section 6.2 provides an overview of how ZeRØ works. Section 6.3 enumerates the ZeRØ instruction set ex-

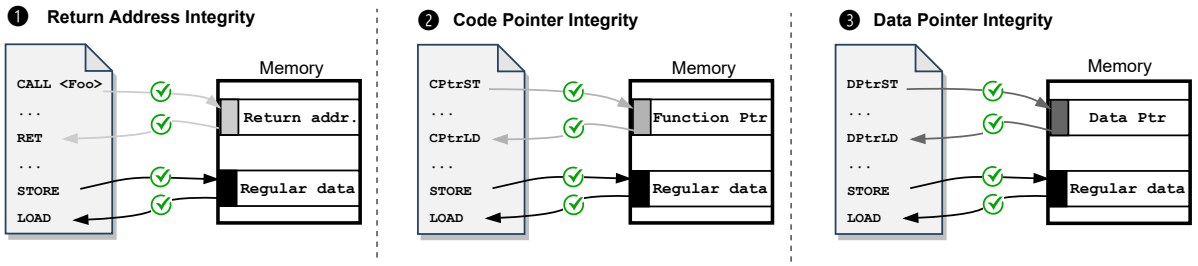
tensions. Section 6.4 details the microarchitectural design of ZeRØ whereas Section 6.5 specifies its software design. Section 6.6 analyzes the security guarantees provided by ZeRØ and its current limitations. Section 6.7 evaluates the hardware and software costs of ZeRØ. Section 6.8 summarizes the chapter.

6.1 Motivation

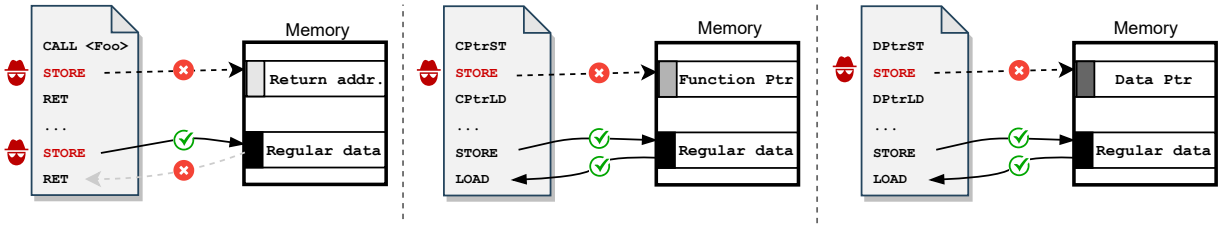
Pointers give programmers the raw ability to work with particular memory locations. The power and flexibility of pointers makes programs written in C and C++ very efficient as long as programmers are careful with their usage. Unfortunately, errors in pointer usage (e.g., out-of-bounds access) can lead to memory corruption vulnerabilities [1]. These memory corruption vulnerabilities have provided attackers with significant opportunities for exploitation. For example, attackers abuse memory safety vulnerabilities to overwrite code pointers and hijack the control flow of the program [53, 54, 55, 56]. Similarly, attackers target data pointers to build up sequences of operations (also known as data-oriented gadgets) without modifying the program's control flow [57]. The prevalence of pointer manipulation attacks against modern software has prompted processor manufacturers to implement hardware mitigation primitives, such as Intel's CET [9] and ARM's PAC [10]. For example, PAC uses cryptographic message authentication codes (MACs) to protect the integrity of pointers—namely return addresses, code pointers, and data pointers. Unfortunately, PAC's usage of cryptographic primitives presents a non-zero performance and energy penalty. In addition, PAC remains vulnerable to speculative execution attacks where arbitrary pointers can be speculatively authenticated [142].

6.2 System Overview

In this section, we describe how ZeRØ enforces pointer integrity for different program assets: return addresses, code pointers, and data pointers.



(a) ZeRØ enforces access control rules that maintain pointer integrity for return addresses, function pointers, and data pointers.



(b) ZeRØ mitigates code-reuse attacks through its use of access control preventing regular `STORE`s from corrupting pointers.

Figure 6.1: A high level overview of how ZeRØ's pointer integrity mechanism works.

6.2.1 How Does ZeRØ Work?

To better understand ZeRØ's security guarantees, let us consider the example in Figure 6.1. Under normal program execution, ZeRØ enforces three different classes of data integrity as shown in Figure 6.1a, namely return address integrity, code pointer integrity, and data pointer integrity. Return address integrity aims at preventing the attackers from overwriting return addresses on the stack (i.e., return oriented programming, or ROP [53, 54]). Return address integrity is provided by extending the functionality of regular `CALL` and `RET` instructions to mark return addresses in memory and prevent other memory instructions from accessing them. As shown in Figure 6.1b ❶, an attacker can attempt to overwrite (i.e., `STORE`) the return address and hijack the control flow of the program. As return addresses are marked such that they can only be accessed by `CALL/RET` pairs, ZeRØ prevents an attacker from hijacking control-flow.

ZeRØ provides code and data pointer integrity by introducing new pairs of memory instructions that are only allowed to access code and data pointers, respectively. For example, we use `CPtrST/CPtrLD` instructions for exclusively accessing code pointers as shown in Figure 6.1a ❷.

If an attacker attempts to overwrite a code pointer using a regular memory instruction, such as `STORE`, as shown in Figure 6.1b ❷, ZeRØ prevents the memory access from occurring. ZeRØ maintains data pointer integrity in the same way as code pointers via specific `DPtrST/DPtrLD` instruction variants as shown in Figure 6.1a ❸.

6.2.2 Main Components

Return Address Integrity. In order to prevent return-oriented programming attacks, ZeRØ protects return addresses on the stack by extending the functionality of regular `CALL` and `RET` instructions to mark return addresses in memory and prevent program loads and stores from accessing them. When a `CALL` instruction is executed, the return address is pushed to the stack alongside the function arguments. ZeRØ sets 2 bits of metadata in the L1 data cache to 01 to mark the 8B return address as *protected*. When a `RET` instruction is executed, the return address is moved from the stack to the program counter if and only if it has the metadata bits set to 01. Once the metadata bits are verified, program execution moves to the new address and ZeRØ sets the metadata bits in the L1 data cache to 00 to mark the memory location as a regular location (i.e., *non-protected*). If any other `LOAD` or `STORE` instruction tries to access a memory location while its metadata bits are set to 01, the hardware generates an advisory exception, effectively preventing return addresses from being leaked or overwritten. The advisory exception is used to notify the system administrator of the access violation without crashing the running process.

Function Pointer Integrity. ZeRØ uses metadata in the L1 data cache to mark function pointers. In order to accurately identify memory instructions that are supposed to access function pointers, ZeRØ uses compiler support and proposes two special instructions, Code Pointer Load (`CPtrLD`) and Code Pointer Store (`CPtrST`), to access function pointers. `CPtrST` marks the function pointer location as *protected* on the first use and assigns a unique state, 10, to it to distinguish function pointers from return addresses. Only `CPtrLD` instructions are allowed to load function pointers from those protected locations. ZeRØ generates an advisory exception if any

regular memory instruction is used to access a memory location that has its metadata bits set to 10.

Data Pointer Integrity. Data pointers work analogously to function pointers. Similarly, ZeRØ proposes two special instructions, Data Pointer Load (`DPtrLD`) and Data Pointer Store (`DPtrST`), to access data pointers. The functionality of these two instructions mirrors the usage of the code pointer variant as described above. While stored in memory, data pointers are assigned a unique L1 metadata state, 11, to avoid confusing them with other protected items (i.e., return addresses and function pointers). We elaborate more on the layout of our ZeRØ metadata and how it is propagated to main memory in Section 6.4.

Pointer-Flow Integrity. In addition to distinguishing between different program assets (i.e., code pointers, data pointers, and regular data), ZeRØ achieves finer protection granularity by distinguishing between elements of the same program asset. To do so, ZeRØ encodes the pointer type in the spare bits (10 bits in our current prototype) of the pointer while executing `DPtrST`. We then verify that the pointer type matches the expected type at a `DPtrLD` location. The pointer type is assigned at compile time and does not require points-to analysis. Two pointers are compatible if their type is the same. As the types are encoded at `DPtrST`/`DPtrLD` sites, an attacker cannot use a vulnerable `DPtrST` instruction to corrupt data pointers of incompatible types, thus reducing the attack surface. The same approach is also applied to code pointers to prevent the attackers from confusing incompatible function pointers. In this case, the function type is used as a unique type per `CPtrST`/`CPtrLD` site.

The next three sections describe the required instruction set extensions, hardware changes, and compiler support, which are needed for ZeRØ.

6.3 Architecture Support

One key aspect in ZeRØ's design is the ability to isolate code and data pointers in memory such that they are not corrupted by attacker-controlled memory instructions. Thus, ZeRØ extends the instruction set architecture to operate exclusively with code and data pointers.

• **`CPtrST/CPtrLD <R1>, <R2>`**: These instructions stand for Code Pointer Store and Code

Pointer Load, respectively. Similar to traditional stores and loads, `CPtrST`/`CPtrLD` use two register operands. The values that are stored in registers `R1` and `R2` point to the store/load address and source/destination register as usual. These instructions are emitted by the compiler only to store/load code pointers. The compiler encodes the code pointer type in the upper bits of `R2`. Upon executing this instruction, the hardware sets/checks the corresponding metadata bits in the L1 data cache and matches the pointer type against the one stored in the upper bits of the memory location (i.e., the store/load address).

• **`DPtrST/DPtrLD <R1>, <R2>`**: These instructions stand for Data Pointer Store and Data Pointer Load, respectively. Similar to `CPtrST`/`CPtrLD`, they are emitted by the compiler to store/load data pointers. Upon executing this instruction, the hardware sets/checks the corresponding metadata bits in the L1 data cache and verifies the pointer type, as described above.

• **`ClearMeta <R1>, <R2>`**: Code and data pointers corresponding metadata bits should be cleared when memory is freed. To support this functionality, we add a Clear Pointer Metadata (`ClearMeta`) instruction that takes two register operands. The value in register `R1` points to the starting address of a 64B cache line. The value in register `R2` is a mask to the corresponding 64B cache line, where 1 allows and 0 disallows changing the state of the corresponding byte. The mask is used to perform partial updates of metadata within a cache line. This instruction is treated similarly to a `STORE` instruction in the processor pipeline since it modifies the architectural state of data bytes in a cache line. Upon executing a `ClearMeta` instruction, the metadata of the target cache line in the L1 data cache is cleared.

Additionally, ZeRØ extends the implementation of regular `CALL` and `RET` instructions to set and check the validity of return address metadata state in the L1 data cache. This functionality is necessary to guarantee the integrity of return addresses. Unlike the other cases discussed prior, this feature does not require a special instruction or additional compiler support. There is no need to explicitly clear the return address metadata as they are cleared upon executing the `RET` instruction. Table 6.1 summarizes the actions taken on various instructions based on the memory location state.

Table 6.1: Actions taken on various instructions based on the memory location state with ZeRØ. 00 represents regular data, 01 represents return address, 10 represents code pointer (i.e., specifically function pointers), and 11 represents a data pointer. X represents "Don't Care".

Instruction	Metadata State	Action
CALL	00	Set the metadata to 01.
	01	Invalid. Cannot overwrite a return address.
RET	00	Invalid. Cannot return from a non-taken address.
	01	Set the metadata to 00.
	10	Invalid. Cannot return from a function pointer.
	11	Invalid. Cannot return from a data pointer.
CPtrST	00	Set the metadata to 10.
	01	Invalid. Cannot overwrite a return address.
	11	Invalid. Cannot overwrite a data pointer.
CPtrLD	10	Load code pointer.
	00	Invalid. Cannot load a non-code pointer.
	X1	Invalid. Cannot load a non-code pointer.
DPtrST	00	Set the metadata to 11.
	01	Invalid. Cannot overwrite a return address.
	10	Invalid. Cannot overwrite a code pointer.
DPtrLD	11	Load data pointer.
	10	Invalid. Cannot load a non-data pointer.
	0X	Invalid. Cannot load a non-data pointer.
LOAD/STORE	00	Load/Store a non-pointer data item.
	01	Invalid. Cannot access a return address.
	10	Invalid. Cannot access a code pointer.
	11	Invalid. Cannot access a data pointer.
ClearMeta	01	Invalid. Cannot free stack memory.
	11	Set the metadata to 00.
	10	Set the metadata to 00.

6.4 Microarchitecture Design

This section describes the microarchitectural changes that are required for implementing ZeRØ.

6.4.1 L1 Data Cache Modifications

Figure 6.2 shows the L1 data cache metadata encoding. ZeRØ uses a 16-bit vector to identify the locations of return addresses, function pointers, and data pointers in a cache line. For example, each two bits of the metadata bit vector represent the state corresponding to each aligned 8B of the cache line. An 8B chunk can either be a return address (01), function pointer (10), data pointer (11),

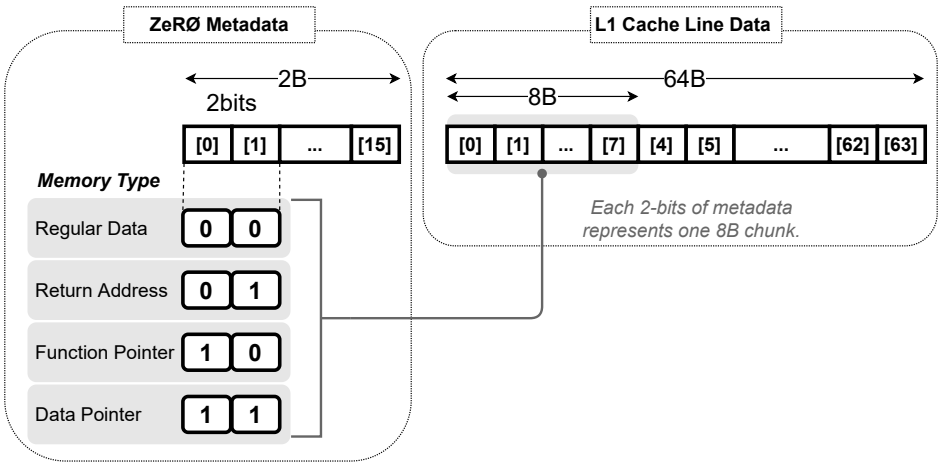


Figure 6.2: ZeRØ's metadata encoding in the L1 data cache. ZeRØ uses a 16-bit vector to indicate whether a chunk of eight bytes is a return address, function pointer, data pointer, or regular (non-pointer) data.

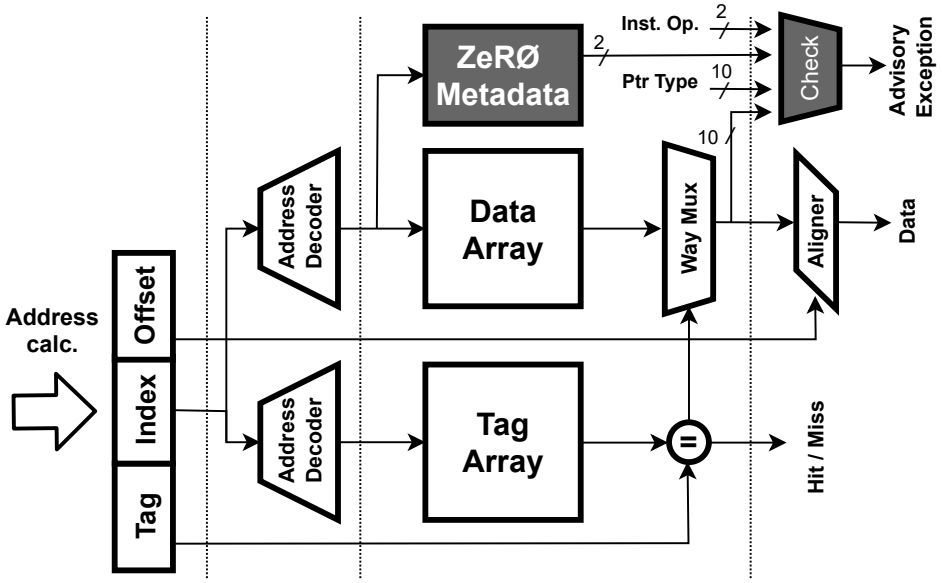


Figure 6.3: Pipeline diagram for the L1 cache hit operation. The shaded blocks correspond to ZeRØ components.

or regular data (00). Our bit vector introduces a 2B storage overhead per 64B cache line (a 3.125% storage for the L1 data cache). As shown in Figure 6.3, if a load/store accesses a protected byte (which is determined by reading the corresponding bit vector), an advisory exception is recorded to be processed when the load/store is ready to be committed.

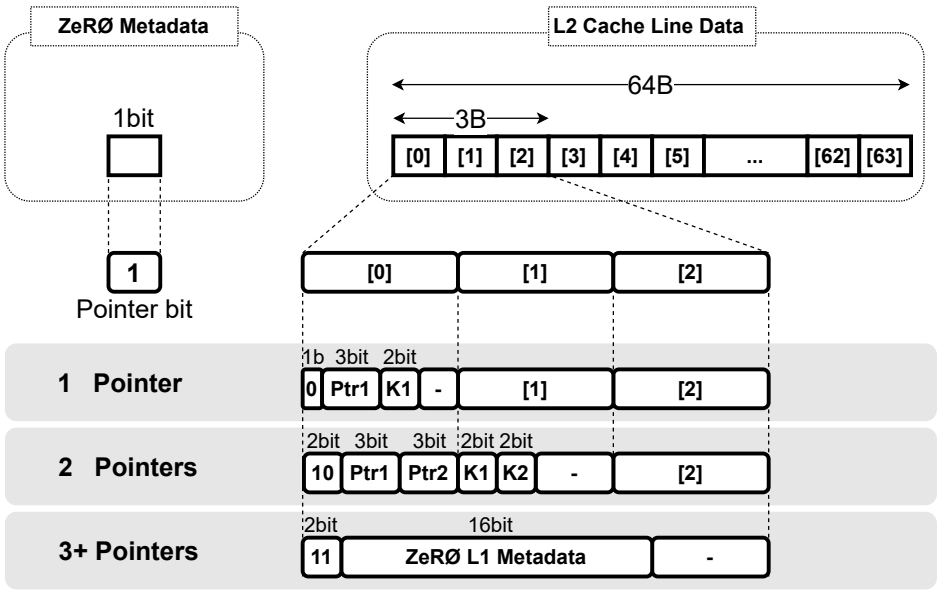


Figure 6.4: ZeRØ's metadata encoding in L2/L3 data cache and main memory. Ptr1/Ptr2 encodes the offset of the pointer in the cache line, whereas K1/K2 encodes its type (return address, code pointer, or data pointer). ZeRØ uses a single bit of metadata to identify protected cache lines.

6.4.2 Exception Handling Circuitry

For certain program functions or libraries, it might be desirable to suppress exceptions (e.g., when the program intentionally accesses pointers with regular `LOAD`/`STORE` instructions). ZeRØ provides hardware support for suppressing the advisory exceptions by using a permit-list. When a binary is loaded the OS writes the starting address and size of the permitted functions/libraries to the permit-list in the exception handling circuitry. Then, when a ZeRØ exception occurs, the hardware checks if the PC of the current memory access instruction is covered by the permit-list or not. If the PC is permitted, the advisory exception is suppressed. Otherwise, the advisory exception is raised and the faulting PC and memory address are passed to the OS exception handler so that they can be used for reporting or investigation purposes. In our design, we use an 8-entry permit-list, where the entry size is 12 bytes (eight bytes for the function starting address and four bytes for the size).

6.4.3 L2/L3 Cache Modifications

Figure 6.4 shows a schematic view of our L2/L3 data cache and main memory metadata encoding. ZeRØ uses a compressed format that requires 1 bit of metadata per 64B cache line (0.2% storage overheads) in L2/L3 caches and main memory. The key idea is that if a cache line has a protected memory location (i.e., return address, code pointer, or data pointer), it will have at least two unused bytes (i.e., the upper 16-bits of the pointer). We use 6 bits from the pointer's upper bits to encode its metadata. For example, if one pointer appears in the cache line, we use 3 bits to store its offset within the line and 2 bits to define its type (return address, code pointer or data pointer). We set the first bit to zero to easily identify this case. The original contents of the first 6 bits of the cache line are moved to the upper 6 bits of the pointer. Finding the location of this pointer requires scanning the bit vector for the occurrence of its state (e.g., 11 for data pointers). This operation is implemented with a priority encoder. We repeat the same approach if two assets of any type exist in a particular cache line.

A natural question to ask is: How do we handle the case in which multiple types of pointers exist in a cache line? For example, a cache line might have three or more code/data pointers. In this case, we have more unused upper pointer bits than needed. Thus, we use 2 bits for recognizing the case and 16 bits for storing the traditional ZeRØ L1 metadata for the entire line. To distinguish formatted lines from regular ones, we use our single metadata bit (i.e., the `Pointer` bit) per cache line as an indicator. If the `Pointer` bit is set to one, that means we have pointers in the cache line. Otherwise, the cache line is normal (i.e., requires no processing).[1]

For DRAM, we store the additional bit per cache line into spare ECC bits, similarly to prior work [4, 34, 6]. We note that the DDR5 standard DIMMs use 80-bit channels, which provides ample space for additional metadata. For non-ECC DRAMs, ZeRØ's eight bytes per 4KB page can be stored in a disjoint location in memory at no additional cost.

[1] We note that ZeRØ's metadata bits can be completely hidden on systems that use caches with error-correction-codes (ECC) support. The techniques from Gumpertz [143] can be used to store ZeRØ's metadata bits for free without compromising the typical ECC functionality. We leave this extension to future work.

```
1: Read the bit vectors of the evicted line and OR them
2: if result is 0 then
3:    Evict the line as is and set its Pointer bit to zero
4: else
5:    Set the Pointer bit to one
6:    Get the location of the first 3 protected addresses
7:    Store the first 6, 12, or 18 bits in the locations obtained in 6
8:    Fill the first 6, 12, or 18 bits based on Figure 5
9: end
```

Algorithm 3: ZeRØ's L1-to-L2 cache line transformation.

6.4.4 L1 to/from L2 Transformation Module

ZeRØ uses two different formats for the L1 data cache and the L2/L3 data caches. As a result, a transformation module is needed to switch between the two formats while cache lines are moving between L1 and L2 in both directions. While the L1-to-L2 transformation module is not on the critical path (only invoked when cache lines are evicted from L1 to L2), the L2-to-L1 transformation module is on the critical path of the processor load operation. Thus, the transformation needs to be carefully designed in order to avoid adding latency to the L2 data cache access.

Algorithm 3 shows the high-level process of the L1-to-L2 transformation module. Figure 6.5 shows the block diagram of the same module. The process starts by ORing all bits from the input L1 bit vector to detect whether a protected address (return address, code pointer, or data pointer) exists in the cache line or not. If the result equals zero (i.e., no protected addresses are detected), we simply set the `Pointer` bit (L2 ZeRØ metadata in Figure 6.5) to zero. If any protected address is detected, we fill in the cache line header according to Figure 6.4. Priority encoders are used to find the index of the first three protected addresses, if they exist. We use the aforementioned locations to store the original contents of the first 6, 12, or 18 bits of the cache line (Line 7 in Algorithm 3) using a cross bar and combinational logic.

Algorithm 4 and Figure 6.6 show the high-level process and block diagram of our L2-to-L1 transformation module. If the cache line has its `Pointer` bit set to one, we check the least significant 2 bits of the first byte to identify the encoding case and reconstruct the original contents of the first 6, 12, or 12 bits of the cache line accordingly. We evaluate the latency and area overheads of our transformation modules in Section 6.7.1.

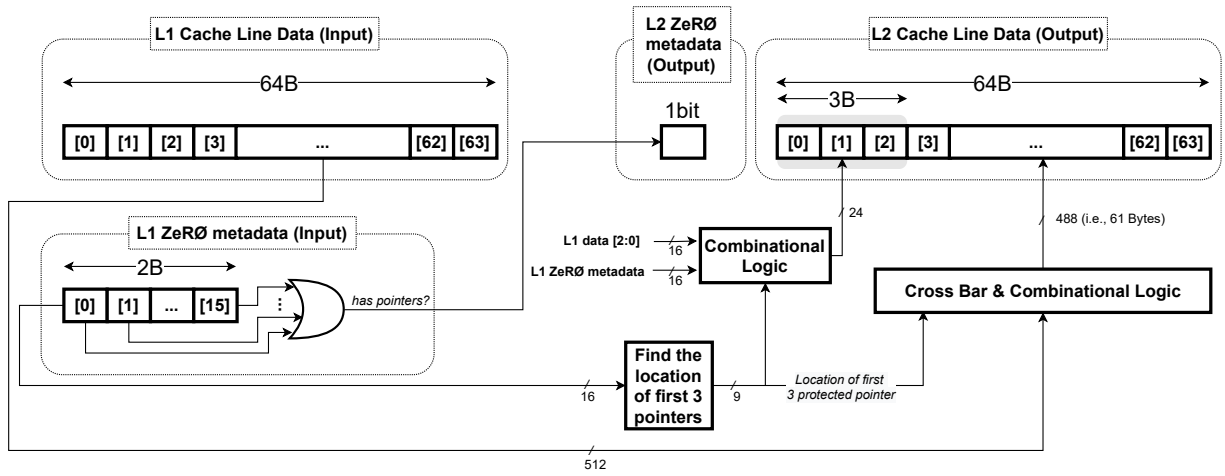


Figure 6.5: Block diagram of the ZeRØ's L1-to-L2 transformation module that is used during the spill operation. The left hand side shows the input L1 cache line data and the corresponding L1 ZeRØ bit vector.

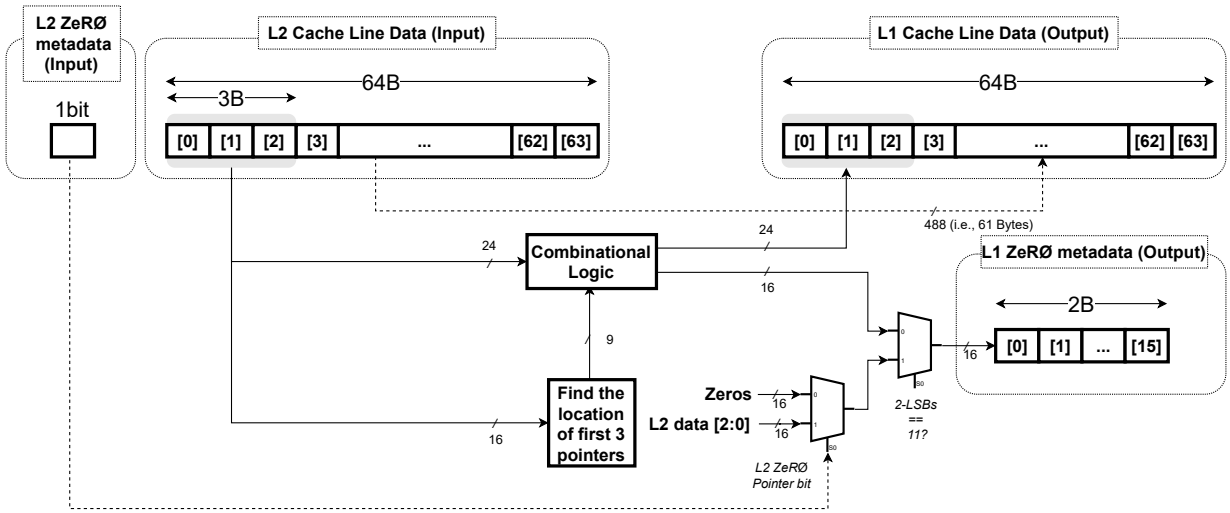


Figure 6.6: Block diagram of the ZeRØ's L2-to-L1 transformation module that is used during the fill operation. The left hand side shows the input L2 cache line data and the corresponding single `Pointer` bit of ZeRØ.

6.4.5 Load/Store Queue Modifications

Since the `CALL`/`RET` instructions generate a store/load micro-op as part of their regular functionality on CISC systems, there is a chance of a load to store forwarding between the stored return address from the `CALL` instruction to a subsequent in-flight load instruction, violating our return address integrity. To avoid this scenario, ZeRØ extends load/store queue entries with 12 bits that

```
 1: Read the Pointer bit for the inserted line
 2: if result is 0 then
 3:    Set the entire bit vector to [0]
 4: else
 5:    Check the least significant 2 bits of byte 0
 6:    if result is 11 then
 7:      Copy bit[2-18] to the L1 bit vector
 8:      Get the location of the first 3 protected addresses
 9:      Set the data of bit[2-18] to the upper 6 bits of location obtained in 8
10:   else
11:      Set the metadata of addresses[Ptr[1-2]] to K[1-2]
12:      Set the data of the first 12 bits to the most
           significant 6 bits of byte[Ptr[1-2]]]
13:   end
14: end
```

Algorithm 4: ZeRØ's L2-to-L1 cache line transformation.

specify whether the entry is associated with a return address, function pointer, data pointer, or regular data (2 bits) in addition to its pointer type (10 bits). This way entries marked as return addresses (or code/data pointers) are written as part of a `CALL` (or `CPtrST`/`DPtrST`) instruction and can only be forwarded to loads that are part of a `RET` (or `CPtrLD`/`DPtrLD`) instruction, respectively. To provide tamper-resistance against side-channel attacks, ZeRØ forwards the value zero from those entries to any matching in-flight load instructions and marks them as potential violators. An advisory exception is thrown only when the potential violating instructions are committed to avoid any false positives due to misspeculation. The checking operation is performed in parallel to the regular address matching process with no performance impact.

6.5 Software Design

In this section we describe the memory management, compiler, and operating system support necessary to enable ZeRØ.

6.5.1 Memory Management

ZeRØ is agnostic to the memory allocator. ZeRØ intercepts any program calls to `free()` or `delete[]` and emits `ClearMeta` instructions to clear code- and data-pointers metadata from the free'd regions if it exists. Additionally, ZeRØ emits `ClearMeta` instructions on function returns to cleanup the stack frame.

6.5.2 Compiler Support

Pointer Integrity. To provide pointer integrity, we need to accurately identify `LOAD` and `STORE` instructions that access pointer values. To do so, our current prototype uses the Clang/LLVM compiler infrastructure to replace program code- and data-pointer loads and stores with our new instructions, `CPtrLD`/`CPtrST` and `DPtrLD`/`DPtrST`.

In order to protect code pointers that are initialized prior to runtime (e.g., entries in C++ virtual tables), we add a `handleGlobals` function that emits `CPtrST` instructions for all global pointers and invoke it at the start of the `main` function as part of program initialization. This way we protect all code pointers that have no explicit `STORE` instructions executed at runtime.

Pointer-Flow Integrity. To prevent pointer confusion between data pointers, we encode the type of the pointer in its most significant 10 bits prior to executing `DPtrLD`/`DPtrST`. We use the pointer's LLVM ElementType, which depends on the type of the pointed-to data structure. The ElementType of each pointer is readily available and does not require points-to analysis. Similarly, we encode code pointer types in the most significant 10 bits of a code pointer prior to executing `CPtrLD`/`CPtrST` to prevent the attackers from confusing incompatible function pointers. In this case, we use the function type as a unique code pointer type.

Return Address Integrity. No compiler support is needed for return address integrity as ZeRØ extends the functionality of traditional `CALL`/`RET` instructions.

Finally, a recent work [31] shows that it is feasible to track data pointers in hardware with no compiler support. This pointer tracking feature should enable ZeRØ to relax its compiler support requirement for data pointer integrity.

6.5.3 Operating System Support

Advisory Exceptions. When ZeRØ's hardware detects an access violation, it throws an exception once the instruction becomes non-speculative. Our exceptions are advisory in nature. In other words, they do not halt program execution. Instead, they just notify the operating system of the

invalid behavior and continue program execution (after rejecting the violating memory access).[2] ZeRØ provides hardware support for suppressing the advisory exceptions by using a permit-list, as described in Section 6.4. For example, it might be desirable to add functions that copy plain bytes (e.g., `memcpy` and `memmove`) to the permit-list as they may generate exceptions upon accessing protected addresses (e.g., pointers). Wherever possible, our compiler pass emits type-aware copying functions that do not need any special exception handling. Of the 16 SPEC CPU2017 C/C++ benchmarks, only two (`502.gcc_r` and `526.blender_r`) have cases where a permit-list is needed. For the rest of the benchmarks, our compiler pass successfully identifies the copied operands types and emits our special instructions for copying the protected fields and regular memory access instructions for the non-pointer fields.

Page Swaps. ZeRØ requires 1 bit of metadata per 64B cache lines. When a page is swapped out from main memory, the page fault handler needs to store the metadata for the entire page into a reserved address space managed by the OS; the metadata is reclaimed upon swap in. The kernel has enough address space in practice (the kernel's virtual address space is 128TB for 64-bit Linux with 48-bit virtual address space) to store the metadata for all the processes on the system since the size of the metadata is minimal (8B for a 4KB page or 0.2%).

Stack Unwinding. The C standard uses `setjmp` and `longjmp` in order to add exception-like functionality to C. `setjmp` saves the current environment including the return address and stack pointer to a memory buffer (`jmp_buf`) while `longjmp` restores the previously saved environment from `jmp_buf`. To guarantee return address (and stack pointer) integrity while saved in `jmp_buf`, ZeRØ instruments `setjmp`/`longjmp` to insert `CPtrST`/`CPtrLD` instructions for those protected addresses. This way an attacker cannot use regular memory instructions to overwrite the return address and stack pointers in `jmp_buf`. The same approach can be applied to the C++ exception handling mechanism by instrumenting the appropriate APIs.

Context Switching. The permit-list contents (96 bytes) are maintained across context switches—

[2]Only faulty store instructions are rejected to guarantee pointer integrity. We do not skip faulty loads as they do not change the control/data flow of the program.

as part of the process control block—if the process uses a permit-list. This step is likely to add minimal overhead (a few `LOAD` and `STORE` instructions takes $\leq 0.1\mu$S) to the OS context switch (typically $3-5\mu$S). Other OS-related tasks remain intact, such as inter-process data sharing, copy-on-write, and memory-mapped files.

Finally, as ZeRØ's metadata is inlined within the pointers themselves, they require no extra work for supporting multi-threaded applications.

6.6 Security Analysis

In this section, we first define the threat model. Next, we analyze the security guarantees provided by ZeRØ and its current limitations.

6.6.1 Threat Model

Adversarial Capabilities. We assume that the adversary is aware of the applied defenses and has access to the source code, or binary image, of the target program. Furthermore, the target program suffers from memory safety-related vulnerabilities that allow the adversary to read from, and write to, arbitrary memory addresses. The attacker's objective is to (ab)use memory corruption and disclosure bugs, mount a code-reuse attack, and achieve privilege escalation. Furthermore, we include DOP [57] attacks in our threat model. We exclude pure data corruption attacks from our threat model as they target non-pointer data. This limitation applies to prior work as well [46, 144, 47, 141]. Due to their prominence, we include speculative execution attacks in our threat model [43].

Hardening Assumptions. We assume that the underlying operating system (OS) is trusted. If the OS is compromised and the attacker has kernel privileges, the attacker can execute malicious code without making CRAs; a simple mapping of the data page as executable will suffice. However, our technique can be applied to the operating system code itself for protecting code and data pointers. We assume that ASLR and W^X protection are enabled—i.e., no code injection is allowed (non-executable data), and all code sections are non-writable (immutable code). Thus, attacks that

modify program code at runtime, such as rowhammer [77] and CLKSCREW [98], are out of scope.

Secrets. Unlike prior work, ZeRØ requires no secret parameters or configuration keys. The security is purely derived from runtime enforcement.

6.6.2 Security Discussion

Return Oriented Programming Attacks. Corrupting code pointers has been the most common and preferred attack vector over the last two decades. For instance, ROP attacks [53] and their just-in-time variant [145] typically start by corrupting the return address of a function to hijack the control flow of a program. ZeRØ's return address integrity effectively mitigates those attacks as it stops the adversary from leaking/overwriting return addresses using 1 bit of metadata per cache line in L2/L3 and main memory. For example, when an attacker tries to overflow a buffer to write to an adjacent return address, ZeRØ rejects the action and raises an advisory exception as the access violates the rules in Table 6.1.

Jump- and Call-Oriented Programming Attacks. Protecting return addresses alone is not sufficient for more advanced attack variants. A variation of the ROP attack uses indirect branch instructions (`JMP`) to transfer control between gadgets. This attack technique is called jump-oriented programming (JOP) [56]. Another similar attack variant is call-oriented programming (COP) [55], which uses gadgets ending with an indirect `CALL` instruction. What makes JOP and COP similar is their use of code pointers for the indirect `JMP`/`CALL` instructions. As ZeRØ's code pointer integrity protects code pointers from being manipulated in memory, an attacker cannot use the pointers to launch a JOP/COP attack.

Counterfeit Object-Oriented Programming Attacks. Unlike ROP/JOP/COP attacks, which (re)use short instruction sequences, in COOP attacks, whole C++ functions are invoked through code pointers in read-only memory, such as `vtables` [146]. Each C++ object keeps a pointer (`vptr`) to its `vtable` (a table containing pointers to virtual methods). A method invocation, therefore, requires (a) dereferencing the `vtable` pointer, (b) computing the respective table in-

dex, and (c) executing an indirect CALL instruction with the table entry of the previous step as an operand. COOP attacks typically hijack program control-flow by overwriting vptrs of existent C++ objects and/or crafting counterfeit C++ objects with arbitrary vptrs. ZeRØ prevents COOP attacks by protecting code pointers inside the vtables and by using data pointer integrity to harden the vptr inside the C++ objects. For instance, using a regular STORE instruction to create a vptr will cause an advisory exception when the counterfeit vptr is accessed with a DPtrLD instruction.

Data-Oriented Programming Attacks. Unlike control-flow hijacking attacks, data-oriented programming (DOP) attacks do not alter the control flow of the program [57, 58, 59]. Instead, DOP attacks abuse data pointers to simulate the attacker's arbitrary computations using the original control flow of the victim program. Mitigating DOP has been a real challenge for prior defenses due the attack's use of data pointers. As data pointers are much more common than code pointers, the overheads of protecting them can be significant. ZeRØ's inlined metadata allows us to provide data pointer integrity with no performance cost. ZeRØ prevents those attacks by ensuring that regular LOAD/STOREs cannot corrupt data pointers.

Pointer Confusion Attacks. An attacker who has access to a DPtrST instruction may potentially overwrite any data pointer as all data pointers use the same encoding state (i.e., 11). To mitigate this issue, ZeRØ assigns a unique 10-bit identifier for every data pointer type and verifies it at DPtrST/DPtrLD call sites. This identifier prevents an attacker from using a vulnerable DPtrST instruction to corrupt arbitrary data pointers in memory. Instead, attackers will be restricted to accessing data pointers of the same (or compatible) type, thus reducing the attack surface. To comply with the C standard [147], ZeRØ permits accessing any data pointer using void* and char* without flagging a violation. All other data pointer types are considered incompatible. Similarly, ZeRØ mitigates code pointer confusion attacks by using the function type as a unique identifier at CPtrST/CPtrLD call sites. We report the total number of unique data- and code-pointer types for SPEC CPU2017 benchmarks in Section 6.7.

Speculative Execution Attacks. Speculative execution attacks represent a major challenge for all

modern security solutions [43]. They allow the attacker to leak program memory by first speculatively executing instructions that are not supposed to execute under normal conditions. The traces left behind in the microarchitecture by the speculatively executed instructions are then used to leak information covertly. While ZeRØ does not prevent speculative execution attacks, ZeRØ takes multiple steps to ensure speculative execution attacks cannot be used to bypass it. For example, a recent work (SpecROP [148]) shows that an attacker can speculatively chain multiple ROP gadgets. SpecROP uses speculative execution to prime the targets of indirect jump instructions and uses them to construct a gadget chain that leaks secrets. As all gadgets are speculatively executed, current defenses do not raise exceptions upon executing them. On the other hand, ZeRØ is resilient against SpecROP as we do not forward regular data (i.e., has a 00 state) to protected addresses (e.g., code pointers with a 10 state) in the processor pipeline. Instead, we mark those cases as potential violations and only raise our advisory exception when they become non-speculative. Thus, SpecROP gadgets will not be able to receive the attacker's primed targets.

In addition, it has been shown that speculative execution attacks can bypass ARM's PAC by speculatively executing pointer signing instructions as gadgets to sign arbitrary pointers [142]. Once the pointers are signed, an attacker can leak the signature via a covert channel and use it to create a forged pointer. This forged pointer is then used to bypass ARM's PAC authentication. This raises the question of whether an attacker can use a speculative execution attack to bypass ZeRØ. The short answer is No. Speculatively executing `CPtrLD`/`DPtrLD` instructions can only leak the pointer value. Leaking code and data pointers cannot alter the control/data flow of the program. On the other hand, overwriting the pointer requires a `STORE` instruction, which cannot be speculatively executed. Finally, `ClearMeta` instructions cannot be speculatively used to clear the pointer metadata bits as they are treated similarly to `STORE` instructions.

6.6.3 Limitations

Non-pointer Data Corruption. The main focus of this work is preventing the corruption of different pointer classes. It is to say, ZeRØ does not prevent regular (non-pointer) data from being

corrupted through program LOAD/STOREs. Non-pointer (or non-control) data attacks that tamper with or leak security-sensitive memory are possible [60]. Defeating non-pointer data attacks requires full memory safety, which typically comes with significant memory and performance overheads. Similar to recent hardware-based solutions (e.g., Intel's CET [9], ARM's PAC [10], and Morpheus [141]), we opt to exclude pure data attacks to simplify our design and performance requirements.

Third-party Code. Similar to prior work [47], ZeRØ provides pointer integrity for instrumented code only. Third-party libraries cannot take advantage of ZeRØ without recompilation. To facilitate communication with unprotected third-party code, ZeRØ provides a couple of options. The first option is to add the starting address and size of the third-party code to the permit-list. This way LOAD/STORE instructions from the third-party code operate normally without generating advisory exceptions. The second option is to clear the code- and data-pointer metadata in memory regions that are shared with third-party code before invoking external libraries. This is done by recognizing external library calls at the compiler level and inserting ClearMeta instructions accordingly. This way regular LOAD/STORE instructions in uninstrumented libraries can access the passed pointers without raising exceptions. ZeRØ, however, never clears the return address metadata bits if they are set, as return address integrity requires no program recompilation and thus can be provided for third party libraries and legacy binaries.

Memory Aliasing. Two memory access instructions can access the same memory location using different types. For example, a C union with a pointer and an integer member can be accessed using both regular STORE and DPtrST instructions. To avoid raising false alarms, ZeRØ statically detects such occurrences at compile time and emits regular STORE instructions for all union accesses. Similarly, we emit regular STOREs for pointers that are "cast" to integers before being stored to memory. While emitting regular STOREs for potential pointers reduces the security coverage, we opt for this solution to eliminate any false positives even if such C idioms are uncommon.

6.7 Evaluation

In this section, we first measure the hardware overheads of implementing ZeRØ. Then, we compare ZeRØ's performance against prior solutions using the SPEC CPU2017 benchmark suite.

6.7.1 Hardware Measurements

ZeRØ adds additional operations to the L1 data cache and the interface between the L1 and L2 caches. Qualitatively, the area overhead of ZeRØ's L1 metadata is 3.125% as it adds 2B per 64B. As the metadata lookup happens in parallel to the L1 data and tag accesses, ZeRØ should have no impact on the L1 access latency. We verify this hypothesis by implementing ZeRØ on top of a 32KB direct mapped L1 data cache. We synthesize the baseline L1 data cache and the ZeRØ modified cache with the Synopsys Design Compiler and the 45nm NangateOpenCell library. We use OpenRAM [88] to generate SRAMs for both the data/tag arrays in L1 data cache and the bit vector arrays for ZeRØ. We report our VLSI measurements results in Table 6.2.

Table 6.2: Area, delay and power overheads of ZeRØ (GE represents gate equivalent).

ZeRØ	**Area (GE)**	**Delay** (ns)	**Power** (mW)
L1 Overheads	[+5.41%] 531,175	[+0.05%] 1.99	[+3.37%] 30.7
L2-to-L1 Transformation	299	1.45	0.04
L1-to-L2 Transformation	326	1.72	0.04

As expected, the overheads associated with the ZeRØ pointer integrity are minor in terms of delay (0.05%) and power consumption (3.37%). The latency of the L2-to-L1 transformation module is less than the L1 data cache latency. This small latency implies that our transformation module can be folded completely within the pipeline stages and will not impact the performance-critical cache line fill operation. On the other hand, the latency of the L1-to-L2 transformation module is slightly higher (1.72ns). This is acceptable as the spill operation is not on the processor critical path. Thus, adding one more cycle to cache line evictions will not impact program execution time. Finally, the area and power overheads of our transformation modules are negligible compared to the L1 data cache.

Table 6.3: Number of unique LLVM function pointer types (**FPtrType**) and data pointer types (**DPtrType**) for the SPEC CPU2017 benchmark suite.

Benchmark Name	**Number of CPtrTypes**	**Number of DPtrTypes**	**Benchmark Name**	**Number of CPtrTypes**	**Number of DPtrTypes**
perlbench	17	72	xalancbmk	837	703
gcc	78	451	x264	50	23
mcf	0	6	blender	566	705
namd	5	10	deepsjeng	0	2
parest	48	611	imagick	21	54
povray	31	148	leela	1	16
lbm	0	2	nab	0	19
omnetpp	298	133	xz	14	18

6.7.2 Software Performance

We use the VLSI measurements as a guideline for our software evaluation. Our VLSI measurements show that ZeRØ's hardware changes have no impact on L1/L2 access latency. Thus, no extra clock cycles are needed to perform our integrity operations. In terms of program instructions, ZeRØ's return address integrity does not add any special instructions. Instead, it extends the functionality of regular `CALL`/`RET` instructions. On the other hand, ZeRØ uses special instructions to access code and data pointers. We note that the `CPtrLD`/`CPtrST` and `DPtrST`/`DPtrST` instructions simply replace traditional loads and stores for code and data pointers, respectively. They do not require any extra registers. We insert `MOV` instructions to encode the pointer types into the upper 10 bits of the `CPtrLD`/`DPtrLD` destination register and `CPtrST`/`DPtrST` source register (all are 48-bit wide pointers). We report the total number of unique data/code pointer types in Table 6.3. Finally, ZeRØ inserts `ClearMeta` instructions upon heap/stack memory deallocation to remove the tags from the code- and data pointers, if they exist. We emulate the overheads of the `ClearMeta` instructions by inserting dummy `STORE` instructions in the corresponding (deallocation) code segments.

6.7.3 Comparison with Prior Work.

To demonstrate the need for implementing ZeRØ, we compare it against the state-of-the-art pointer integrity technique, ARM's PAC using the SPEC CPU2017 workloads. Prior work [47] showed that ARM's PAC can be used to enforce code- and data-pointer integrity. As ARM's PAC is only available in certain Apple SoCs with no support for third party code at the time of writing, we use the same emulation methodology as used by Liljestrand et al. [47] to estimate the performance overheads. We write a LLVM/Clang compiler [89] pass to insert four exclusive-or (`xor`) operations to account for the 4 cycle latency introduced by the PAC instructions. In addition to ZeRØ, we run three different instrumentation configurations:

- *PAC-FPtr.* In this configuration, ARM's pointer authentication is applied to function pointer usages (i.e., forward-edge protection). Our compiler pass inserts the dummy instructions whenever a function pointer is loaded from memory (to emulate code pointer authentication) or stored to memory (to emulate code pointer signing).

- *PAC-RET.* In this configuration, ARM's pointer authentication is applied to return addresses (i.e., backward-edge protection). Our compiler pass inserts the dummy instructions when a `CALL` instruction is executed (to sign the return address before pushing it to the stack memory) and when a `RET` instruction is executed (to authenticate the return address after loading it from memory).

- *PAC-Full.* In this configuration, ARM's pointer authentication is applied to return addresses, code pointers, and data pointers. In addition to the first two configurations, we instrument all data pointer `LOAD` and `STORE` instructions to insert the dummy PAC instructions.

Evaluation Setup. We run our experiments on a bare-metal Intel Skylake-based Xeon Gold 6126 processor running at 2.6GHz with RHEL Linux 7.5 (kernel 3.10). We use the SPEC CPU2017 benchmarks with `ref` inputs and run to completion. To minimize variability, each benchmark is

executed 5 times and the average of the execution times is reported. We notice negligible variance between the different runs for each benchmark configuration.

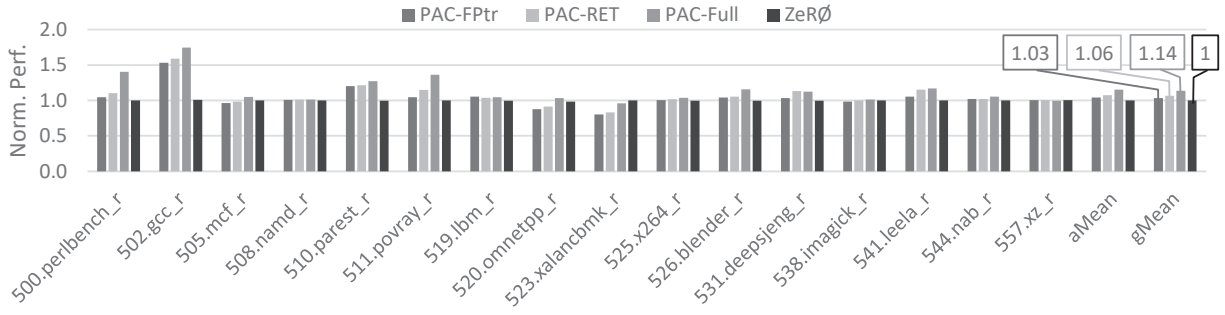


Figure 6.7: Performance overheads of ZeRØ and three different ARM's PAC configurations for the SPEC CPU2017 benchmark suite.

Performance Results. Figure 6.7 shows the runtime overhead of the different design approaches (all normalized to baseline execution with no defenses). As the name suggests, ZeRØ introduces 0% performance overheads on average with a maximum of 0.6%. The overhead of PAC-FPtr is 3% on average with a maximum of 53%. The overhead of PAC-RET is 6% on average with a maximum of 59%.

Protecting all code and data pointers with PAC-Full results in 14% performance overheads on average with a maximum of 75%, which in many situations is considered too costly for ARM's PAC to be practically deployed for data pointer protection. AOS [32] reduces the performance cost of data pointer integrity by using bounds tables and on-chip caches instead of signing & authenticating every data pointer. AOS reports an average performance overhead of 8.4% on SPEC CPU2006 workloads (by running the first 3 billion instructions on the gem5 simulator [134]). ZeRØ reduces the pointer integrity costs to zero by using minimal L1 metadata and only a 0.2% memory overhead.

6.8 Summary

In this chapter we proposed ZeRØ, a hardware primitive for resilient operation under memory corruption attacks with zero overhead. ZeRØ enforces code and data pointer integrity with minimal

metadata. Specifically, using 1 bit per 64 bytes in L2 and beyond, and 3.125% area overhead in the L1, ZeRØ is able to protect the integrity of both code and data pointers. As a result, ZeRØ incurs 0% performance degradation compared to 14% for the state-of-the-art ARM's PAC when applied to its full extent. ZeRØ matches or offers better security guarantees than ARM's PAC and Intel's CET. Moreover, our VLSI results showed that ZeRØ can be implemented with minimal latency, area, and power overheads.

The techniques described in the chapter offer exploit mitigation at no cost. While extant memory safety techniques are more suitable for testing before apps are distributed to customers where higher overheads can be tolerated, exploit mitigation techniques such as ZeRØ which offer no overheads and resilient operation are more suitable for end user deployment.

Chapter 7: Efficient Pointer Integrity For Securing Embedded Systems

With the rise of the Internet of Things and cyber-physical systems, the usage of embedded devices has witnessed a rapid increase. Unfortunately, memory-safety based attacks remain a major concern for embedded systems as they are typically programmed in memory unsafe languages, such as C. The limited processing and storage resources of embedded devices hinders the efforts of securing them using server-grade defenses.

Thus, this chapter presents Efficient Pointer Integrity (EPI), a hardware-based technique that mitigates memory safety-based attacks by ensuring the integrity of valuable application assets (i.e., pointers). The key observation that enables our EPI encoding is that leveraging common software properties allows for harvesting extra bits from pointers on 32-bit architectures. For example, compilers typically align stack frames to 16-byte boundaries. That means the maximum number of 32-bit return addresses per 64-byte cache lines is four instead of 16, reducing the metadata that is needed for enforcing return address integrity. Additionally, fixed-width instructions on RISC architectures, such as the RISC-V four-byte instructions, mean that any instruction address (e.g., return address or function pointer) will be four-byte aligned and will have its two least significant bits set to zero. EPI harvests those bits (and inserts extra padding bits if necessary) to efficiently store the pointer integrity metadata on 32-bit architectures.

EPI takes multiple steps to address the power and reliability challenges of embedded systems [149]. First, as many embedded systems nowadays are battery operated, they typically have low power consumption budget. EPI mitigates the power overheads by avoiding frequent crypto operations [46, 10, 47] and continuous randomization [141] approaches. Second, as embedded devices are often used in safety-critical environments, they typically have strong reliability requirements. EPI maintains the reliability of the protected system by avoiding terminating the victim process upon detecting an attack. Instead, EPI continues program execution after skipping

the violating instruction. As a result, EPI is resilient against denial-of-service attacks, which are commonly used against embedded devices. Third, EPI does not require any secret parameters or configuration keys that need to be explicitly protected.

The remainder of this chapter is organized as follows. Section 7.1 further motivates the need for EPI. Section 7.2 provides an overview of how EPI works. Section 7.3 describes the hardware changes that are required to implement EPI whereas Section 7.4 specifies the software design. Section 7.5 analyzes the security guarantees provided by EPI and its limitations. Section 7.6 evaluates the EPI performance overheads. Section 7.7 summarizes the chapter.

7.1 Motivation

Memory corruption attacks represent a major threat for embedded systems software. For example, overwriting code pointers such as return addresses and function pointers allows an attacker to hijack the control-flow of an application and achieve arbitrary code execution [53]. Moreover, overwriting data pointers can alter an application's benign behavior without changing its control-flow [57]. Both control- and data-flow manipulation attacks cause significant damage to the victim system. In 2019, researchers showed how to exploit a series of buffer overflow vulnerabilities, named QualPwn [150], in the Qualcomm WLAN and modem firmware that ships in millions of Android devices. The vulnerabilities allow for code execution on the victim device by sending specially-crafted packets to an Android's device modem. In 2020, another series of zero-day vulnerabilities, dubbed Ripple20 [151], targeted a TCP/IP library found at the base of many embedded devices. The impacted devices include smart home devices, power grid equipment, routers, satellite communications equipment, and many others.

One strategy to harden embedded systems against memory safety-based attacks is to deploy exploit mitigation techniques, such as address space layout randomization (ASLR) [152] and ARM's pointer authentication (PAC) [10]. These techniques raise the bar for the attacker by making it harder to exploit memory safety vulnerabilities while keeping the performance and memory costs lower than the full memory safety solutions [33]. Unfortunately, state-of-the-art exploit mitigation

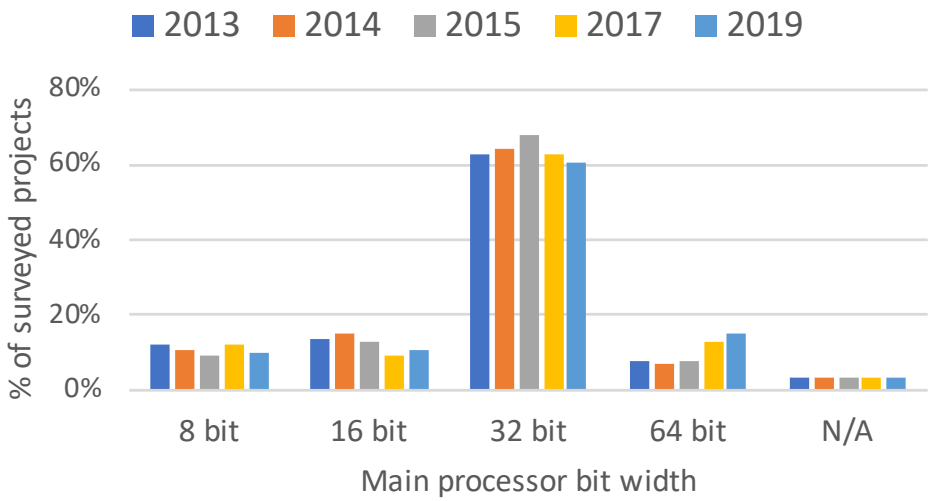


Figure 7.1: Embedded systems market trend from 2013 to 2019 [48].

techniques are mainly designed for 64-bit processors. For example, randomization-based solutions, such as ASLR [152], take advantage of the massive 64-bit virtual address space to hide the valuable assets. Other solutions, such as ARM's PAC [10], leverage the currently unused upper bits in 64-bit pointers to store metadata. As a result, such solutions perform poorly when deployed on non 64-bit processors, which are the common choice for embedded systems. Figure 7.1 shows that the embedded world is dominated by 32-bit processors [48, 49]. As a result, there is a need for solutions to the problem of securing embedded 32-bit systems with minimal performance, power, and area overheads.

7.2 System Overview

In this section, we show how EPI protects the main application assets: function pointers, data pointers, and return addresses on 32-bit architectures. Then, we describe how EPI manages its metadata.

7.2.1 Function Pointer Integrity

As function pointers are stored in application memory (i.e., stack, heap, and globals), they can be overwritten due to memory safety-based vulnerabilities. Changing a function pointer alters the

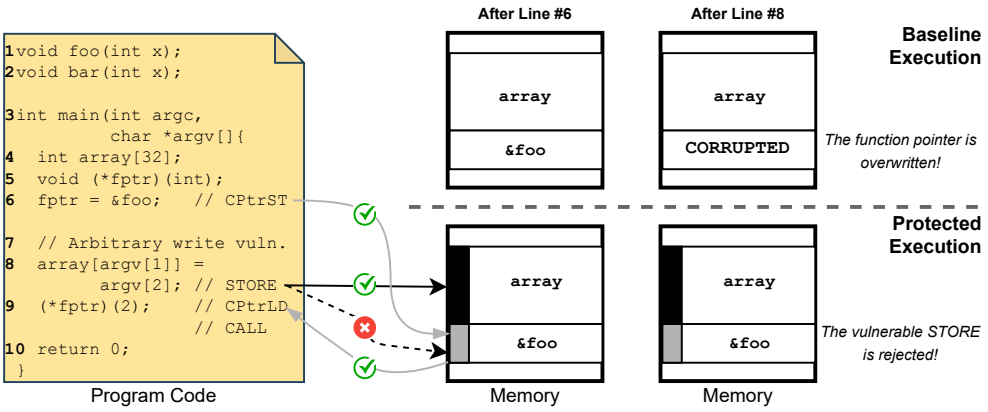


Figure 7.2: A sample C application highlighting how EPI protects function pointers in memory. The left hand side shows a code snippet with a memory safety-based vulnerability in Line 8. Under baseline execution conditions (top-right corner), the vulnerability can be exploited to corrupt the function pointer, `fptr`. With EPI (shown in the bottom-right corner), `fptr` can only be accessed by the code pointer store and load instructions (Lines 6 and 9). Thus its integrity is protected by rejecting the violating `STORE` instruction from Line 8. The same technique is used to protect data pointers and return addresses.

application control flow. Therefore, function pointers are common targets for attackers. In order to guarantee the integrity of function pointer (and any instruction-based address that is stored in memory, such as indirect jump targets), EPI uses two special instructions, Code Pointer Load (`CPtrLD`) and Code Pointer Store (`CPtrST`), to access function pointers. If any other memory access instruction is used to target a function pointer, EPI will reject the violating instruction, effectively preventing function pointers from being overwritten.

To better understand how our function pointer integrity works, let us consider the example in Figure 7.2. The code snippet (shown on the left hand side) shows a simple C application with a memory safety-based vulnerability that gives the attacker arbitrary write capabilities (Line 8). As a result, the attacker can write arbitrary values to arbitrary locations in memory. The attacker's goal is to hijack the control flow of the application by overwriting the function pointer, `fptr`, to point to a different function other than `foo`. The attack succeeds under baseline execution (shown on the top right corner) as the violating `STORE` instruction is able to access any memory location with no restrictions. EPI provides pointer integrity by only granting the `CPtrST`/`CPtrLD` instructions exclusive access to function pointers. As shown in the bottom-right corner of Figure 7.2, `CPtrST`

marks the function pointer location with a unique tag (e.g., 10) on the first use. Only CPtrLD instructions are allowed to load function pointers from those specially-tagged locations. Thus, the attacker fails to overwrite fptr with the vulnerable STORE instruction. Our unique tags are stored in bit vectors in the L1 data cache and are encoded within the application data when transferred to the L2 cache and/or main memory, as will be described in Section 7.3.

7.2.2 Data Pointer Integrity

EPI enforces the integrity of data pointers in a similar fashion to function pointers. Two new instructions, Data Pointer Load (DPtrLD) and Data Pointer Store (DPtrST) are used to access data pointers. We use a special tag (e.g., 11) to mark data pointers in the L1 data cache. The tag is assigned upon executing the DPtrST instruction and is verified upon executing the DPtrLD instruction. Accessing data pointers with regular LOAD/STORE instructions is rejected to prevent attackers from manipulating data pointers.

In order to avoid confusing data pointers (i.e., replacing one data pointer with another pointer of an incompatible type), DPtrST/DPtrLD uses an additional register operand, RegX. The compiler writes the data pointer type to RegX at each data pointer load and store location. This step is done at the compiler intermediate level by using the readily available pointer's ElementType as defined by the compiler without requiring any points-to analysis. The hardware verifies that the value in RegX matches the type metadata, which is stored adjacent to the data pointer in memory. This way an attacker cannot exchange two different data pointers with each other to hijack the application data flow. The same approach can also be applied to function pointers to avoid type confusion. In this case, we (1) use the function type as a unique function pointer type and (2) write it to the RegX operand of function pointer load and store locations at compile time.

7.2.3 Return Address Integrity

In order to mitigate return-oriented programming (ROP) attacks, EPI enforces the integrity of return addresses by extending the functionality of regular CALL and RET instructions without any

compiler support. Upon executing a `CALL` instruction, our hardware pushes the return address to memory and marks it with a unique tag (e.g., 01) in the L1 data cache. Any memory access instructions, including the traditional `LOAD` and `STORE` instructions and our special code- and data-pointer variants, cannot access a memory location as long as it is tagged as a return address. When a `RET` instruction is executed, our hardware pops the return address from memory and simultaneously clears its corresponding metadata if and only if it is originally marked with the return address tag (i.e., 01). This way EPI prevents the attackers from using arbitrary data in memory as potential return addresses. By limiting the return address accesses to `CALL` and `RET` instructions, EPI mitigates ROP without using shadow stacks or recompiling the application.

7.2.4 Metadata Management

Ensuring the integrity of the metadata is a key requirement for EPI to (1) prevent the attackers from manipulating the metadata and (2) avoid causing false positives during normal application execution. While return address tags are exclusively written and cleared by the `CALL` and `RET` instructions, the function- and data-pointer metadata needs special treatment as pointers can be written and read multiple times. EPI introduces one more instruction, `ClearMeta <R1>, <R2>`, to explicitly clear the function- and data-pointer metadata when a heap object is freed or a stack frame is deallocated. The `ClearMeta` instruction takes two register operands, `R1` and `R2`. `R1` holds the starting address of a 64B cache line whereas `R2` holds a binary mask to the corresponding 64B cache line, where one allows and zero disallows changing the state of the corresponding byte. We use the mask to perform partial updates of metadata within a cache line. At compile time, we insert `ClearMeta` instructions to clear the metadata of the stack frames that hold function and/or data pointers upon function return. We also create a runtime wrapper around the memory deallocation functions, `free` and `delete`, to clear the function- and data-pointers metadata from the deallocated regions if it exists.

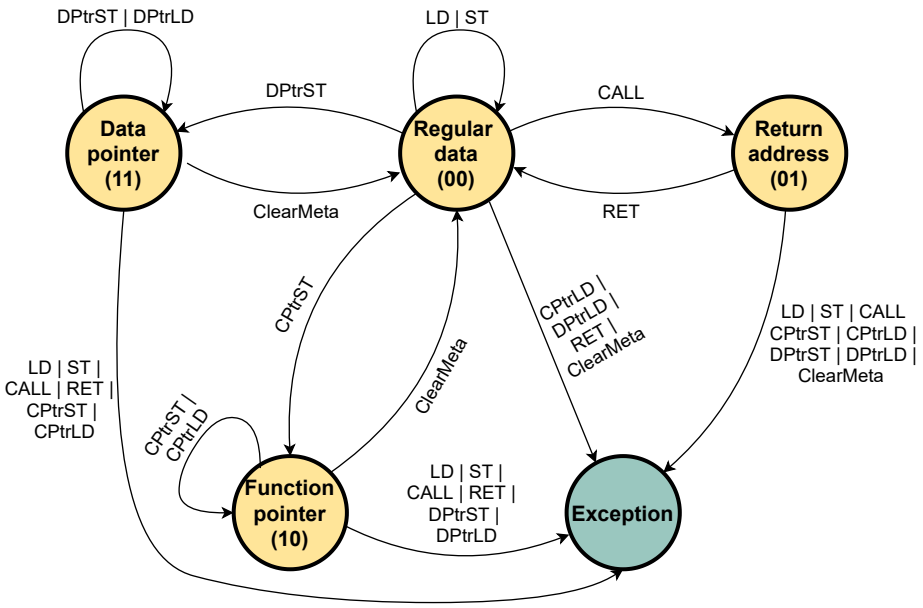


Figure 7.3: Finite state machine of the different EPI metadata (represented by states) and instructions (shown as transitions). The main idea is restricting access to memory locations, which are marked with similar metadata state, to a subset of memory instructions. Incompatible memory accesses (i.e., accesses that use the wrong instruction type) are rejected, as represented by the exception state.

7.2.5 Putting It All Together

Figure 7.3 shows a finite state machine that summarizes how our different EPI instructions interact with the EPI memory tags. The first four states (shown in yellow) represent the different application assets: regular data, return addresses, function pointers, and data pointers. Any valid memory access instruction moves the target memory location tag from one state to another. However, all invalid memory accesses cause a violation, as represented by the exception state in Figure 7.3. For example, a `CPtrST` instruction that targets a memory location with a tag equals 00 will change the tag state to 10. However, the same `CPtrST` will be rejected if it targets a memory location that has a 01 tag. In order to provide the operating system (or the system administrator) with more information about the violating instruction, EPI uses advisory exceptions. Unlike traditional exceptions, EPI's advisory exceptions do not crash the running process. Instead, they simply notify the operating system and provide the address and operands of the violating instruction, if more forensics is needed.

7.3 Microarchitecture Design

This section describes the hardware changes that are required to implement EPI.

7.3.1 Processor Modifications

In order to add EPI to an embedded device processor, the following extensions are needed. First, we extend the instruction decoder to support the `CPtrST`/`CPtrLD`, `DPtrST`/`DPtrLD`, and `ClearMeta` instructions. Second, we modify the logic for the `CALL` and `RET` instructions to update and validate the return address metadata. Third, we add an exception handling module that is responsible for (a) notifying the operating system when an access violation occurs and (b) checking the address of the violating instruction against the permit-list contents, if it is not empty, to trigger or suppress the exception accordingly. Fourth, a set of registers can be (optionally) introduced to avoid causing any register pressure on the main register file due to the extra operand of EPI's memory access instructions.

7.3.2 Memory Hierarchy Modifications

A subset of embedded processors, especially the ones that run lightweight operating systems, use data caches for enhancing performance. Depending on how many levels of caches are available, EPI requires the following extensions. The key design goal is to speed-up metadata lookup in the upper-level caches that are closer to the processor (i.e., L1) by using bit-vector metadata and reduce the memory overheads in the lower-level caches that are closer to the main memory (i.e., L2) by using compressed metadata.

L1 Data Cache. In our design, we use a 32-bit vector of metadata, `L1Vec`, per each 64B cache line in the L1 data cache (i.e., a 6.25% extra storage). Each 2 bits indicate whether a 4B chunk is a regular data, function pointer, data pointer, or a return address, as shown in Figure 7.4. The per-pointer identifier needs no dedicated storage as it is readily available in the padding bytes, as described in Section 7.4. The metadata is checked—in parallel to regular data access—when a

memory instruction reaches the L1 data cache. If an access violation is detected, a signal is sent to the exception handling module.

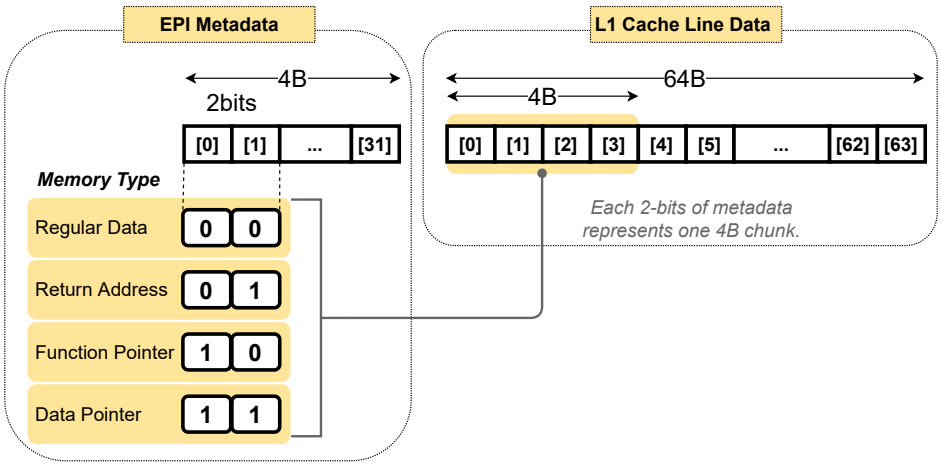


Figure 7.4: EPI's metadata encoding in the L1 data cache on 32-bit architectures. We use a 2 bits to indicate whether any 4B is a regular data, function pointer, data pointer, or return address.

Systems with ECC-enabled caches for better reliability can completely avoid the storage overheads of our `L1Vec`. The key idea is to tweak the original ECC encoding and decoding algorithms to compute the ECC using the 32-bit data and 2-bit metadata altogether. When a memory access occurs, the 2-bit metadata is implicitly known (e.g., a `CPtrLD` instruction expects a metadata of 11) and can be added to the 32-bit data before computing the ECC. If the computed ECC matches the stored ECC value, then the data is correct and the access is valid. If a mismatch occurs, either a data corruption or an EPI access violation occurs. Both cases requires exception handling. Prior work shows how implicitly encoding metadata bits in ECC works without compromising reliability [143].

L2 Cache and Main Memory. For the lower-level components of the memory hierarchy (i.e., the L2 cache and main memory), we use a compressed metadata layout with only 2 bits, `EPI`[1:0], per each 64B cache line. If a cache line has no function pointers, data pointers, or return addresses, we do not modify its contents and set its corresponding metadata bits to 00. If a cache line has any pointers, we encode the pointer offset within the cache line and its type in the first three bytes of the cache line as a header, as shown in Figure 7.5. The original contents of the header are

copied to the spare bits of the pointers, which we harvest with software optimizations, as described in Section 7.4. As our compressed metadata adds minimal storage overheads (i.e., 0.39%), it can be efficiently stored into spare ECC bits or in a disjoint memory region for non-ECC memories.

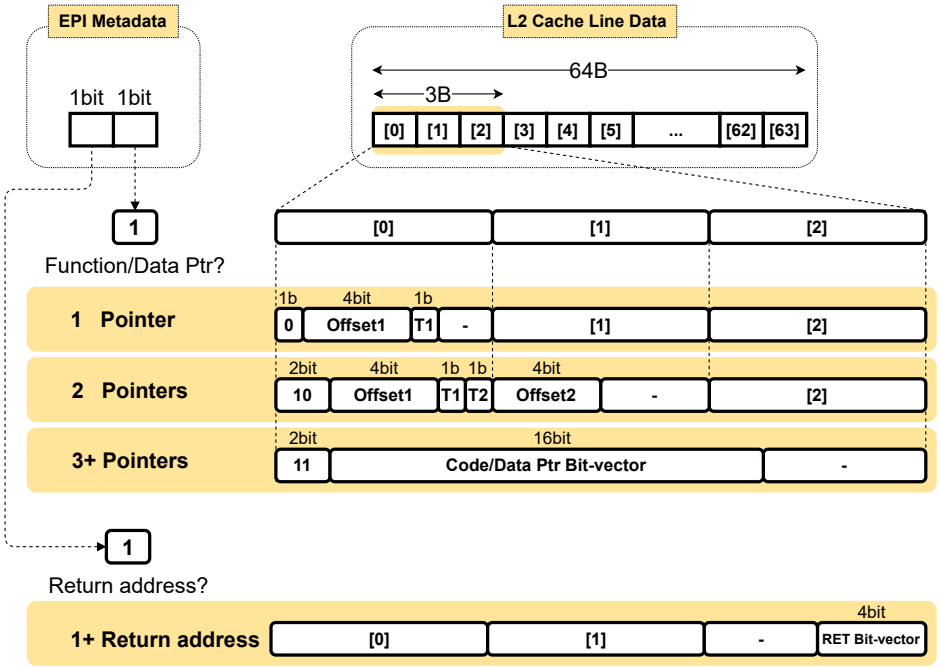


Figure 7.5: EPI's metadata encoding in the L2 cache and main memory. We use 2 bits per cache line to indicate whether the cache line has function/data pointers and/or return addresses. We use the first three bytes of the cache line as a header where Offset1/Offset2 encodes the offset of the pointer in the cache line and T1/T2 encodes its type (i.e., function pointer or data pointer). A 4-bit vector is used to encode the metadata of return addresses (i.e., whether a 16B chunk has a return address or not).

Metadata Encoding & Decoding Modules. In order to switch between the EPI's bit vector metadata and its compressed layout, we introduce metadata encoding and decoding modules between the L1 data cache and the L2 cache (or main memory). Algorithm 5 shows the steps of the metadata encoding process, whereas Algorithm 6 shows the steps of the metadata decoding process. Both modules can be implemented with simple combinational logic. The performance overheads of the two modules are evaluated in Section 7.6.

Input : A 64-byte L1 cache line and a 32-bit vector, `L1Vec`.
Output: A 64-byte L2 cache line and a 2-bit EPI metadata, where `EPI`[0] is the return address bit and `EPI`[1] is the pointer bit.

```
Read all bits from L1Vec and OR them
if result is 0 then
    Evict the line as is and set its EPI[1:0] to 00
else
    Count the number of pointers in L1Vec
    if pointer count is 0 then
        Set EPI[1] to 0
    else if pointer count is 1 then
        Set EPI[1] to 1
        Write the location of the pointer and its type in the lower 6-bits of byte[0]
        Copy the lower 6-bits of byte[0] to the 6-spare bits of the pointer
    else if pointer count is 2 then
        Set EPI[1] to 1
        Write the location of the 2 pointers and their type in the lower 12-bits of byte[0:1]
        Copy the lower 12-bits of byte[0:1] to the 12-spare bits of the 2 pointers
    else                                    // pointer count is 3 or more
        Set EPI[1] to 1
        Write the pointers' type as a 16-bit vector in byte[0:2]
        Copy the lower 18-bits of byte[0:2] to the spare bits of the first 3 pointers
    end
    Count the number of return addresses in L1Vec
    if return addresses count is 0 then
        Set EPI[0] to 0
    else
        Set EPI[0] to 1
        Write the return addresses locations as a 4-bit vector in the upper bits of byte[2]
        Copy the upper bits of byte[2] to the 4-spare bits of the first return address
    end
end
```

Algorithm 5: EPI metadata encoding steps (L1-to-L2).

Input : A 64-byte L2 cache line and a 2-bit EPI metadata, where `EPI`[0] is the return address bit and `EPI`[1] is the pointer bit.
Output: A 64-byte L1 cache line and a 32-bit vector, `L1Vec`.
Read the `EPI`[1:0] bits of the inserted line
if *result is 00* **then**
  Set the entire `L1Vec` to [0]
else
  if *has return address* **then** `// EPI[0] is 1`
    Get the return addresses locations from the upper 4-bits of byte[2]
    Set the corresponding places in `L1Vec` to 01
    Copy the 4-spare bits of the first return address to the upper bits of byte[2]
  end
  if *has pointer* **then** `// EPI[1] is 1`
    Check the least significant 2 bits of byte[0]
    Get the locations of the pointers and their type from byte[0:2] as shown in Figure 5
    Set the corresponding places in the `L1Vec` to 10 or 11 based on the pointers' type
    Copy the 18-spare bits of the first 3 pointers to the lower 18-bits of byte[0:2]
  end
  Set the rest of bits in the `L1Vec` to zeros.
end

Algorithm 6: EPI metadata decoding steps (L2-to-L1).

7.4 Software Design

In this section we explain the main software properties that are used to enable EPI. Then, we describe our compiler and operating system support.

7.4.1 Software Properties

One constraint that prevents the immediate porting of the state-of-the-art exploitation mitigations (e.g., ARM's PAC [10] and ZeRØ [11]) to 32-bit architectures is the lack of unused in-pointer bits to store the metadata. While the upper bits of 64-bit pointers are currently unused by software, there are no unused bits in 32-bit pointers. To address this problem, EPI leverages common software properties to harvest more bits to use on 32-bit architectures without affecting the application correctness or introducing significant memory overheads.

Aligning Stack Frames. Each program function has its own stack frame, in which local variables and return address are stored. Current compilers typically align stack frames to N-bytes boundaries

as a performance optimization. The number of alignment bytes, N, defines the maximum number of return addresses that can appear in a single cache line. For example, a 64B cache line can only store a maximum of $64/N$ different return addresses. We leverage this compiler optimization to reduce the size of the metadata bit vector that is associated with return addresses. By using the default stack frame alignment (i.e., N = 16B), a 4-bit vector is sufficient to track the locations of potential 32-bit return addresses in any 64B cache line. We show how EPI's compressed encoding takes advantage of this feature in Section 7.3.

Aligning Program Functions. Compilers, such as LLVM, provide compile-time options (e.g., `-falign-functions`) and function attributes (e.g., `__attribute__((aligned(S)))`) for specifying the minimum alignment for the first instruction of a function. As function pointers typically point to function starting addresses, the number of alignment bytes, S, affects the least significant bits of each function pointer. For example, using a function alignment, S = 16B, means that the $log_2(16) = 4$ least significant bits of any function pointer are always set to zero. EPI harvests those bits to store the tags when function pointers are spilled from the L1 data cache to the L2 cache and main memory.

Compacting Code Space. On 32-bit architectures, the maximum size of the code address space in virtual memory is 4GB. However, the majority of embedded applications do not use the entire code space. Even for statically linked applications, code size is typically in orders of MBs. We propose compacting the size of the code address space to 1GB in order to leverage the two most significant bits of code pointers, including return addresses and function pointers. We note that this optimization does not apply to data pointers. Thus, data items on heap and stack can still use the entire 4GB of virtual memory on 32-bit architectures as before.

Furthermore, as instructions on RISC architectures have fixed width, some of the least significant bits of code pointers can be used for metadata encoding. For example, RISC-V instructions are all of 32-bit width, meaning that the two least significant bits of return addresses and function pointers are always set to zero. EPI harvests those bits as well to facilitate the metadata encoding.

Inserting Padding Bytes. While the above optimizations work for code pointers (i.e., instruction-

based addresses), they cannot be applied for harvesting bits in 32-bit data pointers. On one hand, the most significant bits of data pointers are not always set to zero. Compressing the data address space might cause problems for embedded applications that operate on large chunks of data. On the other hand, the least significant bits of data pointers are only set to zero in case of a allocation base address (i.e., pointers returned by `malloc` or `new`). However, applications may arbitrarily create derived pointers that point to any byte-aligned location within the allocation and store it to memory. Thus, derived pointers will not have their least significant bits set to zero, preventing us from harvesting them for metadata storage. As a result, we opt to explicitly insert two padding bytes adjacent to data pointers to save the data-pointer metadata tag (i.e., 11) and type. We quantify the performance overheads of the inserted padding bytes in Section 7.6.

Finally, Figure 7.6 shows the layout of different application assets on 32-bit architectures after applying the above optimizations. The number of harvested bits in return addresses and function pointers equals four and six, respectively. Furthermore, EPI-protected function and data pointers can utilize an additional two padding bytes.

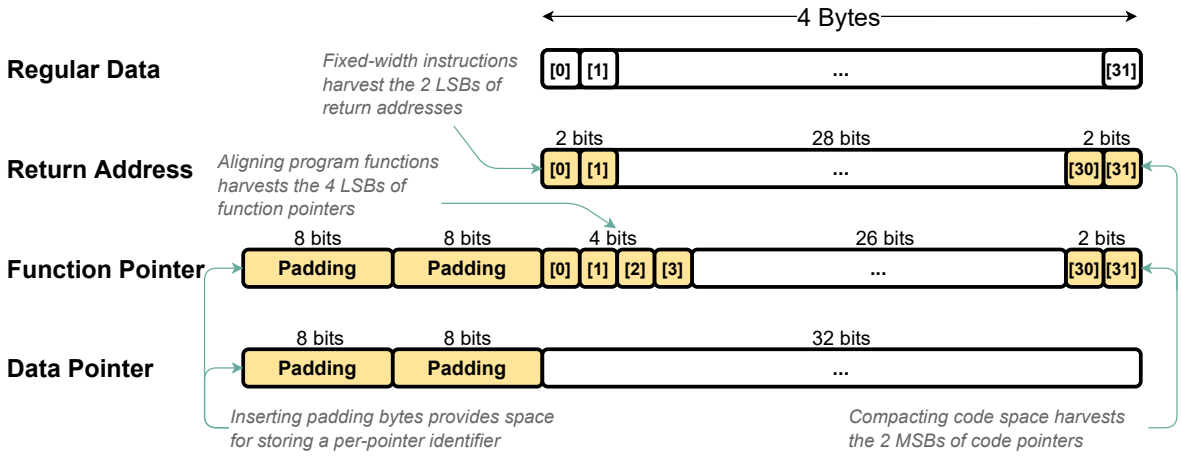


Figure 7.6: Different pointers layout on 32-bit architectures after applying EPI's optimizations.

7.4.2 Compiler Support

While EPI guarantees return address integrity for legacy binaries without recompilation, we use compiler support for enforcing function- and data-pointer integrity, as described below.

Code Instrumentation. We use the Clang/LLVM compiler infrastructure [89] to instrument the application code. First, we modify the compiler front-end to insert two padding bytes per function and data pointers, if desired, to mitigate pointer confusion attacks. This is an optional feature that could be turned off if the application does not heavily use data and/or function pointers. Second, we write a compiler pass that works at the intermediate (IR) level. Our compiler IR pass identifies pointer access instructions and replaces them with our new `CPtrLD`/`CPtrST` and `DPtrLD`/`DPtrST` instructions. Then, we use the pointer's LLVM ElementType, which depends on the type of the pointed-to data structure, as a unique identifier per each data pointer load and store instruction. The size of the identifier depends on the number of different data-pointer types in the application. Based on our experiments, we set the size of the identifier to ten bits. Function types are similarly used as unique identifiers in function-pointer load and store locations. Finally, our compiler pass identifies functions that store function and/or data pointers as local variables and emits `ClearMeta` instructions on function returns to cleanup the stack frame.

Runtime Wrappers. While the majority of embedded applications use statically allocated memory for maximizing efficiency, some applications might use dynamic memory allocations. In this case, EPI creates a wrapper around memory deallocation functions (e.g., `free` and `delete`). Inside the wrapper, EPI invokes the standard `malloc_usable_size` function to get the size of the free'd memory object and iterates over all cache lines of the free'd object clearing its function- and data-pointers metadata with our `ClearMeta` instruction.

7.4.3 Operating System Support

Microcontrollers and embedded devices typically run bare-metal applications with no operating system support. In this case, EPI can be directly deployed to protect the bare-metal application

with no further changes. However, some microcontrollers have an operating system that schedules and runs multiple applications on the device. To support such devices, EPI requires minimal modifications to the operating system code similar to prior work [11].

Exception Handling. EPI provides the option to trigger an advisory exception when a memory access violation occurs. Instead of crashing the running applications, our advisory exceptions send the violating instruction information (i.e., instruction address and operands) to the operating system. The operating system then takes the decision of either terminating the application or not. Furthermore, EPI provides an optional per-application permit-list that can store the address ranges of code sections for which the advisory exceptions should be suppressed. This feature can be used to avoid false alarms in case of functions that treat pointer and non-pointer data similarly, such as `memcpy` and `memmove`. The permit-list is created during the application loading and is mapped to the hardware exception circuitry to allow the hardware to decide on when advisory exceptions are triggered. The operating system is responsible for maintaining the contents of the permit-list (eight 8-bytes entries) during context switches. For example, it can be stored as part of the process control block or saved in an attacker-inaccessible memory region.

Page Swapping. If multiple processes run on the same embedded device, the operating system swaps certain memory pages to disk in order to create enough space in main memory for supporting the currently running processes. If a swapped-out page belongs to EPI-protected applications, the operating system needs to store the metadata of this page in a separate memory region until the page is swapped in again. This step adds minimal memory overheads as EPI uses 2-bits of metadata per 64B cache lines (or 16B for a 4KB page).

7.5 Security Analysis

In this section, we first define the threat model. Next, we reason about how EPI mitigates state-of-the-art pointer manipulation attacks. Then, we discuss the EPI limitations.

7.5.1 Threat Model

We consider a threat model that is consistent with the state-of-the-art defenses against pointer manipulation attacks [47, 141, 11]. Specifically, we assume that the victim application is written in a memory unsafe languages, such as C/C++, and suffers from one or more memory safety vulnerabilities, such as buffer overflow or use-after-free. The above vulnerabilities grant the attacker arbitrary read/write capabilities to the application memory.

Additionally, we assume that the source code of the victim application and/or its binary image are known to the attacker. However, the attacker cannot manipulate the victim application source code or binary instructions (i.e., code sections are verified at boot time and are non-writable at runtime). The attacker's goal is to leverage the memory safety-based vulnerabilities to mount an attack and hijack the control and/or data flow of the victim application. This includes using control-flow hijacking attacks, such as ROP [53, 54], COP [55], JOP [56], and COOP [146] and data-oriented programming attacks such as DOP [57] and BOP [58], which are all included in our threat model. Similar to prior exploit mitigations, pure data corruption attacks, such as flipping regular non-pointer data [60], are out-of-scope. Mitigating non-pointer data manipulation attacks requires full memory safety solutions, which come with high performance overheads.

Finally, we assume that all hardware components including the ones proposed in this chapter are trusted and tamper-resistant. Attacks that exploit hardware vulnerabilities, such as rowhammer [77] and CLKSCREW [98] are out of scope.

7.5.2 Security Discussion

Control-Flow Hijacking Attacks. Attacks, such as ROP [53], JIT-ROP [145], COP [55], and JOP [56], compromise the victim system by corrupting code pointers, such as return addresses and function pointers. As EPI enforces all pointers' integrity while stored in the application memory, it effectively mitigates these attacks.

A different type of code-reuse attacks is counterfeit object-oriented programming (COOP), in which the attacker reuses whole C++ functions by either (1) manipulating the contents of the

virtual function tables, (2) overwriting the virtual pointers (vptr) of existent C++ objects, or (3) tricking the victim application to use counterfeited objects that include attacker-controlled data and vptr. EPI provides natural protection against all COOP approaches. First, our function pointer integrity protects all virtual function table entries. Second, our data pointer integrity prevents the attacker from both: overwriting the vptr of existent C++ objects and creating fake objects as vptrs can only be created via a `DPtrST` instruction.

Moreover, EPI works against a powerful attacker who controls a `CPtrST` instruction as our identifier, which is encoded as a register operand in the vulnerable instruction, limits the attacker's ability to overwrite arbitrary function pointers. Instead, each `CPtrST` instruction can only access function pointers which share the same function type, highly reducing the attack surface.

Data-Flow Hijacking Attacks. The common theme of all known data-flow hijacking attacks, such as DOP [57] and BOP [58], is their ability to manipulate data pointers to achieve arbitrary computations without modifying the application control-flow. While such attacks have not been demonstrated yet in embedded environments, EPI's data pointer integrity provides an efficient way to mitigate their threat. Furthermore, the additional data pointer identifier that is used by EPI ensures that a vulnerable `DPtrST` instruction has limited attack surface (i.e., only memory locations with compatible data pointer types are reachable). Prior work showed that a ten-bit unique identifier is sufficient to cover different data pointer types in the SPEC CPU2017 benchmarks [11].

Spectre Attacks. While speculative execution is not common in resource constrained devices (due to energy limitations), EPI's security guarantees remain valid under speculative execution. This is simply because altering the application control or data flow requires overwriting a code or data pointer using a violating `STORE` instruction, which cannot be speculatively executed. To mitigate the risk of speculatively leaking code and data pointers (or speculatively chaining multiple code-gadgets [148]), EPI does not allow violating instructions to speculatively forward their results if they violate the rules of Figure 7.3. For example, attackers cannot use speculative `RET` instructions to load memory from a regular memory location (i.e., has a 00 state) that is controlled by the attacker.

7.5.3 Limitations

Addressing Pure Data Corruption Attacks. Similar to prior exploit mitigation techniques [10, 141, 11], EPI does not prevent non-pointer data attacks [60]. While addressing this attack vector for all variables comes with the high cost of enforcing full memory safety, EPI provides an option to guard a subset of the application non-pointer data under certain conditions. For example, if the application contains security-critical non-pointer data (e.g., an `is_admin` global variable) that needs to be protected, EPI may treat those variables similarly to data pointers. In other words, the security-critical fields are padded to 6 bytes (4B of a regular pointer and 2B for storing the identifier). Then, all memory instructions that access the security-critical fields are replaced with `DPtrLD` and `DPtrST` instructions. Finally, a unique identifier is assigned to each security-critical field at compile time to prevent confusing them with any other pointers.

Handling External Libraries. If the protected application uses external libraries, EPI will enforce return address integrity for such libraries. For enforcing function- and data-pointer integrity, we provide three options for handling external libraries with different security-usability guarantees. First, the user can choose to compile the libraries with EPI's compiler passes to enjoy the same security coverage as the main application. Alternatively, we can identify all calls to external library code at compile time and ensure that any data that is passed externally has no code or data pointers. If such data exists, it is sufficient to clear the pointer metadata of the shared objects using `ClearMeta` instructions. The third option is to simply add the instruction address ranges of the external libraries to the permit-list in order to avoid generating false alarms if the external library code accesses a protected code or data pointer.

7.6 Evaluation

In this section, we evaluate the performance overheads of EPI on a real machine using the SPEC CPU2017 workloads. Then, we compare EPI against a 32-bit variant of the state-of-the-art exploitation mitigation technique, ARM's PAC. Finally, we estimate the hardware overheads

of EPI using the CACTI modeling tool.

7.6.1 Experimental Setup

In order to run real workloads to completion in a reasonable time, we opt to use real machines to emulate the performance overheads of our proposal instead of using microarchitectural simulators, which typically suffer from long simulation times. Thus, we run our experiments on a machine equipped with an Intel Skylake-based 2.6GHz Xeon Gold 6126 processor, running RHEL Linux 7.5. We use `Clang-4.0` to compile the SPEC CPU2017 benchmarks using the following baseline flags, "`-m32 -fPIE -pie -fno-strict-aliasing -Wno-everything -O3`". For all experiments, we run the `ref` inputs of the SPEC CPU2017 workloads to completion. Each benchmark is executed five times and the average of the execution times is reported.

7.6.2 Performance Results

Methodology. EPI uses regular `CALL` and `RET` instructions to verify return addresses and introduces new memory access instructions, `CPtrLD`/`CPtrST` and `DPtrST`/`DPtrST`, to handle different pointers. As `CALL` and `RET` instructions already exist in the vanilla (i.e., unmodified) program, they do not require any software modifications. Similarly, our code- and data-pointer load and store instructions simply replace regular loads and stores in the vanilla program. As the EPI metadata is accessed in parallel to the L1 data access, our special instructions does not introduce any latency at the hardware level that requires special treatment during the performance evaluation.

As EPI requires padding bytes to encode the type of data and function pointers as a mitigation against pointer confusion attacks, we modify the compiler front-end to insert two padding bytes per pointer to emulate the performance overheads of the extra memory utilization. furthermore, we insert a `MOV` instruction before pointer loads and stores to encode the pointer types in a dummy register to emulate the performance overheads of accessing the additional register

operand, `RegX` [1]. Additionally, we emulate the performance overheads of clearing the EPI metadata (i.e., the `ClearMeta` instruction) by inserting dummy `MOV` instructions that write a fixed value to memory every time (1) a heap object is deallocated or (2) a stack frame, which contains function/data pointer, is destroyed.

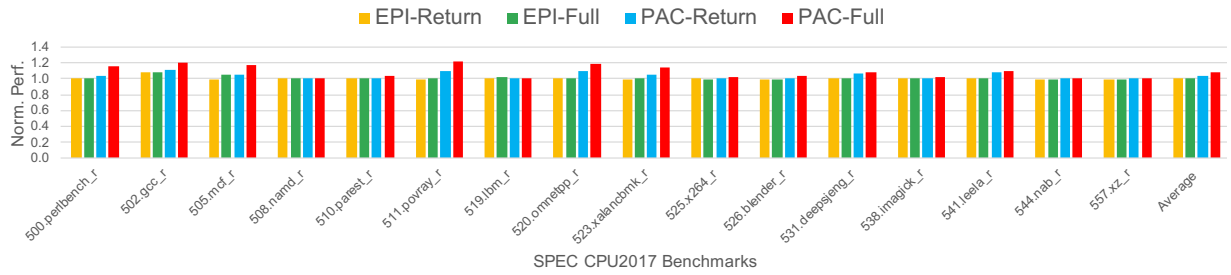


Figure 7.7: Performance overheads of the SPEC CPU2017 workloads for EPI and ARM's PAC normalized to baseline execution.

Results. The first two bars in Figure 7.7 show the runtime overheads of EPI-Return and EPI-Full normalized to baseline execution, respectively. EPI-Return provides return address integrity (i.e., backward-edge protection) without any per pointer padding bytes or additional operations while adding 0.47% performance overheads on average (with a maximum of 7% in case of `gcc_r`). On the other hand, EPI-Full represents our full pointer integrity protection, including return addresses, function pointers, and data pointers. Two padding bytes are inserted in this configuration as explained before. The results show that EPI-Full introduces 0.88% performance overheads on average with a maximum of 8%.

7.6.3 Comparison with ARM's PAC

Methodology. In addition to our EPI configurations, we evaluate a 32-bit variant of ARM's pointer authentication technique. As ARM's PAC is only available for 64-bit processors in certain Apple devices, we use the same emulation methodology adopted by prior work [47, 11] to estimate the performance overheads of ARM's PAC on a real machine. Specifically, we modify the compiler to emit four `XOR` instructions to account for the 4 cycle latency introduced by the PAC instructions.

[1]The extra register pressure, which may be introduced by `RegX` can be mitigated by proposing dedicated EPI physical registers that the compiler can use only for encoding pointer types.

Additionally, we insert two padding bytes per pointer to emulate the overheads of explicitly storing a 16-bit message authentication code (MAC) per each 32-bit pointer.

Results. The last two bars in Figure 7.7 show the runtime overheads of PAC-RET and PAC-Full normalized to baseline execution, respectively. PAC-RET emulates the overheads of signing and authenticating return addresses on the stack whereas PAC-Full emulates the overheads of applying ARM's PAC to its full-extent (i.e., protecting return addresses, function pointers, and data pointers). Our experimental results show that PAC-RET and PAC-Full introduce an average of 4% (with a maximum of 11%) and 8.5% (with a maximum of 21%) runtime overheads compared to baseline execution, respectively. The above results show that using cryptographic-based solutions introduces non-negligible performance overheads (in addition to a high energy budget), making them unsuitable for embedded environments.

7.6.4 Hardware Overheads

EPI requires minor changes to the processor and data caches. Qualitatively, the area overhead of EPI's L1 metadata is 6.25% as we add 2 bits per every four byte in the cache line. As the metadata lookup happens in parallel to the L1 data and tag accesses, EPI should have no impact on the L1 access latency. We use CACTI [153] to validate this hypothesis. By using 8-way associate caches, we measure the access time difference between a 34KB cache (with a 68B cache line) and a 32KB cache (with a 64B cache line). We compare with the access latency of the larger (34KB) cache to provide a fair comparison for two caches that provide the same amount of data storage. For these measurements we assume normal cache access mode without the late way select optimization and thus this estimate is conservative in terms of access times. The access time difference at 22nm is 0.00196ns (0.18% additional time). At the level of detail modeled by CACTI these differences are well within the modeling error range, and as such conclude that both caches can be accessed in the same amount of time. The dynamic read and write energies increase by 0.1% and 0.26%, respectively.

For lower level caches (i.e., L2 and L3), EPI adds minimal area overhead (2-bits per 64B cache

lines or 0.39%). The metadata encoding and decoding modules are used at the L1-L2 interface to change the cache line layout during the typical cache line spill and fill operations. As the spill operation is not on the processor critical path, adding extra logic—for encoding the metadata—to cache lines evictions will not impact the execution time of the applications. On the other hand, the metadata decoding module uses simple combinational logic and thus can be folded completely within the pipeline stages without impacting the cache line fill operation.

7.7 Summary

In this chapter I illustrated, EPI, a hardware-based technique that can protect embedded systems from a wide variety of code reuse and data-oriented programming attacks, at negligible runtime and hardware costs. Specifically, EPI enforces pointer integrity using minor changes to the processor logic, 6.25% area overheads in the L1 data cache, and two bits per 64-bytes cache lines in the L2 caches and main memory. While state-of-the-art commercial solutions for 64-bit architectures rely on cryptographic operations (e.g., ARM's PAC) or disjoint storage (e.g., Intel's CET shadow stacks) for mitigating memory safety-based attacks, EPI achieves better security guarantees on the more constrained 32-bit architectures without dedicating a performance or energy budget to cryptographic co-processors or disjoint stacks. Our evaluation results show that EPI has 0.88% runtime overheads on the SPEC CPU2017 benchmarks while having negligible latency and energy overheads.

Part V

Comparison With Prior Work On Architectural Support For Memory Safety

Chapter 8: Related Work

This chapter describes how the techniques presented in this thesis are unique compared to prior work in the area of memory safety error detection and exploit mitigation.

8.1 Memory Safety Error Detection

Hardware support for detecting memory safety errors can be broadly categorized into the following three classes: blocklisting, memory tagging, and permitlisting. Each of these classes requires a different set of metadata that are either stored per allocation or per pointer. As shown in Table 8.1, the metadata can be maintained in one of the following forms: disjoint, inlined, co-joined, or implicit. In this section we briefly describe the categories of memory safety error detectors and show how Califorms (Chapter 3) and No-FAT (Chapter 4) outperform the state-of-the-art solutions. Table 8.2 compares Califorms and No-FAT with prior techniques on memory safety error detection. The comparison points include the security guarantees, binary compatibility with unprotected code, multi-threading support, hardware complexity, memory requirements, and performance overheads. Since prior work is not easy to reproduce (due to a plethora of software and hardware platforms), we opt to include the main sources of memory and performance overheads instead of including the reported overheads in each paper.

8.1.1 Memory Blocklisting

This class of memory safety defenses (also known as tripwires) aims to detect overflows by marking the memory regions on either side of an allocation, and flagging accesses to them. For example, SafeMem [24] implements tripwires by repurposing ECC bits in memory to mark memory regions invalid, thus trading off reliability for security. On processors supporting speculative

Table 8.1: Categorization of prior work on spatial memory safety based on how they handle the security metadata.

	Permitlisting		**Blocklisting**
	Per-Allocation	**Per-Pointer**	
Disjoint Metadata	Baggy Bounds [75]	Hardbound [26] Softbound [154] Watchdog [27] Intel's MPX [38] AOS [32] CHEx86 [31]	ASan [3]
Inlined Metadata	EffectiveSan [37] In-Fat [155] **C-5** [8]	CHERI [29, 35] Compact-Ptrs [28] Intel's C^3 [44]	SafeMem [24] REST [30] **Califorms** [6]
Co-joined Metadata	ARM's MTE [5] SPARC's ADI [4]		
Implicit Metadata	Native-Ptrs [156, 76] **No-FAT** [7]		

execution, however, it might be possible to speculatively fetch blocklisted lines into the cache without triggering a faulty memory exception. Unless these lines are flushed immediately after, SafeMem's blocklisting feature can be trivially bypassed. Alternatively, REST [30] achieves the same by storing a predetermined large random number, in the form of a 8–64B token, in the memory to be blocklisted. Violations are detected by comparing cache lines with the token when they are fetched. Compatibility with unprotected modules is easily achieved as well, since tokens are part of the program's address space and all access are implicitly checked. However, intra-object spatial memory safety is not supported by REST owing to fragmentation overhead such heavy usage of tokens would entail.

In order to provide temporal memory safety, blocklisting-based solutions use memory quarantining. Freed memory region is moved to a quarantine pool, such that this region will not be allocated by `malloc` any time soon. Using memory quarantining typically increases performance overheads as it prevents the program from reusing recently freed memory to satisfy new allocation requests

Table 8.2: Comparison with prior work on memory safety error detection.

Proposal	Spatial Protection [*] Inter	Spatial Protection [*] Intra	Temp. Prot. [§]	Binary Comp. [†]	MT Support [¶]	Hardware Modifications	Metadata Overhead	Memory Overhead	Performance Overhead
Hardbound [26]	●	●[‡]	○	○	◒	μop injection, L1$ & TLB for tags	0-2 words per ptr & 4bit per word	∝ # of ptrs	∝ # of ptr derefs
Baggy Bounds [75]	●	○	○	●	●	N/A	N/A.	∝ padding objects to the nearest size	∝ # of ptr ops
Compact-Ptrs [28]	●	○	○	○	●	One extra pipeline stage for bounds check & update	N/A	∝ padding objects to the nearest size	∝ # of ptr ops
Watchdog [27]	●	◒[‡]	●	○	◒	Renaming logic, μop injection logic and Lock location$	4 words per ptr	∝ # of ptrs and allocs	∝ # of ptr derefs
WatchdogLite [157]	●	◒[‡]	●	○	◒	N/A	4 words per ptr	∝ # of ptrs and allocs	∝ # of ptr ops
Native-Ptrs [156, 76]	●	○	○	●	●	N/A	N/A.	∝ padding objects to the nearest size	∝ # of ptr ops
Intel's MPX [38]	●	◒[‡]	○	◒	◒	Unknown (closed platform)	2 words per ptr	∝ # of ptrs	∝ # of ptr derefs
BOGO [158]	●	◒[‡]	●	◒	◒	Unknown (closed platform)	2 words per ptr	∝ # of ptrs	∝ # of ptr derefs
CHERI [29, 35]	●	◒[‡]	○	○	●	Capability coprocessor, Tag$ and Capability Unit	Ptr size is 2-4X	∝ # of ptrs	∝ # of ptr ops
CHERIvoke [159]	○	○	●	○	●	Capability coprocessor, Tag$ Tag controller, and Capability Unit	Ptr size is 2-4X	∝ # of ptrs	∝ # of ptr ops
PUMP [160]	●	○	●	◒	●	Extend all data units by tag width, new miss handler and Rule$	8B per cache line	∝ prog. mem. footprint	∝ # of ptr ops
ARM's MTE [5]	●	○	◒	●	●	Unknown (closed platform)	4bit per 16B objects	∝ prog. mem. footprint	∝ # of tag (un)set ops
REST [30]	◒	○	◒	●	●	1-8B per L1D line, 1 comparator	8-64B token	∝ blacklisted memory	∝ # of (dis)arm insns.
AOS [32]	●	○	◒	◒	◒	ARM's PAC instructions, memory check queue, bounds$, and bounds way buffer	8B bounds per ptr	∝ # of ptrs	∝ # of ptr derefs
CHEx86 [31]	●	◒	◒	●	◒	μop injection logic, 256-entry Alias$ Capability$, and Speculative Pointer Tracker	2 words per ptr	∝ # of allocs & ptrs	∝ # of ptr derefs
Califorms [6]	◒	◒	◒	●	●	8B per L1D line, 1bit per L2/L3 line	1-7B per critical field	∝ blacklisted memory	∝ # of BLOC insns.
No-FAT [7]	●	●	●	◒	●	bounds checking module, and base address register file	1KB per process Table	∝ padding objects to the nearest size	∝ # of ptr derefs
C-5 [8]	●	●	●	◒	●	bounds checking module, QARMA cipher at L1/L2, and extended register file	1KB per process Table and 1 word per ptr	∝ padding objects and # of ptrs	∝ # of ptr derefs

[*] ● - Complete (Linear and non-linear overflows); ◒ - Linear only; ○ - No protection.
[§] ● - Complete; ◒ - Partial protection (until `realloc`); ○ - No protection.
[†] ● - Fully compatible; ◒ - Execution compatible, but protection dropped when external modules modify pointer; ○ - No support.
[¶] ● - Supported (stateless); ◒ - Supported (requires synchronization on global metadata); ○ - No support.
[‡] Achieved with bounds narrowing.

As a blocklisting-based technique, the primary advantages of Califorms are: (1) being faster than disjoint metadata based systems as our metadata resides with program data and does not require explicit propagation, (2) having lower performance and energy overheads since it neither requires multiple memory accesses, nor does it incur any significant checking costs, (3) providing fine-grained protection at the byte granularity, which is necessary for intra-object memory safety, and (4) being architecture width agnostic (i.e., not limited to 64-bit architectures) making it better suited for deployment over a more diverse device environment.

8.1.2 Memory Tagging

This class of techniques associates a "color" with newly allocated memory, and stores the same color in the upper bits of the data pointer that is used to access the allocated memory. At runtime, the hardware enforces spatial memory safety by comparing the colors of the pointer and accessed memory. For example, SPARC's ADI [4] assigns 4-bit colors to every 64B of memory (i.e., limiting the minimum allocation size to 64B), while ARM's MTE [5] uses 4-bit colors per every 16B of memory [34]. Since metadata bits are acquired along with the corresponding data, no extra memory operations are needed.

Temporal safety is enforced by assigning a different color when memory regions are reused. The number of tag bits in memory tagging defenses is limited as the tags are used for pointers and memory locations. As a result, prior techniques offer less entropy for temporal protection. For example, in SPARC's ADI colors are repeated every 15 allocations, raising the attacker's chances of bypassing the defense.

8.1.3 Memory Permitlisting

This class of memory safety defenses (also known as base & bounds) enforces spatial memory safety by verifying memory accesses against allocation bounds. The bounds information can be explicitly stored or implicitly derived.

Explicit Base & Bounds. This class of memory safety defenses attaches bounds metadata to every

pointer or allocation. The metadata can be stored in a shadow (i.e., disjoint) memory region (e.g., Hardbound [26], Intel's MPX [38], CHEx86 [31], and AOS [32]) or be marshaled with the pointer by extending its size (e.g., CHERI [29]). Temporal memory safety can be added to the above techniques by either storing an additional "identifier" along with the pointer metadata and verifying that no stale identifiers are ever retrieved (e.g., CETS [127], Watchdog [27], and WatchdogLite [157]) or invalidating all pointers to freed regions in the lookup tables (e.g., BOGO [158]).

While explicit base and bounds systems offer strong security guarantees, they introduce other complexities. For example, disjointly storing the metadata in a shadow memory [26, 38, 32, 31] requires extra memory accesses to fetch and update the metadata and introduces atomicity problems for multithreading applications. On the other hand, increasing the pointer width to include the metadata [29, 35] changes object layouts and breaks compatibility with the rest of the system (e.g., unprotected libraries). On the contrary, No-FAT performs simple arithmetic computations to derive the allocation bounds and uses a fixed area cost for `MAST`. Furthermore, the metadata-less aspect of No-FAT allows it to support multi-threading applications with no false positives/negatives, which occur in disjoint metadata schemes (e.g., Intel's MPX [38]). Additionally, the No-FAT's Buf2Ptr transformation implicitly resolves the intra-allocation memory safety problem, which is overlooked by recent memory safety techniques [31, 32].

Software-based Implicit Base & Bounds. This class of memory permitlisting solutions avoids the cost of maintaining the base and bounds information per each allocation or pointer. Instead, they derive allocation bounds from the pointer itself. For example, guarded pointers divided memory into powers-of-two segments and encoded the segment size into the pointer's upper bits [161]. Similarly, baggy bounds [75] restricts allocation sizes to powers-of-two and encodes the binary logarithm of the allocation size in the pointer's upper bits. Unlike No-FAT, this design choice significantly increases the program's memory footprint due to padding allocations to the nearest powers-of-two size. Moreover, neither guarded pointers nor baggy bounds offers temporal memory safety protection.

Compact-pointers [28] tried to avoid the powers-of-two restriction by using a floating-point

representation to encode allocation bounds in the pointer itself. CHERI-concentrate [35] adopts a similar approach to compress metadata to 128 bits (instead of 256 bits) by changing a pointer's layout and introducing instructions to manipulate them. Due to the pointer layout manipulation, both techniques neither support temporal memory safety nor maintain binary compatibility.

Similar to No-FAT's binning allocator, Native-Pointers [156, 41, 76] divides the program's virtual address space into several regions of equal size and uses each region to allocate objects of similar non powers-of-two sizes. As a software-only solution, Native-Pointers suffers from high performance overheads. Additionally, Native-Pointers does not naturally provide temporal protection. A follow-up work (EffectiveSan [37]) adds temporal protection (and intra-allocation memory safety) to Native-Pointers but with expensive per-allocation metadata. Concurrent to my work, Xu et al. add hardware support for EffectiveSan, dubbed In-Fat [155]. The key idea is to maintain a per-allocation metadata table and use the pointer's upper bits to index into this table for intra-allocation bounds retrieval. In-Fat uses different metadata schemes for different program objects (e.g., stack, heap, and globals) to reduce the lookup overhead. Unlike No-FAT, In-Fat does not provide temporal protection as it utilizes the pointer's upper bits for indexing into the metadata tables. A key advantage of No-FAT over EffectiveSan and In-Fat is that it does not require any per pointer/allocation metadata. Thus, it runs with almost native performance, making it best suited to be an always-on memory safety mitigation.

To summarize, as a memory permitlisting technique, the main advantages of No-FAT are: (1) having simpler hardware design as it does not require any changes to the memory subsystem, (2) being faster than prior base & bounds systems as it uses no metadata for spatial/temporal memory safety, (3) supporting multi-threading applications due to its metadata-less nature, and (4) being able to catch both adjacent and non-adjacent spatial memory violations. C-5 further reduced the performance costs of No-FAT and expands its security benefits to include resiliency against physical attacks.

8.2 Memory Safety Exploit Mitigation

Instead of enforcing the full memory safety rules, exploitation prevention techniques aim at enforcing relaxed security rules in order to prevent the attacker from compromising the system while keeping the associated overheads low. We categorize exploit mitigations into two classes: shadow stack-based systems and encryption-based solutions. Table 8.3 compares ZeRØ (Chapter 6) and EPI (Chapter 7) versus prior techniques on memory safety exploit mitigation. Unlike error detectors, which catch memory safety violations when they occur, exploit mitigation techniques prevent the second attack phase, which is overwriting the program assets. Thus, the key comparison points in Table 8.3 are the assets protected by each system, main operation, hardware complexity, memory requirements, and performance/energy overheads.

8.2.1 Shadow Stack-Based Techniques

A straightforward solution to guarantee the integrity of return addresses is to adopt a shadow call stack [165]. Every time a `CALL` instruction is executed, the return address is pushed to the regular stack and an additional memory instruction stores a copy of the return address to the shadow stack. When a function returns, the original return address is restored from the stack and compared against the shadow return address. If an attacker manipulates the return address while stored on the stack, a mismatch occurs as the shadow stack is not accessible by the attacker. For example, Intel's Control-flow Enforcement Technology (CET) [9] makes its shadow stack inaccessible to program loads and stores, while CFI CaRE protects the shadow stack using ARM's TrustZone-M security extensions [162]. Similar to ZeRØ's return address integrity, shadow stacks can be applied to legacy binaries with no compiler modifications. However, shadow stacks add an extra memory access operation for every function call and return, increasing energy and memory overheads.

Unlike return addresses, code pointers are not accessed in pairs of `CALL`/`RET` instructions. As a result, shadow stack-based defenses require an additional component to protect code pointers (also known as forward-edge transitions). For example, Intel's CET adds a new `ENDBRANCH`

Table 8.3: Comparison with prior work on memory safety exploit mitigation. The assets protected by each system is listed: **RET** stands for return address integrity/protection, **CPtr** stands for function pointer integrity, **DPtr** stands for data pointer integrity, and **Data** stands for non-pointer data integrity.

Proposal	Protected Assets				Main Operations	Hardware Changes	Metadata Overhead	Performance Overhead	Energy Overhead
	RET	CPtr	DPtr	Data					
Intel's CET [9]	✓	✗	✗	✗	Memory access per CALL/RET and EndBranch Check	Isolated shadow stack and indirect branch FSM tracker	1 word per return address	low	high
CFI CaRE [162]	✓	✓	✗	✗	Memory access per CALL/RET and Branch verification	TrustZone-protected shadow stack	1 word per return address	high	high
CPI [163]	✓	✓	✗	✗	Extra memory accesses per every sensitive pointer access	No changes	4 words per sensitive pointer	moderate	moderate
Branch Regulation [164]	✓	✓	✗	✗	Memory access per CALL/RET and Branch verification	Isolated Secure Call Stack and a function bounds cache	3 words per stack frame	low	high
CCFI [46]	✓	✓	✗	✗	Pointer signing/authentication	AES co-processor	ptr-inlined	high	high
ARM's PAC [10, 47]	✓	✓	✓	✗	Pointer signing/authentication	QARMA co-processor	ptr-inlined	moderate	high
HDFI [140]	✓	✓	✗	✗	Tag check	1B per L1D line, Tag$, and DFITagger unit	1-bit per word	moderate	moderate
Morpheus [141]	✓	✓	✓	✗	Pointer encryption/decryption and pointer displacement	QARMA co-processor, Tag$, and churn unit	2-bit per word	low	high
ZeRØ [11]	✓	✓	✓	✗	Tag check	2B per L1D line, 1bit per L2 line	ptr-inlined	none	low
EPI [12]	✓	✓	✓	✗	Tag check	2B per L1D line, 1bit per L2 line	ptr-inlined	low	low

instruction, which is placed at the entry of each basic block that can be invoked via an indirect branch. When an indirect forward branch occurs, the following instruction is expected to be an `ENDBRANCH`, otherwise an attack is assumed. On the other hand, CFI CaRE instruments binaries in a manner which removes all function calls and indirect branches and replaces them with dispatch instructions that trap control flow to a branch monitor. The branch monitor verifies the control-flow transition by comparing it against a pre-determined (i.e., compile time) control flow graph (CFG) of the program. Trapping into the branch monitor for every indirect call causes CFI CaRE's performance overheads to range between 13% and 513%. More importantly, techniques that rely on static analysis to construct a CFG and enforce it at runtime are ultimately limited by the precision of the analysis [144]. ZeRØ's simple instruction set extensions implicitly protect forward edge transitions by guaranteeing code pointer integrity with zero cost.

Code Pointer Integrity (CPI) [163] and its relaxed variant (Code Pointer Separation) use compiler analysis and instrumentation to isolate code pointers into a separate region of memory. The idea is similar to the concept of shadow stacks, but extends it to include code pointers in globals and heap objects. Unlike ZeRØ's inlined metadata, CPI requires extra memory accesses per every sensitive pointer access to fetch its corresponding metadata. Moreover, prior work showed that CPI's safe region can be leaked and then maliciously modified by using data pointer overwrites, undermining the security guarantees of the solution [166].

8.2.2 Encryption-Based Techniques

To eliminate the memory costs associated with shadow stacks, prior work used encryption to randomize the pointer layout before storing it to memory. As long as the attackers have no access to the encryption key, they cannot reliably leak/overwrite the pointer. Early work used XOR-based encryption to avoid adding performance costs to every pointer load/store operation [167, 168]. As XOR-based encryption is vulnerable to known plaintext attacks, modern work utilizes strong encryption, such as AES in cryptographic control-flow integrity (CCFI) [46] and QARMA ciphers in ARM's PAC [10, 47] and Morpheus [141].As shown in Section 6.7, ZeRØ completely eliminates

the runtime overheads associated with ARM's PAC for code and data pointer protections.

Another example of an encryption-based defense is Morpheus [141], an architecture that (i) displaces code and data pointers in the address space (ii) diversifies the representation of code and pointers using strong encryption, and (iii) periodically repeats the above steps using a different displacement and key. Similar to ZeRØ, Morpheus does not protect non-pointer data corruption and provides low performance overheads. Unlike ZeRØ which is a secret-less solution, Morpheus must keep two parameters secret until they are changed: displacements for the code and data regions, and keys for encrypting/decrypting pointers. Additionally, a key limitation of encryption-based techniques is the additional energy costs per pointer operation. One AES operation can cost up to 48.02pJ/bit (or 3073.28pJ per 64-bit encryption) at 1 MHz while one QARMA operation costs 7.78pJ/bit (or 497.92pJ per 64-bit encryption) [169], which is at least an order of magnitude higher than ZeRØ's 2-bit metadata read and check in the L1 data cache (energy consumption of L1 data access ranges between 64 and 105pJ/Byte, i.e., 16 and 26.25pJ per two bits [170]).

In short, as an exploit mitigation technique, the key advantages of ZeRØ are: (1) requiring no extra memory accesses for verifying return address integrity (unlike the shadow stack-based techniques shown in the first half of Table 8.3), (2) eliminating the energy and runtime overheads associated with using encryption for protecting code/data pointers, and (3) having much lower performance and memory overheads compared to full memory safety techniques. EPI, on the other hand, enforces pointer integrity rules on 32-bit architectures by leveraging common software properties for harvesting different types of code- and data-pointer bits. Thus, EPI can be efficiently applied to embedded systems, which are currently dominated by 32-bit processors [49, 48].

Part VI

Conclusion

Chapter 9: Conclusion

Despite the massive efforts undertaken by commercial vendors and academic researchers in the past three decades, the memory unsafety problem in C and C++ remains unsolved. The first chapter of this thesis explained why memory safety is still a major concern nowadays. Reasons include (1) the immaturity of memory-safe languages compared to C/C++ in addition to the vast amount of legacy code, (2) the limited capabilities of pre-deployment testing, and (3) the impracticality of current hardware-assisted memory safety mechanisms. In order to overcome the limitations of prior techniques (e.g., having complex metadata, lacking binary compatibility, offering incomplete protection, and being vulnerable to side-channels), I took an entirely different approach. Instead of adding more features to a program in order to make it memory safe, this thesis explored *leveraging common software trends and existent program features and turning them into security primitives by rethinking computer microarchitectures*. This way it is possible to *efficiently circumvent the problems of traditional memory safety solutions for C and C++* without introducing performance or memory overheads.

This thesis advanced the state of the art in three different directions for mitigating memory safety violations—namely memory blocklisting, memory permitlisting, and exploit mitigation.

9.0.1 Advancing Memory Blocklisting

I presented Califorms (Chapter 3), a novel approach that uses dead spaces in program memory to store memory blocklisting metadata without increasing the program's memory footprint. Califorms changes the cache line layout to allow for encoding the blocklisting metadata within the program data itself at the byte granularity. The compressed layout of Califorms avoids the overheads associated with fetching the metadata from disjoint memory locations, significantly reducing the costs of enforcing fine-grained memory safety. The benefits of Califorms go well beyond

memory safety. For example, the in-place metadata can be used for marking program secrets and hence throttling speculative execution when a program secret is transiently accessed. This way, it is feasible to prevent the attackers from leaking a program's secrets without incurring the large overheads of traditional Spectre mitigations.

9.0.2 Advancing Memory Permitlisting

I proposed No-FAT (Chapter 4), a novel technique that leverages current technological trends in software development, namely the usage of binning memory allocators, to effectively enforce memory safety. No-FAT made the novel contribution of making the memory allocation size (e.g., malloc size) an architectural feature so that the hardware can implicitly derive allocations bounds information (i.e., the base address and size) from the memory address itself without relying on expensive metadata. This way No-FAT achieves several benefits, such as (1) providing an efficient always-on memory safety defense, (2) improving the fuzz testing bandwidth by up to 10x compared to ASan, and (3) improving resilience to Spectre-V1 attacks. My C-5 work (Chapter 5) further optimized No-FAT and enhanced its security guarantees by increasing its temporal safety entropy and integrating data encryption, making C-5 an ideal security solution for thwarting software and physical-based attacks. Furthermore, while this thesis has focused on the security benefits of No-FAT, the concept of exposing the allocation sizes to the hardware can enable performance benefits as well by enhancing the predictability of memory prefetchers and DRAM controllers.

9.0.3 Advancing Exploit Mitigation

I explained ZeRØ (Chapter 6), a novel hardware primitive that enforces pointer integrity with no runtime costs. ZeRØ exposes the different data types that are available during program compilation (e.g., regular data, code pointers, and data pointers) to the hardware to protect valuable program assets from traditional memory safety vulnerabilities. Furthermore, ZeRØ leverages the currently unused upper pointer bits on 64-bit architectures for encoding the necessary metadata. While the concept of encoding the security metadata by changing the cache line format was also

used in Califorms (Chapter 3), ZeRØ relied on a simpler encoding that reduces the complexity of the L1-to-L2 transformation modules. Moreover, Califorms' metadata was used to deny accesses to dead bytes whereas ZeRØ's metadata was used to enforce access control rules on neighboring data (i.e., the rest of bytes in the code/data pointer). As security is a full system property, it is mandatory to protect all system components, including wimpy devices and micro-controllers. Thus, to extend ZeRØ's protection to non-64-bit systems, I proposed EPI (Chapter 7), which offers the same level of exploit mitigation to resource constrained devices. EPI uses several optimizations to harvest unused bits on 32-bit RISC processors so that they can be repurposed for security.

Finally, I envision that the way to put an end to the memory safety problem is to (1) always use memory-safe languages for developing new software and (2) add the hardware extensions proposed in this thesis to future processors in order to secure legacy C and C++ code.

References

[1] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *S&P '13: Proceedings of the 2013 IEEE Symposium on Security and Privacy*, San Francisco, CA, USA, 2013, pp. 48–62.

[2] A. Jain, *CVE-2021-3156: Heap-based buffer overflow in Sudo (Baron Samedit)*, 2021. [Online]. Available: `https://blog.qualys.com/vulnerabilities-research/2021/01/26/cve-2021-3156-heap-based-buffer-overflow-in-sudo-baron-samedit`.

[3] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *ATC '12: Proceedings of the 2012 USENIX Annual Technical Conference*, 2012.

[4] Oracle, *Hardware-assisted checking using silicon secured memory (SSM)*, 2015. [Online]. Available: `https://docs.oracle.com/cd/E37069_01/html/E37085/gphwb.html`.

[5] ARM, *Memory tagging extension: Enhancing memory safety through architecture*, 2019. [Online]. Available: `https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/enhancing-memory-safety`.

[6] H. Sasaki, M. A. Arroyo, M. Tarek Ibn Ziad, K. Bhat, K. Sinha, and S. Sethumadhavan, "Practical byte-granular memory blacklisting using Califorms," in *MICRO'52: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, Columbus, OH, USA, 2019, 558—571.

[7] M. Tarek Ibn Ziad, M. A. Arroyo, E. Manzhosov, R. Piersma, and S. Sethumadhavan, "No-FAT: Architectural support for low overhead memory safety checks," in *ISCA-48: Proceedings of the 48th Annual International Symposium on Computer Architecture*, Worldwide Event, 2021, pp. 916–929.

[8] M. Tarek Ibn Ziad, E. Manzhosov, and S. Sethumadhavan, *C-4: Compromising cryptographic capability computing*, Currently under review.

[9] V. Shanbhogue, D. Gupta, and R. Sahita, "Security analysis of processor instruction set architecture for enforcing control-flow integrity," in *HASP '19: Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, Phoenix, AZ, USA, 2019.

[10] Qualcomm Technologies, *Pointer authentication on ARMv8.3*, 2017. [Online]. Available: https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf.

[11] M. Tarek Ibn Ziad, M. A. Arroyo, E. Manzhosov, and S. Sethumadhavan, "ZeRØ: Zero-overhead resilient operation under pointer integrity attacks," in *ISCA-48: Proceedings of the 48th Annual International Symposium on Computer Architecture*, Worldwide Event, 2021, pp. 999–1012.

[12] M. Tarek Ibn Ziad, M. A. Arroyo, E. Manzhosov, V. P. Kemerlis, and S. Sethumadhavan, "EPI: Efficient pointer integrity for securing embedded systems," in *SEED '21: Proceedings of the 2021 International Symposium on Secure and Private Execution Environment Design*, Worldwide Event, 2021, pp. 163–175.

[13] J. P. Anderson, *Computer security technology planning study*, 1972. [Online]. Available: http://seclab.cs.ucdavis.edu/projects/history/papers/ande72.pdf.

[14] M. Miller, *Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape*, https://github.com/microsoft/MSRC-Security-Research/tree/master/presentations/2019_02_BlueHatIL, [Online; accessed 15-April-2022], 2019.

[15] C. Cimpanu, *Chrome: 70% of all security bugs are memory safety issues*, 2020. [Online]. Available: https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues/.

[16] B. Hawkes, *0day in the wild*, https://googleprojectzero.blogspot.com/p/0day.html, [Online; accessed 15-April-2022], 2019.

[17] M. Böhme and B. Falk, "Fuzzing: On the exponential cost of vulnerability discovery," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 713–724.

[18] Mozilla Security, *VIRGO: Crowdsourced fuzzing cluster*, https://github.com/MozillaSecurity/virgo, [Online; accessed 15-April-2022], 2019.

[19] V. Tsyrklevich, *GWP-ASan: Sampling heap memory error detection in-the-wild*. [Online]. Available: https://www.chromium.org/Home/chromium-security/articles/gwp-asan.

[20] J. Metzman, *ClusterFuzzLite: Continuous fuzzing for all*, 2021. [Online]. Available: https://security.googleblog.com/2021/11/clusterfuzzlite-continuous-fuzzing-for.html.

[21] Microsoft, *OneFuzz: A self-hosted fuzzing-as-a-service platform*, 2020. [Online]. Available: https://www.microsoft.com/security/blog/2020/09/15/microsoft-onefuzz-framework-open-source-developer-tool-fix-bugs/.

[22] D. L. Schacter and E. Tulving, *Memory Systems 1994*. MIT Press, 1994.

[23] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," in *PLDI '94: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, Orlando, FL, USA, 1994, pp. 290–301.

[24] F. Qin, S. Lu, and Y. Zhou, "SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs," in *HPCA'05: Proceedings of the IEEE 11th International Symposium on High Performance Computer Architecture*, 2005.

[25] M. Kharbutli, X. Jiang, Y. Solihin, G. Venkataramani, and M. Prvulovic, "Comprehensively and efficiently protecting the heap," in *ASPLOS XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, USA, 2006, pp. 207–218.

[26] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "HardBound: Architectural support for spatial safety of the C programming language," in *ASPLOS XIII: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[27] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Watchdog: Hardware for safe and secure manual memory management and full memory safety," in *ISCA'12: Proceedings of the 39th International Symposium on Computer Architecture*, 2012.

[28] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight Jr, and A. DeHon, "Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," in *CCS'13: Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security*, 2013.

[29] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. H. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. M. Norton, M. Roe, S. D. Son, and M. Vadera, "CHERI: A hybrid capability-system architecture for scalable software compartmentalization," in *S&P '15: Proceedings of the 2015 IEEE Symposium on Security and Privacy*, San Jose, CA, USA, 2015, pp. 20–37.

[30] K. Sinha and S. Sethumadhavan, "Practical memory safety with REST," in *ISCA'18: Proceedings of the 45th Annual International Symposium on Computer Architecture*, Los Angeles, CA, USA, 2018, pp. 600–611.

[31] R. Sharifi and A. Venkat, "CHEx86: Context-sensitive enforcement of memory safety via microcode-enabled capabilities," in *ISCA '20: Proceedings of the 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture*, Valencia, Spain, 2020, pp. 762–775.

[32] Y. Kim, J. Lee, and H. Kim, "Hardware-based always-on heap memory safety," in *MICRO'53: Proceedings of the 53nd Annual IEEE/ACM International Symposium on Microarchitecture*, Global Online Event, 2020, pp. 1153–1166.

[33] Intel, *Intel® memory protection extensions enabling guide*, 2016.

[34] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov, *Memory tagging and how it improves C/C++ memory safety*, Feb. 2018. arXiv: 1802.09517v1 [cs.CR].

[35] J. Woodruff, A. Joannou, H. Xia, A. Fox, R. Norton, D. Chisnall, B. Davis, K. Gudka, N. W. Filardo, A. T. Markettos, M. Roe, P. G. Neumann, R. N. M. Watson, and S. Moore, "CHERI concentrate: Practical compressed capabilities," *IEEE Transactions on Computers*, vol. 68, no. 10, pp. 1455–1469, Oct. 2019.

[36] J. Bialek, K. Johnson, M. Miller, and T. Chen, *Security analysis of memory tagging*, 2020. [Online]. Available: https://github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/Security%20analysis%20of%20memory%20tagging.pdf.

[37] G. J. Duck and R. H. C. Yap, "EffectiveSan: Type and memory error detection using dynamically typed C/C++," in *PLDI '18: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018.

[38] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel mpx explained: A cross-layer analysis of the intel mpx system stack," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 2, no. 2, 28:1–28:30, 2018.

[39] S. Ghemawat and P. Menage, *TCMalloc: Thread-caching malloc*, 2007. [Online]. Available: http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[40] J. Evans, "A scalable concurrent malloc(3) implementation for FreeBSD," in *Proceedings of the Technical BSD Conferene*, 2006. [Online]. Available: https://www.bsdcan.org/2006/papers/jemalloc.pdf.

[41] G. J. Duck and R. H. C. Yap, *An extended low fat allocator api and applications*, 2018. arXiv: 1804.04812 [cs.PL].

[42] D. Leijen, B. Zorn, and L. de Moura, "Mimalloc: Free list sharding in action," Microsoft, Tech. Rep. MSR-TR-2019-18, 2019. [Online]. Available: https://www.microsoft.

com/en-us/research/publication/mimalloc-free-list-sharding-in-action/.

[43] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *S&P'19: Proceedings of the 40th IEEE Symposium on Security and Privacy*, 2019.

[44] M. LeMay, J. Rakshit, S. Deutsch, D. M. Durham, S. Ghosh, A. Nori, J. Gaur, A. Weiler, S. Sultana, K. Grewal, and S. Subramoney, "Cryptographic capability computing," in *MICRO-54: Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*, Virtual Event, Greece, 2021, pp. 253–267.

[45] I. Sysoev, *Nginx*, http://nginx.org/en/, 2004.

[46] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "CCFI: Cryptographically enforced control flow integrity," in *CCS '15: Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, Denver, Colorado, USA, 2015, pp. 941–951.

[47] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan, "PAC it up: Towards pointer integrity using ARM pointer authentication," in *SEC '19: Proceedings of the 28th USENIX Security Symposium*, Santa Clara, CA, USA, Aug. 2019, pp. 177–194.

[48] AspenCore, *Embedded markets study: Integrating iot and advanced technology designs, application development & processing environments*, 2019. [Online]. Available: https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf.

[49] P. Clarke, *MCU market turns to 32-bits and ARM*, 2013. [Online]. Available: https://www.eetimes.com/mcu-market-turns-to-32-bits-and-arm/.

[50] D. Lea, *A memory allocator*, 2000. [Online]. Available: http://gee.cs.oswego.edu/dl/html/malloc.html.

[51] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "SoK: Sanitizing for security," in *S&P'19: Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[52] U. Drepper, *Security enhancements in redhat enterprise Linux (beside SELinux)*, 2005. [Online]. Available: https://akkadia.org/drepper/nonselsec.pdf.

[53] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *CCS '07: Proceedings of the 14th ACM Conference on Computer and Communications Security*, Alexandria, Virginia, USA, 2007, pp. 552–561.

[54] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to RISC," in *CCS '08: Proceedings of the 15th ACM Conference on Computer and Communications Security*, Alexandria, Virginia, USA, 2008, pp. 27–38.

[55] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *S&P'14: Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014, pp. 575–589.

[56] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *ASIACCS '11: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, Hong Kong, China, 2011, pp. 30–40.

[57] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *S&P'16: Proceedings of the 2016 IEEE Symposium on Security and Privacy*, San Jose, CA, USA, 2016, pp. 969–986.

[58] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks," in *CCS '18: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, Toronto, Canada, 2018, pp. 1868–1882.

[59] J. Pewny, P. Koppe, and T. Holz, "STEROIDS for DOPed applications: A compiler for automated data-oriented programming," in *EuroS&P '19: Proceedings of the 2019 IEEE European Symposium on Security and Privacy*, Stockholm, Sweden, 2019.

[60] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *SSYM '05: Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, Baltimore, MD, USA, 2005.

[61] D. Weston and M. Miller, *Windows 10 mitigation improvements*, Black Hat USA, 2016.

[62] A. Milburn, H. Bos, and C. Giuffrida, "SafeInit: Comprehensive and practical mitigation of uninitialized read vulnerabilities," in *NDSS '17: Proceedings of the 24th Annual Network and Distributed System Security Symposium*, 2017.

[63] K. Cook, *Introduce struct layout randomization plugin*, `https://lkml.org/lkml/2017/5/26/558`, [Online; accessed 15-April-2022], May 2017.

[64] N. Hussein, *Randomizing structure layout*, `https://lwn.net/Articles/722293/`, [Online; accessed 15-April-2022], May 2017.

[65] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *S&P '14: Proceedings of the 2014 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2014, pp. 227–242.

[66] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *ISCA '13: Proceedings of the 40th International Symposium on Computer Architecture*, 2013.

[67] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large Intel® Itanium® programs with dynamic instrumentation," 2004.

[68] L. Eeckhout, *Computer architecture performance evaluation methods*, 1st. 2010.

[69] L. K. John, "More on finding a single number to indicate overall performance of a benchmark suite," *ACM SIGARCH Computer Architecture News*, vol. 32, no. 1, pp. 3–8, Mar. 2004.

[70] M. Aizatsky, K. Serebryany, O. Chang, A. Arya, and M. Whittaker, *Announcing OSS-Fuzz: Continuous fuzzing for open source software*, 2016. [Online]. Available: `https://security.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html`.

[71] N. Vijaykumar, A. Jain, D. Majumdar, K. Hsieh, G. Pekhimenko, E. Ebrahimi, N. Hajinazar, P. B. Gibbons, and O. Mutlu, "A case for richer cross-layer abstractions: Bridging the semantic gap with expressive memory," in *ISCA '18: Proceedings of the 45th Annual International Symposium on Computer Architecture*, Los Angeles, CA, USA, 2018, pp. 207–220.

[72] R. Hundt, S. Mannarswamy, and D. Chakrabarti, "Practical structure layout optimization and advice," in *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, New York, NY, USA, 2006, pp. 233–244.

[73] P. Roy and X. Liu, "StructSlim: A lightweight profiler to guide structure splitting," in *CGO '16: Proceedings of the 2016 International Symposium on Code Generation and Optimization*, Barcelona, Spain, 2016, 36–46.

[74] C. Yu, P. Roy, Y. Bai, H. Yang, and X. Liu, "LWPTool: A lightweight profiler to guide data layout optimization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 11, pp. 2489–2502, 2018.

[75] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors," in *SEC '09: Proceedings of the 18th USENIX Security Symposium*, 2009.

[76] G. J. Duck, R. H. C. Yap, and L. Cavallaro, "Stack bounds protection with low fat pointers," in *24th Annual Network and Distributed System Security Symposium, NDSS*, San Diego, CA, USA, 2017.

[77] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *ISCA '14: Proceeding of the 41st Annual International Symposium on Computer Architecuture*, Minneapolis, Minnesota, USA: IEEE Press, 2014, 361–372.

[78] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *SEC '14: Proceedings of the 23rd USENIX Conference on Security Symposium*, San Diego, CA, USA, 2014, pp. 719–732.

[79] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *CCS '10: Proceedings of the 17th ACM Conference on Computer and Communications Security*, Chicago, Illinois, USA, 2010, pp. 559–572.

[80] L. Cheng, H. Liljestrand, M. S. Ahmed, T. Nyman, T. Jaeger, N. Asokan, and D. Yao, "Exploitation techniques and defenses for data-oriented attacks," in *SecDev'19: Proceedings of the 2019 IEEE Cybersecurity Development*, Tysons Corner, VA, USA, 2019, pp. 114–128.

[81] C. Carruth, *RFC: Speculative load hardening (a spectre variant 1 mitigation)*, 2018. [Online]. Available: `https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html`.

[82] O. Oleksenko, B. Trach, T. Reiher, M. Silberstein, and C. Fetzer, *You shall not bypass: Employing data dependencies to prevent bounds check bypass*, 2018. arXiv: `1805.08506 [cs.CR]`.

[83] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative taint tracking (STT): A comprehensive protection for speculatively accessed data," in *MICRO-52: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, Columbus, OH, USA, 2019, pp. 954–968.

[84] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, "Isolating speculative data to prevent transient execution attacks," *IEEE Computer Architecture Letters*, vol. 18, no. 2, pp. 178–181, 2019.

[85] X. Ren, L. Moody, M. Taram, M. Jordan, D. M. Tullsen, and A. Venkat, "I see dead μops: Leaking secrets via Intel/AMD micro-op caches," in *ISCA-48: Proceedings of the 48th Annual International Symposium on Computer Architecture*, Worldwide Event, 2021.

[86] E. M. Koruyeh, S. H. A. Shirazi, K. N. Khasawneh, C. Song, and N. B. Abu-Ghazaleh, "SpecCFI: Mitigating spectre attacks using CFI informed speculation," in *S&P '20: Proceedings of the IEEE Symposium on Security and Privacy*, San Francisco, CA, USA, 2020, pp. 39–53.

[87] D. Chisnall, C. Rothwell, R. N. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann, "Beyond the PDP-11: Architectural support for a memory-safe c abstract machine," in *ASPLOS '15: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, Istanbul, Turkey, 2015, pp. 117–130.

[88] M. R. Guthaus, J. E. Stine, S. Ataei, B. Chen, B. Wu, and M. Sarwar, "OpenRAM: An open-source memory compiler," in *ICCAD '16: Proceedings of the 35th International Conference on Computer-Aided Design*, Austin, TX, USA, 2016.

[89] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis transformation," in *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, San Jose, CA, USA, 2004, pp. 75–86.

[90] J. Sukumaran, *Syrupy*, `https://github.com/jeetsukumaran/Syrupy`, [Online; accessed 15-April-2022], 2008.

[91] LLVM, *Scudo hardened allocator*. [Online]. Available: `https://llvm.org/docs/ScudoHardenedAllocator.html`.

[92] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *SEC '15: Proceedings of the 24th USENIX Security Symposium*, Washington, D.C., USA, Aug. 2015, pp. 897–912.

[93] M. Hähnel, W. Cui, and M. Peinado, "High-resolution side channels for untrusted operating systems," in *ATC '17: Proceedings of the 2017 USENIX Annual Technical Conference*, Santa Clara, CA, USA, Jul. 2017, pp. 299–312.

[94] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX," in *CCS '17: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, Dallas, Texas, USA, 2017, 2421–2434.

[95] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: Cold-boot attacks on encryption keys," *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[96] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[97] M. Kounavis, S. Deutsch, S. Ghosh, and D. Durham, "K-Cipher: A low latency, bit length parameterizable cipher," in *ISCC '20: Proceedings of the 2020 IEEE Symposium on Computers and Communications*, Rennes, France, 2020, pp. 1–7.

[98] A. Tang, S. Sethumadhavan, and S. Stolfo, "CLKSCREW: Exposing the perils of security-oblivious energy management," in *SEC '17: Proceedings of the 26th USENIX Conference on Security Symposium*, Vancouver, BC, Canada, 2017, pp. 1057–1074.

[99] A. N. Sovarel, D. Evans, and N. Paul, "Where is the FEEB? the effectiveness of instruction set randomization," in *SEC '05: Proceedings of the 14th USENIX Security Symposium*, Baltimore, MD, USA, Jul. 2005.

[100] Y. Weiss and E. G. Barrantes, "Known/chosen key attacks against software instruction set randomization," in *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference*, Miami Beach, FL, USA, 2006, pp. 349–360.

[101] *CVE-2020-18771*, 2020. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-18771.

[102] *CVE-2020-24977*, 2020. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-24977.

[103] *CVE-2019-13222*, 2019. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-13222.

[104] Common Weakness Enumeration, *CWE-134: Use of externally-controlled format string*. [Online]. Available: https://cwe.mitre.org/data/definitions/134.html.

[105] *CVE-2012-0809*, 2012. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0809.

[106] *CVE-2016-4071*, 2016. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-4071.

[107] *CVE-2020-29018*, 2018. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-29018.

[108] *CVE-2021-30145*, 2021. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-30145.

[109] *CVE-2021-20307*, 2021. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-20307.

[110] M. Y. Mo, *Triggering garbage collection with rejected promises to cause use-after-free in chrome*, https://securitylab.github.com/research/garbage-collection-uaf-chrome_gc/, [Online; accessed 15-April-2022], 2020.

[111] E. Cao, *CVE-2020-17053: Use-after-free IE vulnerability*, https://www.trendmicro.com/en_us/research/20/k/cve-2020-17053-use-after-free-ie-vulnerability.html, [Online; accessed 15-April-2022], 2020.

[112] J. Horn, *How a simple linux kernel memory corruption bug can lead to complete system compromise*, https://googleprojectzero.blogspot.com/2021/10/how-simple-linux-kernel-memory.html, [Online; accessed 15-April-2022], 2021.

[113] S. Martinez, *Analysis of a use-after-free vulnerability in adobe acrobat reader DC*, https://blog.exodusintel.com/2021/04/20/analysis-of-a-use-after-free-vulnerability-in-adobe-acrobat-reader-dc/, [Online; accessed 15-April-2022], 2021.

[114] M. Talbi and P. Fariello, *VM escape: QEMU case study*, Phrack Inc., Issue #70, 2021. [Online]. Available: http://www.phrack.org/issues/70/5.html#article.

[115] C. Evans, *What is a good memory corruption vulnerability?* 2015. [Online]. Available: https://googleprojectzero.blogspot.com/2015/06/what-is-good-memory-corruption.html.

[116] *CVE-2018-4192*, 2018. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-4192.

[117] Apple, *WebKit*. [Online]. Available: https://github.com/WebKit/webkit.

[118] M. Gaasedelen, *Timeless debugging of complex software: Root cause analysis of a non-deterministic JavaScriptCore bug*, 2018. [Online]. Available: https://blog.ret2.io/2018/06/19/pwn2own-2018-root-cause-analysis/.

[119] K. Miller, *Race condition in arrayProtoFuncReverse() causes wrong results or crash*, 2018. [Online]. Available: https://github.com/WebKit/WebKit/commit/ba892e5725f4842a0bd50a53b6cc23d82b1fb783.

[120] A. Burnett, P. Biernat, and M. Gaasedelen, *Weaponization of a JavaScriptCore vulnerability: Illustrating the progression of advanced exploit primitives in practice*, 2018. [Online]. Available: https://blog.ret2.io/2018/07/11/pwn2own-2018-jsc-exploit/.

[121] S. GroB, *Attacking JavaScript engines: A case study of JavaScriptCore and CVE-2016-4622*, 2021. [Online]. Available: http://www.phrack.org/issues/70/3.html#article.

[122] S. GroB and N. Baumstark, *Pwn2Own 2017: UAF in JSC::CachedCall (WebKit)*, 2017. [Online]. Available: https://phoenhex.re/2017-05-04/pwn2own17-cachedcall-uaf.

[123] Lookout, *Technical analysis of the Pegasus exploits on iOS*, 2016. [Online]. Available: https://info.lookout.com/rs/051-ESQ-475/images/pegasus-exploits-technical-details.pdf.

[124] Z. Badawi, *Diving deep into a Pwn2Own winning WebKit bug*, 2019. [Online]. Available: https://www.zerodayinitiative.com/blog/2019/11/25/diving-deep-into-a-pwn2own-winning-webkit-bug.

[125] S. GroB, *Attacking client-side JIT compilers (v2)*, 2018. [Online]. Available: https://saelo.github.io/presentations/blackhat_us_18_attacking_client_side_jit_compilers.pdf.

[126] R. Avanzi, "The QARMA block cipher family. almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes," *IACR Transactions on Symmetric Cryptology*, vol. 2017, no. 1, pp. 4–44, 2017. [Online]. Available: https://tosc.iacr.org/index.php/ToSC/article/view/583.

[127] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "CETS: Compiler enforced temporal safety for C," in *ISMM '10: Proceedings of the 2010 International Symposium on Memory Management*, Toronto, Ontario, Canada, 2010, pp. 31–40.

[128] M. Kurth, B. Gras, D. Andriesse, C. Giuffrida, H. Bos, and K. Razavi, "NetCAT: Practical cache attacks from the network," in *S&P '20: Proceedings of the 2020 IEEE Symposium on Security and Privacy*, 2020, pp. 20–38.

[129] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "CrossTalk: Speculative Data Leaks Across Cores Are Real," in *S&P '21: Proceedings of the 2021 IEEE Symposium on Security and Privacy*, May 2021.

[130] R. C. Merkle, "Protocols for public key cryptosystems," in *S&P '80: Proceedings of the 1980 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 1980, pp. 122–134.

[131] G. E. Suh, D. Clarke, B. Gassend, M. v. Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in *MICRO-36: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, San Diego, CA, USA, 2003.

[132] R. Elbaz, D. Champagne, R. B. Lee, L. Torres, G. Sassatelli, and P. Guillemin, "Tec-tree: A low-cost, parallelizable tree for efficient defense against memory replay attacks," in

CHES '07: Proceedings of the 9th International Workshop on Cryptographic Hardware and Embedded Systems, Vienna, Austria, 2007, pp. 289–302.

[133] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, J. A. Joao, and M. K. Qureshi, "Morphable counters: Enabling compact integrity trees for low-overhead secure memories," in *MICRO '18: Proceedings of the 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture*, Fukuoka, Japan, 2018, pp. 416–427.

[134] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 simulator," *SIGARCH Computer Architecture News*, 2011.

[135] S. Vaarala, *Duktape*, `https://github.com/svaarala/duktape`, 2013.

[136] HTTP, *The 2019 Web Almanac: Http archive's annual state of the web report*, `https://almanac.httparchive.org/en/2019/`, 2019.

[137] J. Fulmer, *Siege*, `https://github.com/JoeDog/siege/`, 2012.

[138] Google, *Octane 2*, `https://github.com/chromium/octane`, 2012.

[139] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "RIPE: Runtime intrusion prevention evaluator," in *ACSAC '11: Proceedings of the 27th Annual Computer Security Applications Conference*, Orlando, Florida, USA, 2011, pp. 41–50.

[140] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "HDFI: Hardware-assisted data-flow isolation," in *S&P'16: Proceedings of the 2016 IEEE Symposium on Security and Privacy*, 2016, pp. 1–17.

[141] M. Gallagher, L. Biernacki, S. Chen, Z. B. Aweke, S. F. Yitbarek, M. T. Aga, A. Harris, Z. Xu, B. Kasikci, V. Bertacco, S. Malik, M. Tiwari, and T. Austin, "Morpheus: A vulnerability-tolerant secure architecture based on ensembles of moving target defenses with churn," in *ASPLOS '19: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Providence, RI, USA, 2019, pp. 469–484.

[142] B. Azad, *Examining pointer authentication on the iphone XS*, 2019. [Online]. Available: `https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html`.

[143] R. H. Gumpertz, "Combining tags with error codes," in *ISCA '83: Proceedings of the 10th Annual International Symposium on Computer Architecture*, Stockholm, Sweden, 1983, pp. 160–165.

[144] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys (CSUR)*, vol. 50, no. 1, p. 16, 2017.

[145] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *S&P '13: Proceedings of the 2013 IEEE Symposium on Security and Privacy*, Berkeley, CA, USA, 2013, pp. 574–588.

[146] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *S&P '15: Proceedings of the 2015 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 2015, pp. 745–762.

[147] International standardization working group for the programming language C, *ISO/IEC 9899:202x. ISO/IEC*, `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2346.pdf`, [Online; accessed 15-April-2022], 2019.

[148] A. Bhattacharyya, A. Sánchez, E. M. Koruyeh, N. Abu-Ghazaleh, C. Song, and M. Payer, "SpecROP: Speculative exploitation of ROP chains," in *RAID '20: Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses*, San Sebastian, Spain, 2020, pp. 1–16.

[149] A. Abbasi, J. Wetzels, T. Holz, and S. Etalle, "Challenges in designing exploit mitigations for deeply embedded systems," in *EuroS&P '19: Proceedings of the 2019 IEEE European Symposium on Security and Privacy*, Stockholm, Sweden, 2019, pp. 31–46.

[150] X. Gong and P. Pi, "Exploiting Qualcomm WLAN and modem over-the-air," in *Black Hat*, Las Vegas, CA, USA, 2019.

[151] JSOF Research Lab, *Ripple20: 19 zero-day vulnerabilities amplified by the supply chain*, 2020. [Online]. Available: `https://www.jsof-tech.com/disclosures/ripple20/`.

[152] O. Whitehouse, "An analysis of address space layout randomization on windows vista," 2007.

[153] N. Muralimanohar, A. Shafiee, and V. Srinivas, *Cacti 7.0*, `https://github.com/HewlettPackard/cacti`, 2017.

[154] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: Highly compatible and complete spatial memory safety for C," in *PLDI '09: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland, 2009, pp. 245–258.

[155] S. Xu, W. Huang, and D. Lie, "In-Fat pointer: Hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection," in *ASPLOS '21: Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems*, Virtual Event, 2021.

[156] G. J. Duck and R. H. C. Yap, "Heap bounds protection with low fat pointers," in *CC '16: Proceedings of the 25th International Conference on Compiler Construction*, Barcelona, Spain, 2016, pp. 132–142.

[157] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "WatchdogLite: Hardware-accelerated compiler-based pointer checking," in *CGO '14: Proceedings of the 12th IEEE/ACM International Symposium on Code Generation and Optimization*, 2014.

[158] T. Zhang, D. Lee, and C. Jung, "BOGO: Buy spatial memory safety, get temporal memory safety (almost) free," in *ASPLOS '19: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Providence, RI, USA, 2019, 631–644.

[159] H. Xia, J. Woodruff, S. Ainsworth, N. W. Filardo, M. Roe, A. Richardson, P. Rugg, P. G. Neumann, S. W. Moore, R. N. M. Watson, and et al., "CHERIvoke: Characterising pointer revocation using cheri capabilities for temporal memory safety," in *MICRO '52: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, Columbus, OH, USA, 2019, 545–557.

[160] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight Jr, B. C. Pierce, and A. DeHon, "Architectural support for software-defined metadata processing," in *ASPLOS '15: Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[161] N. P. Carter, S. W. Keckler, and W. J. Dally, "Hardware support for fast capability-based addressing," *SIGPLAN Not.*, vol. 29, no. 11, pp. 319–327, Nov. 1994.

[162] T. Nyman, J.-E. Ekberg, L. Davi, and N. Asokan, "CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers," in *Research in Attacks, Intrusions, and Defenses*, Springer International Publishing, 2017, pp. 259–284.

[163] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R Sekar, and D. Song, "Code-pointer integrity," in *OSDI'14: Proceedings of the 11th {USENIX} Symposium on Operating Systems Design and Implementation*, 2014, pp. 147–163.

[164] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, "Branch regulation: Low-overhead protection from code reuse attacks," in *ISCA'12: Proceedings of the 39th Annual International Symposium on Computer Architecture*, Portland, Oregon, USA, 2012, pp. 94–105.

[165] N. Burow, X. Zhang, and M. Payer, "SoK: Shining light on shadow stacks," in *S&P '19: Proceedings of the 2019 IEEE Symposium on Security and Privacy*, San Francisco, CA, USA, 2019, pp. 985–999.

[166] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi, "Missing the point(er): On the effectiveness of code pointer integrity," in *S&P '15: Proceedings of the 2015 IEEE Symposium on Security and Privacy*, San Jose, CA, USA, 2015, pp. 781–796.

[167] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "Pointguard: Protecting pointers from buffer overflow vulnerabilities," in *SSYM '03: Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, Washington, DC, USA, 2003.

[168] N. Tuck, B. Calder, and G. Varghese, "Hardware and binary modification support for code pointer protection from buffer overflow," in *MICRO-37: Proceedings of the 37th International Symposium on Microarchitecture*, Portland, OR, USA, 2004, pp. 209–220.

[169] S. J. Ghangro, "Block ciphers for low energy," Ph.D. dissertation, KU Leuven, 2017. [Online]. Available: `https://www.esat.kuleuven.be/cosic/publications/thesis-293.pdf`.

[170] D. Molka, D. Hackenberg, R. Schöne, and M. S. Müller, "Characterizing the energy consumption of data transfers and arithmetic operations on x86_64 processors," in *Proceedings of the International Conference on Green Computing*, Chicago, IL, USA, 2010, pp. 123–133.