Hardware-Assisted Defenses for Data Integrity and Confidentiality

by

Hossam ElAtali

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2025

© Hossam ElAtali 2025

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: David Lie
Professor, Dept. of Electrical and Computer Engineering
University of Toronto

Supervisor: N. Asokan
Professor, Cheriton School of Computer Science
University of Waterloo

Internal Members: Mei Nagappan
Associate Professor, Cheriton School of Computer Science
University of Waterloo

Meng Xu
Assistant Professor, Cheriton School of Computer Science
University of Waterloo

Internal-External Member: Arie Gurfinkel
Professor, Dept. of Electrical and Computer Engineering
University of Waterloo

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This dissertation is based on my contributions to the following six papers.

Publication I:
Hossam ElAtali et al. "BliMe: Verifiably Secure Outsourced Computation with Hardware-Enforced Taint Tracking". In: *Proceedings of the 2024 Network and Distributed System Security Symposium*. NDSS '24. San Diego, CA, USA: Internet Society, 2024. ISBN: 1-891562-93-2. DOI: 10.14722/ndss.2024.24105

I designed and implemented all hardware changes except the BliMe-BOOM ChaCha20 encryption module, which was designed and implemented by Lachlan J. Gunn (external collaborator). Lachlan also conceived the idea and implemented the formal model in F*. I designed and implemented all instruction set architecture (ISA) changes required in LLVM to generate the new instructions with help from Hans Liljestrand (postdoctoral researcher). I designed, planned and carried out all evaluation experiments. I took the lead in writing with help from all other authors.

Publication II:
Hossam ElAtali et al. "Data-Oblivious ML Accelerators using Hardware Security Extensions". In: *Proceedings of the 2024 IEEE International Symposium on Hardware Oriented Security and Trust*. Tysons Corner, VA, USA: IEEE, 2024. DOI: 10.1109/HOST55342.2024.10545398

I conceived the idea. I designed the hardware changes. John Z. Jekel (Undergraduate Research Assistant) implemented the majority of the hardware changes. I helped finish the implementation and performed all evaluations. I also implemented the formal model in F* with help from Lachlan J. Gunn (external collaborator). I led the writing with contributions from Prof. Asokan.

Publication III:
Hossam ElAtali et al. "BliMe Linter". In: *Proceedings of the 2024 IEEE Secure Development Conference (SecDev)*. Pittsburgh, PA, USA: IEEE, Oct. 2024, pp. 46–53. DOI: 10.1109/SecDev61143.2024.00011

Xiaohe Duan (MMath student) designed and implemented all LLVM changes with help from me and Hans Liljestrand (postdoctoral researcher). Hans also conceived the idea.

Xiaohe and I contributed to the evaluation and I led the writing with help from Hans and Prof. Asokan.

Publication IV:
Hossam ElAtali and N. Asokan. "CacheSquash: Making caches speculation-aware". In: *arXiv preprint: 2406.12110* (2025). DOI: 10.48550/arXiv.2406.12110

I conceived the idea, designed and implemented all changes, performed all evaluations and led the writing.

Publication V:
Hossam ElAtali et al. "BLACKOUT: Data-Oblivious Computation with Blinded Capabilities". In: *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security.* CCS '25. Taipei, Taiwan: Association for Computing Machinery, Oct. 2025. DOI: 10.1145/3719027.3765169

The idea was jointly conceived by all co-authors. I designed and implemented all hardware changes to CHERI-Toooba and helped Merve Gülmez (external collaborator) with the evaluation. Merve designed and implemented all changes to LLVM, CHERI-BSD and the software stack, and led the evaluation. All authors contributed to the writing.

Publication VI:
Hossam ElAtali, Hans Liljestrand, and Jan-Erik Ekberg. "PBI: Program-Counter-Based Isolation". In: *To Be Submitted.* 2025

I conceived the idea. Jan-Erik Ekberg (internship supervisor) and Hans Liljestrand (internship colleague) helped refine the idea. I designed and implemented all changes, including QEMU, and the Linux kernel. I performed all evaluations and led the writing.

Abstract

The increasing complexity of modern computing systems and their exposure to the internet expose sensitive data to a range of security threats from remote adversaries. Bugs in software can lead to run-time attacks that gain direct access to sensitive data in memory, compromising its integrity and confidentiality. Furthermore, hardware and/or compiler optimizations can introduce data-dependent behavior that expose sensitive data to side-channel leakage, even in the absence of software bugs, breaking confidentiality.

As business needs evolve, different usage scenarios, such as outsourced computation, have gained popularity, making the task of protecting data integrity and confidentiality more complex. This dissertation investigates how the integrity and confidentiality of sensitive data at run-time can be efficiently preserved through hardware-assisted mechanisms. I consider a range of usage scenarios and threat models, from protecting data sent to remote servers for outsourced computation by untrusted code, to protecting data processed locally from other vulnerable or malicious parts of the system.

Specifically, this dissertation addresses: 1. how to efficiently protect data confidentiality against side-channel leakage with negligible overheads. Existing solutions to side-channel leakage suffer from significant overheads, making their deployment difficult in situations where performance is critical. I address this problem with CacheSquash, a software-transparent hardware mechanism to effectively harden against transient side-channel attacks such as Spectre and Meltdown with near-zero overheads. 2. How to combine protections against *both* direct access and side channels. I propose BliMe, a novel architecture that relies on remote attestation, taint-tracking and hardware-enforced data obliviousness to protect sensitive data processed by untrusted code in an outsourced computation setting. 3. For integrity, I propose PBI, a novel hardware primitive that enables efficient memory protection for sandboxing and in-process isolation, thereby safeguarding *both* data confidentiality and integrity. 4. Finally, I address how to efficiently combine memory safety and side-channel protection mechanisms for data integrity and confidentiality. For this, I propose `BLACKOUT`, a hardware-software extension to CHERI that enforces data-oblivious computation on sensitive data, and inherits the memory safety properties of CHERI, all while introducing minimal overheads.

The proposed solutions confirm that hardware-assisted mechanisms can indeed be used to efficiently protect data at run-time, both from direct access and side-channel leakage. I conclude my dissertation with promising directions for future work.

Acknowledgements

There are a lot of people without whom this dissertation would not have been possible. To say that this degree required a lot of patience (from everyone) would be an understatement.

I would like to start first and foremost with my wife, Noha, who has sacrificed a lot along the way. At many points I wondered how she could have such patience, and yet she always surprised me with more. Thank you for your patience, love, support and always believing in me. You were there to support me at the hardest moments, and there to laugh and cry with me at the happiest. P.S. As I have been saying for the past five years, I promise to help out more now that this is done.

My parents have been an anchor of support, not only during this degree, but throughout my entire academic life. I would not be the man I am today without them. They taught me honesty, perseverance, and self-confidence, and always believed I could excel at anything I do. Mum, Dad, I know you always wanted me to get a PhD, and I am glad you encouraged and supported me to pursue one.

I would also like to give a special thank you to my mother-in-law, Hadia. Thank you for all the times you helped carry the weight off our shoulders. Thank you for all the times you fed me great food. And sorry for making you worry about me so much.

I would also like to thank my sister, Dina. I was born a single child, but I got the best little sister along the way. And despite the additional degree and its accompanying maturity, I will continue to make fun of you. I also want to give a big thank you to the Khafagy family. You make it feel like home away from home, and our move to Canada would have been almost impossible without you.

Another thank you goes to my friends. I know this degree kept me from a lot of gatherings, vacations, and North Coast trips, but the Pharaoh is now ready to party. A special thank you goes to Ibrahim Ahmed, who has quite literally sponsored me throughout this degree. This whole thing would have been cut quite short if it were not for you.

Even though they are not old enough to read this yet, I would like to give a shout out to my sons, Omar and Youssef. Hopefully some day we can read this together and I can tell you why not to ever ever do a PhD. Just kidding of course...somewhat. We can discuss this later.

I would also like to thank my supervisor, Prof. Asokan. Thank you for being patient with me. I have learned a lot from you. You have one of the highest levels of ethics (both work and personal) I have ever seen. You also have great writing skills, and I hope some of

that has rubbed off on me. I also hope you can see that when comparing my writing now to when I started. I will take your silence as a yes.

The acknowledgements would be incomplete if I did not mention the great people I worked with. A big shout-out to Hans Liljestrand, Lachlan Gunn, Thomas Nyman and Merve Gülmez. I have learned a lot from each of you. Another big shout-out goes to the great people I worked with at Huawei Helsinki System Security Lab (HSSL): Jan-Erik Ekberg, Valentin Manea, Mohammed Hassan and everyone else at the lab who gave me valuable insights and feedback.

If I have forgotten anyone, please forgive me. This is the last piece of text going into my dissertation, and I am incredibly excited to finish. I have tried to give the acknowledgements the credit they deserve, but it will never be enough. Thank you again to everyone.

الحمد لله

Table of Contents

Examining Committee **ii**

Author's Declaration **iii**

Statement of Contributions **iv**

Abstract **vi**

Acknowledgements **vii**

List of Figures **xv**

List of Tables **xvii**

List of Abbreviations **xix**

1 Introduction **1**

2 Background **6**

2.1 Run-time attacks 6

2.1.1 Memory safety 6

2.1.2 Exploitation 8

2.2 CPU optimizations 9

2.2.1 Caches 9
2.2.2 Speculation and transient execution 10
2.3 Side channels 10
2.3.1 Cache & contention timing attacks 11
2.3.2 Transient execution attacks 11
2.3.3 Data-oblivious code 12
2.4 Trusted execution environments 13

3 CacheSquash 15

3.1 Introduction 15
3.2 Problem description 17
3.2.1 Goals & Objectives 17
3.2.2 Threat model 18
3.3 CacheSquash: Design 18
3.3.1 Cache flow chart 19
3.3.2 Forwarding cancellations upstream 21
3.3.3 TLBs & I-Caches 21
3.4 CacheSquash: Implementation 22
3.5 Security evaluation 22
3.5.1 Case studies 22
3.5.2 Security limitations 27
3.6 Performance Evaluation 29
3.7 Discussion & Future Work 30
3.7.1 Meltdown 30
3.7.2 Cancellation broadcasts 30
3.7.3 Cancellation of memory bus transactions 31
3.7.4 Overlapping cancelled and uncancelled requests 31
3.8 Related work 31

4 BliMe **34**
4.1 Introduction . . . 34
4.2 Problem Description . . . 36
4.2.1 Usage scenario . . . 36
4.2.2 Goals and objectives . . . 36
4.3 Design . . . 37
4.3.1 System overview . . . 37
4.3.2 Adversary model . . . 37
4.3.3 Protocol . . . 38
4.3.4 Taint-tracking policy . . . 40
4.3.5 BliMe-compliant software . . . 42
4.4 Implementation . . . 43
4.4.1 Architectural changes . . . 43
4.4.2 BliMe-BOOM . . . 44
4.5 Performance & resource usage evaluation . . . 47
4.5.1 BliMe-BOOM . . . 47
4.5.2 Memory optimization . . . 48
4.6 Compatibility evaluation . . . 50
4.7 Discussion & Future Work . . . 51
4.7.1 Handling large numbers of clients . . . 51
4.7.2 Enabling safe local processing . . . 52
4.7.3 Handling secret-dependent faults . . . 52
4.8 Related work . . . 53

5 BliMe Extensions **54**
5.1 Dolma . . . 55
5.1.1 Introduction . . . 55
5.1.2 Background – Gemmini . . . 56

5.1.3 Assumptions & Threat Model 57
5.1.4 Design & Implementation 58
5.1.5 Evaluation 63
5.1.6 Discussion & Future Work 66
5.2 BliMe Linter 67
5.2.1 Introduction 67
5.2.2 Background – SVF 68
5.2.3 Design & Implementation 68
5.2.4 Evaluation 71
5.2.5 Discussion & Future Work 76

6 Program-Counter-Based Isolation 79

6.1 Introduction 79
6.2 Background 82
6.2.1 Permission overlays 82
6.2.2 Shadow stack 83
6.3 Threat Model 84
6.4 Goals and Challenges 85
6.5 PBI Design 86
6.5.1 PBI-Core 86
6.5.2 PBI-PI 92
6.6 Implementation 94
6.7 Evaluation 95
6.7.1 Security 95
6.7.2 Performance 98
6.8 Discussion & Future Work 101
6.9 Related work 103

7 BLACKOUT **106**
7.1 Introduction . . . 106
7.2 The CHERI capability architecture . . . 108
7.3 System and Adversary Model . . . 110
7.4 Goals and Challenges . . . 111
7.4.1 Goals and Primary Challenges . . . 111
7.4.2 Additional Challenges . . . 112
7.5 Blinded Capability Design . . . 113
7.5.1 Software Architecture . . . 114
7.5.2 Hardware architecture . . . 115
7.5.3 Motivating Examples . . . 118
7.6 BLACKOUT Blinded Capability Implementation . . . 119
7.6.1 BLACKOUT CHERI-RISC-V CPU . . . 119
7.6.2 BLACKOUT Software Stack . . . 121
7.7 Evaluation . . . 122
7.7.1 Power & resource usage . . . 123
7.7.2 Performance . . . 123
7.7.3 Security . . . 128
7.8 Discussion & Limitations . . . 130

8 Discussion & Future Work **133**
8.1 Solution comparisons . . . 133
8.2 Limitations . . . 136
8.3 Future work . . . 136

9 Related Work **140**
9.1 Taint tracking . . . 140
9.2 Data-oblivious execution . . . 140
9.3 Point solutions for side-channel attacks . . . 141

10 Conclusion **143**

References **145**

List of Figures

3.1 CacheSquash sequence diagrams 19

3.2 CacheSquash flow chart 20

3.3 Central processing unit (CPU) events for Spectre attack instances. 25

3.4 Results for SPEC CPU 2017 in instructions-per-cycle (IPC) with `ref` input size on 1- and 4-core configurations of baseline and CacheSquash. 30

3.5 Results for PARSEC in ticks with medium and large input sizes on 1- and 4-core configurations of baseline and CacheSquash. 33

4.1 BliMe Overview 38

4.2 Overhead of BliMe-BOOM-1 and BliMe-BOOM-8 48

4.3 Overhead before and after applying the optimization from Section 4.4.2 . . 50

5.1 System Overview 58

5.2 Dynamic information flow tracking (DIFT) in parallel for the weight-stationary data flow inside a 2x2 systolic array 61

5.3 Pipelined writes 63

5.4 Performance results of running ResNet-50 image classification 64

5.5 Resource usage results for Dolma relative to unmodified Gemmini. 65

5.6 The BliMe Linter Overview. 69

6.1 Permission overlay mechanism 83

6.2 PBI permission check mechanism 87

6.3 PBI architecture overview 88
6.4 Sandboxing example with two isolated sandboxes in domains A and B . . . 89
6.5 Example showing sandbox exits using trampolines. 90
6.6 Confused deputy attack scenario 92
6.7 Extended shadow stack structure for PBI-PI 93
6.8 PBI-PI architecture overview 94
6.9 Context switch overhead on SPEC CPU2017 100
6.10 Execution run-time for Javascript microbenchmark 101

7.1 In-memory representation of Capability Hardware Enhanced RISC Instructions (CHERI) capabilities adapted from Watson et al. [199] 109
7.2 High-level overview of the components of the blinded capability software stack 113
7.3 High-level overview of the blinded capability-enhanced CHERI-RISC-V CPU hardware components 114
7.4 Run-time in seconds of the OISA benchmarks 126
7.5 Run-time in seconds of the OISA `binary_search` benchmark in dynamically-linked, statically-linked, and bare-metal configurations 127

8.1 Spider chart comparing proposed solutions 134
8.2 Trade-offs made by trusted execution environments (TEEs), BliMe and BLACKOUT between integrity guarantees, side-channel protection and threat model assumptions. 137

List of Tables

1.1 Coverage of proposed solutions across integrity and confidentiality. 5

3.1 Configurations used in case study experiments 24

3.2 The values of *CC* for all experiments. 27

3.3 Parameters used in performance evaluation. 29

4.1 BliMe taint-tracking policy rules for all instruction types 41

4.2 Effect of BliMe-BOOM modifications on power consumption and FPGA resource usage (unmodified BOOM vs. BliMe-BOOM-1 and BliMe-BOOM-8). 47

4.3 Average overheads of running SPEC2017 on BliMe-BOOM-1 and -8, BliMe-gem5 and BliMe-gem5 Optimized 49

5.1 Effect of modifications on FPGA power consumption. 65

5.2 Results of running the BliMe linter on the oblivious instruction set architecture (OISA) benchmarks, PARSEC benchmarks and TensorFlow Lite . . . 72

7.1 Blindedness bit propagation and side-channel prevention rules enforced by `BLACKOUT` hardware 117

7.2 Area and power costs on VCU118 @ 25MHz expressed in number of LUTs and registers, and Watts respectively. 122

7.3 Performance cost on VCU118 @ 25MHz expressed as CoreMark test results for $5x10^3$ iterations 123

7.4 SpectreGuard benchmark performance average measured over 20 runs of each test case (maximum $\sigma = 0.08\%$). 128

7.5 Transient-execution attacks successfully prevented on CHERI-Toooba and Blinded CHERI-Toooba 129

List of Abbreviations

GEP GetElementPtr

BLACKOUT Blinded Architectural Capabilities and Kernel for Oblivious Userspace Tasks

clc load capability via capability

csc store capability via capability

csp stack pointer capability

pcc program counter capability

ABI application binary interface

ALU arithmetic logic unit

AMi Architectural-Mimicry

API application programming interface

ASLR address-space layout randomization

BCB bounds check bypass

BliMe Blinded Memory

BRR blinded register record

CFI control-flow integrity

CFL control-flow linearization

CHERI Capability Hardware Enhanced RISC Instructions

CISA Cybersecurity & Infrastructure Security Agency

COTS commercial off-the-shelf

CPU central processing unit

CSC Capability Speculation Contract

DFI data-flow integrity

DFL data-flow linearization

DIFT dynamic information flow tracking

DIT data-independent timing

DMA direct memory access

DOP data-oriented programming

DRAM dynamic random-access memory

DVFS dynamic voltage and frequency scaling

eBPF extended Berkeley Packet Filter

ECC error-correcting code

FHE fully-homomorphic encryption

FPGA field-programmable gate array

GCS Guarded Control Stack

GPU graphics processing unit

HSM hardware security module

IP intellectual property

IPC instructions-per-cycle

IPI in-process isolation

IR intermediate representation

ISA instruction set architecture

JIT just-in-time

JOP jump-oriented programming

LLC last-level cache

LSU load-store unit

ML machine learning

MMU memory management unit

MPK Memory Protection Keys

MSHR miss status holding register

MTE Memory Tagging Extension

NCSC National Cyber Security Centre

OISA oblivious instruction set architecture

OS operating system

PBI Program-Counter-Based Isolation

PBIR PBI register

PC program counter

PE processing element

PHT pattern history table

PI permission inheritance

PoC proof-of-concept

POE Permission Overlay Extensions

POI permission overlay index

POR permission overlay register

PTE page table entry

PTW page table walker

REE rich execution environment

ROP return-oriented programming

RSB return stack buffer

RTL register-transfer level

SEP Secure Enclave Processor

SFI software-based fault isolation

SoC system-on-chip

SP stack pointer

STT Speculative Taint Tracking

SVFG sparse value-flow graph

TA trusted application

TCB trusted computing base

TEE trusted execution environment

TLB translation lookaside buffer

VM virtual machine

WLLVM whole-program LLVM

WNS worst negative slack

Chapter 1

Introduction

As modern computing systems grow more complex and interconnected through the global internet, they become increasingly vulnerable to security threats that can compromise sensitive data through *remote* attacks, in which attackers have no physical access to the system. One of the most pressing and persistent security challenges is run-time attacks due to inadequate *memory safety* that can lead to attackers gaining *direct access* to data in memory, compromising both *integrity* and *confidentiality*. Programming languages like C and C++, while foundational to modern computing, lack built-in memory-safety mechanisms. This has made them historically prone to security flaws such as buffer overflows, use-after-free errors, and other forms of memory corruption. Despite decades of awareness, these issues remain among the most prevalent and dangerous in cybersecurity [138].

In recent years, the urgency to confront the lack of memory-safety at a systemic level has grown, particularly due to heightened regulatory oversight. Prominent cybersecurity agencies, including the U.S. Cybersecurity & Infrastructure Security Agency (CISA) and the U.K. National Cyber Security Centre (NCSC), have publicly emphasized the need to transition toward memory-safe solutions. One solution is the adoption of modern programming languages like Rust, which enforces memory safety at compile time. However, while memory-safe languages provide strong guarantees at minimal performance costs, the enormous volume of legacy code written C and C++—which underpins much of today's software infrastructure—makes a wholesale transition economically and logistically unfeasible in the near term.

Until memory-safe languages become ubiquitous, if at all, alternative solutions are required. Software solutions exist to *prevent* run-time attacks, either by improving memory safety (e.g., Checked C [61], SoftBound [142]) or by providing defense-in-depth (e.g., control-

flow integrity, sandboxing, in-process isolation) to reduce the effectiveness of memory safety exploits. These software-only approaches offer notable advantages, including rapid deployment and ease of integration into existing systems, making them the best choice in certain situations where time-to-market or compatibility constraints are paramount. However, hardware-assisted solutions [223] provide a better security-performance trade-off, albeit at the cost of increased development time and complexity due to hardware changes. For example, SoftBound provides spatial memory safety at an average overhead of 67%, whereas HardBound [51], which provides architectural hardware support for bounded pointers, gives the same guarantees at an average overhead of $7 - 9\%$ (with a reported maximum of 23%). In recent years, hardware vendors have been adding an increasing number of security primitives to their processors [135, 15, 121], and their adoption into software has gained traction, as demonstrated by many industry-scale software projects (e.g., Android, V8 [81], Linux Kernel).

In addition to run-time attacks which give attackers *direct access*, the confidentiality of sensitive data can be compromised by *side-channel leakage*. Adversaries can observe systems during execution, allowing them to infer sensitive data from programs that have data-dependent behavior. A prominent example of side-channel leakage is timing side channels, which result in data-dependent execution timing. For instance, if the outcome of a conditional branch is determined by a sensitive value, and each possible branch path takes a different amount of time to execute, an attacker may be able to deduce information about that sensitive value by measuring how long it takes for execution to complete. Similarly, if a sensitive value is used to index into an array, the resulting memory access pattern—which reveals the specific memory address accessed—can alter the state of shared resources like the CPU cache, or generate detectable traffic on the main memory bus, both of which can be observed and exploited by an attacker.

Constant-time programming is a technique designed to defend against timing side-channel attacks by ensuring that the execution time of code does not vary based on sensitive data. However, writing truly constant-time code manually is extremely challenging. Even in software specifically engineered to resist side-channel attacks, such as cryptographic libraries, numerous vulnerabilities have been discovered, underscoring the difficulty of getting it right. Furthermore, modern optimizing compilers can alter code during translation into machine instructions in ways that unintentionally compromise constant-time behavior. For instance, they might introduce branches based on secret values, undermining the original intent of the programmer. In addition, hardware-level features like *speculative execution*—which are designed to improve performance and are largely invisible to software—can be exploited by attackers to extract secret data using known side-channel techniques. Unlike memory safety bugs, which often lead to visible crashes or incorrect behavior when triggered, flaws

in constant-time code are usually silent and non-obvious. They may remain undetected until an attacker successfully exploits them—if they are detected at all.

With the two threats to data integrity and confidentiality above in mind, I put forward the following thesis, which I will defend in this dissertation:

Thesis

Hardware-assisted mechanisms can efficiently protect the integrity and confidentiality of sensitive data against remote adversaries, both from direct access and side channels.

Prior works have individually addressed parts of this thesis, but there has been limited research towards achieving a unified solution, which I will address towards the end of the dissertation.

As business requirements change, new use cases, such as *outsourced computation* where clients send data to a remote server for processing, have become increasingly common, adding complexity to the challenge of safeguarding data. In this dissertation, I consider a range of usage scenarios and threat models, from protecting data sent to remote servers for outsourced computation by untrusted code, to protecting data processed by trusted code from other vulnerable or malicious parts of the system.

To defend the above thesis, I pose the following research questions.

RQ1: *(How) can we protect data confidentiality against side-channel leakage with* ***negligible overheads****?*

Prior solutions [216, 106, 165, 120, 219] to side-channel leakage provide strong security guarantees, but come at significant performance costs. In situations where performance is critical and *complete* protection against side channels is not required, it is more desirable to deploy a security mechanism that hardens against (rather than completely prevents) side-channel leakage at no performance cost. I address this challenge with **CacheSquash** (Chapter 3), a novel software-transparent mechanism to harden against *transient execution attacks* such as Spectre [111] and Meltdown [122]. CacheSquash works by cancelling outstanding read requests to the cache hierarchy that the central processing unit (CPU) has determined to by mis-speculated. Once mis-speculation is detected, the CPU sends a cancellation that propagates down the cache hierarchy and prevents changes to caches that have not yet received a response, thereby reducing the overall change to cache state and reducing the likelihood of success for transient execution attacks.

RQ2: *(How) can we efficiently protect data confidentiality against **both direct access and side-channel leakage**?*

While CacheSquash protects data against side-channel leakage, it does not address direct memory access, e.g., through run-time attacks or untrusted code. For this, I propose **BliMe**(Blinded Memory) (Chapters 4 and 5), a new architecture that uses remote attestation, taint-tracking and hardware-enforced data obliviousness to safeguard data handled by untrusted code—protecting it from both direct exposure and leakage through side channels. Blinded Memory (BliMe) does this efficiently, imposing minimal overheads of $\approx 8\%$ compared to native execution. I also propose **Dolma**, an enhancement of the Gemmini *machine learning (ML) hardware accelerator* that extends BliMe guarantees (i.e., confidentiality against direct access and side channels) to data processed on Gemmini.

RQ3: *(How) can we efficiently protect data confidentiality and **integrity**?*

BliMe effectively protects data confidentiality, but does not provide integrity guarantees. For RQ3, I propose **PBI** (Program-Counter-Based Isolation) (Chapter 6), a novel hardware primitive that enables efficient memory protection for sandboxing and in-process isolation, thereby safeguarding data confidentiality *and integrity*. Program-Counter-Based Isolation (PBI) works by including the program counter (PC) into the access permission check, removing the need for manual memory permission changes when transitioning between security domains. This improves security by avoiding the presence of unprivileged permission changing instructions (such as for Intel Memory Protection Keys (MPK) [135]) and by reducing the number of code gadgets available for an return-oriented programming (ROP) attack at any single point in the codebase.

RQ4: *(How) can we efficiently **combine memory protection and side-channel protection** to ensure confidentiality and integrity of sensitive data?*

After answering the first three research questions, on one hand, we have BliMe, which protects data confidentiality against both direct access and side channels, and on the other hand, we have PBI, which provides strong memory protection for data integrity and confidentiality, but does not prevent side-channel leakage. For RQ4, I address the challenge of efficiently *combining* memory protection and side-channel protection for confidentiality and integrity by proposing **Blinded Architectural Capabilities and Kernel for Oblivious Userspace Tasks (`BLACKOUT`)** (Chapter 7), a hardware-software extension to CHERI that enforces data-oblivious computation on sensitive data, and inherits the memory safety properties of CHERI, at minimal overheads.

In Table 1.1, I show how the different solutions I propose address the goals of data

<table>
<tr><th rowspan="2">Integrity</th><th colspan="2">Confidentiality</th></tr>
<tr><th>Direct Access</th><th>Side Channels</th></tr>
<tr><td></td><td></td><td>CacheSquash</td></tr>
<tr><td></td><td colspan="2">BliMe & Dolma</td></tr>
<tr><td colspan="2">PBI</td><td></td></tr>
<tr><td colspan="3">BLACKOUT</td></tr>
</table>

Table 1.1: Coverage of proposed solutions across integrity and confidentiality.

integrity and confidentiality. In addition, other considerations may apply. For example, in outsourced computation settings, the code operating on the data may be untrusted. While BliMe and PBI are robust against untrusted code, BLACKOUT is not. In terms of performance, CacheSquash has the lowest overhead, followed by PBI, BliMe and then BLACKOUT (when comparing to a non-CHERI baseline). Each of the proposed solutions satisfies a specific combination of performance impact, threat model and security guarantees. In Chapter 8, I delve into a more detailed comparison of the solutions and propose synergistic opportunities for future work.

Chapter 2

Background

In this chapter, I introduce common background information that is essential to understanding the remainder of the dissertation. In later chapters, I additionally include project-specific background where necessary.

2.1 Run-time attacks

2.1.1 Memory safety

Memory safety represents one of the most critical challenges in systems security. Memory safety violations occur when programs access memory in ways that violate the intended semantics of the programming language or the underlying memory model. These violations can lead to undefined behavior, data corruption, information disclosure, and arbitrary code execution, making them a primary target for attackers seeking to compromise system security. Memory safety violations can generally be split into spatial and temporal violations.

Spatial violations Spatial memory safety violations occur when programs access memory outside the bounds of allocated objects or valid memory regions. Out-of-bounds accesses, such as buffer overflows, represent the most common form of spatial violation, occurring when programs write data beyond the boundaries of allocated buffers. These violations can corrupt adjacent data structures, overwriting critical program state (e.g., bypassing authentication), or modifying control-flow information. Stack-based violations can overwrite return addresses stored on the stack, allowing attackers to redirect program execution to

arbitrary code, achieving complete control over the system. Heap-based violations target dynamically allocated memory regions, potentially corrupting heap metadata or adjacent allocated objects. This can lead to more sophisticated attacks such type-confusion, where an attacker manipulates the program to treat an object in memory using an incorrect type, violating program semantics.

Temporal violations Temporal memory safety violations occur when programs access memory regions at inappropriate times, typically after the memory region has been deallocated or before it has been properly initialized. Use-after-free (UAF) vulnerabilities represent the most significant class of temporal violations, and arise from the mismatch between object and pointer lifetimes. When an object is deallocated, any existing pointers to that object become "dangling pointers" that reference memory that may have been reallocated for different purposes. Subsequent use of these dangling pointers can lead to data corruption, information disclosure, type confusion, or arbitrary code execution, depending on how the freed memory has been reused.

Mitigations A relatively "simple" approach to prevent all memory safety violations is to *always* use a *memory-safe* programming language, such as Rust. However, this does not seem to be feasible in the foreseeable future: unsafe languages such as C and C++ are pervasive and comprise a large part of legacy (and current) code. Furthermore, partial adoption of memory-safe languages, while helpful, is not sufficient, as vulnerable parts of a program written in an unsafe language can cause memory safety violations in parts written in safe languages, breaking their guarantees. Therefore, other defenses are also required.

One form of mitigation is static analysis techniques, which attempt to identify potential memory safety violations during compilation or before program execution. These approaches range from simple pattern matching to sophisticated abstract interpretation and symbolic execution techniques. Other defenses include stack canaries (guard values placed between local variables and return addresses), which can detect stack buffer overflows at runtime, and bounds checking, which can be implemented through compiler instrumentation but typically incurs significant performance overhead. Address-space layout randomization (ASLR) randomizes the memory layout of programs, making it difficult for attackers to predict the locations of code, data, or system functions. While not preventing memory safety violations, ASLR significantly increases the difficulty of reliable exploitation (discussed below). Dynamic analysis tools, such as AddressSanitizer [171] and Valgrind, can detect memory safety violations during program execution. While these tools can incur substantial performance overhead, they are invaluable for debugging and testing.

Hardware-based defenses against memory safety violations have also gained traction. Memory tagging approaches, such as Arm's Memory Tagging Extension (MTE) [15], associate metadata tags with memory allocations and pointers, enabling detection of spatial memory violations. Each memory *granule* is extended with additional bits to carry the tags, which are usually stored in a separate *shadow region* of memory. This requires double the number of memory accesses to merge data with its tags but can be optimized using techniques such as tag caching and eliding unnecessary tag accesses [188, 100]. In Arm MTE, memory is tagged with a "color" and that color is assigned to corresponding memory pointers. If an attempt is made to access the memory region with a pointer containing the wrong color, the hardware recognizes the mismatch as a memory safety violation. CHERI [199, 200, 211] is another hardware-based defense against memory safety violations that extends conventional memory systems with hardware-supported capabilities to verify memory access and management. Capabilities are represented at double the width of native pointers and ensure spatial memory safety by associating memory allocations with valid address ranges and access permissions. CHERI's capabilities are derived from existing ones, with constraints ensuring they cannot exceed the permissions of their parent. Temporal memory safety is provided via extensions such as Cornucopia [204] for capability revocation.

2.1.2 Exploitation

Memory safety violations are an essential first step towards a successful run-time attack, but, in general, they alone are not sufficient. Programs can suffer from memory safety violations that are not "exploitable", i.e., are not useful to an attacker, likely due to constraints on the violations such as limited scope or accessible memory addresses. As mentioned in the previous section, only violations that result in useful changes to the program state (e.g., overwritten return addresses) can lead to run-time attacks. Therefore, attackers must carefully select inputs to the program to induce such useful violations.

Assuming a useful memory write primitive is available for an attacker to use, several types of run-time attacks are possible. While simply listing the types of attacks and their corresponding defenses is possible, I believe a chronological presentation is more useful as it demonstrates the constant race between attacks and defenses. One of the first types of attacks that appeared in the wild was *code-injection* attacks, which work by directly writing code to memory and then forcing the program to execute that code, e.g., by overwriting a function pointer or a return address on the stack. In response, systems introduced W⊕X, which prevents code execution in writable memory regions. Attackers circumvented this by using return-oriented programming (ROP) [172] and jump-oriented programming (JOP) [28], which identify useful instruction sequences (*gadgets*) in *existing* program code

and chain their execution to obtain arbitrary functionality. This gave rise to control-flow integrity (CFI), which aims to limit program execution paths to expected paths along a control-flow graph. Varying degrees of granularity exist for CFI. Fine-grained approaches limit each control-flow change to a very small set, but suffer from high complexity and/or overheads [132]. Coarse-grained approaches trade-off precision for performance, and have recently gained traction with several hardware vendors introducing primitives to support them (e.g., Intel CET and Arm BTI) [2, 32, 174, 14]. Attackers responded to CFI by introducing data-only attacks, such as data-oriented programming (DOP), which require no violation of the program's control-flow graph, and instead rely on corrupting program data to exercise useful (existing) paths in the program. Proposals to combat DOP by enforcing data-flow integrity (DFI) exist [36], but have not yet gained widespread adoption.

The current state of affairs is that control-flow attacks such as ROP and JOP are still a significant threat, as current defenses are either incomplete due to using a coarse-grained approach (creating holes for attackers to exploit), or have prohibitive performance overheads (preventing adoption). Furthermore, the rise of just-in-time (JIT) compilation techniques has rolled back the benefits gained from W⊕X, complicating defenses. Nowadays, modern systems rely on *defense-in-depth* strategies. For example, operating systems (OSs) use layers of isolation to prevent malicious or compromised processes from affecting other processes or the OS itself. Another example is *in-process isolation*, which attempts to sandbox vulnerable sections of the program such that successful attacks inside the sandbox have limited effects on the rest of the program.

2.2 CPU optimizations

2.2.1 Caches

Caches are small, high-speed memory structures close to the central processing unit (CPU) cores that store frequently-accessed data and instructions, thus reducing the time the CPU needs to fetch this information again in the future. Data is stored in caches in chunks called "lines" or "blocks", usually 64 bytes in size. Each cache line stores the data itself as well as a tag that identifies which address the data belongs to. When a cache receives a read or write request, it searches its tags for one matching the request. If a match is found, this is called a "cache hit", and the cache can respond with the data. Otherwise, a "cache miss" has occurred and must be handled by a miss status holding register (MSHR) [90, 8, 221]. MSHRs are used in modern caches to enable *non-blocking* miss handling by keeping track of outstanding misses, issuing requests downstream, and servicing the misses once a

response is received. Each active MSHR is in charge of a single tag (i.e., 64-byte aligned address). Multiple misses with the same tag are added as "targets" to the same MSHR. Concretely, if there is already an outstanding MSHR with the same tag, the miss is added to it as an additional target. Once a response is received for this MSHR, all its targets are serviced. If there is no matching outstanding MSHR, an empty MSHR is allocated to the miss. Caches have a fixed number of MSHRs and a maximum number of targets per MSHR; if there are no empty MSHRs to handle a new miss or the matching MSHR has reached its maximum number of targets, the cache must stall.

2.2.2 Speculation and transient execution

Modern processors employ *speculative execution* to improve performance by predicting and executing instructions ahead of time. This allows the processor to continue processing instructions even when there is a branch instruction whose outcome is uncertain. When predictions are correct, performance gains are significant as the CPU does not need to wait for a branch outcome to be determined before fetching the next instruction. However, when a *mis-prediction* occurs, the CPU must *squash* (i.e., discard) the speculatively executed instructions and return to fetch instructions from the correct branch path. The *speculation window* refers to the period during which instructions are executed speculatively. The larger the speculation window, the more instructions that can be executed before a squash occurs. Transient execution is a more general term for any instructions that are executed and then discarded before being committed. This can refer to speculative execution, which uses speculations made on branching instructions and memory dependencies, as well as instructions executed microarchitecturally and then discarded due to faults, such as page faults.

2.3 Side channels

Side channels are outputs of the system that are not part of the developers' intended outputs. Prominent examples of CPU side channels are execution time, memory access patterns, observable microarchitectural state (such as the state of shared caches, branch predictors and performance counters), power consumption and electromagnetic radiation [150, 112, 71, 126, 123, 108]. Side-channel leakage can occur when an adversary is able to infer information about the sensitive data by observing the system while the data is processed. For example, if a conditional branch instruction depends on a sensitive value and the execution time of each branch is different, an adversary can infer some information about the sensitive

value by monitoring the time it takes to complete the branch. Another example is when a sensitive value is used to index an array in memory; the memory access pattern, which is the sensitive address in this case, can change the observable state of a shared cache, or result in an observable request on the main memory bus.

Several side channels, such as power consumption and electromagnetic radiation, rely on measurement of physical parameters and therefore require physical access to the system under attack. In this dissertation, I consider such side channels out of scope. I focus on side channels that can be observed remotely: specifically, timing side channels.

2.3.1 Cache & contention timing attacks

Cache timing attacks, such as Flush+Reload [217] and Prime+Probe [149], exploit variations in the time it takes for CPUs to access cached vs. uncached data. Cache hits take less time to complete than cache misses. Attackers can compare the time it takes to access a certain address against a threshold to infer whether the corresponding data is cached. If a process uses secret-dependent memory addresses, this affects the cache state (i.e., which addresses are cached), which the attacker can then probe to leak information about the secret.

Unlike cache timing attacks that focus on cache state changes, contention timing attacks target the scheduling and utilization patterns of shared microarchitectural resources. For example, an attacker can maintain full utilization of a shared functional unit in the CPU, and monitor when another victim process contends for this unit. The contention causes the attacker to wait until the resource is freed, and this delay informs the attacker that the victim is using the resource. If the victim's behavior is secret-dependent (i.e., whether the victim uses the resource depends on a secret value), the delay measured by the attacker can now leak the victim's secrets.

2.3.2 Transient execution attacks

Transient execution attacks use instructions executed transiently to leak secret data. Spectre [111] attacks are a class of side-channel attacks that exploit speculative execution. They use speculative loads to leak sensitive information across security boundaries. By manipulating the CPU's branch prediction mechanism, an attacker can force the execution of speculatively loaded instructions that access sensitive data. Even though these instructions are eventually discarded, the accessed secret can be leaked before the mis-speculation or fault is discovered. Leaking the secret can be done using existing side channels, such as cache timing (by causing persistent secret-based cache changes) or contention-timing (by

accessing shared resources in a secret-dependent manner). Meltdown [122] is a similar attack that relies on exploiting *faults* rather than speculation to force transient execution of malicious instruction sequences.

2.3.3 Data-oblivious code

Data-oblivious code (also called constant-time code) employs a set of programming paradigms used in cryptographic libraries in order to mitigate timing side-channel leakage. The idea is to prevent sensitive data from being used in ways that affect execution time, which can manifest in two ways: control flow, and data flow. The techniques employed to prevent such leakage are called control-flow linearization (CFL) and data-flow linearization (DFL), respectively.

For control flow, state-of-the-art solutions use a program counter security model (PC-security) [139]. PC-security states that the program counter cannot become sensitive-data-dependent at any point during the execution of the program. This effectively means that an adversary cannot use a trace of the program's execution to infer sensitive data values. PC-security prevents using sensitive data to select conditional branches or determine whether a fault occurs. CFL attempts to provide PC-security by executing both paths of a sensitive conditional branch (a real path and a decoy path), while maintaining correct functionality of the program. *Predicated execution* does this by maintaining a predicate throughout execution that represents whether this path is a real or decoy path. Every sensitive operation is then masked with this predicate to ensure that it only changes the program state if it is on the real path. *Transactional execution*, on the other hand, executes both paths as-is but attempts to buffer and then discard state updates from decoy paths.

PC-security alone, however, is not sufficient. Sensitive-data-dependent data flows can occur that affect execution time. For example, some CPU implementations of floating point instructions have a variable number of cycles that depends on the operand values. If sensitive values are used as operands to such instructions, the execution time (in cycles) can leak the values to an adversary. Furthermore, memory accesses can have different latencies depending on whether the requested address is cached. A leak can therefore also occur if the addresses of memory accesses are sensitive. One simple DFL technique for instructions with a variable number of cycles is software emulation; the unsafe CPU instruction is replaced with a safe instruction sequence that performs the same functionality. For memory accesses, DFL is more difficult. The access latency must be indistinguishable for the set of all possible addresses that could be accessed at this point in the program execution. For arrays, one way to achieve this is simply by fetching the entire array into the cache before

each access. This ensures that each access, irrespective of the sensitive address, will always have the cache-hit latency. Another more general approach is to identify, for each memory instruction, the set of all possible memory addresses that can be accessed by the instruction, and simply access the entire set, masking away all values except the desired one.

Implementing data-oblivious code by hand is tedious and error-prone. Compilers can optimize away operations they deem redundant, even though they might be necessary to prevent side-channel leakage. They can also produce binaries that use instructions with sensitive-data-dependent behavior as described above. To this end, there has been significant efforts to 1) automate data-oblivious programming using specific compiler transformations, and 2) enforce data-oblivious invariants during execution. I discuss the state-of-the-art solutions related to data-oblivious execution in Section 9.2.

2.4 Trusted execution environments

As mentioned in Chapter 1, trusted execution environments (TEEs) are used in outsourced computing scenarios to protect data integrity and confidentiality against direct access. TEEs isolate trusted applications (TAs)—programs within TEEs—from software outside the TEE as well as from other TAs. Software running outside the TEE is called the rich execution environment (REE) and includes the OS. In addition to TA isolation, some TEEs also provide *remote attestation* to assure clients that the code and configuration of server-side components are what they expect, as well as the ability to securely and persistently store data in the REE.

TEEs contain a root of trust for attestation, typically in the form of a unique attestation key embedded in the hardware at the time of manufacture. The remote attestation protocol first authenticates the hardware by having the server prove that its TEE possesses this key, which is certified by the device manufacturer. After authentication, remote attestation is provided by "measuring" the system: the code, configuration, and state of the system are checked by the TEE to assure the client that they are as expected.

Examples of commercial TEEs are Arm TrustZone [154] and Intel SGX [47], each implementing TEEs in a different manner. TrustZone uses a single bit to split the processor state and data into a "secure world" and a "non-secure world". A security monitor, which is a piece of software within the trusted computing base (TCB), is used to switch between the two worlds securely. Since only a single bit is used, TrustZone TAs share the same security domain by default and require a TEE OS to isolate them from one another. Intel SGX, on the other hand, provides isolation between TAs, called enclaves in Intel SGX terminology,

by default. In SGX, any data moved outside the CPU chip (e.g., to main memory) is encrypted with an enclave-specific key. Intel SGX also provides code attestation. The integrity of enclave code is checked before it is executed by hashing the loaded executable and comparing the hash to an expected value.

Chapter 3

CacheSquash

The contents of this chapter are based on the following publication:

Hossam ElAtali and N. Asokan. "CacheSquash: Making caches speculation-aware". In: *arXiv preprint: 2406.12110* (2025). DOI: 10.48550/arXiv.2406.12110

3.1 Introduction

As modern computing systems become increasingly vulnerable to remote attacks that compromise sensitive data, one of the most insidious threats comes from side-channel leakage that allows adversaries to infer sensitive information without direct access to memory. Among these threats, transient execution attacks such as Spectre and Meltdown represent a particularly challenging class of vulnerabilities that exploit the performance-oriented design of modern processors to extract sensitive data through microarchitectural side channels.

To address RQ1 —how we can protect data confidentiality against side-channel leakage with negligible overheads—we must confront a fundamental tension between security and performance in modern processor design. Hardware-level features like speculative execution, designed to improve performance and largely invisible to software, can be exploited by remote adversaries to extract secret data using known side-channel techniques. Unlike memory safety bugs, which often lead to visible crashes when triggered, flaws in side-channel protection are usually silent and may remain undetected until successfully exploited.

Speculation is a fundamental technique employed in modern central processing units (CPUs) to optimize performance by predicting and executing instructions ahead of time.

Correct predictions eliminate stalls in the processor pipeline, providing significant performance gains. An incorrect prediction, or *mis-speculation*, causes the offending instructions to be squashed, their results discarded and the pipeline flushed to restart execution at the correct location. However, while speculation is tightly integrated into CPU cores, the cache hierarchy in modern CPUs remains largely unaware of speculative state.

This creates a low attack barrier for Spectre-class attacks. Spectre attacks exploit speculative execution to leak sensitive information, such as cryptographic keys, by training the CPU's branch-prediction mechanism to transiently access architecturally-inaccessible secrets in memory. The attack consists of three steps: accessing the secret, transmitting it through a side channel (such as changing cache state), and receiving/extracting it from the side channel (such as probing cache state). Critically, cache state changes caused by the transmit step persist even after mis-speculation is detected, leaving observable traces that remote adversaries can still extract.

Prior solutions [216, 106, 165, 120, 219] to transient execution attacks provide strong security guarantees but come at significant performance costs, as described in RQ1. In situations where there can be no compromises on performance, it is more desirable to deploy a security mechanism that hardens against side-channel leakage without any overhead rather than one that completely prevents side-channel leakage at high overheads. However, this is where existing defenses fall short.

To address this challenge, I propose CacheSquash, a novel software-transparent mechanism that hardens against transient execution attacks at *near-zero overheads* by making the cache hierarchy speculation-aware. CacheSquash works by cancelling outstanding read requests to the cache hierarchy that the CPU has determined to be mis-speculated. Once mis-speculation is detected, the CPU sends a cancellation that propagates down the cache hierarchy and prevents changes to caches that have not yet received a response, thereby reducing the overall change to cache state and reducing the likelihood of success for transient execution attacks. CacheSquash only requires minimal control-logic changes to the CPU's load-store unit and the caches' miss-handling circuitry, with *no additional state storage*, unlike prior filtering or "cache-undo" approaches [5, 164]. It is applicable to any instruction set architecture (ISA) and requires *no changes to software or external hardware interfaces*.

I implemented CacheSquash in gem5 [27] and evaluate its performance under different configurations on the SPEC CPU 2017 [177] and PARSEC [26] benchmarks. I also evaluate its efficacy against various Spectre proofs-of-concept (PoCs) and provide an analysis on its efficacy against real-world attacks.

My contributions are:

1. CacheSquash, an **ISA-agnostic** mechanism for cancelling read requests upon squashing, requiring **no changes to software or external hardware interfaces** (Section 3.3),
2. its implementation in gem5 (Section 3.4)
3. performance evaluations showing a negligible geometric mean overhead (0.48%) on SPEC CPU 2017, and a geometric mean **speedup** of 2.06% and overhead of 0.37% on the medium and large PARSEC benchmarks, respectively, (Section 3.6), and
4. case studies showing that CacheSquash is **effective** against several Spectre PoCs (Section 3.5).

3.2 Problem description

3.2.1 Goals & Objectives

Ideally, desiderata for speculation-aware caches are:

R1—Performance: *no negative run-time performance impact on realistic workloads.*

R2—Software Compatibility: *require* no changes *to software and be fully compatible with existing program binaries.*

R3—Hardware Compatibility: *require* no changes *to external interfaces (e.g., DRAM) or hardware components, other than the CPU.*

R4—ISA Compatibility: *be applicable to any ISA.*

R5—Effectiveness: *reduce leakage of secret data through cache state changes.*

We define a *cache change* metric CC for effectiveness (R5):

$$CC = \frac{\sum_i^K N_i \times (K - i + 1)}{N_{total} \times \sum_i^K i} \tag{3.1}$$

K is the number of cache levels in the system. N_{total} is the total number of squashed access and transmit instructions in a program. N_i is the number of squashed access and transmit instructions that cause a change in the ith cache. We assign more weight to changes to caches closer to the CPU because they are easier to exploit via cache timing [124]. $CC \in [0,1]$. Any non-zero value implies that Spectre attacks may succeed.

3.2.2 Threat model

We consider the strongest Spectre threat model, where the attacker and the victim execute within the same process, sharing the same address space and having the same process context. The attacker is unable to directly access the victim's secret (e.g., due to in-process isolation mechanisms such as sandboxing), but can train the branch predictor to access the secret speculatively. This corresponds to "same-address in-place" (SA-IP) as defined by Canella et al [34]. A defense that works under this strong threat model, is also secure under weaker threat models such as where the attacker and victim are in different processes. We only consider cache-timing channels; other channels, e.g., contention-based channels, are out of scope.

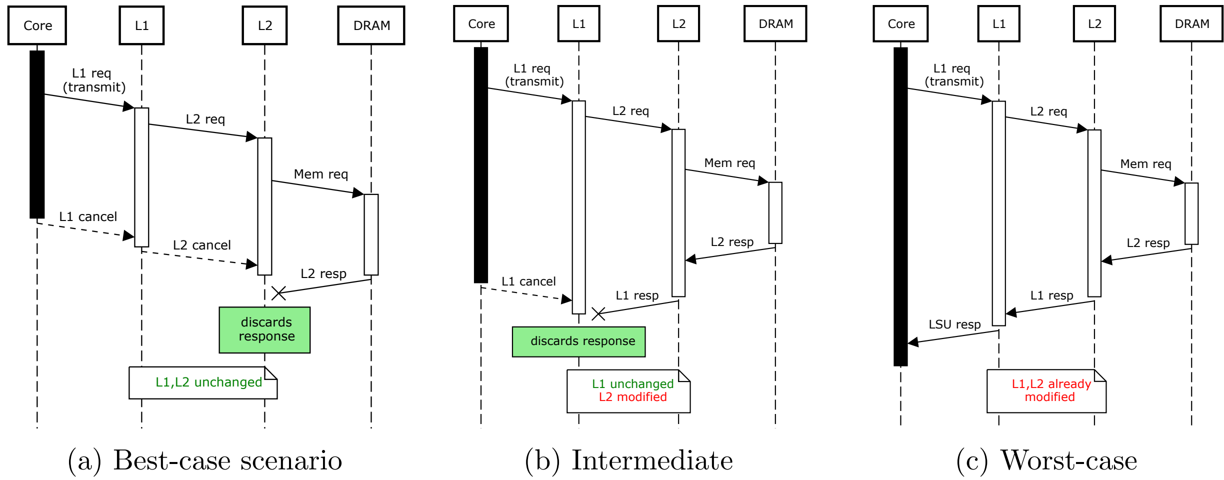
3.3 CacheSquash: Design

The idea behind CacheSquash is to minimize cache state changes by issuing cancellations to outstanding speculative read requests as soon as they are squashed. Whenever a cache receives a cancellation for an outstanding request, it drops the request from its MSHRs[1] (and ignores any responses it receives for it in the future), and, if appropriate, forwards the cancellation to caches downstream.

The final effect of the cancellation depends on the state of the read request/response within the cache hierarchy at the time the cancellation is sent. We present all possible cases in Figures 3.1a to 3.1c. In the best case, Figure 3.1a, the cancellation reaches the last-level cache (LLC), L2, before it receives a response from memory. This prevents any changes to the cache hierarchy as any response received by the LLC from memory is ignored; subsequently, the LLC does not provide further responses to caches upstream (which have already cancelled the request and the corresponding evictions[2] themselves).

[1]CacheSquash works with any mechanism for keeping tracking of outstanding requests. We refer to MSHRs here as they are the most prevalent.

[2]Evictions do not require special handling as they are only completed when the response arrives, rather than when the MSHR is first allocated.



(a) Best-case scenario (b) Intermediate (c) Worst-case

Figure 3.1: The solid black activation bar (for core) represents the speculation window. The hollow activation bars (for caches) represent lifetimes of the requests' miss status holding registers (MSHRs). The hollow activation bar (for DRAM) represents the memory-only access latency.

In the worst case, Figure 3.1c, the cancellation is either never made (because the response is received by the CPU core before the speculation window ends), or it reaches L1 after it has received a response. If the CPU has more than one cache level, other intermediate cases between Figure 3.1a and Figure 3.1c can occur: Figure 3.1b shows a cancellation reaching L1, but not L2, before the response; only L2 is modified by the request. If this request is from a Spectre transmit instruction, only attacks targeting the LLC can succeed; those targeting L1 will not.

3.3.1 Cache flow chart

Figure 3.2 shows the CacheSquash flowchart. It works with any cache coherence protocol (including both snooping and directory-based protocols). Beside the addition of handling cancellations, which is only relevant when a cache has an outstanding request to the level below it, the rest of the coherence protocol remains unchanged. Outstanding requests can either be reads (if the block is currently in the invalid state) or upgrades (if the block is valid, but does not have the required permissions, e.g., writable). As soon as we receive a cancellation, we only need to check whether there is a corresponding MSHR (MatchMSHR), and if so, remove the cancelled request from it. If the MSHR then becomes empty (i.e., no

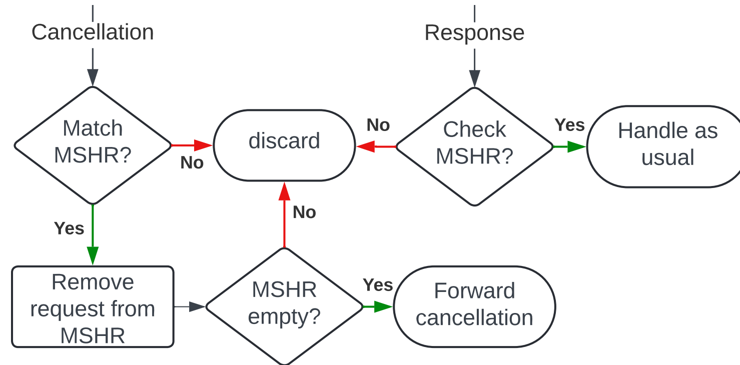


Figure 3.2: CacheSquash flow chart.

other requests are waiting for this cache block), we can send a cancellation to the lower cache level. For a response, if a matching MSHR is not found, it is discarded (CheckMSHR).

MatchMSHR

MatchMSHR searches the MSHR queue for a match, i.e., one that has the same cache block address. The circuitry to perform this search already exists in modern caches. It is required to check if incoming misses match an outstanding MSHR, and coalesce requests to the same block. It is possible for cancellations to find no matching MSHR due to simultaneous transfers of responses and cancellations.

Simultaneous response and cancellation. Cache buses can have several channels, allowing simultaneous bidirectional communication, e.g., TileLink [176] or Arm AMBA CHI [11]. This means that a cancellation can be received while a response is being sent. This can occur in two cases: 1) if the original request hits in this cache, or 2) the original request misses, but, before the cancellation arrives, the cache receives a response and the MSHR is serviced and freed. The cancellation in both cases will have no corresponding MSHR and therefore must be discarded. This requires an MSHR search to detect.

CheckMSHR

Without CacheSquash, MSHRs are locked to a single downstream request until a response is received. The downstream request contains an MSHR index which is copied into the response by the downstream cache. This makes it easy to determine the exact MSHR corresponding to a response. With CacheSquash, this assumption no longer holds. The

cache must double-check the designated MSHR when receiving a response to ensure that it has not been freed and possibly reassigned to another block. This can happen either due to simultaneous response and cancellation transfer (as described above), or due to responses from memory.

Responses from memory. CacheSquash does not require modifications to DRAM or memory controllers (satisfying R3). Cancellations are not supported on the memory bus; LLCs must therefore not forward cancellations to memory. Thus, a request from the LLC to memory will eventually receive a response, even if it is cancelled in all caches. LLCs need to detect whether the MSHR in the response still corresponds to the original request, hence the need for CheckMSHR.

3.3.2 Forwarding cancellations upstream

Cache coherence protocols ensure that requests for a cache line get an up-to-date response. The protocol can probe upstream caches for dirty cache lines, causing them to allocate an MSHR while they probe caches further upstream. With CacheSquash, forwarding cancellations is thus required to ensure these MSHRs are freed. Cache coherence protocols are mainly split into two groups: snooping-based and directory-based. In snooping, caches "snoop" the bus to detect requests that require their intervention. This is usually done using additional circuitry and can span multiple cache levels. For CacheSquash, this circuitry can also be used to snoop on cancellations. In directory-based protocols, a directory tracks the caches holding each block. A cache receiving a cancellations must consult its directory and forward a cancellation to all caches with an outstanding MSHR of a corresponding target.

3.3.3 TLBs & I-Caches

An important cache-like structure in CPUs is the translation lookaside buffer (TLB), which caches virtual-to-physical address translations. Prior work has also shown that TLBs can be used to leak information in a way similar to cache timing attacks [82]. Therefore, with CacheSquash, we also send cancellations for mis-speculated address translations that result in page-table walks, reducing changes to TLB state.

The instruction cache is also affected by (mis-)speculation since the branch predictor speculates on which path of instructions will be executed. We therefore send cancellations to the instruction cache when an instruction fetch is mis-speculated.

3.4 CacheSquash: Implementation

We implement CacheSquash on gem5 [27], a cycle-accurate computer system simulator. gem5 includes a speculative out-of-order CPU model, O3, and supports arbitrary cache configurations. It provides two separate cache hierarchy implementations: classic and ruby. Ruby caches are newer and allow configurable cache coherence protocols. However, they currently do not support cache maintenance operations, such as flushing, and therefore cannot be used with the Flush+Reload cache timing attack. As this is the attack used by the majority of the publicly available Spectre PoCs, we choose to implement CacheSquash on the classic caches. The classic caches have a fixed snooping-based cache coherence protocol, so we implement cancellation snooping as described in Section 3.3.2.

MatchMSHR & CheckMSHR. The logic for MatchMSHR already exists for handling cache misses. We use the same latency of searching the MSHR queue for cancellations. CheckMSHR adds new functionality. However, since no search is required, only a simple check, we assume it is combinational logic and can be performed in the same cycle. CheckMSHR thus incurs no additional latency.

O3 CPU. In addition to modifying the caches, we add CacheSquash support to gem5's O3 model. Whenever an instruction is squashed, the load-store unit checks if there are any outstanding memory requests for the instruction, and if so, sends a cancellation to the cache hierarchy. Note that we do not make any changes to ISA-specific CPU models; CacheSquash is ISA-agnostic, satisfying R4.

3.5 Security evaluation

3.5.1 Case studies

We present case studies with two Spectre variants to empirically show the effectiveness of CacheSquash. We analyze the first (Section 3.5.1) to show the CPU events occuring throughout a Spectre attack and how CacheSquash affects them. For the second, we only report our findings for brevity; the attacks use the same access-transmit-receive mechanism. Note that since cancellations can be used for any squashed memory request, regardless of the speculation condition, CacheSquash is equally applicable to *all Spectre variants*.

All PoCs define a secret string as the target of the attack. For each character of the string, all PoCs continue attempting to leak the character until the extracted value matches certain criteria, e.g., value is a valid English ASCII character. This means that when attacks are successful, the program terminates quickly. On the other hand, if attacks do not succeed, the program runs until it times out. The default timeout periods for the PoCs are infeasibly long when run on gem5. We therefore shorten all timeouts to allow the simulation to complete in a reasonable amount of time. To ensure a fair comparison, we verify that, without CacheSquash, both PoCs are still able to leak the secret using the shortened timeouts.

Google SafeSide – Spectre PHT

SafeSide [80] is a Google code repository containing several Spectre and Meltdown [122] PoCs. We use the `spectre_v1_pht_sa` PoC, which mistrains the pattern history table (PHT) and then exploits it to achieve a bounds check bypass (BCB). The PHT is a component of the CPU's branch predictor in charge of guessing whether a branch will be taken. The PoC uses Flush+Reload to transmit and receive the secret.

```
1  # bounds check
2  404bec: jae    404bbf <main+0xca>
3  404bee: movsbq 0x0(%r13,%rax,1),%rax
4  404bf4: imul   $0x71,%rax,%rax
5  404bf8: add    $0x64,%rax
6  # access secret: data[local_offset]
7  404bfc: movzbl %al,%eax
8  404bff: add    $0x1,%rax
9  404c03: mov    %rax,%rsi
10 404c06: shl    $0x6,%rsi
11 404c0a: add    %rsi,%rax
12 404c0d: shl    $0x6,%rax
13 404c11: add    0x28(%rsp),%rax
14 # transmit: timing_array[secret]
15 404c16: movzbl (%rax),%eax
16 404c19: jmp    404bbf <main+0xca>
```

Listing 1: `spectre_v1_pht_sa` x86 assembly extract.

Listing 1 shows the x86 disassembly of the speculatively executed instructions. Lines 3-15 are executed speculatively until the failed bounds check on line 2 is detected and the instructions are squashed. Before the squash occurs, the secret is accessed (line 7) and transmitted across a side channel by modifying the cache state (line 15). Extracting the

secret from the cache state is done non-speculatively afterwards. We now describe four experiments to evaluate the PoC with and without CacheSquash.

Parameter	C1	C2
Core count	2	2
Core frequency (GHz)	3	0.1
Private L1I/D size (kB)	32	32
Shared L2 size (kB)	512	512
L1I/L1D/L2 associativity	8/8/16	8/8/16
L1I/L1D/L2 latency (cycles)	4/4/14	80/80/80

Table 3.1: Configurations used in case study experiments. For C1, realistic values are used for L1 and L2 based on Intel IceLake [65]. C2 is modified from C1 to intentionally make CacheSquash ineffective. For main memory, we use 3GB of dual channel DDR4-2400.

Experiment 1 – C1, Baseline. We first run the PoC in gem5 using the O3 model and the C1 gem5 configuration shown in Table 3.1 without CacheSquash. The PoC is able to extract the entire secret. We dump all load-store unit and cache events from gem5, and identify and extract events related to pairs of the access and transmit instructions in Listing 1. Each pair represents a single Spectre attack. Figure 3.3a plots relevant events for the transmit instruction of each attack, showing that for all attacks where the transmit instruction is executed, the squash occurs before any response is returned to the LLC. Further, in almost all of those cases, there is a significant delay between the squash occurring, and the LLC receiving a response. This provides an opportunity for cancellations, and hints that CacheSquash might prevent this attack (see our next experiment). Exceptions to this are the cases highlighted with black boxes. These can pose a problem because cancellations might not reach L2 in time before the response from memory. We do not find this in our following experiments with C1, but we force this situation to occur in Experiment 4.

In attack 1 ($y = 1$ in Figure 3.3a), the transmit instruction is never executed. This occurs because the secret is not yet cached and the access instruction does not complete in time to allow the transmit instruction to execute. But subsequent attacks (e.g., attack 3) can access the now-cached secret quickly and thus have enough time to execute the transmit. In attack 3, the access instruction misses in L1 but hits in L2.

Experiment 2 – C1, CacheSquash. We run the PoC with C1 and CacheSquash enabled, and indeed find that the program times out without leaking any secrets. Our analysis

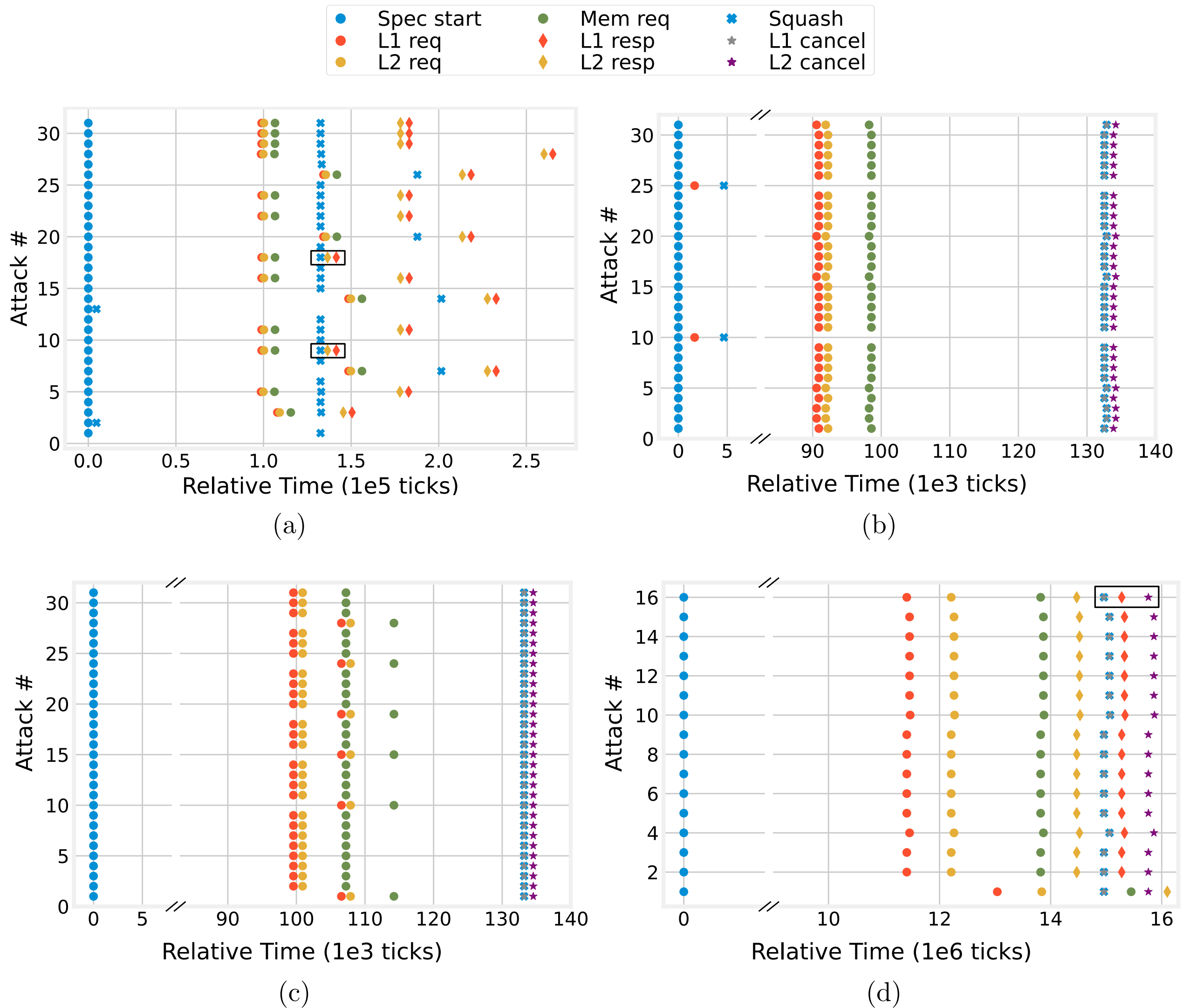


Figure 3.3: CPU events for Spectre attack instances. Each row represents a single attack, the y-axis showing the attack number. All event times are relative to speculation start: for each attack, events are shifted to make speculation start at t=0. The plots show a) transmit instructions for all attacks in Experiment 1, b) access instructions for attacks #1–31 in Experiment 2, c) transmit instructions for attacks #1–31 in Experiment 3, and d) transmit instructions for all attacks in Experiment 4.

leads to an interesting discovery: *no transmit instructions are executed.* Upon further

investigation, we find that this occurs because CacheSquash cancels the *access instructions*, preventing even the first step of the attack. In its entire run, the PoC only finds the secret in the L1D cache twice, and in both cases, speculation ends before the transmit is executed. In all other cases, the attack has the same result as attack 1 in Experiment 1 (Figure 3.3a). This is shown in Figure 3.3b, where we plot the access instructions instead, and see that they hit in L1D only twice. Note that because Experiment 2 times out, the total number of attacks (102) is much larger than in Experiment 1 (31). However, for clarity, we only plot attacks #1–31.

Experiment 3 – C1, CacheSquash, secret cached. Here, we want to test the effectiveness of CacheSquash even when the secret is cached and the access instruction completes. We modify the PoC to access the secret non-speculatively on every iteration. Running the PoC again results in a timeout with no secrets leaked. Our analysis confirms that the access instructions now hit in L1D, and the transmit instructions execute. We show all relevant events for the transmit instructions in Figure 3.3c. Despite the execution of the transmit, no Spectre attack succeeds. This is because the cancellations always reach the caches before the response, even for the last-level L2 cache, hence preventing any changes to cache state.

Experiment 4 – C2, CacheSquash, secret cached. Here, we intentionally change the system configuration to C2 to reduce the effectiveness of CacheSquash. We drastically increase the L1 and L2 latencies, and reduce the CPU clock frequency to 100MHz, shown in red in Table 3.1. Reducing the clock frequency effectively increases the speculation window relative to main memory latency (as main memory uses a separate clock), making responses from memory more likely to arrive within the speculation window.

Running the modified Spectre PoC (with secret caching), we find that the entire secret is indeed extracted. Figure 3.3d shows the relevant events. The key change, highlighted with a black box, compared to Figure 3.3c is that the response reaches L2 before the cancellation. This is not the case for L1; however, the cache state change in L2 alone is sufficient for the extract step of the Spectre attack to succeed. This is due to the large difference in hit access latencies for L1 and L2 (80 vs. 160, respectively) caused by the large latencies used for C2.

Here, the PoC requires fewer attacks than in Experiment 1 to extract the secret because we use the modified PoC where we intentionally cache the secret before each Spectre attack.

Effectiveness. Table 3.2 shows the cache change metric, CC (Equation (3.1)) for all experiments. Experiment 1 gives a value of 1 because CacheSquash is not enabled and any transmit instruction executed results in changes to all caches. For experiments 2 and 3, we get a value of 0 because no cache changes occurred. Experiment 4 shows that when transmit instructions manage to cause a partial change to cache state, the value of CC is between 0 and 1.

Experiment	N_1	N_2	N_{total}	**CC**
1	29	29	29	1
2	0	0	104	0
3	0	0	204	0
4	0	15	32	0.234

Table 3.2: The values of CC for all experiments.

Google SafeSide – ret2spec

As before, we evaluate CacheSquash on the `ret2spec_sa` PoC and verify that it thwarts the attack: no part of the secret is leaked. Ret2spec, also called Spectre-RSB [115, 34], targets the return stack buffer (RSB), which predicts the addresses of return instructions. It exploits the fact that the RSB has a limited number of entries, and must remove addresses once it is filled due to deeply nested functions, causing mispredictions of the return addresses. These mispredictions lead to speculatively executed instructions that are abused by ret2spec to leak the secret.

3.5.2 Security limitations

Cache hits for transmit instructions

Flush+Reload uses flushing to prepare caches before transmit instructions are executed, preventing the cache line from being present at any level. However, an attacker can instead use evictions [149] to remove the data from one level but keep it in lower levels. For example, an attacker can evict the data from L1 but not L2 before launching the Spectre attack, and later use the timing difference between an L1 access and an L2 access to leak the secret. While CacheSquash is less effective against such attacks, they are also more difficult to

launch due to the smaller timing difference. Further, for architecturally inaccessible secrets that are flushed out on context switches, the attacker must first successfully cache the secret, which is made difficult with CacheSquash (see Experiment 2).

Windowing gadgets

We demonstrated the effectiveness of CacheSquash against Spectre PoCs, which were originally created to simply show the feasibility of the attack. Attackers can use several techniques to make attacks more robust. One is to increase the speculation window [136]. Gigerl [78], Mambretti et al. [130] and Xiao et al. [212] identify empirical limits on speculation window sizes achievable on different platforms via different instructions for speculation conditions.

While increasing speculation window size can reduce the effectiveness of CacheSquash, attackers must also overcome other practical challenges that maintain CacheSquash's effectiveness. In the PoCs, both the victim and attacker code are under our control. In practice, attackers are forced to rely on speculative *gadgets* [96, 101, 208, 25] present in the victim's code, in a manner similar to return-oriented programming (ROP) gadgets [172]. This includes *disclosure* gadgets, used to access and transmit the secret, and *windowing* gadgets, needed to increase the speculation window as described above. Significant prior work has been done to investigate and reduce the availability of speculative gadgets in critical software targets such as the Linux Kernel [101, 95]. As a result, 1) attackers are increasingly forced to use less-than-ideal *disclosure* gadgets that can have many redundant instructions (reducing the effective available speculation window), and 2) fewer windowing gadgets are available, making it harder to circumvent CacheSquash. CacheSquash serves as complementary work to reduce the effective attack surface of critical software targets.

Speculative-interference attacks

A crucial requirement for CacheSquash is that the transmit instruction is speculative, and is therefore squashed when the speculation window ends. In speculative interference attacks [20], however, this is not the case. Instead, speculative execution is used to affect the order of *non-speculative* loads/stores, resulting in a cache state difference that can later be measured to leak the secret. As non-speculative memory requests cannot be cancelled, CacheSquash cannot thwart such attacks. We consider them out-of-scope and rely on orthogonal defenses, e.g., full pipelining to prevent speculative instructions from affecting older ones, as suggested by Behnia et al [20].

Non-cache-based side channels

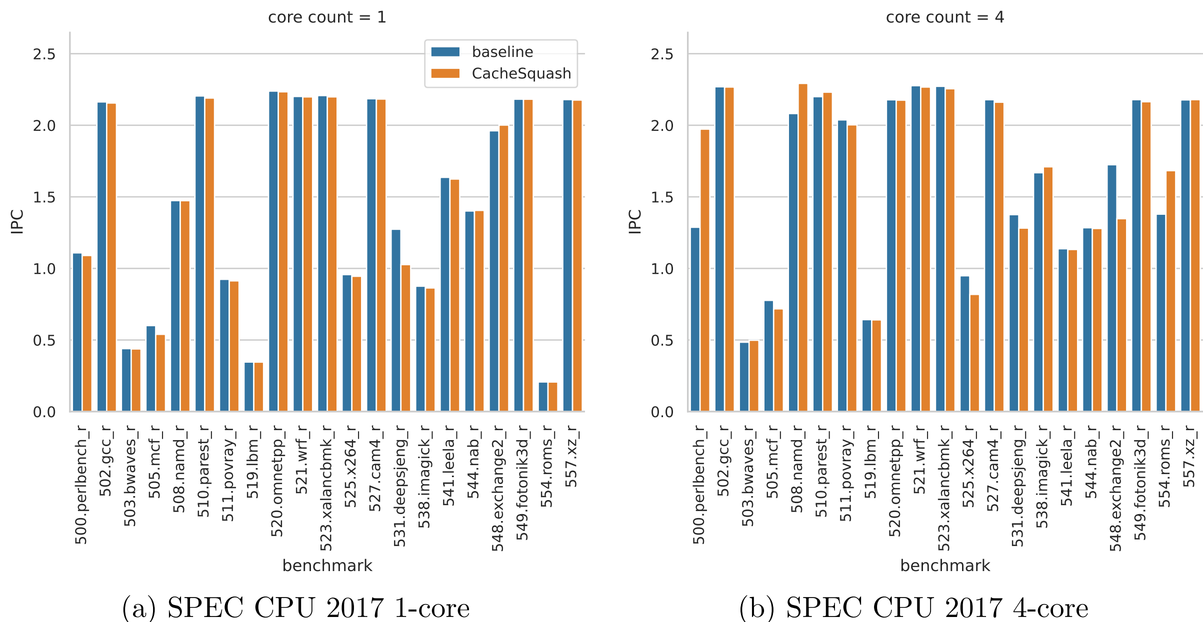
CacheSquash works by reducing *persistent* secret-dependent changes to *cache state*. As a result, CacheSquash only covers cache-based side channels. Other side channels, e.g. contention-based channels [187, 224, 150], are out-of-scope, as in many invisible speculation schemes (Chapter 9).

3.6 Performance Evaluation

We evaluate CacheSquash's performance on 1- and 4-core configurations using similar parameters to InvisiSpec [216] and CleanupSpec [164]. We first run the SPEC CPU 2017 benchmarks [177] with the `ref` input size, using a warm-up period of 10B instructions and measuring the instructions-per-cycle (IPC) for the next 1B instructions. This follows standard procedure from prior work [219, 216, 164]. We exclude `507.cactuBSSN_r` as it crashes on the baseline and CacheSquash. The results are shown in Figures 3.4a and 3.4b. The geometric mean IPC overhead across all benchmarks and configurations is −0.48%. However, the results show a high geometric standard deviation of 1.10. As our simulations are deterministic, running the benchmarks more than once produces identical results. We therefore evaluate further using the PARSEC benchmarks [26]. We run them to completion with the medium and large input sizes and measure the number of gem5 ticks (Figures 3.5a to 3.5d) yielding a geometric mean *speedup* of 2.06% and overhead of 0.37% with lower geometric standard deviations of 1.07 and 1.008, respectively.

Parameter	**Value**
Core count	1,4
Core frequency (GHz)	3
Private L1I/L1D size (kB)	32/64
Shared L2 size per core (MB)	2
L1/L2 associativity	2/8
L1I/L1D/L2 latency (cycles)	1/2/20

Table 3.3: Parameters used in performance evaluation.



(a) SPEC CPU 2017 1-core

(b) SPEC CPU 2017 4-core

Figure 3.4: Results for SPEC CPU 2017 in IPC with `ref` input size on 1- and 4-core configurations of baseline and CacheSquash.

3.7 Discussion & Future Work

3.7.1 Meltdown

CacheSquash provides support for read request cancellations *regardless of the reason for cancellation*. While we tackle speculative execution in our implementation, CacheSquash is also applicable to fault-based transient attacks such as Meltdown [122]. Any transmit instruction executed transiently during the Meltdown attack can be cancelled once the fault is detected, thereby reducing cache state changes and reducing the attack's chance of success.

3.7.2 Cancellation broadcasts

Dedicated circuitry, similar to that used for snooping, can be added to the CPU die to broadcast cancellations to all caches, even if a snooping protocol is not used. This can drastically improve the security provided by CacheSquash, by eliminating the dependency on cache forwarding latency. However, this adds complexity to cancellation handling because

lower-level caches would now get cancellations even if the corresponding upstream MSHR is not empty. A mechanism must therefore be added to allow lower-level caches to track upstream MSHRs and only act on cancellations once the upstream MSHR is deallocated.

3.7.3 Cancellation of memory bus transactions

One of the goals of CacheSquash was to explicitly avoid changes to external modules and interfaces such as main memory to enable backward compatibility. However, introducing cancellations to memory buses can be an opportunity to improve system performance. By aborting transactions that are no longer needed, we can free up the memory bus for other transactions. Furthermore, for memories with integrated on-chip caches, this can improve security by cancelling changes to the on-chip cache.

3.7.4 Overlapping cancelled and uncancelled requests

If there are n requests waiting for the same cache block, canceling up to $(n-1)$ of them will have no effect on the cache as the MSHR must still be serviced. Information leakage can occur if the *first* request to allocate the MSHR is cancelled, but the MSHR cannot be deallocated due to the existence of other non-speculative targets that arrived later. There is a timing difference between when the response arrives in this case, and when the response would have arrived had there not been the first request (i.e., the MSHR was instead allocated by the non-speculative second request). While this timing difference can theoretically be used to leak information, our threat model assumes that this second request is not under the attacker's control, and they cannot determine the time at which it occurs (and therefore cannot accurately measure the timing difference). Note that if the attacker can control this non-speculative request, they have no need for a Spectre attack and can simply use a non-transient cache timing attack.

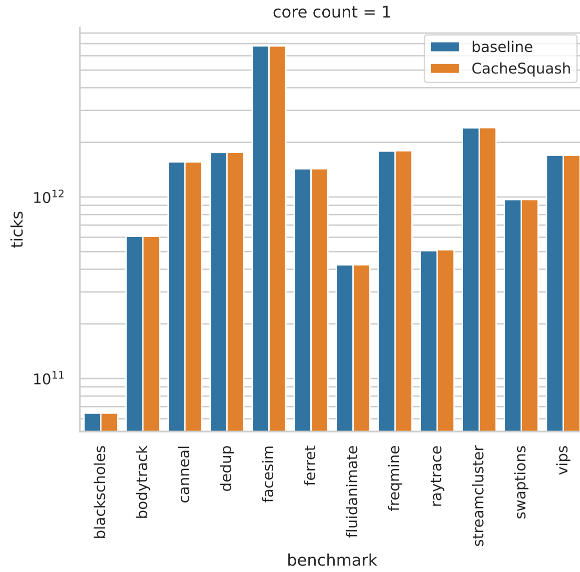
3.8 Related work

Invisible speculation mechanisms [216, 106, 165, 120, 5] attempt to hide speculative side effects until they are determined to be non-speculative. While hidden, speculative changes are stored in shadow structures which are invisible to the rest of the non-speculative system. CleanupSpec [164] takes an "undo", rather than a hiding, approach, allowing speculative changes to be seen by the system, but undoing them on squashes. However, prior schemes

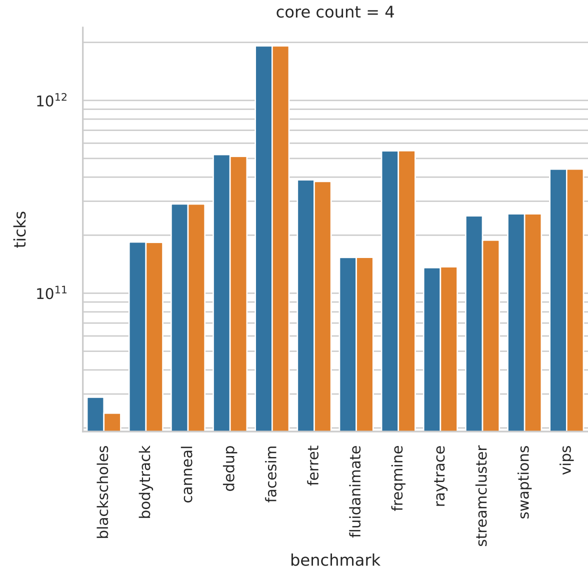
suffer from significant overheads, (e.g., $21 - 72\%$ for InvisiSpec) and/or require the addition of expensive on-chip storage to track speculative changes (e.g., L0 in MuonTrap, buffers in CleanupSpec). In comparison, CacheSquash does not require any structures to track speculative changes and has negligible overheads.

Speculative Taint Tracking (STT) [219] is another Spectre mitigation technique that taints data loaded by Spectre access instructions and delays any instructions that use it until the access instruction becomes non-speculative. While STT is not limited to protecting only cache-based side channels, it can result in significant overheads ($8.5 - 14.5\%$) compared to CacheSquash and does not cover the case where only the transmit, but not the access, instruction is speculative.

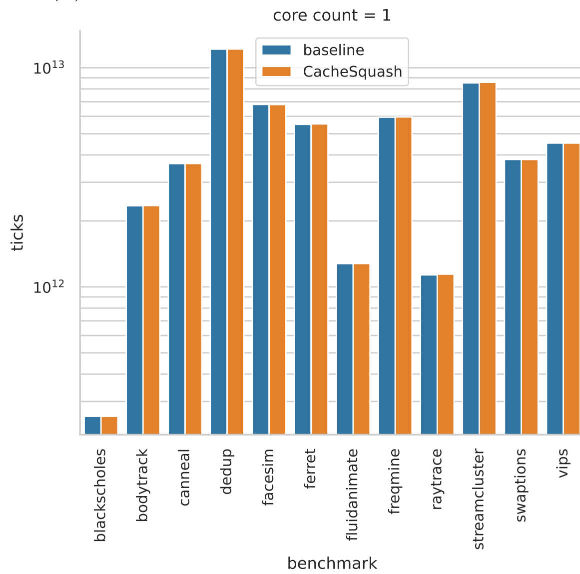
Prior work that defends against non-speculative cache timing attacks is discussed in Section 9.3.



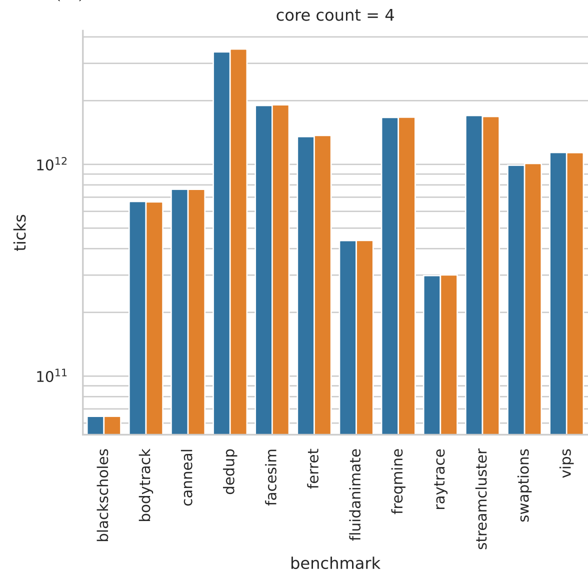
(a) PARSEC 1-core medium input size



(b) PARSEC 4-core medium input size



(c) PARSEC 1-core large input size



(d) PARSEC 4-core large input size

Figure 3.5: Results for PARSEC in ticks with medium and large input sizes on 1- and 4-core configurations of baseline and CacheSquash.

Chapter 4

BliMe

The contents of this chapter are based on the following publication:

Hossam ElAtali et al. "BliMe: Verifiably Secure Outsourced Computation with Hardware-Enforced Taint Tracking". In: *Proceedings of the 2024 Network and Distributed System Security Symposium*. NDSS '24. San Diego, CA, USA: Internet Society, 2024. ISBN: 1-891562-93-2. DOI: 10.14722/ndss.2024.24105

4.1 Introduction

As computing systems become increasingly interconnected and data processing is often delegated to remote or untrusted environments, protecting sensitive data from both direct access and side-channel leakage has become critical. Outsourced computation, where clients send sensitive data to remote servers for processing, exemplifies this challenge. While cost-effective, such arrangements expose client data to multiple threat vectors: malicious or vulnerable software can compromise data through run-time attacks, and even trusted software may inadvertently leak information through side channels during execution.

Traditional cryptographic approaches like fully-homomorphic encryption (FHE) provide strong confidentiality guarantees by allowing computation on encrypted data, ensuring the server never processes plaintext. However, FHE incurs prohibitive performance overheads—orders of magnitude worse than native execution. Hardware-assisted security mechanisms, such as trusted execution environments (TEEs), offer better performance characteristics while providing isolation. However, existing TEE implementations remain vulnerable to both software bugs that enable run-time attacks and sophisticated side-channel

attacks that can extract sensitive information through timing variations, cache behavior, or other observable system states.

To address RQ2 —how we can efficiently protect data confidentiality against both direct access and side-channel leakage—we propose Blinded Memory (BliMe). BliMe represents a hardware-assisted approach that provides comprehensive protection through minimal extensions to the RISC-V instruction set architecture (ISA). Unlike prior solutions that address either memory safety or side-channel resistance in isolation, BliMe simultaneously protects against both threat vectors while maintaining minimal performance overhead.

BliMe allows a client to send conventionally encrypted data to a remote server, so that the central processing unit (CPU) can decrypt and process the data without allowing it or any data derived from it to be exfiltrated from the system. Computation results are returned only after encryption with the client's key. BliMe works by having the CPU enforce a taint-tracking policy preventing client data from being exported from the system. Prior work on hardware-enforced taint tracking [218] provide an untaint instruction to extract results, implicitly assuming that software invoking this instruction is trusted, making them vulnerable to run-time or speculative execution attacks [111]. In contrast, BliMe uses a small attestable, fixed-function hardware security module (HSM) and an encryption engine to facilitate secure import and export of data between clients and servers; decryption on server-side always results in tainted plaintext. This allows BliMe to provide its security guarantees to *multiple independent clients*, who *do not need to trust server software*, including the operating system (OS). Consequently, BliMe protects not just against side channels, but even against malware and run-time attacks that allow an attacker to execute arbitrary server software. Our contributions are:

- BliMe, an architecture with a set of taint tracking ISA extensions for preventing exfiltration of sensitive data (Section 4.3).
- BliMe-BOOM, an RTL implementation of the BliMe ISA extensions, incorporated into the speculative out-of-order RISC-V core BOOM (Section 4.4).
- A performance evaluation of BliMe-BOOM showing minimal run-time, power and area overheads (Section 4.5).

4.2 Problem Description

4.2.1 Usage scenario

The scenario we target is where several clients send data to a remote server for outsourced computation. Each client starts a session with the server, sends its data, and the server invokes some *potentially untrusted, third-party* software ("server software") to perform computation on the data (possibly in combination with the server's own data). Once computation completes, the results are sent to the client. Data import/export and computation can repeat multiple times per session. Data exchange between clients and servers is secured using authenticated encryption. Multiple clients may connect to the server simultaneously, leading to multiple parallel sessions.

Execution of server software that can leak any information about client data must be prohibited, even when attackers can run malware on the server, exploit vulnerabilities in the server software, or use side channels to extract data. In other words, sensitive client data must not flow to any observable output, nor to any other client.

4.2.2 Goals and objectives

We now identify design requirements. The first relates to security.

SR—Confidentiality: *When a client provides sensitive input data to the server, no party other than that same client can infer anything about this data, other than its length.*

Malware running on the server may attempt to gain access to the data, or the software processing the sensitive data may itself be malicious, but must not be allowed to reveal sensitive data outside the system, or to anyone other than the original client.

The next requirement relates to performance.

PR—Fast execution: *The design will not significantly reduce the performance of software accepted by the CPU, compared to running the same software on a similar processor without such protections.*

It is important to ensure that any solution does not excessively degrade performance. Elimination of side channels may prevent certain optimizations, resulting in some overhead, but some high-performance security-critical software has already been hand-written in

assembly to eliminate side channels, and solutions that significantly reduce its performance may prove unsuitable in applications that make heavy use of such software.

The final requirement relates to backwards-compatibility.

CR—Backwards compatibility: *Software that does not leak sensitive data, by covert channels or otherwise, will run successfully.*

A large portion of server software does not process sensitive data. It is important from a practical point that existing software can run on the new hardware: if this is not the case, then a new software stack will be needed, greatly limiting utility.

Moreover, there is also software that handles sensitive data but that already does so safely, such as side-channel-resistant cryptographic software. It is equally desirable that this secure and well-tested software will continue to run on our hardware.

4.3 Design

4.3.1 System overview

Figure 4.1 shows an overview of BliMe. The server's software, including the OS, runs on top of a CPU that contains 1) the BliMe extensions, which enforce a taint-tracking policy on all software running on the CPU (Section 4.3.4), and 2) a BliMe encryption engine, used for data import and export (Section 4.3.3). A HSM is used to perform remote attestation, assuring the client that the server uses BliMe. The HSM is a separate fixed-function component sharing few resources with the CPU, reducing its exposure to side-channel attacks. The client knows how to verify the root of trust for attestation embedded within the HSM, which contains a key exchange engine responsible for negotiating a session key between with the client. The HSM provides this session key securely to the encryption engine without exposing it to the server software.

4.3.2 Adversary model

We suppose that the server hardware, including the BliMe extensions, encryption engine, and the HSM, is implemented correctly.

The adversary has control over all server software, including the OS, and can make it behave as it sees fit, including making inferences based on side channel information such as memory access patterns and instruction traces.

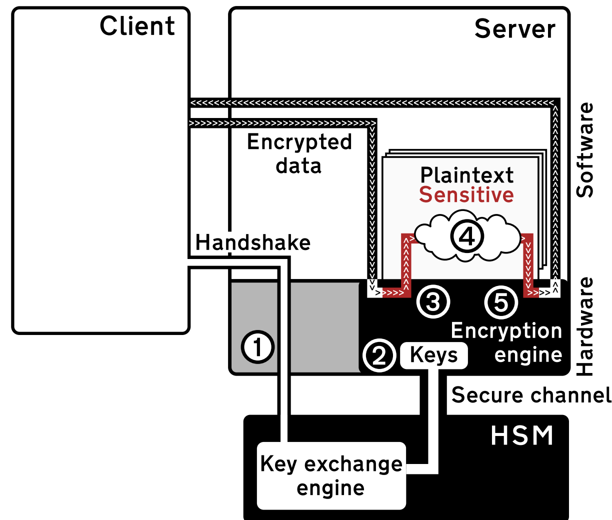


Figure 4.1: BliMe Overview. The client completes a cryptographic handshake with HSM ①, which stores the resulting shared key in a protected register for use by the encryption engine ②. The client encrypts its secret data, which the server software decrypts with the aid of the encryption engine, while atomically marking the decrypted data as blinded ③. The server software can then perform the requested computation on the resulting blinded data in a verifiably leakage-resistant manner ④, and encrypt the output using the encryption engine ⑤ for the client.

Attacks against the hardware itself are currently out-of-scope; the client assumes that the attacker cannot use physical means to make the hardware act differently from its specification. Side-channel attacks that require physical access (e.g., differential power analysis) are also out-of-scope. We discuss this in Section 8.3.

4.3.3 Protocol

In this section, we outline the steps needed to perform safe outsourced computation using BliMe.

Remote attestation and key agreement

Before sending any data to the server, a client first performs a handshake (Figure 4.1-①) with the HSM, which consists of remote attestation and agreement on a session key. Remote

attestation is provided by the HSM to assure the client that the assumptions made in the adversary model hold. It attests two properties. First, the HSM attests that it is genuine using the root-of-trust embedded within it at manufacture, as described in Section 2.4. Second, the HSM attests to the client that it is embedded in server hardware incorporating BliMe.

At the end of remote attestation, the client and HSM agree on a client-specific session key. The HSM stores this key inside the encryption engine using a secure channel (Figure 4.1-②), along with a unique blindedness tag identifying the client from which data was sourced; this is most simply implemented cryptographically using a sealing key shared by the two components. The encryption engine uses the session-specific key in two atomic functions that it exposes to the server software: *data import* and *data export*.

Data import

Once the handshake is complete, the client locally encrypts its data using the session key, and sends the resulting ciphertext to the server. The server software calls the encryption engine's data import function on the ciphertext. The encryption engine then *atomically* decrypts the ciphertext using the sealed session key, and taints the resulting plaintext by *marking* it as *"blinded"* (Figure 4.1-③). This is done by setting a *blindedness* tag attached to the data in registers and memory. The blindedness tag is an n-bit integer, which takes the value zero when its associated data is not blinded. Other values indicate that the data is blinded, and from which of $2^n - 1$ clients the data is sourced. In the minimal case where $n = 1$, this tag is a single bit that indicates whether or not the data is blinded.

> Data import atomically decrypts data using the session key and marks it as *blinded*.

Safe computation

Now, the server software can perform the requested computation on the blinded plaintext data (Figure 4.1-④). The BliMe CPU extensions apply a taint-tracking policy, limiting the operations that can be done on blinded data; this prevents the server software from directly exfiltrating the data or even leaking it through side channels. Any operations forbidden by the taint-tracking policy cause the server to fault. The policy ensures that the final results and any intermediate results derived from the data are also blinded. More details are in Section 4.3.4.

Data export

Once the computation is complete, the server software calls the encryption engine's data export function to atomically encrypt the blinded results and mark the ciphertext as non-blinded (Figure 4.1-⑤), which is done by zeroing the blindedness tag. The server software can then send the ciphertext back to the client, who can decrypt it. The encryption engine ensures that blinded data is only encrypted with the session key corresponding to its blindedness tag. As the adversary controls the data to be encrypted, this encryption must be secure against adaptive chosen-plaintext attacks in order to prevent the adversary from using this ability to identify the ciphertexts corresponding to particular plaintexts.

> Data export atomically encrypts data using the session key and marks it as *non-blinded*.

4.3.4 Taint-tracking policy

We use taint tracking to *prevent sensitive data from flowing to observable outputs*. First, we define which parts of the system can be tainted. We split the system state into two types:

- *Blindable* state consists of the values (not addresses) of lines in the cache, values in registers except the program counter, and values in main memory, as well as all busses and queues used to transfer these values. Each of these is extended with a blindedness tag.
- *Visible* state consists of information that may be exposed outside the system, and must therefore never contain sensitive data. It includes *all microarchitectural state that does not have a blindedness tag associated with it*, e.g., the program counter, addresses of lines in the caches, branch predictor state, and performance counters. As the program counter encapsulates control flow, making it part of visible state forbids blinded data from affecting control flow.

We then define the list of observable outputs we consider in BliMe: visible state, non-blinded blindable state, the addresses of memory operations sent to main memory, the execution time of an instruction or set of instructions, and fault signals. Note that once blindable state becomes blinded, it is no longer observable. We exclude outputs that require physical access to be observed, such as voltage and electromagnetic radiation.

Instruction	**Tag (Ø=unblinded, ?="don't care")**		**Decision**	**Result tag**
Arithmetic / Logic†	If # of cycles depends on value of any blinded operand		Fault	-
	Otherwise:			
	Op1	**Op2**		
	Ø	Ø	Propagate	Ø
	a	Ø	Propagate	a
	a	a	Propagate	a
	a	b	Fault	-
Branching	**Addr**	**Condition ops**		
	Ø	Ø	Propagate	Ø
	a	?	Fault	-
	?	a	Fault	-
Load	**Addr**	**Data in memory**		
	Ø	Ø	Propagate	Ø
	a	?	Fault	-
	Ø	a	Propagate	a
Store	**Addr**	**Data in register**		
	Ø	Ø	Propagate	Ø
	a	?	Fault	-
	Ø	a	Propagate	a

† See Section 4.7.3 for a discussion on data-dependent faults.

Table 4.1: BliMe taint-tracking policy rules for all instruction types. Tags a and b are arbitrary but different. Swapping them produces an equivalent result.

The taint-tracking policy is defined as follows (and as shown in Table 4.1). An instruction with a blinded input (i.e., which takes at least one input with a non-zero blindedness tag) yields a blinded value for each output that depends on a blinded input, with the same blindedness tag as the input. An instruction that receives blinded data from multiple sources will raise a fault. If an instruction attempts to affect any observable output except non-blinded blindable state in a manner that depends on the value of a blinded input, a fault is raised, since this can otherwise be used to exfiltrate sensitive data. This effectively means that the program cannot use a blinded value as the address of a jump, branch or memory access, or use instructions whose completion time or fault status depends on a blinded value. For example, BliMe prevents

- *Cache-timing side-channel attacks* by prohibiting blinded values from being used as addresses for loads/stores, and
- *Timing attacks* by forbidding blinded-value-dependent control flow (prohibiting both altering PC based on blinded values, and instructions of variable duration from using

blinded values). Measuring program execution time will reveal nothing about the blinded values; the adversary will obtain the same information as if they ran the same program over a blinded array of zeroes of the same length.

- *Transient execution attacks* by preventing any blinded values from being used in speculation decisions. Speculatively executed instructions can still use blinded values (and maintain their blindedness tags throughout this transient execution), but the result of any prediction *decision* (e.g., whether a branch is taken, or what the next return address might be) cannot rely on blinded values. This same concept applies to all data-dependent hardware optimizations; blinded values cannot be used to *decide* on whether or how an optimization is applied.

The blindedness of any given value is not sensitive; this means that it is safe for the ISA to include instructions that query whether or not a given value is blinded.

4.3.5 BliMe-compliant software

BliMe's restrictions on application software are the same as those required of anyone developing secure side-channel-resistant code (even without BliMe hardware modifications): they must adhere to constant-time coding principles [18], including data-oblivious control flow and memory access, and must not explicitly exfiltrate sensitive data outside the system. An example of such practices being used today can be found in the development of cryptographic libraries, the compatibility of which we show in Section 4.6. Concretely, for software to be BliMe-compliant, it must not attempt to:

1. use blinded data in any control-flow decisions,
2. use blinded data as the target address for any jump or branch instructions,
3. use blinded data as the address for any memory operations, including using blinded data as an offset or index,
4. use blinded data as an operand for an instruction whose execution time depends on the value of that operand (e.g., variable-time division instruction),
5. mix blinded data belonging to separate security domains, i.e., with different blindedness tags, and
6. write blinded data to any peripherals, e.g., displays, network devices, disk drives.

Note that BliMe does *not* require software developers to be aware of how the CPU behaves speculatively. BliMe ensures that all blinded data cannot be leaked by hardware optimizations, including speculation (Section 4.3.4).

4.4 Implementation

In this section, we first present the common architectural changes required to implement BliMe (Section 4.4.1), and then describe how these were applied to the out-of-order RISC-V BOOM core [222] to obtain BliMe-BOOM (Section 4.4.2).

The implementation covers the hardware needed for safe computation (Section 4.3.3), including taint tracking and enforcement of the security policy (Section 4.3.4). We do not include the HSM in the implementation. Components with similar functionality already exist, such as Google's Titan-M chip [134] or Apple's Secure Enclave Processor (SEP) [10]; the HSM is configured to perform the attested handshake and provide correct client IDs, and the system integrator provides the secure channel between HSM and encryption engine in the form of a shared key embedded in each component.

4.4.1 Architectural changes

The architectural changes needed to implement BliMe can be grouped into two main categories: taint tracking, and handling of policy violations.

Taint-tracking is performed on registers and memory. Due to the load-store architecture of RISC-V, these can be handled separately.

Registers and ALU operations. For each register, we maintain a blindedness tag. Any ALU operation that reads from a blinded register, blinds its output register by default. However, this is a conservative approximation, and implementations can make exceptions in order to more accurately model instruction dataflows. For example, if a register is XORed with itself, the result is not blinded because the result is always zero, irrespective of the input value. Similar exceptions can be used for other situations in which an instruction takes blinded inputs but its output does not depend on said blinded inputs.

Memory. Blindedness is tracked in using a tagged-memory approach, with a blindedness tag being attached to each physical address, in both main memory and in each cache. Whenever a value is stored into memory from a register, the memory bytes inherit the blindedness of the register. The reverse holds for memory reads. To reduce overheads,

multiple consecutive bytes in memory, forming a *tag granule*, can share the same tag. For tag granule sizes larger than a single byte, we introduce an additional BliMe policy rule to prevent mixing blinded data belonging to separate security domains: if a store instruction attempts a *partial* write of blinded data (e.g., a byte) with tag a to a granule in memory (e.g., of size 8 bytes) with tag b, the instruction must fault.

New Instructions. We introduce two new instructions, `blnd` and `rblnd`, that correspond to the data import and export operations, respectively (Section 4.3.3). The instructions encrypt and decrypt data with a supplied key, writing the result into memory in unblinded or blinded form respectively.

Handling violations is needed in four situations:

1. Attempting to write blinded values to the `PC` register: Jumps and conditional branches relying on blinded registers, either as a jump destination address or as part of a conditional check, are forbidden.
2. Attempting to use blinded values in an instruction whose execution time depends on the blinded input value.
3. Attempting to read from or write to memory using a blinded value as an address: This occurs when a load or store uses a blinded register either as the address base or offset.
4. Attempting to write blinded data to an "unblindable" memory location. This allows a system to specify whether certain memory-mapped peripherals have access to blinded data.

Any of the above causes an illegal instruction fault.

We implemented the architectural changes in Spike [163], a C++ RISC-V ISA simulator, to enable quick testing of our extensions.

4.4.2 BliMe-BOOM

To demonstrate BliMe on a complex and realistic computation platform that relies heavily on speculation to achieve high performance, we implemented BliMe in RTL on the BOOM [222] core and Chipyard [8] system-on-chip (SoC). BliMe taint-propagation rules are also enforced during any speculative execution thereby preventing sensitive values from being leaked even by Spectre-type attacks [111].

We implemented two variations of BliMe with different tag configurations: BliMe-BOOM-1 with single-bit blindedness tags and byte-level granularity, and BliMe-BOOM-8 with eight-bit blindedness tags and 8-byte (i.e., word-level) granularity.

The BOOM core is written in Chisel, which is a hardware construction language embedded in Scala [19]. Chisel defines common built-in data types, such as unsigned integers (UInt) and Boolean values (Bool), used to attach semantic meaning to bits in digital circuits. Chisel also allows user-defined data types to be created by composition and inheritance. Taking advantage of this feature, we create a *Blinded* data type as a composition of a blindedness tag and a variable-length vector of bits, and use this type throughout the design. As Scala (and, by extension, Chisel) is strongly-typed, this allows the compiler to ensure that blinded values cannot be separated from their blindedness tags except by code that is aware of the *Blinded* type. It also allowed us to identify *all locations in the code* where blinded data can propagate and prevent any unintended leakage caused by hardware optimizations such as speculation.

Registers

Register files are modified to use the *Blinded* type, thereby storing a blindedness tag alongside the register contents.

Main Memory & Caches

A region of memory is allocated for the storage of blindedness tags, and made inaccessible to software. We expand the L1 data and L2 caches to include blindedness tags alongside each granule of data. The L1 instruction cache does not need blindedness tags; blinded values are not allowed to be executed, so whenever a blinded value is read into the cache, it is replaced with a zero, and its 'valid instruction' metadata bits are unset, making it inaccessible. The L2 cache includes logic to translate misses into *two* operations to main memory: one for the actual data, and one for the blindedness tags.

It is possible to optimize the design for the specific case where the blindedness tags are small relative to their associated data, with carefully-designed logic reducing the ratio of blindedness tag requests to data requests, thereby reducing overhead. However, the TileLink implementation in Chipyard reads data from memory at an 8-word granularity by default, yielding a 64-bit effective minimum tag size. Therefore, even for 8-bit tags, the memory controller ties up the bus for seven additional cycles to transmit unneeded words. We forego implementing the extensive modifications required to implement this optimization in Chipyard and instead investigate its effect using gem5 [27] in Section 4.5.2.

Pipeline Stages

In BOOM, instructions are decoded in the instruction decoder stage into micro-ops, which are stored in the reorder buffer until they are issued to an execution unit and committed. Execution units contain different functional units that can perform specific operations. We modified all functional units to either propagate blindedness (e.g., addition of blinded values), or fault when the operation is unsafe to perform with blinded operands (e.g. memory address generation).

The functional unit responsible for memory address generation faults when blinded operands are used to generate a virtual address. Furthermore, if the page table walker (PTW) finds a blinded page table entry (PTE), it zeroes it before storing it in the translation lookaside buffer (TLB). This ensures that virtual-to-physical address translation by the load-store unit cannot be used to load from a blinded physical address. Any load or store attempting to use a blinded physical address directly will fault.

Encryption engine

The encryption engine, which performs atomic blind-and-decrypt and encrypt-and-unblind operations using the ChaCha20 stream cipher, is built as a RoCC accelerator [16, p. 3] containing a pipelined ChaCha20 implementation with a 19-cycle latency. This allows this functionality to be accessed using a set of instruction opcodes reserved for accelerators, and allows it to be used by an unmodified C compiler using RISC-V intrinsics.

Speculation

Speculatively executed instructions are subject to the same checks as non-speculative instructions regarding blinded operands. Branch prediction can also never depend on blinded data. This is done by blocking all blinded data flows into the prediction logic. We 1) zero out any blinded instruction fetched into the instruction cache, 2) prevent any feedback from the execution units to the prediction logic regarding faulting branch decisions based on blinded data. By ensuring that both the speculation decision *and* the speculative execution do not leak sensitive data, we block all speculative side channels. This is in a similar vein to Speculative Taint Tracking (STT) [219] but, unlike STT, we do not untaint after the instructions leave the speculative state and the security policy on the data remains in effect.

Note that, for BliMe-BOOM, only non-constant-time operations, which always fault or are rolled back, affect speculation. Constant-time code, with no secret-data-dependent

	Unmod.	BliMe-BOOM-1		BliMe-BOOM-8	
	Value	Value	Δ [%]	Value	Δ [%]
Power (W)	**37.783**	**38.137**	**+0.9**	**38.319**	**+1.4**
Resources					
LUTs	**460480**	**478950**	**+4.0**	**501847**	**+9.0**
Registers	**357179**	**371024**	**+3.9**	**388956**	**+9.0**

Table 4.2: Effect of BliMe-BOOM modifications on power consumption and FPGA resource usage (unmodified BOOM vs. BliMe-BOOM-1 and BliMe-BOOM-8).

branching or memory accesses, speculates normally *without overhead*. This is because BOOM does not have any other data-dependent speculation (e.g., data-dependent prefetchers). For microarchitectures with such speculation, it must be disabled, but only on blinded data.

4.5 Performance & resource usage evaluation

4.5.1 BliMe-BOOM

Incorporating BliMe into BOOM affects logical complexity (hence maximum clock rate), and the number of cycles that certain operations will take due to the additional memory accesses required to fetch the blindedness tags. Therefore, we evaluate the overall performance of BliMe-BOOM using both the maximum clock rate, and the number of clock cycles taken to execute a program.

We synthesized BliMe-BOOM using the FireSim [104] FPGA-hosted simulation tool to measure the change in resource consumption and maximum clock rate. Table 4.2 shows the power estimate and FPGA resource usage provided by Vivado, indicating low overheads for both BliMe-BOOM-1 and BliMe-BOOM-8. Vivado timing analysis reported a worst negative slack (WNS) of 0.018ns for both modified and unmodified cores, indicating no significant reduction to the maximum clock rate.

To determine the effect of BliMe-BOOM's memory modifications on performance, we ran the SPEC2017 Integer benchmark suite on both BliMe-BOOM variations using the `ref` workload. BliMe-BOOM-1 was run with 8GiB of main memory, and BliMe-BOOM-8 with 14GiB of main memory, so that both systems have 7GiB of addressable memory. This avoids the risk that the performance of the benchmarks is affected by differences in

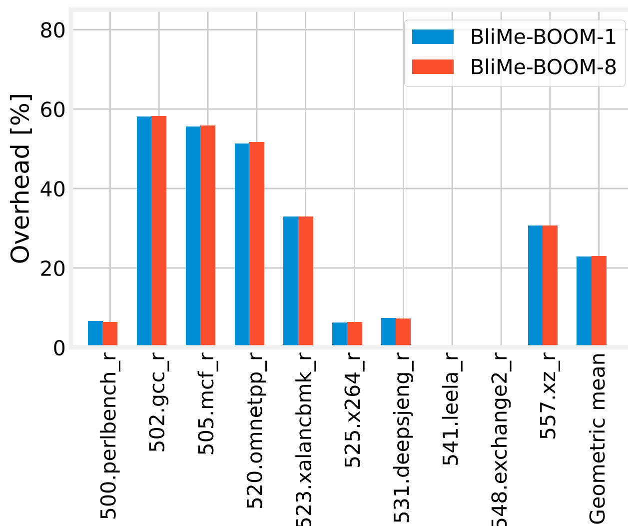


Figure 4.2: Overhead of BliMe-BOOM-1 and BliMe-BOOM-8 relative to unmodified BOOM, as measured using SPEC2017. The average overhead was **23%** for both BliMe-BOOM-1 and BliMe-BOOM-8.

available memory. The results are shown in Figure 4.2; the average overhead across all benchmarks is **23%** for both BliMe-BOOM-1 and -8. This overhead, however, stems from the limitation mentioned in Section 4.4.2. As we show below in Section 4.5.2, appropriate memory optimization can substantially reduce this overhead.

4.5.2 Memory optimization

In Section 4.4.2, we mentioned that an implementation optimized for small blindedness tags can improve its performance by reducing the ratio of blindedness tag requests to data requests. We investigate this by simulating both the optimized and unoptimized designs using the gem5 simulator [27], where it is straightforward to reduce the memory request size for tags, and comparing their performance.

Gem5 is run in RISC-V full-system mode with the `O3` out-of-order processor model. The cache configuration was chosen to match that used in Section 4.4.2: 16kB each for private L1D and L1I, and 256kB for private L2. Main memory was 7GB of Dual Channel DDR3-1600 DRAM. In addition, we modified the memory model to generate the extra requests used to read and write blindedness tags, with the ratio of blindedness tag to data request sizes being configurable as either 1:1 (as used in BliMe-BOOM) or 1:8 (as in the

Implementation	Avg. overhead (%)
BliMe-BOOM-1	**23**
BliMe-BOOM-8	**23**
BliMe-gem5	**25**
BliMe-gem5 Optimized	**8**

Table 4.3: Average overheads of running SPEC2017 on BliMe-BOOM-1 and -8, BliMe-gem5 and BliMe-gem5 Optimized. The averages are calculated for the full benchmarks on BliMe-BOOM-1 and -8, and for 1 billion instructions on BliMe-gem5 and BliMe-gem5 Optimized. The gem5 simulation of the unoptimized system shows an average overhead of 25%, similar to that of both BliMe-BOOM-1 and -8 (23%).

proposed optimization).

Since gem5 is much slower than native execution using a field-programmable gate array (FPGA), following the approach taken in prior work [219], we ran the SPEC CPU 2017 Integer benchmark suite with the `ref` workload in each configuration and measured the number of cycles taken to execute and commit 1 billion instructions after skipping the first 10 billion instructions in order to allow the benchmark proper to start. Table 4.3 compares the performance results for BliMe-BOOM-1, BliMe-BOOM-8 and BliMe-gem5. With the gem5 model configured in unoptimized mode, the average overhead of 25% is similar to that of both BliMe-BOOM-1 and -8 (23%). With the model configured in optimized mode, the overhead reported by gem5 is reduced to **8%**. The detailed benchmark results comparing the two configurations is shown in Figure 4.3. Therefore, we conclude that if the optimizations are implemented in hardware (by making the needed modifications to the memory subsystem in the Chipyard SoC as mentioned in Section 4.4.2), similar overhead reductions can be achieved. As a result, for a tag-to-data size ratio of 1:8, 8% is a fairer representation of BliMe's overhead in real deployments by hardware manufacturers. Use of tag caches [100] would reduce overheads even further. The earlier figure of 23%, on the other hand, corresponds to a tag-to-data size ratio of 1:1. Note that the optimization that leads to reduced overheads *does not trade-off any security guarantees nor require a weaker threat model*.

The closest comparisons to BliMe, oblivious instruction set architecture (OISA) [218] and FHE, suffer from significantly higher overheads. OISA's fastest benchmarks (matrix-mult and neural-net) incur approximately 35% overhead with larger-than-cache inputs, whereas FHE is several orders of magnitude slower [195]. Overall, given the moderate impacts on performance and resource usage discussed in Sections 4.5.1 and 4.5.2, we conclude that requirement PR is satisfied.

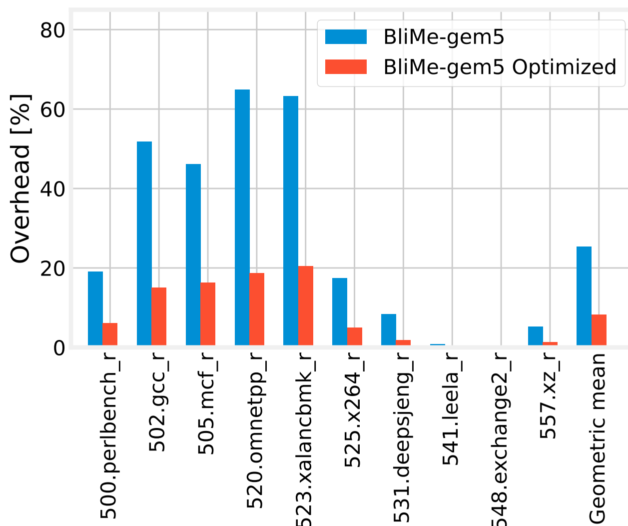


Figure 4.3: Overhead before and after applying the optimization from Section 4.4.2, as measured using gem5 simulator to execute 1 billion instructions of SPEC2017. The optimization reduces the average overhead on gem5 from 25% to 8%.

4.6 Compatibility evaluation

For backwards compatibility with existing code (requirement CR), two types of code are important: code that processes exclusively non-blinded data, and code that is already side-channel resistant and processes blinded data.

BliMe's taint tracking policy (Section 4.3.4) changes the behavior of the processor only where an instruction depends upon a blinded input value. Therefore, code that exclusively processes non-blinded data is unaffected.

Code that processes blinded data will fault if it attempts to modify an observable output in a way that the processor determines to be dependent on the blinded data. To evaluate whether the BliMe implementations have met requirement CR, we must therefore determine whether side-channel-resistant code will comply with the BliMe taint-tracking policy.

BliMe faults when blinded data attempts to flow to non-blindable observable outputs (Section 4.3.4). Existing side-channel-resistant code already prevents the program counter and the addresses of memory operations from being affected by blinded data, e.g., as in [23]. Data flows of blinded data to other visible state, execution time and fault signals (Section 4.3.4) are prevented by design. Therefore, side-channel-resistant code is compatible with BliMe's taint-tracking policy, so long as only sensitive data is blinded. We successfully

ran stream cipher encryption/decryption with a blinded key using the TweetNaCl [24] library, demonstrating the backwards compatibility of BliMe.

However, the CPU cannot identify cases where the result of a computation remains blinded even though it no longer contains any sensitive data. For example, it will not be possible to branch on the result of the decryption/verification of an authenticated encryption, meaning that its cryptographic application programming interfaces (APIs) will need to be modified so that its control flow does not depend on whether the verification was successful. In practice, this means that any computation must always continue as though verification was successful, with any failure being indicated by a flag that is returned to the client (Section 4.7.3).

4.7 Discussion & Future Work

4.7.1 Handling large numbers of clients

BliMe can handle $2^n - 1$ simultaneous clients with an n-bit blindedness tag. This means that the ability to handle larger numbers of clients will require a logarithmic increase in memory overhead. This can be made configurable at boot-time, allowing servers to trade main memory capacity for a larger number of simultaneous clients. Another approach, however, is to overcome this overhead by multiplexing the same blindedness tag value across multiple clients, and using an OS trusted by clients—e.g., by using remote attestation of a well-known certified OS—to prevent data from flowing between programs that share blindedness tag values. In the extreme case, this reduces to a single-bit blindedness tag, as implemented in BliMe-BOOM-1.

This OS has several tasks. On context switch, the OS asks the encryption engine to export the current client key by "sealing" it (i.e., encrypting it using a key known only to the encryption engine), and provides a previously sealed key, corresponding to the incoming process, which the encryption engine can unseal and set as the new current client key. Thus, even though the OS never sees any client key in the clear, it must be trusted to load the correct sealed key onto the encryption engine.

The normal process isolation features of the OS ensure that a client cannot access another client's blinded data directly or any of the client session keys held by the OS. Because different processes may share blindedness tag values, the OS must further ensure that an application cannot transfer blinded data to other applications via system calls or other inter-process communication mechanisms. Preventing a client from accessing another

client's blinded data ensures two things: 1) an application serving one client cannot use the encryption engine to encrypt and unblind blinded data originating from another client, and 2) computation can never use blinded data belonging to two different clients, since there would be no clear way to determine the ownership of the result. Therefore, processed data is only encrypted and exported with a key corresponding to the client from which the input data was originally imported. Note that we would only rely on the OS to prevent clients from having *direct* access to other clients' blinded data. We would not rely on the OS code being side-channel resistant, because blinded data is protected from side channels by the hardware.

4.7.2 Enabling safe local processing

Future work can apply BliMe on the local machine as well. Since an application processing blinded data cannot infer anything about the data other than its length, it can safely process data belonging to other users or applications. One possibility is for the OS to allow an application to read data from a file that is normally inaccessible to the application with the constraint that any data read will be marked as blinded. This makes it possible to build useful computational pipelines while strongly adhering to the principle of least privilege.

4.7.3 Handling secret-dependent faults

In BliMe, a fault's occurrence cannot depend on a secret value, since the fact that it occurs (or not) can leak information. For example, if a div-by-zero fault will occur if the divisor is a blinded data item with a value zero, and this faults leads to an interrupt handler being called, the change in control flow will reveal that the blinded value was zero. On the other hand, if it does not occur, it reveals that the value was not zero. We therefore do not allow blinded values to be used in such situations. To avoid this limitation, we might instead suppress faults depending on blinded data, so that the control flow remains *as if the fault did not occur*. The client can be informed of this fault and that the computation results are invalid by setting a blinded bit in some protected storage (e.g., a special register) when the fault occurs, so that the software can convey this bit to the client as part of the returned encrypted results.

4.8 Related work

Lee *et al.* [119] propose DOVE to protect sensitive data used in outsourced computation from side channels. DOVE uses a frontend to transform the client application code to a custom data-oblivious representation called a Data Oblivious Transcript (DOT). The DOT is then sent to a trusted interpreter (the backend) on the server, which verifies that the DOT is data oblivious and then runs it on the sensitive data. The trusted interpreter must be run within a TEE, such as an Intel SGX enclave, as it is part of the trusted computing base (TCB).

Yu *et al.* [218] develop an OISA that performs run-time taint tracking of sensitive values and adds a duplicate set of instructions to the ISA. Each operand of the additional instructions is defined as either safe or unsafe. Using any tainted values as unsafe operands results in a fault. The hardware guarantees that computation is oblivious to safe operands and, therefore, that any sensitive values used as safe operands are not leaked through side channels. OISA offers taint and untaint instructions and relies on the application code to use them correctly to taint/untaint sensitive values during computation. Consequently OISA is *not applicable for our usage scenario* described in Section 4.2.1, where outsourced computation is carried out by potentially *untrusted, third-party*, application code, for multiple simultaneous clients. Although OISA could use remote attestation to verify the server-side application code, remote attestation of arbitrary third-party application code is neither realistic nor scalable. Furthermore, software vulnerabilities in the application code can allow adversaries to untaint arbitrary sensitive values through the OISA's untainting instruction, which is exposed to any software running on the main CPU.

Chapter 5

BliMe Extensions

While BliMe demonstrates the effectiveness of hardware-assisted taint tracking for protecting data confidentiality against both direct access and side-channel leakage, its practical deployment faces two critical challenges that extend beyond the core architecture itself. First, as computing increasingly relies on specialized hardware accelerators for performance-critical tasks, BliMe's protection must extend beyond general-purpose CPUs to these specialized processing units. Second, the stringent data-oblivious programming requirements imposed by BliMe's security policy create a barrier to adopting existing software, as most codebases do not adhere to the constant-time programming principles necessary for BliMe compliance.

This chapter addresses both challenges through two key extensions to the BliMe ecosystem. Dolma extends BliMe's hardware-assisted protection to machine learning accelerators, demonstrating how the taint-tracking approach can be systematically applied to specialized computing hardware while maintaining both security guarantees and performance characteristics. The BliMe Linter addresses the software compatibility challenge by providing automated static analysis tools that can identify potential policy violations in existing code, significantly reducing the manual effort required to adapt software for BliMe deployment.

Together, these extensions demonstrate how the BliMe approach can scale beyond its initial CPU-focused design to address the broader challenges of deploying hardware-assisted protection mechanisms in diverse computing environments. They represent critical components in bridging the gap between BliMe's strong theoretical guarantees and its practical applicability in real-world systems where specialized hardware and legacy software are fundamental constraints.

The contents of this chapter are based on the following publications:

Hossam ElAtali et al. “Data-Oblivious ML Accelerators using Hardware Security Extensions”. In: *Proceedings of the 2024 IEEE International Symposium on Hardware Oriented Security and Trust*. Tysons Corner, VA, USA: IEEE, 2024. DOI: 10.1109/HOST55342.2024.10545398 ©2024 IEEE

Hossam ElAtali et al. “BliMe Linter”. In: *Proceedings of the 2024 IEEE Secure Development Conference (SecDev)*. Pittsburgh, PA, USA: IEEE, Oct. 2024, pp. 46–53. DOI: 10.1109/SecDev61143.2024.00011 ©2024 IEEE

5.1 Dolma

5.1.1 Introduction

As modern computing increasingly relies on specialized hardware accelerators for performance-critical tasks such as machine learning inference, the challenge of protecting sensitive data against remote adversaries extends beyond traditional CPU architectures. While BliMe demonstrates how hardware-assisted mechanisms can efficiently protect data confidentiality against both direct access and side-channel leakage in general-purpose processors, the growing prevalence of hardware accelerators in remote computing environments creates new attack surfaces.

The rise of outsourced computation has made machine learning inference a particularly compelling use case for data protection mechanisms. Clients wishing to classify sensitive images or process confidential data using a service provider’s machine learning (ML) model face the fundamental challenge established in RQ2: how to protect data confidentiality against both direct memory access through run-time attacks or untrusted code, and indirect leakage through side channels. Unlike general-purpose central processing units (CPUs), accelerators for ML workloads often implement specialized computational patterns, memory hierarchies, and data flows that can create new opportunities for both direct access attacks and side-channel leakage. Simply extending CPU-based protection mechanisms to accelerators is insufficient, as the unique architectural characteristics of these devices require tailored approaches that preserve both their computational efficiency and their security properties.

To address this challenge as part of our investigation into RQ2, we propose Dolma, an enhancement to the Gemmini machine learning accelerator [74] that extends BliMe’s hardware-assisted taint tracking and data obliviousness guarantees to specialized ML hardware. Dolma demonstrates how the principles established in BliMe—remote attestation,

hardware-enforced taint tracking, and data-oblivious computation—can be systematically extended to hardware accelerators while maintaining their performance characteristics and preserving the comprehensive protection against both direct access and side-channel threats that modern remote computation environments demand.

In Dolma, we use hardware-based dynamic information flow tracking (DIFT) to track secret client data (and any derivative of it) and enforce a security policy that aborts any attempt, intentional or otherwise, by software to leak this data. Implementing DIFT in accelerators using mechanisms such as GLIFT [189] is possible, but introduces large overheads due to the unnecessarily small granularity (gate-level) at which taint is propagated. The predictable behavior of systolic arrays common in accelerator architectures presents unique optimization opportunities. We can safely propagate taint at a much higher granule leading to significantly reduced area overheads.

Our contributions are the following:

- We present Dolma, a minimal extension to matrix multiplication hardware accelerators that enables efficient DIFT and guarantees confidentiality of secret client data, including against side-channel attacks and even in the presence of malicious software. We provide a register-transfer level (RTL) implementation of Dolma on Gemmini [75], a flexible hardware accelerator integrated into the RISC-V Chipyard [8] system-on-chip (SoC) (Section 5.1.4).
- An evaluation of performance using realistic ML workloads showing minimal overheads, orders of magnitude lower than state-of-the-art cryptographic solutions (4.4% in 32x32 configuration), and an evaluation of resource usage showing that Dolma is scalable and feasible for accelerators with large dimensions. We also extend Blinded Memory (BliMe)'s F* model and provide a machine-checked proof of its security. (Section 5.1.5)

5.1.2 Background – Gemmini

Gemmini is a RISC-V accelerator allowing flexible configuration of accelerator parameters, e.g., systolic array dimensions, data flow (weight stationary or output stationary), and scratchpad sizes. It is connected to the host CPU using the RoCC interface [8]. It consists of a systolic array (a "mesh") for matrix multiplication, internal scratchpads for data storage, and components for performing activation functions such as ReLU. The circuit is controlled by three controllers: two for direct memory access (DMA) and one for execution. These controllers can act independently in a decoupled access-execute architecture. The systolic array has two levels: the mesh consists of a 2D array of tiles, each tile a combinational 2D

array of processing elements (PEs). The main computation performed by the systolic array is the following multiply-and-accumulate (multiply-and-accumulate) operation (for a 2x2 array):

$$C = A * B + D \tag{5.1}$$

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} + \begin{pmatrix} d_{11} & d_{12} \\ d_{21} & d_{22} \end{pmatrix} \tag{5.2}$$

Input matrices are loaded from memory and stored in the scratchpads, whose width corresponds to the width of the systolic array. The inputs then flow into the array, where each PE performs a multiply-and-accumulate on a single element, e.g., $a_{11} * b_{11} + d_{11}$. The inputs must be delayed appropriately using registers to ensure the correct elements are multiply-and-accumulateed in every cycle. The outputs are also delayed to ensure the resulting matrix is written into the scratchpads *one row at a time*.

Gemmini supports two data flow modes: weight stationary and output stationary. In the former, the weights, B, are preloaded into the array and A and D flow in. In the latter, D is preloaded and A and B flow in. With weight-stationary data flow, inputs A and D can be expanded to allow back-to-back computations on *independent rows*, as shown in Equation (5.3). This is particularly useful in neural networks where multiple inputs, e.g., images, are given to the same model. The model weights, B, remain in place and the input vectors (A_1, A_2, A_3, etc.) are fed in as independent rows with no stalls between them.

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \\ c_{31} & c_{32} \\ \dots & \dots \\ c_{K1} & c_{K2} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ \dots & \dots \\ a_{K1} & a_{K2} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} + \begin{pmatrix} d_{11} & d_{12} \\ d_{21} & d_{22} \\ d_{31} & d_{32} \\ \dots & \dots \\ d_{K1} & d_{K2} \end{pmatrix} \tag{5.3}$$

5.1.3 Assumptions & Threat Model

Our trusted computing base (TCB) includes the BliMe CPU hardware, which we assume correctly enforces the BliMe DIFT policy, and the accelerator hardware as well as the interconnect between them, whose data we assume cannot be sniffed or modified. We also assume that BliMe can securely perform remote attestation and manage client session keys and key-to-tag mappings.

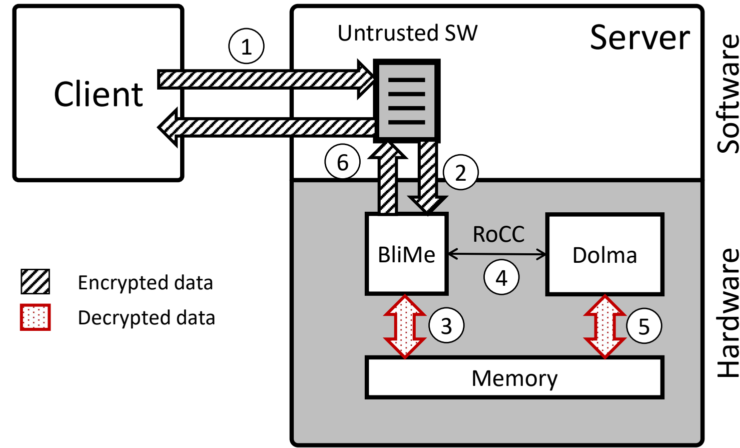


Figure 5.1: System Overview. The client encrypts and sends their secret data to the untrusted software on the server ①, which calls BliMe's data import operation ②. BliMe decrypts-and-blinds the data, tagging it with the session-specific tag, and stores it in memory ③. The untrusted software can then use RoCC instructions ④ to make Dolma operate on the blinded data. Dolma accesses the data using DMA and enforces the DIFT security policy ⑤. Once processing is complete, the untrusted software can call BliMe's data export function to encrypt-and-unblind the data, and then send it back to the client ⑥.

All software, including the operating system (OS) and applications, is untrusted: the adversary can control all software running on the server. As in [59], attacks requiring intrusive physical access, such as bus snooping or differential power analysis, are out of scope.

5.1.4 Design & Implementation

Overview

An overview of the system is shown in Figure 5.1. Dolma is connected to a BliMe CPU through the RoCC interface. The remote attestation performed by BliMe in the handshake step is augmented to include attestation of Dolma using an additional root of trust embedded in Dolma. The attestation of both roots of trust assures the client that the server contains a genuine BliMe CPU connected to a genuine Dolma accelerator. The client then encrypts its data using the session key shared with BliMe and sends it to the server. The processing software on the server can then call BliMe's data import instruction to atomically blind and decrypt the data. At this point, the secret client data is blinded in memory and can be

read by Dolma for processing. Note that combining blinded and non-blinded data (e.g., an input image and a model, respectively) within Dolma is allowed and the result (e.g., the inference output) is blinded. Once processing is complete, the blinded result can be atomically unblinded and encrypted using BliMe's data export instruction and sent back to the client.

Our DIFT policy is defined such that if any of the following have the same non-zero tag, then the output row receives the same non-zero tag: 1) the corresponding input row $\mathbf{a}_i$, 2) the corresponding input row $\mathbf{d}_i$, 3) any weight matrix row $\mathbf{b}_j$. If any two of these rows have differing non-zero tags, then the operation faults and does not emit any further output.

DIFT is used by the BliMe CPU to forbid software from using secret data in a way that affects memory access patterns, control flow, or I/O. The DIFT policy above ensures that Dolma cannot be used to circumvent these restrictions. In the remainder of this section, we will present the challenges we encountered while implementing this policy in Gemmini and the optimizations we used to reduce overheads. Even though we discuss Gemmini in particular, these challenges and optimizations are applicable to a wide range of hardware accelerators as they concern generic commonly used mechanisms. In particular, our design is directly applicable to any systolic-array-based accelerator.

Tag bits

We extend the RoCC interface to include tag bits with register values passed to Gemmini from the CPU. We also extend the memory interface and internal scratchpads to include tag bits. The TLB is *not* extended with tag bits: page table entries must not be blinded. We check all new entries into the TLB; if an entry is blinded, it is zeroed and a fault is raised.

We ensure the completeness of our modifications by using a `Blinded` data type [59], wrapping untagged Chisel data bundles. This wrapper causes the Chisel compiler to raise an error wherever non-BliMe-aware logic attempts to use tagged data. Upon detecting such a mismatch, we decide whether to 1) safely propagate the tags, or 2) check for a security violation and if one is detected, zero the blinded value and raise a fault.

RoCC commands

Sending blinded instructions to the accelerator is prohibited because, otherwise, the type of operation performed by the accelerator, e.g., loading values or starting execution, would

leak the values of the instructions. Furthermore, in Gemmini's case, RoCC commands pass two register values to Gemmini as 'rs1' and 'rs2', which are used either as pointers to data in memory or as opcodes for subfunctions. In both cases, blinded values for rs1 and rs2 are prohibited by the hardware because they can be leaked through the memory access patterns or by observing the subfunction performed by Gemmini, respectively. We thus raise a fault if any blinded data (i.e., instructions or register values) is sent directly through the RoCC interface.

DIFT in the systolic array

The most straightforward way to perform DIFT is to insert tracking logic inside every PE. However, this introduces unnecessarily large overhead. Instead, we implement DIFT at row granularity. This fits well with how Gemmini operates, as explained in Section 5.1.2, and, in weight-stationary mode, enables inputs from different clients to be streamed back-to-back into the systolic array while maintaining isolation between them. We analyze the systolic behavior of the tiles in Gemmini and deduce how tags will propagate from inputs to outputs, which is possible due to the tiles' fixed functionality. We calculate the tags for each output row according to our DIFT policy before the inputs enter the systolic array, and then propagate the tags using a minimal circuit in parallel to the data. This allows us to avoid adding logic to each tile and PE. The propagation of output tags is visualized in Figure 5.2. In this example, the parallel circuit allows reducing tag storage registers from four (one in each tile) to three. It also reduces tag propagation logic from 4xNxM instances, where N and M are the dimensions of the PE array within each tile, to only three. These reductions scale quadratically with larger systolic arrays. We show how these reductions benefit larger configurations in Section 5.1.5.

Each matrix row carries a single blindedness tag, where a zero tag indicates non-blinded, and every non-zero tag value identifies a separate security domain. Output rows always take a constant number of cycles to propagate out of the systolic array. We therefore propagate the tag using a parallel queue with a latency that matches that of the output rows. The queue input tag is determined by an OR of the tags corresponding to the three input matrices. This is done after checking that the three inputs do not have mismatching non-zero tags, as explained in Section 5.1.4.

Scratchpads & context switches

The internal scratchpads are extended with tags to mark secret data inside Gemmini. This is necessary because the OS is in charge of context switches. If the internal scratchpads are

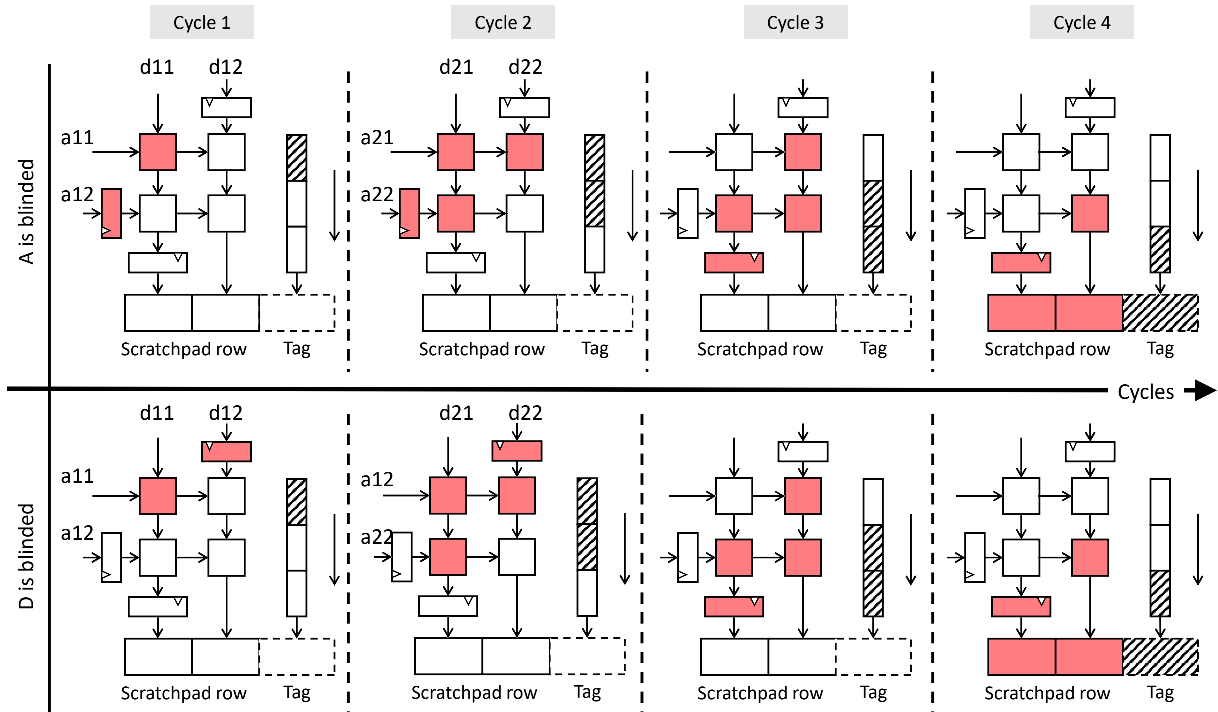


Figure 5.2: DIFT in parallel for the weight-stationary data flow inside a 2x2 systolic array. The top half of the figure shows the case where A is blinded. The bottom half shows the case where D is blinded. The subfigures from left to right show how computation proceeds over successive cycle. Secret values are shown in red. Note that the tiles and registers themselves do not carry any additional logic. The tags corresponding to the secret values are shown as striped. Output tags are calculated before input values enter the systolic array and propagate alongside the corresponding secret input and intermediate values. The propagation is synchronized such that output rows receive the correct tag. In the case where B is blinded (not shown here), all output from the systolic array would be blinded since B is preloaded into the array.

not tagged, a malicious OS can wait for a process to load secret data into Gemmini, and then perform a context switch to give another process (or itself) access to the data. We extend the scratchpads with one tag per entry, which is equal in size to the rows of the systolic array. As in Section 5.1.4, this corresponds to one tag per row and therefore, all elements in the row share the same tag.

Tag mixing

We do not allow mixing of data with different non-zero tags. Therefore, when calculating the tag for each output row, we check the tags of the three corresponding inputs (rows of A and D, and column of B) and only allow propagation when there are no two or more different non-zero tags. Furthermore, when reading data from main memory and writing it to the scratchpads, mismatching tags can occur if the width of the scratchpad rows is larger than 64 bits. We therefore also check for and forbid mismatching tags between the current tag of the scratchpad row and the tag of incoming data.

Read-Check-Write

As discussed in Section 5.1.4, partial writes require first checking that there is no violation between the current and incoming row tags. However, scratchpad memory is implemented using SRAM, which provides a synchronous read/write interface and requires that reads and writes take one cycle. This means that we cannot read the current tag, check for violations, and then request a write all in one cycle. A simple solution to this would be to implement the tag memory (only) using registers to support asynchronous reads. However, this is infeasible due to the large capacity of the scratchpads leading to high area overhead. We therefore pipeline the writes, introducing an additional cycle to read the current tag and delaying the write to the second pipeline stage. This is shown in Figure 5.3. The scratchpad processes at most one read or write per cycle (not both), with priority given to writes. For a read in Stage 1, we issue corresponding reads to data and tag memories, which are available for output after one cycle at Stage 2. For a write in Stage 1, we issue a read to tag memory to check the current tag in memory. In Stage 2, we output read responses or issue writes to memory after checking for violations. In the case that the scratchpad receives a write followed by a read to the same address, this would mean that we issue a read (in Stage 1) and a write (in Stage 2) to the same address *in the same clock cycle* (i.e., a Read-Over-Write). This is not supported by the underlying SRAMs. We therefore manually implement a bypass to forward writes to reads. Data is only forwarded if no violations are detected in the corresponding write.

Activation functions

One important aspect of accelerator architectures is that common activation functions are implemented in hardware using combinational logic. This is in contrast to general-purpose architectures where activation functions are normally computed in software. Computation of

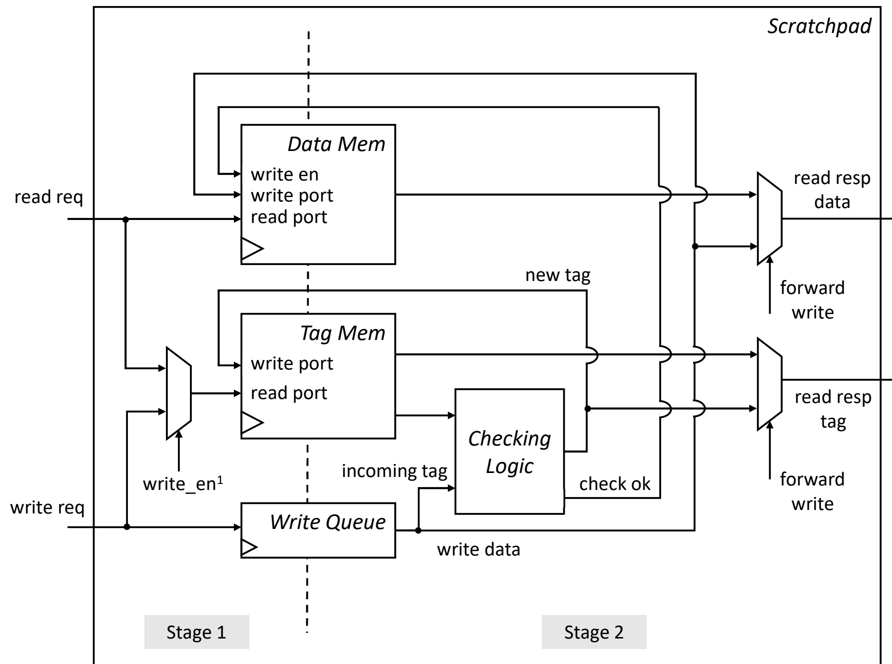


Figure 5.3: Writes are pipelined to maintain throughput while enabling read-check-write to prevent tag mixing. A 2-stage pipeline is implemented. The dashed line visualizes the separation between the two stages.

activation functions, such as ReLU, in software is often implemented using branching logic, which leads to timing side channels. This is because the control flow becomes dependent on the data values (Section 2.3). For example, in the case of ReLU, whether to multiply by a constant factor or set to zero depends on whether the input is positive or negative. By implementing the functions using combinational logic, the function becomes constant-time for all inputs, eliminating all timing side channels. This allows us to simply propagate taint from inputs to outputs without any cycle overheads.

5.1.5 Evaluation

Performance

Performance overheads consist of two parts: cycle overhead, and clock frequency overhead. Cycle overhead is caused by the additional clock cycles required to access tag bits in

main memory, and the added latency for read-check-write (Section 5.1.4). Clock frequency overhead is caused by the additional circuitry, which can cause a reduction in maximum clock frequency if it is on the critical path.

We measure cycle overhead by running ML inference in the form of image classification on ResNet-50 using a random 100-image sample of the ImageNet data set. The experiments were executed on a Xilinx VCU118 FPGA running Linux Buildroot on the Chipyard design, which was configured with a single BOOM core. We compare the performance of Dolma to baseline Gemmini without BliMe. The results are shown in Figure 5.4. We measure an average overhead of 5.6%. This is similar to overhead reported by BliMe [59] compared to unmodified BOOM, indicating that the majority of the overhead is not inherent to Gemmini, but rather to the memory tagging mechanism used by BliMe.

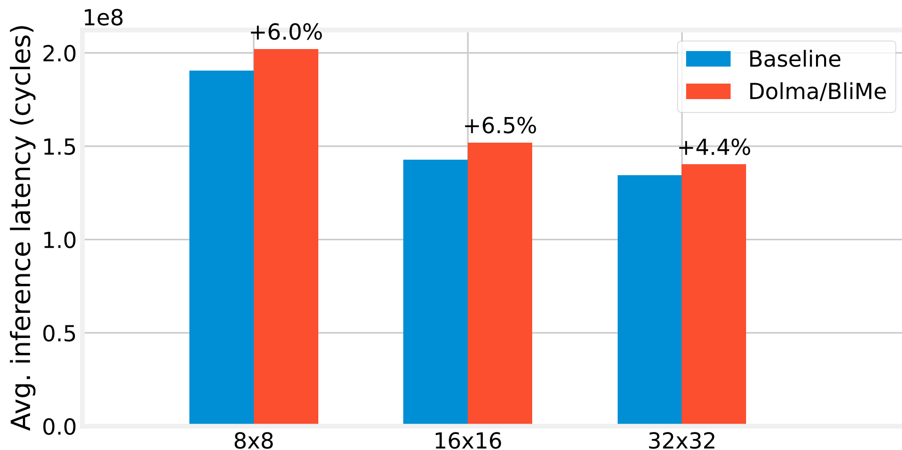


Figure 5.4: Performance results of running ResNet-50 image classification for Dolma relative to unmodified Gemmini. We obtain an average overhead of 5.6% over all three configurations.

All configurations for both Dolma and the baseline were successfully built for the default Chipyard VCU118 frequency of 50MHz. Xilinx Vivado reported baseline worst negative slack (WNS) (in ns) as 0.263, 0.367 and 0.261 for the 8x8, 16x16 and 32x32 configurations, respectively. WNS for Dolma was 0.309, 0.300 and 0.166, respectively. This indicates no significant overhead for Dolma.

Power

We obtain Vivado power reports for Dolma and unmodified Gemmini for all configurations. The results are shown in Table 5.1, with estimated power consumption increasing by 12.2−16.5%.

	Power consumption (W)		
Config	**Baseline**	**Dolma**	Δ [%]
8x8	4.951	5.701	15.2
16x16	5.426	6.085	12.2
32x32	6.994	8.147	16.5

Table 5.1: Effect of modifications on FPGA power consumption.

Resource usage

We obtain FPGA resource usage reports from Vivado for Dolma and unmodified Gemmini for different mesh sizes. The results are shown in Figure 5.5. The results here highlight the benefits of the optimization described in Section 5.1.4. We see a significant decrease in relative resource usage overheads as the mesh dimensions increase. Note that the additional resource usage reported here includes the tag storage added to the scratchpads, whose size is constant across the three configurations.

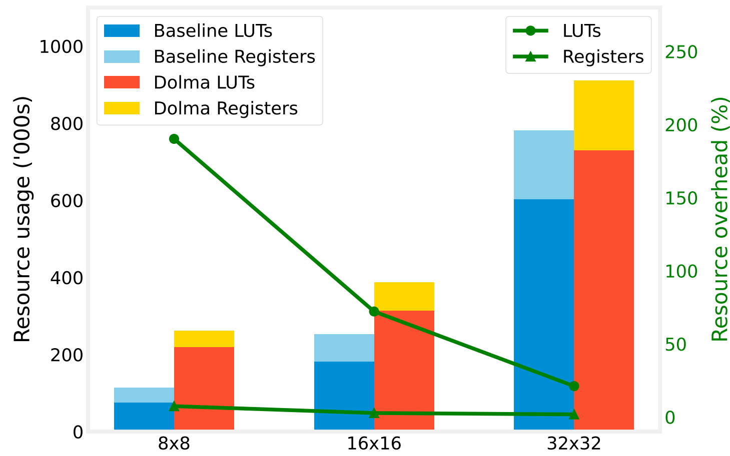


Figure 5.5: Resource usage results for Dolma relative to unmodified Gemmini.

Security

BliMe includes a formal model and a machine-checked security proof in F* [182]. It proves that 'secure' computation, as defined by BliMe, does not leak any information on data marked as blinded. However, Dolma differs from normal instructions in that it directly modifies memory, rather than being limited to data stored in registers, and we therefore extend BliMe's formal model to include this type of operation. In particular, we prove that adding 'secure' accelerator operations to the system also does not leak any information.

First, we define DMA accelerator operations as functions that map a list of input words (read from memory) to a list of outputs (written to memory). We then define 'secure' accelerator operations as those that do not allow for information to flow from their blinded inputs to their observable outputs; more specifically, if two inputs to the accelerator differ only in their blinded values, then their corresponding outputs can differ only in their blinded values.

We modify the execution model to allow execution units to trigger such an accelerator, and prove that any accelerator meeting this safety requirement does not compromise the safety of the system. We then modify BliMe's concrete instruction set architecture model to incorporate a model Dolma in the form of a matrix multiply-and-add accelerator, which if given any blinded input produces a fully blinded output. This accelerator can be proven secure, thereby showing that incorporating our model Dolma into BliMe's model instruction set architecture (ISA) does not undermine its security guarantees.

5.1.6 Discussion & Future Work

Data-dependent processing

Baseline Gemmini does not support any data-dependent control flow. As a result, the only potential for policy violation arises from tag mixing (Section 5.1.4). However, state-of-the-art accelerators often provide data-dependent optimizations, e.g., skipping multiplications when zeros are detected is useful when processing sparse matrices [184]. These can result in data-dependent computation latencies leading to data leakage through timing side channels. Furthermore, some accelerator designs enable on-chip control flow [102]. This allows different control paths to be taken depending on the data, which can again leak data through side channels. Implementing Dolma on such accelerators must consider *all* data-dependent timing side channels and enforce data-obliviousness when processing blinded data. To avoid unnecessarily enforcing worst-case data-oblivious performance on all data, additional logic will likely be needed to disable optimizations on blinded data *only*.

Graphics processing units (GPUs)

GPUs are another type of accelerator, commonly used for ML workloads. Originally intended for graphics processing, GPUs contain large numbers of small cores executing small sequences of code called "kernels" in parallel. After the introduction of general-purpose GPUs, which enable arbitrary code execution, GPU architectures have recently shifted towards incorporating dedicated hardware components for accelerating ML workloads [146].

In contrast to accelerators such as Gemmini, GPUs inherently support on-chip control flow for kernel execution, making them vulnerable to control-flow-based timing side channels. As discussed in the previous section, this requires additional care to enforce data-obliviousness on branches. Another source of timing side channels is memory accesses using secret-dependent addresses. Access to off-package memory is subject to the same data-oblivious policy, i.e., secret-dependent addresses are forbidden. However, GPUs use high-capacity on-package memories and provide explicit control over data movement in and out of these memories using a separate local address space (unlike CPU caches). Secret-dependent accesses to these on-package memories (using blinded local addresses) can be executed without causing timing differences visible to an adversary, providing opportunities for optimizing data-oblivious algorithms. On the other hand, GPUs are more general-purpose than fixed accelerators such as Gemmini and might not be able to benefit from row-wise DIFT. We leave such research as future work.

5.2 BliMe Linter

5.2.1 Introduction

The practical deployment of hardware-assisted mechanisms for protecting sensitive data against remote adversaries faces a fundamental challenge: bridging the gap between hardware security guarantees and existing software ecosystems. While BliMe demonstrates how hardware-enforced taint tracking and data obliviousness can efficiently protect data confidentiality against both direct access and side-channel leakage, its stringent security policy requires software to adhere to constant-time programming principles that most existing codebases do not follow. In order for such software to run on BliMe, its executable needs to be adapted into a form that avoids secret-dependent control-flow branches and memory accesses. Since manually identifying potential data leaks is a complex and error-prone task, a compiler-based tool is needed to automatically identify potential violations in source code.

In this work, we introduce the BliMe linter, a set of compiler extensions that analyze LLVM bitcode to identify possible Blinded Memory (BliMe) violations. At the core of our constant-time code linter is a taint-tracking engine that propagates taint statically but with the same policy as enforced by the BliMe hardware at runtime. To detect information flows within the program, we build on a state-of-the-art static analysis tool, SVF [180], that uses value flows to track relations in programs (Section 5.2.2). SVF alone, however, is not enough to detect all BliMe violations. We discuss the reasons for this, as well as how we improve the analysis to account for it, in Section 5.2.3. Concretely, our contributions are as follows.

1. The BliMe linter, a set of compiler extensions to identify potential BliMe violations in LLVM bitcode (Section 5.2.3)[1] .
2. An evaluation of the BliMe linter on the oblivious instruction set architecture (OISA) benchmarks and TensorFlow Lite, including two case studies that demonstrate the effectiveness of the BliMe linter in identifying the root cause of violations (Section 5.2.4).
3. A discussion of challenges and possible improvements for future work, including automatic transformations of non-compliant code (Section 5.2.5).

5.2.2 Background – SVF

Compilers often need to determine how values affect or propagate to other values. One way to do this is via static program analysis, specifically value flow analysis. Value flow analysis resolves dependencies between variables in the program. SVF [180] is a state-of-the-art value-flow analysis tool. It splits the analysis into two steps that are performed iteratively: pointer analysis, and value-flow construction. Pointer analysis determines the set of locations each pointer can refer to. Value-flow construction takes the results of the pointer analysis and uses it to create a sparse value-flow graph (SVFG), which represents the dependencies between values as a directed graph. If further precision is required, the SVFG can be fed back to the pointer analysis step to improve its precision, resulting in further improvement to the SVFG.

5.2.3 Design & Implementation

The main requirement for the BliMe linter is that the analysis must be sound but not necessarily complete. This means that the analysis may fail to mark certain BliMe-compliant

[1] Open-sourced at https://github.com/ssg-research/BliMe-linter

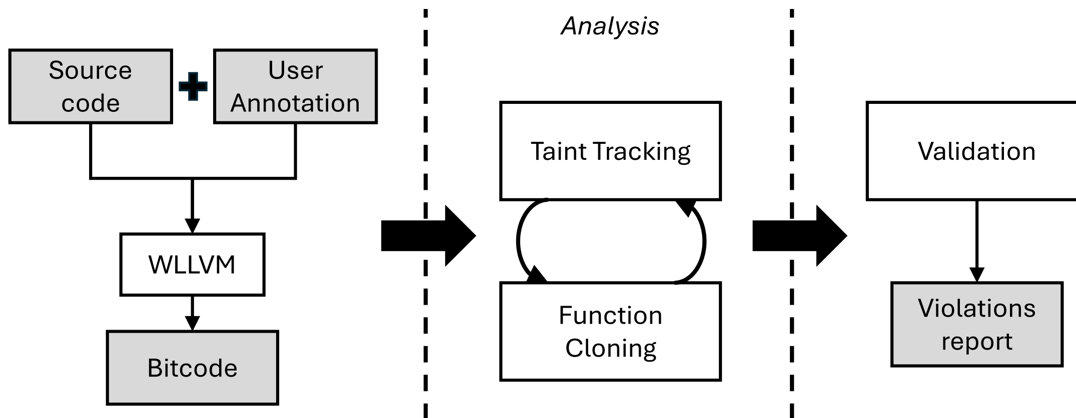


Figure 5.6: The BliMe Linter Overview.

```
1  __attribute__((blinded)) char x = 123;
2  __attribute__((blinded)) char *y = new char[10];
```

Listing 2: Examples of using the `blinded` attribute.

code as safe, but code reported as safe by the linter must have no violations on the actual BliMe hardware.

The high-level design of the BliMe linter is shown in Figure 5.6. The input to the BliMe linter is the source code, annotated to mark which data is sensitive, e.g., a secret key. Internally, the BliMe linter is divided into two steps (Analysis and Validation). The Analysis step uses the sensitive data annotated by the developer as *taint sources* and performs static taint-tracking analysis to identify where the taint can propagate throughout the program. The Validation step then identifies all locations in the program that might use tainted data in a non-BliMe-compliant manner, and creates a violations report. Note that since the analysis is not complete, the presence of violations in the final report does not necessarily mean that the executable will not run on BliMe; the violations might be a result of over-approximations of the analysis as discussed in Section 5.2.5. On the other hand, if the violations report is empty, the executable is guaranteed to run on BliMe.

Developer Interface

Since the BliMe linter relies on developer annotations to identify sensitive data, we introduce a new C++ Clang attribute (`blinded`) that the developer can attach to variables in the source code. The attribute works similar to the C/C++ type qualifiers such as `const`

and can similarly be applied to primitive types and complex types such as structures and pointers. The example in Listing 2, shows a blinded primitive type (`char x`) and a pointer to an array of blinded characters.

We have extended the Clang front-end to recognize the `blinded` attribute and pass them on in the lowered LLVM intermediate representation (IR), which is then passed to the Analysis step. As the LLVM compiler pipeline may drop unknown attributes, we schedule our analysis pass before other LLVM IR passes that may discard them.

Whole-program LLVM

When compiling C++ projects with more than a single `.cpp` file, build systems usually compile each file to a separate object file and then links all object files together at the end to produce an executable. Thus analyses or optimizations *across* modules at the *LLVM IR* level are not possible. To solve this issue, we use whole-program LLVM (WLLVM) [205]. During compilation, WLLVM generates LLVM bitcode for each compilation module and adds it to a custom ELF section in the corresponding object file. When the final executable is linked, WLLVM concatenates all bitcode sections from the linked files and adds them to a section in the ELF executable. A WLLVM tool can then be used to extract and link the bitcode. The result is a bitcode file that represents the entire program. This allows us to easily analyze the entire program (i.e., across modules) using a single input bitcode file.

Analysis

As shown in Figure 5.6, the Analysis step consists of two parts that are also performed iteratively: Taint Tracking and Function Cloning. Taint Tracking uses static value-flow tracking to propagate taint from the taint sources to the rest of the program. If there are any functions with newly tainted arguments, they are cloned, and the compiler then start another round of Taint Tracking. If there are no functions with newly tainted arguments, the compiler exits the Analysis step.

Taint Tracking We adapt the state-of-the-art SVF [180] tool to perform static value-flow tracking in the Taint Tracking step. From SVF, we first obtain a SVFG, which shows the dependencies between values. We then traverse the SVFG breadth-first and propagate taint from the taint sources to the rest of the program along the edges of the SVFG.

There is one case, however, which is not covered by SVF: *implicit flows*, which are information flows from a condition to value assignments that depend on this condition. As

a *partial* remedy for this gap, we propagate taint for LLVM `select` instructions. We use LLVM's internal def-use chains to detect such cases and propagate taint from the condition to the output. However, the BliMe linter intentionally does not handle all implicit flows as this would produce many warnings, making it harder to identify the root branching violation. Instead, we rely on the developer using an iterative "find-and-fix" approach until the BliMe linter reports no violations (indicating that the program is now BliMe-compliant). We discuss this further in Section 5.2.4.

Function Cloning We introduce function cloning to improve the context-sensitivity of the analysis. A single function can be called from many call sites in the code. When taint propagates to the arguments of this function, we must track the taint within the function and, if the return value is tainted, propagate the taint back to the call site. However, doing this without function cloning can cause two issues:

1. **Return value over-tainting:** If a function returns a tainted value at only a subset of its call sites, tainted return values will incorrectly propagate taint back to all call sites, even those not within this subset. For example, if a simple `add` function is called with tainted arguments at site ⓐ and untainted arguments at site ⓑ, taint will propagate back to the return value at both sites. With function cloning, the function called at ⓐ will be a taint-propagating clone of that called at ⓑ, and only the return value at ⓐ would become tainted, as expected. Another example is when two calls use tainted arguments, but only one should have a tainted return value.
2. **Forward call over-tainting:** The situation described above can also occur for forward calls. A function called with tainted arguments can propagate these arguments to other functions calls within it. Consider the `add` example above. Without function cloning, if the `add` function calls another function, e.g., `copy`, with `add`'s tainted arguments, every call to `copy` will result in taint propagation, significantly increasing unnecessary taint.

One drawback to function cloning is the increased code size caused by the additional functions. However, this is a reasonable trade-off considering the issues discussed above.

5.2.4 Evaluation

To evaluate the soundness of the BliMe linter, we first analyze the soundness of SVF. While the authors [180] do not claim any soundness guarantees, we believe it is safe to assume

that SVF is sound with respect to *explicit value flows* as per its design. Therefore, up until the first time tainted data flows into a condition (e.g., the condition in an if-then-else, or the condition in a select instruction), our analysis is sound. Beyond this point, we can no longer provide soundness guarantees. In other words, we make no claims that if violations are reported, that they are the *only* violations. However, once a developer transforms a violating condition to a BliMe-compliant form, and reruns the analysis, our soundness guarantees will hold further along the program's execution paths. Through an iterative process, we can therefore claim soundness as follows: if the BliMe linter does not report any violations, then the program is guaranteed to work on BliMe.

		Linter Results		Spike Results	
Program	LOC	Memory	Branch	Memory	Branch
binary_search	34	1	3	0	2
dijkstra	188	6	5	5	1
dnn	77	0	0	0	0
find_max	30	0	1	0	1
int_sort	98	0	1	0	1
kmeans	106	0	2	0	2
matrix_mult	58	0	0	0	0
page_rank	101	3	0	3	0
PQ	148	0	4	0	4
freqmine	2205	196	131	541	119
swaptions	1163	3	35	230	199
label_image	1,154,882	4,047	13,101	1	27

Table 5.2: Results of running the BliMe linter on the OISA benchmarks, PARSEC benchmarks and TensorFlow Lite. We compare the reported violations with results obtained dynamically on BliMe-Spike. LOC indicates lines of code.

We further evaluate the BliMe linter empirically against a debug-enabled implementation of BliMe on the RISC-V Spike emulator, which we call BliMe-Spike. This debug-enabled implementation does not fault when it identifies BliMe violations, but instead produces warnings that identify the offending instructions. This allowed us to run each executable only once to obtain several violations instead of having to "fix" and rerun the executable for each one. Due to its dynamic nature, BliMe-Spike has the following properties.

- It can only identify violations that are on the execution path. This results in a list of violations that is a subset of that produced by the BliMe linter.

- It intentionally does not propagate taint through implicit flows. It will correctly report a violation on a branching condition that uses a blinded value, but will not propagate this violation to assignments performed within the violating then/else branches that would be unreachable with BliMe enforcements enabled. This is because BliMe-Spike does not understand high-level program semantics and therefore cannot reason about where the branches end. The other design choice would have been to propagate taint to *all* assignments after the branching violation, but this would produce many false positives and would reduce the overall usefulness of the output.

- The same violation can result in many duplicate outputs. For example, a branching violation within a for loop that performs 100 iterations will produce 100 warnings instead of one. BliMe-Spike's output therefore requires post-processing to remove duplicates.

We selected several applications for the empirical evaluation: the OISA benchmarks [218], two PARSEC benchmarks, and an image classification example from TensorFlow Lite [1].

OISA

Yu et al. use a suite of benchmarks to evaluate OISA by comparing the performance before and after the benchmarks are made constant-time. We therefore use the non-constant-time versions as a starting point for our evaluation. We compile the OISA benchmarks, using the same sensitive inputs as Yu et al. and annotating them with the `blinded` attribute to mark them as taint sources. We then compare our analysis results with the violations reported by BliMe-Spike. The results are shown in Table 5.2. We present two case studies from the OISA benchmarks that show the effectiveness of the BliMe linter.

`find_max` Our first case study uses the `find_max` benchmark, which is a simple program that scans an array of secret values and finds the maximum value in that array. We show the source code for `find_max` in Listing 3.

As can be seen in the listing, we add the `blinded` attribute to the `arr` pointer parameter. This marks the data *within* the array as blinded. Our analysis correctly identifies a BliMe violation on line 7. The violation is due to the use of a blinded value (`arr[i]`) in the condition of a branch. The BliMe linter has an optional feature to output LLVM IR that

```
void FindMax(
    __attribute__((blinded)) int arr[],
    int* max_idx,
    int* max_val){
  *max_val = -1;
  for (int i = 0; i < N; i++){
    if (arr[i] > *max_val){
      *max_idx = i;
      *max_val = arr[i];
} } }
```

Listing 3: Source code of OISA benchmark find_max for finding the maximum value.

contains taint attached to instructions as debugging metadata. The IR output from the find_max analysis is shown in Listing 4.

```
%8 = getelementptr inbounds i32,
        i32* %0, i64 %7
%9 = load i32, i32* %8, align 4, !t
%10 = icmp sgt i32 %9, %6, !t
br i1 %10, label %11, label %14, !t
```

Listing 4: LLVM IR of BliMe violation in the find_max benchmark.

Instructions marked with a !t represent blinded data. %8 is the pointer to a[i], i.e., &a[i]. Note that it is correctly not marked as blinded. The following load fetches the blinded data from the array and stores it in %9, which is correctly marked as blinded. Taint is propagated by the analysis to %10, which is then used as a condition in a branching instruction on the following line, resulting in a BliMe violation.

page_rank For the page_rank benchmark, we examine the violation shown in Listing 5. graph is a pointer to a blinded struct representing the secret graph of web pages. This causes two violations on line 3. The first is when numOutEdges is loaded, and the second is when it is stored (after incrementing). The exact taint propagation is shown more clearly in the LLVM IR in Listing 6. %10 loads the value of e->src, which is blinded. A pointer (%12) is derived from this value and used to load numOutEdges (%13). This is marked as a violation due to a load using a blinded address. The same pointer is then used again to

store the incremented value on the line 7, resulting in another violation, this time due to a store using a blinded address.

```
1  for(int i = 0; i < numEdges; i++){
2    Edge* e = &graph->edges[i];
3    graph->vertices[e->src].numOutEdges++;
4  }
```

Listing 5: Source code snippet of load/store violation in OISA benchmark `page_rank`. `graph` is a pointer to a blinded struct. There are two violations reported on line 3: a store and a load.

```
1  %10 = load i32, i32* %9, align 4, !t
2  %11 = sext i32 %10 to i64, !t
3  %12 = getelementptr inbounds %struct.Vertex,
4        %struct.Vertex* %5, i64 %11, i32 2, !t
5  %13 = load i32, i32* %12, align 8, !t
6  %14 = add nsw i32 %13, 1, !t
7  store i32 %14, i32* %12, align 8, !t
```

Listing 6: LLVM IR of BliMe violations in the `page_rank` benchmark.

PARSEC

We select two benchmarks (`swaptions` and `freqmine`) from the PARSEC suite and mark their inputs as taint sources. They are selected based on easiness of RISC-V cross-compilation, which is not readily supported in PARSEC. For `freqmine`, we taint data as it is read from the input file. `swaptions` generates random inputs instead of reading them from a file; we taint the inputs immediately after they are generated. The results for the BliMe linter and BliMe-Spike are included in Table 5.2.

TensorFlow Lite

For TensorFlow Lite, we used the image classification example, `label_image`, marking the input image as sensitive. With BliMe-Spike, we obtained 27 branching violations and 1 memory access violation. 17 of the 27 branching violations were due to uses of the

`std::max` and `std::min` function in libc. 17 of all violations were in the `gemmlowp` matrix multiplication library used internally by TensorFlow.

With the BliMe linter, we faced challenges when running our Analysis iterations on the image classification example. Attempting to run the Analysis until no further iterations were required (i.e., function cloning produced no changes) resulted in a run-time greater than 70 hours and memory usage greater than 110GB. We discuss this in more detail in Section 5.2.5. Our analysis was eventually killed due to lack of resources and we therefore decided to disable function cloning and rerun the analysis. Our memory usage was still high ($\approx$ 80GB) but run-time was significantly lower ($\approx$ 24 hours). As a result, however, the analysis was imprecise and there was a significant amount of taint over-approximation. This is evident in the results shown in Table 5.2. We discuss the challenges of using our approach for large programs in Section 5.2.5.

5.2.5 Discussion & Future Work

SVF execution

One challenge we faced when applying the BliMe linter to a large executable such as TensorFlow was SVF's very long run-time and high memory overhead, which align with reports on SVF's GitHub page. The cause of this is two-fold. The first is that pointer analysis takes a long time to cover the entire executable. The second is the SVFG generation process itself.

One of the reasons the analysis takes a long time is that SVF builds an SVFG of the *entire* program without considering where the taint sources are located. A possible solution is to limit the analysis to the parts of the program that might contain taint and build the SVFG in tandem with taint propagation. For this to work, the SVFG generation process would first start at the taint sources, and incrementally build the value flow paths propagating from those points. The end result is that the linter would ignore the parts of the program that never process taint, therefore avoiding unnecessary processing and improving performance. A potential challenge with this approach is that some steps in the SVFG generation process might already require analyzing a large part of the program, e.g., precise alias analysis might require analyzing the entire program to identify aliasing in seemingly unrelated parts of the code due to global variables.

Taint over-approximation

As with any static analysis technique, our analysis must make approximations to be computationally feasible. We choose to use a conservative approach that over-approximates rather than under-approximates taint. This aligns with our goal of guaranteeing that the program will run if no violations are reported.

While this is sufficient to guarantee BliMe compatibility for a subset of programs, it may result in false negatives. We implemented function cloning to achieve partial context sensitivity and eliminate some false negatives but leave further exploration of more accurate analyses as future work. Additional source code annotations could also be added to allow programmers to explicitly denote data as non-blinded to limit the complexity of the analysis.

Applicability

Although we focus in this paper on BliMe, the BliMe linter is applicable to any system that requires data-oblivious execution. One prominent example is OISA [218], which enforces a similar policy to BliMe. Furthermore, the BliMe linter's analysis is useful even for software that runs on ordinary processors without BliMe or other data-oblivious execution extensions because it can help developers reason about how their software handles sensitive data and identify potential leaks.

Transformations

The BliMe linter provides developers with valuable information about potential side-channel vulnerabilities in their programs. Developers can then use this information to manually transform the programs to remove these vulnerabilities. However, with a large volume of vulnerabilities, manual transformations can be cumbersome and time-consuming. Therefore, a natural next step for future work is to extend the compiler to perform these transformations automatically. While this might not be feasible in all cases (e.g., if static analysis cannot identify proper bounds for a blinded pointer), it can significantly reduce the amount of manual labor required, especially if a program contains many easily-transformed violations such as conditional select statements.

One way to integrate transformations into the BliMe linter is by using Constantine [30]. Constantine is a set of LLVM compiler extensions that perform control-flow linearization (CFL) and data-flow linearization (DFL) transformations to produce constant-time code. CFL is done using a form of predicated execution, which computes a predicate for each

branch, executes both branches, and masks the results using the predicate to obtain the correct result and discard the result from the incorrect branch. DFL linearizes array accesses that use a tainted index by iterating over the entire array using a stride equal to the cache line size, and selecting only the value at the correct index using the predicate.

BliMe faces two challenges in using Constantine as-is:

Array access expansion Constantine's array access expansion results in binaries that are only constant-time on central processing units (CPUs) with a cache line size matching the chosen stride. Furthermore, the offset of each access from the beginning of the cache line is derived from the sensitive index. On BliMe, this will cause all the resulting memory access addresses to also be blinded and thus cause the instructions to fault. We can solve this by modifying the array access expansion transform to use a stride of one, which results in no sensitive offset being added to each address and the addresses remaining unblinded.

Select transform Constantine uses x86-specific optimizations such as the `cmov` instruction. RISC-V does not have such an instruction, which causes IR `select` instructions to be lowered to conditional branching instructions in assembly. Thus integration with Constantine must include special handling for IR select instructions.

Chapter 6

Program-Counter-Based Isolation

6.1 Introduction

In traditional operating systems, workload isolation follows process boundaries and privilege levels, relying on the fundamental abstraction that each process executes in a separate virtual address space with distinct memory mappings. The operating system kernel, running at elevated privilege, mediates all interactions between processes through well-defined system call interfaces, ensuring that untrusted user code cannot directly access another process's memory or resources. This isolation is enforced through page-based memory protection, where the memory management unit (MMU) checks permissions stored in page table entries on every memory access. Page permissions specify whether a page can be read, written, or executed, and whether it is accessible from user mode or kernel mode, enabling the kernel to prevent user processes from accessing kernel memory and to enforce write protection on code segments. Dynamic libraries mapped into a process context are typically trustworthy system services or well-established support libraries, requiring no isolation from the rest of the program.

Modern software architectures, however, increasingly challenge this assumption. Web servers routinely handle sensitive cryptographic keys while processing requests from potentially compromised code paths. Mobile applications integrate third-party libraries for functionality ranging from analytics to payment processing, each representing a potential security vulnerability within the application's address space. These scenarios demand *in-process isolation* mechanisms that can protect sensitive data and trusted code from untrusted components *within the same process*. The real-world importance of robust in-process isolation is underscored by vulnerabilities such as CVE-2021-32629, a sandbox escape in

the Wasmtime and Lucet WebAssembly runtimes that allowed attacker-controlled code to access memory outside designated sandbox boundaries due to a code generation bug in the Cranelift compiler [33].

Software-based fault isolation (SFI) [196] techniques provide in-process isolation by designating specific memory regions for component code and data, then enforcing that all memory references remain within designated boundaries through compile-time instrumentation or binary rewriting. While SFI provides a software-only solution, it suffers from significant performance overhead and requires either compiler instrumentation or expensive runtime binary analysis. Moreover, SFI primarily constrains memory accesses and must be combined with orthogonal control-flow integrity mechanisms to prevent control-flow hijacking attacks.

An alternative approach to in-process isolation uses the existing page-based memory protection infrastructure by assigning different page permissions to trusted and untrusted code and data regions. However, this approach incurs high overheads because changing page permissions requires modifying page table entries and flushing translation lookaside buffer (TLB) entries, operations that can take thousands of cycles. For applications requiring frequent domain transitions—such as web servers accessing cryptographic keys or mobile applications calling into third-party libraries—the cumulative cost of TLB flushes renders page-table-based in-process isolation impractical.

Recognizing these limitations in both software-based and page-table-based approaches, hardware vendors have introduced permission overlay mechanisms to enable efficient in-process isolation. Intel Memory Protection Keys (MPK) [135] and Arm Permission Overlay Extensions (POE) [202] allow software to assign pages to protection domains and rapidly modify permissions for entire domains by writing to a permission overlay register (POR). Unlike traditional page table modifications that require expensive TLB flushes, POR updates can be performed from userspace in tens of cycles, enabling practical sandboxing of performance-critical components.

However, permission overlay mechanisms introduce a fundamental security vulnerability that undermines their effectiveness as standalone isolation primitives. The POR write instructions that modify permissions are executable from userspace, meaning an attacker who achieves arbitrary code execution within a sandbox can directly manipulate the POR to escalate privileges and bypass all isolation guarantees. This architectural flaw forces permission overlay systems to rely on complementary protection mechanisms. ERIM [194] employs binary inspection to identify and rewrite all instances of POR write instructions in application code, ensuring that permission modifications occur only at designated call gates. While ERIM's approach adds minimal runtime overhead, it requires either trusted

compilation or runtime binary analysis, complicating deployment and limiting compatibility with legacy code and proprietary libraries. Hodor [89] takes a different approach, using hardware watchpoints (debug registers) to vet each execution of the POR write instruction at runtime, ensuring it occurs only in authorized contexts. This approach avoids binary rewriting but introduces measurable performance overhead, ranging from 1.5% to 9.85% depending on the switching rate and workload characteristics.

This chapter presents Program-Counter-Based Isolation (PBI), a novel hardware mechanism that eliminates the fundamental security vulnerability in permission overlays by incorporating `source→target` access control semantics into the hardware permission checks, represented by a `src_id` and a `tgt_id`, respectively. The `src_id` is determined by the *location of the executing instruction*, i.e., the program counter (PC). The `tgt_id` is determined by the target address of the load, store or jump instruction.

Beyond the core isolation mechanism, PBI introduces permission inheritance to address a practical challenge that arises when sandboxed code must interact with trusted shared libraries. Traditional isolation approaches face a dilemma: either grant libraries elevated permissions, creating confused deputy vulnerabilities, or maintain separate copies of libraries for each sandbox, introducing memory and cache overheads. PBI's permission inheritance allows library code to automatically adopt the permissions of its caller, enabling efficient code sharing without compromising security. When sandboxed code invokes a trusted library function, the library executes with the sandbox's restricted permissions, preventing it from being exploited to perform unauthorized operations on the sandbox's behalf.

Our evaluation demonstrates that PBI achieves its security goals while maintaining performance competitive with unprotected execution. The primary source of overhead in PBI stems from additional context switch costs for saving and restoring PBI registers. However, our experiments show this overhead is negligible—geometric mean of 0.2% across SPEC CPU2017 benchmarks [177]—because the additional registers represent a small fraction of the total architectural state swapped during context switches. Compared to permission overlay systems that employ binary scanning or hardware watchpoints, PBI eliminates these secondary protection mechanisms entirely, resulting in simpler implementations with smaller trusted computing bases and fewer opportunities for security vulnerabilities.

In summary, this chapter makes the following contributions:

1. We present Program-Counter-Based Isolation (PBI), a hardware mechanism for in-process isolation that incorporates the program counter into permission checks, enabling automatic domain transitions without explicit permission register updates and eliminating the attack surface introduced by userspace-executable permission modification instructions (Section 6.5.1).

2. We introduce permission inheritance, a mechanism that allows trusted library code to safely execute with caller permissions, enabling efficient code sharing across mutually distrusting protection domains without requiring separate library instances or explicit permission updates (Section 6.5.2).

3. We implement (Section 6.6) and evaluate (Section 6.7) PBI, demonstrating that it provides robust isolation without requiring binary scanning, instruction filtering, or other secondary protection mechanisms, while maintaining performance within 0.2% of unprotected execution for CPU-intensive workloads.

6.2 Background

6.2.1 Permission overlays

Permission overlays enable fine-grained memory protection without expensive system calls or page table modifications. As shown in Figure 6.1, each memory page page table entry (PTE) contains a permission overlay index (POI), typically 4 bits, that indexes into a thread-local POR (or PKRU on Intel x86) to obtain RWX overlay permissions. Final permissions result from ANDing page table permissions with overlay permissions. The key advantage is userspace POR modification (EL0 on Arm, userspace on x86). Applications can dynamically change memory permissions for regions containing multiple pages by updating register values rather than invoking kernel operations or page table updates. This avoids TLB invalidations and system call overhead. For example, a just-in-time (JIT) compiler can assign the same POI to all generated code pages, then toggle write permissions by updating a single POR bit. This enables rapid switching between code generation (write-enabled) and execution (write-disabled) phases without kernel involvement. The mechanism provides spatial efficiency since multiple pages share POIs, and temporal efficiency through fast permission updates. However, it requires manual permission management and exposes POR write instructions that attackers could exploit.

While Intel MPK and Arm POE serve similar purposes, they differ in key aspects. Intel MPK supports 16 domains with 4-bit keys but only controls access and write permissions, lacking execute control. Arm POE supports 8 domains with 3-bit indices but provides full RWX control including execute permissions. We build PBI on top of Arm POE as we leverage the execute permissions to control domain transitions (Section 6.5.1).

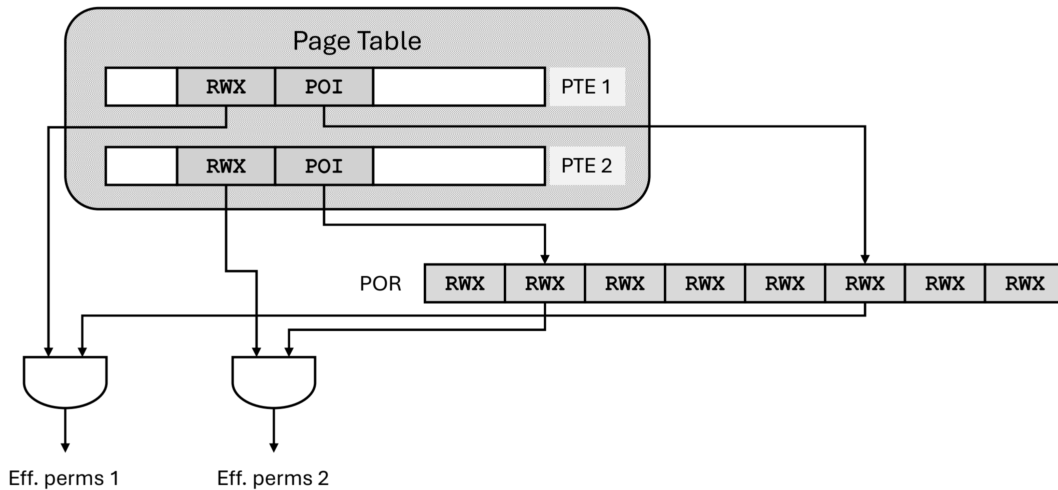


Figure 6.1: Permission overlay mechanism. Each PTE contains a POI that indexes into a thread-local POR to obtain overlay permissions. The final effective permissions are computed by ANDing the PTE permissions with the overlay permissions from the POR.

Apple Fast Permission Restrictions. Apple first introduced a custom implementation of POE in their A11 Bionic and S3 system-on-chips (SoCs) in 2017. This was before Arm introduced their POE specifications in 2022. As such, Apple's implementation differs slightly: the hardware does not allow pages to simultaneously have write and execute permissions (i.e., W$\oplus$X is enforced).

6.2.2 Shadow stack

Shadow stacks are security mechanisms designed to protect control-flow integrity by maintaining a separate, protected copy of function return addresses. Traditional program stacks store return addresses on the stack alongside other data including local variables, function arguments, and saved registers. This creates vulnerability to memory safety violations, particularly buffer overflows that can overwrite return addresses and redirect program execution to attacker-controlled code in return-oriented programming (ROP) attacks.

A shadow stack addresses this vulnerability by maintaining return addresses in a separate memory region with restricted access permissions. When a function is called, the return address is pushed onto both the conventional program stack and the shadow stack. Upon function return, the return address from the program stack is validated against the corresponding entry on the shadow stack. Any mismatch indicates potential tampering and

triggers a security exception, preventing the compromised return address from being used.

Shadow stacks can be implemented either in software or hardware. Software-based shadow stack implementations, such as those provided by LLVM's ShadowCallStack instrumentation [173], use compiler transformations to instrument function prologues and epilogues with code that maintains the shadow stack. These implementations store the shadow stack in a dedicated memory region and rely on software checks and memory protection mechanisms to prevent unauthorized access. While software shadow stacks provide portability and can be deployed on processors without dedicated hardware support, they incur runtime overhead from the additional instrumentation and may be vulnerable to attacks that compromise the software-based protection mechanisms.

Hardware shadow stacks offer stronger security guarantees through architectural enforcement. Intel introduced Control-flow Enforcement Technology [174], which includes a hardware shadow stack implementation, in their processors. Similarly, Arm developed the Guarded Control Stack feature [87], providing analogous protection for Arm architectures. Hardware implementations ensure that shadow stack memory cannot be modified through normal store instructions, requiring dedicated instructions that are tightly controlled by the processor. The processor automatically manages shadow stack operations during function calls and returns, ensuring that even if attackers achieve arbitrary write capabilities within the conventional program stack through memory safety vulnerabilities, they cannot corrupt the protected return addresses stored on the shadow stack.

Beyond protecting return addresses, shadow stacks can be extended to store additional security-critical metadata that must be protected from software modification. The hardware-enforced access restrictions and automatic management by the processor make shadow stacks an ideal location for maintaining security state that transitions across function calls. This extensibility enables shadow stacks to support advanced security mechanisms beyond basic control-flow integrity, as demonstrated by PBI's use of shadow stacks to maintain permission inheritance state across function boundaries (Section 6.5.2).

6.3 Threat Model

We assume a similar threat model to prior work. Attackers have access to an arbitrary write primitive inside a sandboxed region of the program. They can use this primitive to attempt to hijack control of the program, e.g., by modifying function and return pointers on the stack and heap to launch ROP attacks. However, the write primitive is bound by the MMU-enforced page protections. Nevertheless, attackers can attempt to disable such

protections by executing `mprotect` syscalls or POR write instructions as part of a ROP attack.

Out of scope are side channel attacks and attacks requiring physical access to the system. Furthermore, we trust the operating system (OS) to provide adequate isolation between processes; i.e., processes cannot attack other processes directly without first compromising the OS. We also assume hardware implementations are free of defects.

6.4 Goals and Challenges

We now present our main goals and challenges.

Incremental adoption Creating radically different and novel architectures with security in mind from the start, such as CHERI [211], can lead to strong security guarantees. However, as with the use of memory-safe languages, mainstream adoption becomes challenging as entire software stacks and developer programming patterns must change to adapt to the new architectures. With PBI, we therefore purposefully design a security mechanism that is easily adoptable into existing workflows. Page table permissions are ubiquitous, and permissions overlays have already gained support from the Linux kernel community with the introduction of the `pkeys` API in v4.9 [135]. PBI builds on top of permission overlays with minimal software changes: page table entries are augmented with a single additional field (`src_id`), and the POI field is renamed as the `tgt_id`. Furthermore, by using a single `src_id` for all pages, PBI reduces to permission overlays. This allows easy incremental adoption as the codebase can be transitioned from traditional permission overlays to PBI gradually.

Minimal hardware changes In addition to adoption by software developers, we aim to ease adoption of PBI for hardware vendors. As such, PBI requires minimal hardware changes. Existing circuitry for the POI field can be used to implement `tgt_id` handling. Existing permission check circuitry is augmented to handle `src_ids` in parallel, requiring no additional cycles. As we explain in detail in Section 6.5.2, even with PBI-PI, our design relies on existing shadow stack features to implement permission inheritance (PI), adding no additional cycles for hardware that already implements a shadow stack.

Instruction set architecture (ISA) compatibility To ease adoption even further for both software developers and hardware vendors, we require no ISA-specific feature to

implement PBI. For PI, shadow stacks are available in both ISAs (Arm Guarded Control Stack (GCS) and Intel CET).

Flexibility Prior in-process isolation (IPI) designs such as HFI [143] focus on sandboxing untrusted code and introduce a special "sandbox" central processing unit (CPU) mode. While they are efficient for the sandboxing use case, they are difficult to adapt to other use cases such as protecting sensitive keys because that would require an inversion of the trust model assumed by these designs. Traditional MMU page protection provides a large degree of flexibility, allowing developers to assign RWX permissions freely. They can therefore be used as a primitive to implement more sophisticated schemes that target a range of use cases. We aim to maintain this flexibility. Unlike prior work [143], PBI does not require a special "sandbox" mode in the CPU, and does not introduce sandbox enter and exit instructions. Instead, it provides a powerful primitive that can be used to implement sandboxing efficiently, but can also be used to protect memory in other ways, such as creating a lightweight in-process trusted execution environment (TEE) where access control is inverted compared to the sandbox scenario.

6.5 PBI Design

The key idea behind PBI is incorporating the address of the executing instruction into the permission check for read, write and execute operations. We present 2 variants of PBI: PBI-Core, and PBI-PI.

6.5.1 PBI-Core

We incorporate source→target access control semantics into the MMU permission checks, with the source representing *where* the code is executing from (i.e, the program counter (PC)), and the target representing *what* is being accessed (i.e., the target load, store or jump address). This is done by assigning pages `src_ids` and `tgt_ids` (by setting them in the corresponding PTEs). All pages must have a `tgt_id` and all code (i.e., executable) pages must additionally have a `src_id`. On read, write and execute operations, we use the `src_id` of the PC's page and the `tgt_id` of the target address's page to index into a permission matrix to obtain the effective permissions, as shown in Figure 6.2. Code pages require a `tgt_id` (not just a `src_id`) because they are used when jumping *to* the code page, as described below.

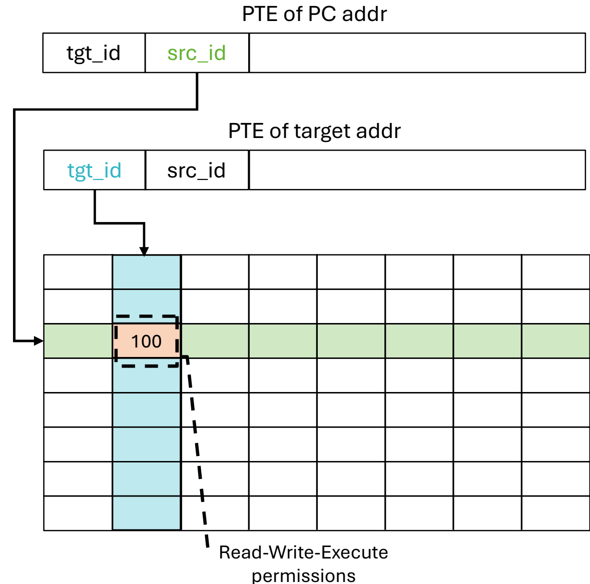


Figure 6.2: PBI permission check mechanism. The `src_id` from the PC's PTE and `tgt_id` from the target address's PTE index into a permission matrix to determine effective RWX permissions for the operation.

The hardware architecture for PBI is shown in Figure 6.3. The `src_id`s and `tgt_id`s are fetched from the PTEs by the page table walker and cached in the TLB. The current `src_id` is maintained by the CPU in a new separate register and is updated as the PC moves between pages. We modify the MMU's permission check unit to use the current `src_id` and the target address's `tgt_id` to index into a permission matrix as described above; this is done by parallel circuit logic, requiring no additional cycles. The permission matrix is stored in a set of new CPU registers called PBIRs, where each row is a separate register containing multiple permission fields. The `src_id` selects the PBIR, and the `tgt_id` selects the permission field within the PBIR. We use the notation [x→y] to identify the cell in the permission matrix corresponding to code executing with `src_id` =x reading, writing or jumping to a page with `tgt_id` =y. We use [x→y][P] to represent the bit of permission P in [x→y], where P$\in \{R, W, X\}$ for read, write and execute permissions, respectively. The number and size of the registers determines the maximum number of source and target IDs, respectively. We reserve `src_id` =0 for trusted code which bypasses PBI checks (but must still adhere to the traditional PTE permissions); our permission matrix's rows therefore start from PBIR1 as PBIR0 is not needed. Each CPU core has a separate permission

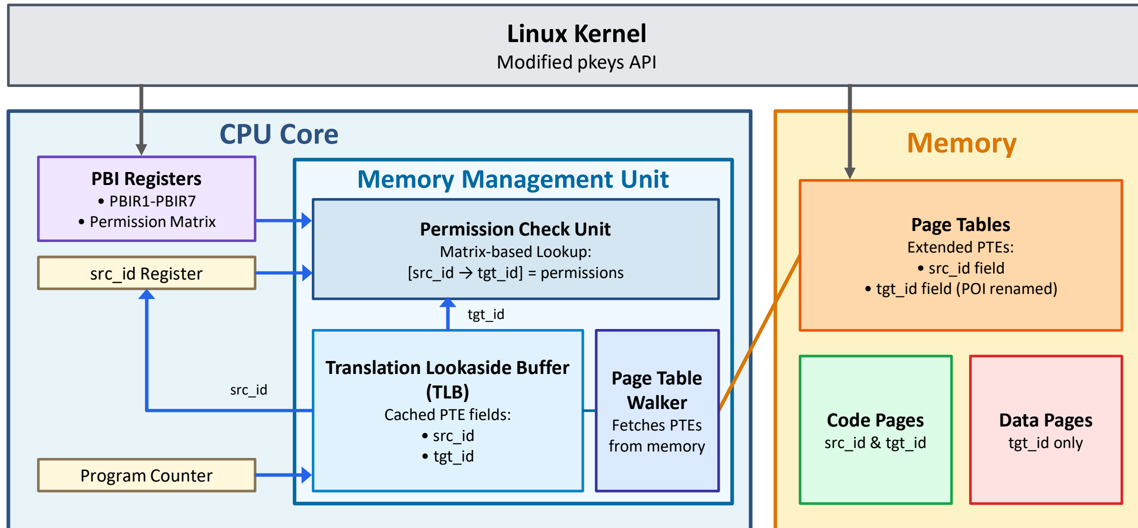


Figure 6.3: PBI architecture overview showing CPU core components including the `src_id` register, PBI registers (PBIRs), and MMU integration with TLB caching of PTE fields (`src_id`, `tgt_id`).

matrix, making the permissions thread-local. One way to think of the permission matrix is as an extension of Arm POE where each row corresponds to a POR, and the `src_id` determines which POR to use. However, unlike Arm POE, PBIRs can only be modified from kernel space.

In addition to reads and writes, the permission matrix also checks control-flow changes and transitions across pages (which occur when the PC is incremented as the program executes), and they are only allowed if [`src_id`→`tgt_id`][X]=1. This allows us to prevent arbitrary control-flow transfers from one executable part of memory to another, and can limit the available ROP gadgets at different points of program execution (Section 6.7.1).

To assign `src_id`s, `tgt_id`s and PBIR permissions systematically, we logically partition memory into *domains*. A domain is defined as a logically separate security context (e.g., a sandbox, or a trusted library), whose code and data pages are assigned a unique set of `src_id`s and `tgt_id`s[1]. Domains do not themselves carry any special meaning in the hardware: they are only logical assignments to systemize how we assign `src_id`s and `tgt_id`s to different security contexts and how we set their permissions. For permission

[1]The values of `src_id`s and `tgt_id`s within the domain are arbitrary (as long as they do not match those in other domains), i.e., there is no requirement to match `src_id` and `tgt_id` values within a domain.

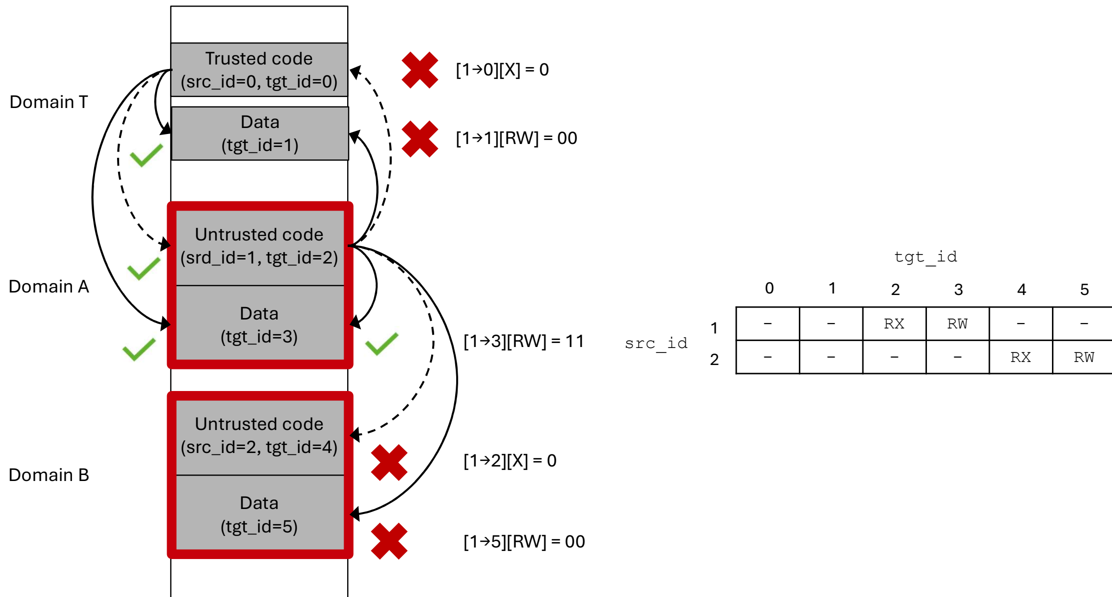


Figure 6.4: Sandboxing example with two isolated sandboxes in domains A and B, with trusted code in domain T having full access. The permission matrix for the example is shown on the right. Solid arrows are reads and writes. Dashed arrows are domain transitions. Ticks and crosses show which reads, writes or transitions are allowed and disallowed by the permission matrix.

checks, the hardware directly uses the `src_id` and `tgt_id` of the PC's and target address's pages, respectively, as discussed earlier.

We demonstrate how PBI-Core can be used for sandboxing in the example in Figure 6.4. Process memory is partitioned to contain trusted code and data in domain T, and two separate sandboxes in domains A and B. The permission matrix and the `src_id`s and `tgt_id`s of the pages in the sandbox domains are set up as shown in Figure 6.4. This configuration disallows any read, write or jump from domains A and B to domain T, and between domains A and B, effectively limiting each sandbox's access to its own code and data (including its own stack and heap). This configuration is written once to the permission matrix at program start-up through a new syscall (as PBIRs are not writable from userspace; syscalls discussed below) and requires no updates throughout the program's execution. Code executing in any of the domains has the domain's set of permissions applied, and transitions from one domain to the next automatically apply the correct set

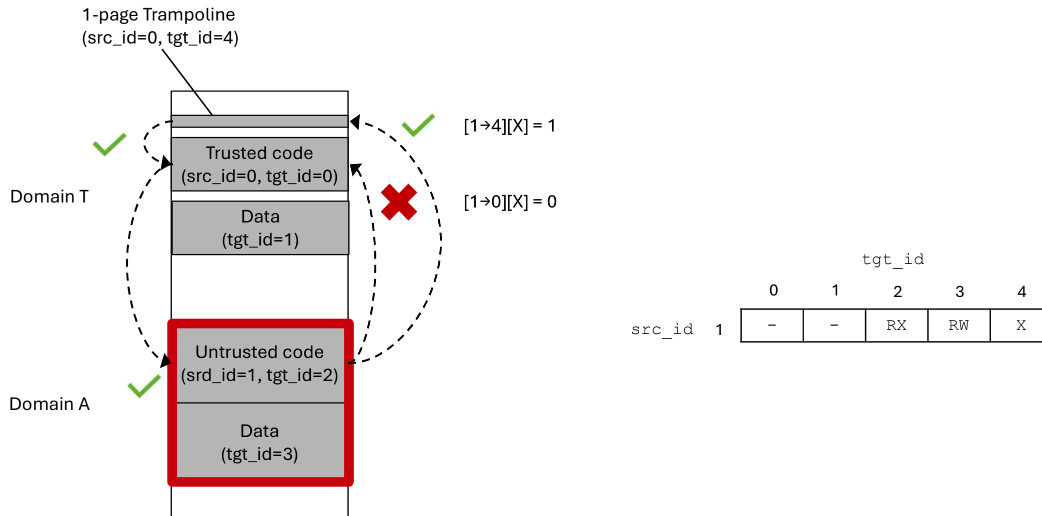


Figure 6.5: Example showing sandbox exits using trampolines.

of permissions without requiring any PBIR updates. This is in contrast to POE, where continuous POR writes are required on domain transitions.

Domain transitions

Domain transitions occur when the program attempts to jump from code with one `src_id` to code with another `src_id`. Transitions from trusted code (with `src_id` =0 for code pages) to untrusted code are straightforward and require no special handling: `src_id` =0 disables PBI checks and the program can simply jump to the untrusted code.

Transitions from untrusted code (with `src_id` =`u`) to trusted code (with `src_id` =0 and `tgt_id` =`t`), on the other hand, are more challenging. Simply setting [`u`→`t`][X] is unsafe because it gives untrusted code the ability to jump to *any part of the trusted code page*, allowing an attacker with arbitrary execution in the untrusted code to easily create gadgets from the trusted code. As the gadgets would execute with `src_id` =0, bypassing PBI checks, the attacker can use them for a confused deputy attack to access all of memory.

To perform transitions from untrusted to trusted code safely, we therefore use *trampolines*, as shown in Figure 6.5. A trampoline consists of a single code page with the same `src_id`

as the target code (`src_id` =0 in Figure 6.5), but a different unique `tgt_id`. This allows us to configure the permission matrix to allow untrusted code to jump to it without allowing untrusted code to jump directly to the target trusted code. Once execution reaches the trampoline, it can then jump safely to the trusted code, which is allowed because the trampoline has `src_id` =0 and can bypass PBI checks.

Inside the trampoline is a single direct jump instruction (to the trusted code's entry point). We use a direct jump instruction, rather than an indirect one, to prevent attackers from changing the target of the jump through register manipulation. The remainder of the page is padded with `nop` instructions to ensure it is gadget-free. We use a single page for the trampoline because that is the smallest granularity that allows us to assign permissions to the direct jump instruction that we need.

System calls

System calls provide the interface through which user-space applications can request services from the operating system kernel. They allow applications to perform privileged operations, such as modifying memory protection attributes or accessing hardware resources, that cannot be performed directly from user space due to security restrictions.

Because the PBIR registers can only be modified from kernel space, we introduce a new system call to allow trusted code to configure the permission matrix. This system call accepts parameters specifying which PBIR to modify and what permission values to set, enabling applications to establish their desired isolation policies at program initialization. The permission matrix, once configured, typically remains static throughout program execution, with domain transitions automatically applying the appropriate permissions without requiring further PBIR updates.

To prevent untrusted code from compromising the security guarantees of PBI, we add mandatory access control checks to all system calls that could modify PBIRs or page table permissions. These system calls read the `src_id` from the calling thread's task struct to determine whether the request originates from trusted code. Only code executing with `src_id` =0 is permitted to invoke these privileged operations. If a system call detects that it has been invoked from code with `src_id` $\neq$0, indicating an attempted call from untrusted code, the system call immediately redirects control to a predetermined signal handler located within the program's trusted code region. This signal handler is registered at program start-up through another new system call, allowing the trusted code to establish appropriate error handling or security responses for such unauthorized attempts. This mechanism ensures that even if an attacker gains arbitrary code execution within an untrusted domain,

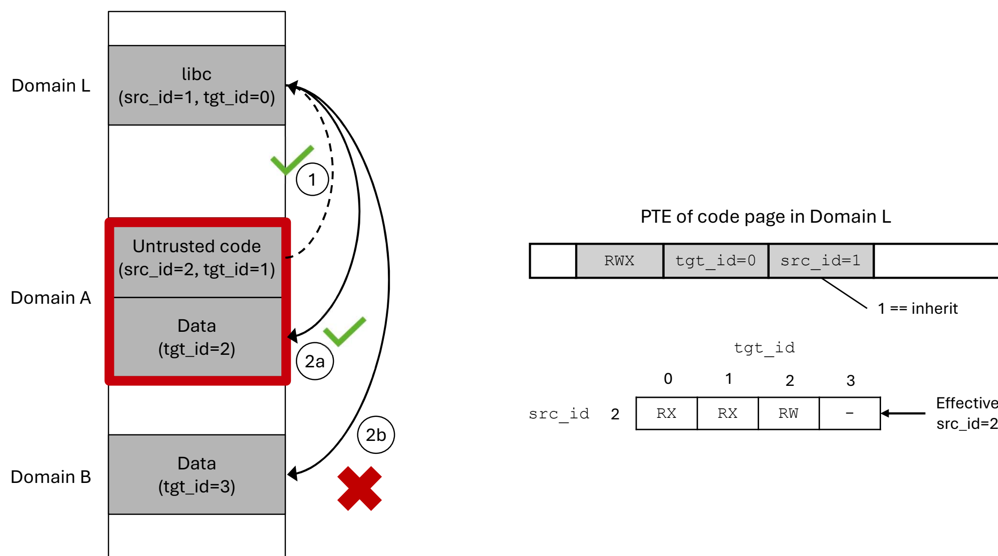


Figure 6.6: Confused deputy attack scenario where sandboxed code in domain A attempts to use shared library code (libc) in domain L to access data outside its sandbox. The permission matrix and PTE for domain L is shown on the right. The `src_id` is set to 1 to mark the code page as inheriting, resulting in the effective `src_id` being 2 when libc is called from domain A. Note that the first row of the permission matrix now starts with PBIR2 as PBIR1 is no longer needed.

they cannot escalate privileges or reconfigure the isolation boundaries enforced by PBI. We discuss additional vulnerable system calls identified in prior work [44] in Section 6.8.

6.5.2 PBI-PI

In many use cases, PBI-Core is a sufficient primitive to provide effective protection. However, consider the situation in Figure 6.6. Domain L contains trusted library code, e.g., libc, that we would like to share with the untrusted code in domain A. With PBI-Core, we can set up the permission matrix to allow the sandboxed code to jump to domain L in order to use the shared library ①. This allows the sandbox to use the shared library to operate on data within the sandbox (2a). However, this also introduces a confused deputy attack: the sandboxed code can also use the shared library to access data outside the sandbox (2b).

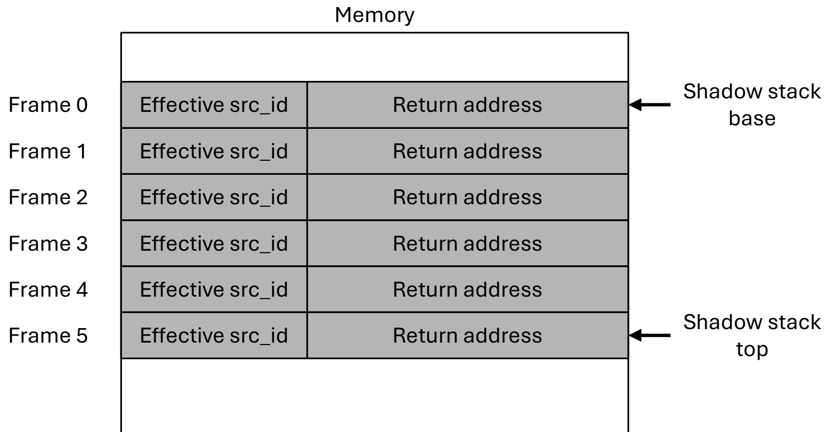


Figure 6.7: Extended shadow stack structure for PBI-PI. Each frame contains the return address and the effective source ID of the caller. The stack grows downwards.

With PBI-Core, it is not possible to allow ①-②a, and at the same time disallow ①-②b.

We therefore propose PBI-PI: PBI with *Permission Inheritance*. Permission inheritance allows the developer to set shared code pages as "inheriting", making them *inherit the* ***src_id*** *of the caller*. We reserve a specific `src_id` value (1) to signify that a page is inheriting. Revisiting Figure 6.6, by assigning `src_id` =1 to the code pages of domain L, we mark it as inheriting. As a result, the jump from the sandboxed code to libc ① does not change the *effective* `src_id`; it remains 2. Therefore, the permission check is applied *as if* the sandboxed code itself is executing, allowing ②a but disallowing ②b, as intended. Since we now also reserve `src_id` =1 for inheritance, we can remove PBIR1 and start the first row of the permission matrix with PBIR2.

Function returns to an inheriting page present a challenge. We cannot simply use the `src_id` of the function to which we are returning. We must instead determine the `src_id` this function has inherited from its own caller. We therefore keep track of the effective `src_id` on the *hardware-protected shadow stack*, e.g., Arm GCS. On each function call, we push a new stack frame consisting of the current effective `src_id` along with the return address, as shown in Figure 6.7. By relying on the hardware-protected shadow stack instead of the traditional program stack, we avoid modifying the application binary interface (ABI) for the traditional program stack, and also enable hardware protection for the effective `src_id`s, which must not be directly modifiable by software.

Figure 6.8 shows the architecture for PBI-PI, illustrating how the effective `src_id` calculation logic is integrated into the CPU pipeline and how the shadow stack is extended

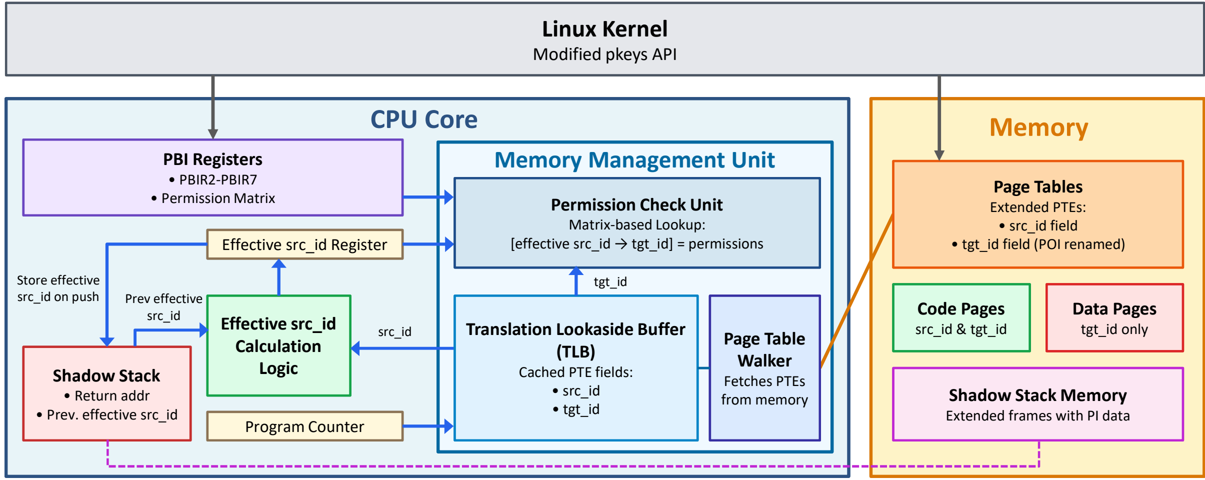


Figure 6.8: PBI-PI architecture overview showing CPU core components including the effective `src_id` register, PBIRs, shadow stack with extended frames, and MMU integration with TLB caching of PTE fields (`src_id`, `tgt_id`).

to store PI information. The architecture maintains PBIRs (PBIR2-PBIR7) that form the permission matrix, while the TLB caches the additional PTE fields (`src_id`, `tgt_id`) to avoid performance penalties during permission checks.

6.6 Implementation

We implement PBI-PI on QEMU [159] for functional testing, and add support for PBI-PI in the Linux kernel by extending the `pkeys` application programming interface (API) [135] and adding system call checks.

QEMU modifications We modified the AArch64 target to support the additional PTE fields required by PBI-PI. The CPU state structure was extended to include the `src_id` and PBI registers (PBIR2-PBIR7) that store the permission matrix. We implemented the permission matrix lookup logic in the memory access paths and extended TLB structures to cache the `src_id` and `tgt_id` fields from PTEs. The effective `src_id` calculation logic is triggered automatically when the PC crosses page boundaries or when a jump, branch or return instruction is executed. For PI, we implemented shadow stack extensions that automatically push and pop the effective `src_id` during function calls and returns.

Linux kernel modifications The Linux kernel's `pkeys` API [135], introduced in version 4.9, provides three primary system calls for managing memory protection keys. The `pkey_alloc` system call allocates an available protection key from the limited hardware-supported set, returning a key identifier that applications can use to tag memory regions. The `pkey_free` system call releases a previously allocated key, returning it to the available pool for subsequent allocation. The `pkey_mprotect` system call extends the traditional `mprotect` interface by accepting an additional key parameter alongside the standard memory address, length, and permission arguments, enabling applications to simultaneously configure both conventional page permissions and protection key assignments for a memory region.

For our implementation, we modified the Linux kernel's `pkeys` API to add support for PBI-PI. We extended the existing `pkey_mprotect` system call to accept an additional `src_id` parameter and reuse the existing `pkey` parameter to be the `tgt_id`, allowing applications to simultaneously set both `src_id`s and `tgt_id`s in the relevant PTEs. We added 2 new system calls: `pkey_set_pbir` for setting PBIRs, and `pkey_register_signal_handler` for setting the signal handler in trusted code for failed system call checks. We added system call checks to `mprotect`, `pkey_mprotect` and `pkey_set_pbir` to read the `src_id` in the calling thread's task struct and check that it is 0. If it is 0, the system call is allowed to continue. Otherwise, the kernel calls the signal handler registered through the `pkey_register_signal_handler` call. We also extended the memory management subsystem to handle the new `src_id` field, and the context switch code to save and restore PBIR values.

6.7 Evaluation

6.7.1 Security

Protection Against Control-Flow Hijacking Attacks

PBI provides inherent protection against control-flow hijacking attacks, including ROP and jump-oriented programming (JOP), by fundamentally limiting the code available to an attacker at any point during program execution. Unlike traditional memory protection schemes where permissions are properties of memory regions alone, PBI incorporates the location of the executing instruction into every permission check, creating execution-context-dependent access control.

In traditional systems without PBI, an attacker who achieves arbitrary code execution within a sandboxed component can construct ROP or JOP chains using gadgets from any executable memory region accessible to the process, including trusted libraries and

```
1 # Trusted Library (Domain T):
2 0x1000: pop rax; ret           # Gadget 1
3 ...
4 0x1100: mov [rdi], rsi; ret    # Gadget 2
5 ...
6 0x1200: syscall; ret           # Gadget 3
```

Listing 7

system calls. The attacker redirects control flow to useful instruction sequences (gadgets) throughout the address space, chaining their execution to achieve malicious objectives while adhering to W⊕X protections.

PBI disrupts this attack model by making the availability of gadgets dependent on the current execution context. When code executes with a restricted `src_id`, the permission matrix determines which memory regions can be accessed or jumped to based on their `tgt_id`. Crucially, an attacker executing within an untrusted domain cannot redirect control flow to trusted code regions, *even temporarily*, because the permission check [`src_id`→`tgt_id`][X] will fail. This occurs regardless of whether the attacker has corrupted return addresses, function pointers, or other control-flow data, because the hardware enforces the constraint that code executing with `src_id` =u cannot transfer control to pages with `tgt_id` =t when [`u`→`t`][X]=0.

Consider a concrete example with a sandboxed component in domain A (`src_id` =2, `tgt_id` =2) and trusted code in domain T (`src_id` =0, `tgt_id` =0). The trusted code contains the instruction sequences in Listing 7. Without PBI, an attacker who compromises the sandboxed component and gains control over the stack or function pointers can construct a ROP chain that includes these gadgets from the trusted library. The attacker simply needs to place the appropriate addresses (`0x1000`, `0x1100`, `0x1200`) on the stack or in corrupted function pointers to chain their execution. Because these addresses reside in executable memory and W⊕X does not distinguish between execution contexts, all three gadgets remain available for exploitation.

With PBI enabled and the permission matrix configured such that [2→0][X]=0, the same attack fails. When code executes from the sandboxed component (PC in a page with `src_id` =2) and attempts to jump to address 0x1000 in the trusted library (a page with `tgt_id` =0), the MMU's permission check unit evaluates [2→0][X] and finds it is set to 0. The jump is denied, triggering a permission fault before the gadget can execute. This protection applies equally to all three gadgets in the trusted code, effectively removing them from the attacker's available gadget space. Importantly, this restriction persists throughout

the entire execution within the sandboxed component; the attacker cannot temporarily escalate privileges to access these gadgets without first successfully transitioning through an authorized entry point (i.e., a trampoline with [2→`trampoline_tgt_id`][X]=1).

The security benefit extends beyond individual gadgets. In traditional systems, attackers can leverage the entire code base of trusted libraries, potentially finding thousands of useful gadgets. PBI eliminates all gadgets in trusted code from the attacker's reach when executing within untrusted domains, dramatically reducing the attack surface and making it substantially more difficult to construct viable exploit chains with the limited gadgets available only within the untrusted domain's own code.

Trampoline security

While trampolines serve as authorized entry points from untrusted to trusted code, their design ensures they do not introduce security vulnerabilities or expand the attack surface available to compromised sandboxed components. The security of trampolines rests on three key properties: minimal functionality, gadget-free construction, and controlled accessibility.

First, trampolines provide minimal functionality by design. Each trampoline consists of exactly one direct jump instruction that transfers control to a specific, predetermined entry point in the trusted code. The use of a direct jump rather than an indirect jump is critical—it prevents attackers from manipulating register values to redirect execution to arbitrary locations within the trusted code. An attacker who successfully redirects control flow to a trampoline can only reach the single intended entry point, not arbitrary addresses within the trusted domain. This stands in stark contrast to allowing direct execution of trusted code pages, which would permit an attacker to jump to any instruction within those pages and construct gadgets at will.

Second, trampolines are designed to be gadget-free. After the single direct jump instruction, the remainder of the page is padded exclusively with `nop` instructions. This padding ensures that an attacker cannot find useful instruction sequences within the trampoline itself to incorporate into ROP or JOP chains. Even if an attacker attempts to jump to an address within the trampoline other than the start of the direct jump, they will encounter only `nop`s followed by a transition to another page (subject to another permission check), thus providing no useful gadget functionality.

Third, trampolines maintain controlled accessibility through the permission matrix. While the trampoline page has `src_id` =0 like other trusted code, it has a distinct `tgt_id` that differs from the main trusted code pages. This distinction allows the permission matrix to be configured such that [`untrusted_src_id`→`trampoline_tgt_id`][X]=1 while

[`untrusted_src_id`→`trusted_tgt_id`][X]=0. Consequently, untrusted code can jump to the trampoline but cannot jump directly to the trusted code. Once execution reaches the trampoline and the direct jump executes, the program counter moves to a page with `src_id` =0, at which point PBI checks are bypassed and the trusted code can execute normally with full privileges. The attacker gains no additional capability by reaching the trampoline that they would not have gained through a legitimate function call to the trusted code's entry point.

The combination of these properties ensures that trampolines function as secure gateways that enforce intended control-flow transitions without compromising the isolation guarantees PBI provides. An attacker executing within a sandbox who corrupts control-flow data can at most redirect execution to a trampoline's entry point, which will then transfer control to the corresponding legitimate entry point in the trusted code. This is precisely the behavior that would occur during normal, intended operation when the sandboxed code legitimately invokes trusted functionality.

6.7.2 Performance

Open-source hardware with permission overlays is not available for modification. We therefore identify several individual factors affecting performance, measure their effect separately on available hardware, and combine them into a total overhead estimate.

We refer to a baseline without PBI or permission overlays as *SimpleBase*, and a system with vanilla permission overlays as *POBase*. The factors contributing to performance changes are:

1. Increased context switch times due to the additional PBIRss can reduce performance compared to SimpleBase and POBase.

2. Removal of POR writes on domain transitions can slightly improve performance compared to POBase.

Note that neither PBI nor PBI-PI add any additional overhead to a system with existing shadow stack support.

Increased context switching

We obtain the context switching overhead compared to SimpleBase by adding dummy loads and stores to the context switch code in the Linux kernel, and measuring the change in

performance for the SPEC CPU2017 [177]. The dummy loads and stores represent the loads and stores needed to swap the values of the PBIRs. For our evaluation, we set the number of PBIRs to 8, corresponding to 4 `stp` (store-pair) and 4 `ldp` (load-pair) instructions. We add the loads and stores in Aarch64 assembly to ensure they are not optimized away by the compiler. We take a conservative approach and assume every context switch swaps the PBIRs. In a realistic setting, swapping would occur lazily, eliding the loads and stores for processes that do not use PBI.

We run all our benchmarks on an `a1.metal arm64` AWS instance (16 cores, 32GB memory) with Ubuntu 24.04 and AWS Kernel v6.8.0-1033. We use a baremetal instance to ensure more reliable results as virtualized instances provide no guarantee on performance consistency across reboots. We configure the benchmarks to run 16 copies simultaneously to match the 16 cores of the `a1.metal` instance, and use the `ref` workload. We set the number of iterations to 2 as recommended by SPEC. We show our results in Figure 6.9 for both base and peak default CPU2017 configurations. The base and peak configurations represent moderate and high compiler optimization levels, respectively. Even with our conservative approach, results show negligible context switching overheads for both, with a combined standard deviation of 0.73%. This is not surprising as 8 additional registers is only a small fraction of the large number of registers (≈100 including System Registers) swapped during context switching on modern Aarch64 processors. Note that in this experiment, we compare against SimpleBase. POBase already swaps the value of the POR on context switching. Therefore, against POBase, PBI would have even lower overheads as we would need to add one fewer dummy load and store. As the overheads against SimpleBase are already negligible, we do not investigate this further.

POR write removal

In this section, we investigate the speedup gained by eliminating POR writes on frequent domain transitions. As Arm GCS is not yet available, we perform our evaluation on Intel MPK-enabled hardware. We modify the Mozilla SpiderMonkey Javascript engine [46], resulting in 3 variants: 1) unmodified SpiderMonkey (SimpleBase), which uses traditional `mprotect` calls to switch permissions for JIT code pages, 2) POBase, which uses Intel MPK instead of the traditional `mprotect` calls for fast permission switching, and 3) a variant that disables all protections, removing `mprotect` calls and Intel MPK, representing PBI. We then run a custom Javascript microbenchmark that calls a dummy Javascript function many times, forcing it to be JIT compiled and causing frequent domain transitions into and out of the sandboxed JIT function code, resulting in frequent POR writes for POBase and `mprotect` calls for SimpleBase. We plot the results for different numbers of function

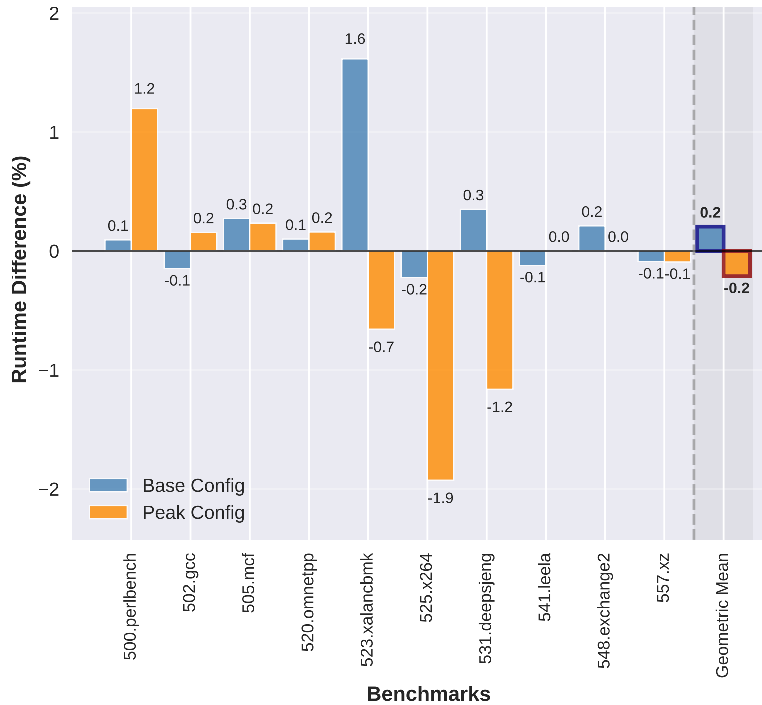


Figure 6.9: Percentage change in SPEC CPU2017 intrate runtime showing overhead of additional PBIR loads/stores during context switches for PBI compared to SimpleBase. Negative numbers represent better performance for PBI compared to SimpleBase. Base and peak configurations are the default configurations for base and peak compile options, representing moderate and high compiler optimizations, respectively. The results show negligible average overheads.

calls in Figure 6.10. SimpleBase consistently performs worse than POBase and PBI, with an average overhead of 6.41% compared to PBI (i.e., no protection). POBase shows no significant overhead (negligible average of -0.45%) compared to PBI, which demonstrates that *POR removal does not provide significant performance improvements*.

The combined results from Section 6.7.2 and Section 6.7.2 show that PBI causes no significant performance change compared to POBase, and improves performance by ≈6% compared to SimpleBase.

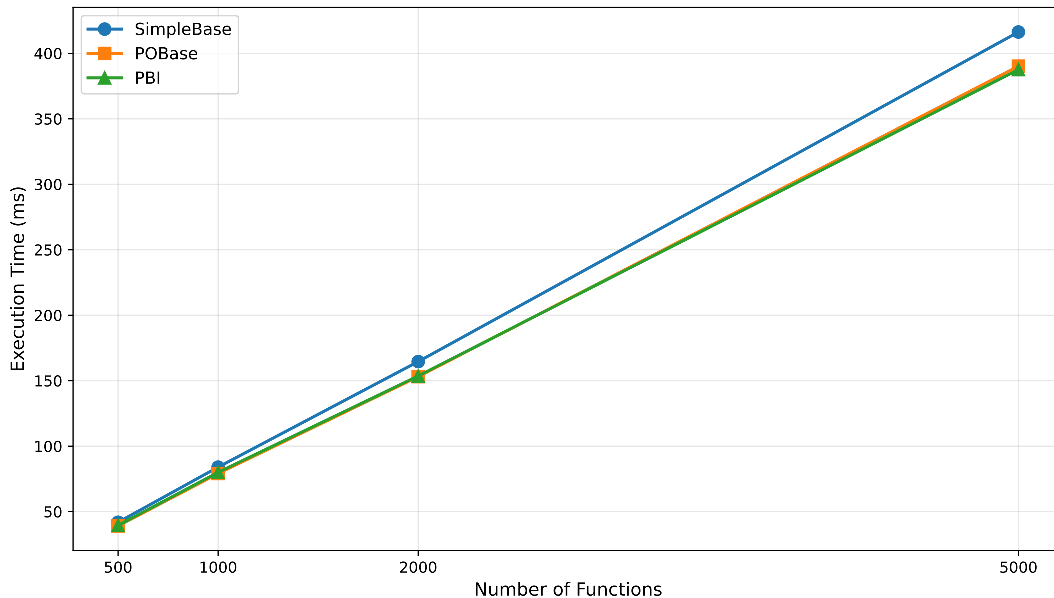


Figure 6.10: Execution run-time for Javascript microbenchmark on SimpleBase, POBase and PBI variants of Mozilla SpiderMonkey for different numbers of function calls.

6.8 Discussion & Future Work

Trampolines between mutually distrusting domains The trampoline mechanism from Section 6.5.1 assumes an asymmetric trust relationship where untrusted code transitions to trusted code. However, many scenarios involve mutually distrusting domains—where neither the caller nor callee trusts the other. This arises when multiple sandboxed libraries interact within the same address space, such as different user-level network stacks or in-memory databases sharing hardware resources while maintaining isolation.

The primary challenge with mutual distrust is safe stack switching. Each domain requires its own protected stack to prevent tampering with return addresses and local variables. When domain A calls domain B, domain A must save its state (stack pointer, thread-local storage base, and other critical registers) to a location that domain B cannot access, and domain B must restore its own previously saved state from a location domain A cannot access. PBI supports mutually distrusting domains by requiring each domain to manage its own state before a transition. State can be stored on the domain's private stack and on entry to the designated entry function (reachable from the trampoline), restored from it. This approach enables efficient transitions between mutually distrusting domains

while preserving PBI's security guarantees.

Compiler tail call optimization Tail call optimization presents a challenge for PBI-PI's extended shadow stack implementation. When a compiler performs tail call optimization, a function transfers control to another function as its final operation, reusing the caller's stack frame rather than creating a new one. This optimization improves performance by eliminating unnecessary stack operations, but creates complications for shadow stack implementations that expect a strict correspondence between calls and returns.

This challenge is not unique to PBI-PI but rather a fundamental issue faced by all hardware shadow stack implementations, including Intel CET and Arm GCS. The core problem arises because tail-optimized calls do not follow the standard call-return pattern that shadow stacks are designed to protect. When a tail call occurs, the compiler avoids pushing a new return address onto the conventional stack, but a naive shadow stack implementation would still push an entry, leading to a mismatch between the shadow stack depth and the actual call chain.

PBI-PI addresses this issue by adopting the same solutions employed by existing hardware shadow stack architectures. These solutions include:

1. **Compiler awareness** – Modern compilers that support shadow stacks emit special instructions or instruction sequences for tail calls. Instead of issuing a standard call instruction followed by an immediate return, the compiler generates code that properly updates the shadow stack to replace the current entry rather than pushing a new one

2. **Shadow stack replacement operations** – Hardware shadow stack implementations provide specialized instructions that atomically update the top shadow stack entry rather than pushing a new frame. For PBI-PI, this means replacing both the return address and the effective `src_id` in the current shadow stack frame with values corresponding to the tail call target.

By relying on these established techniques, PBI-PI maintains compatibility with optimized code while preserving the security guarantees of its extended shadow stack implementation.

Vulnerable system calls Prior work [44] has shown that POE protections can be bypassed by exploiting security holes in system calls. The authors found that certain system

calls do not consider current POE permissions before performing privileged operations on memory, enabling a confused deputy attack where adversaries can access memory they would normally not have access to in user space. The authors argue that the kernel's current security model does not consider in-process isolation, and that kernel support is necessary to ensure proper protection for in-process isolation. PBI adds such support to the kernel by design: modifying the permission matrix can only be done in kernel space, and checks are added to system calls to guarantee that an attacker cannot modify such permissions. While we do not include the changes suggested by Connor et al. [44] in our implementation, a production-ready PBI implementation must include them to prevent the aforementioned confused deputy attacks.

Applicability to eBPF The extended Berkeley Packet Filter (eBPF) enables user-provided code execution within the Linux kernel, but vulnerabilities in its static verifier have led most distributions to disable unprivileged eBPF by default. Recent defenses such as SandBPF and SafeBPF employ software-based fault isolation or Arm's Memory Tagging Extension (MTE) to sandbox eBPF programs at runtime. PBI could provide an alternative approach by assigning eBPF JIT-compiled code distinct `src_id` values, enabling automatic permission transitions when invoking kernel helper functions without requiring binary rewriting. However, eBPF's kernel-space execution presents unique challenges: permission matrix configuration must occur through trusted kernel paths rather than system calls, and helper functions should be assigned `src_id` =0 rather than inheriting restricted permissions from the calling eBPF code.

6.9 Related work

The field of in-process isolation has a rich history, beginning with traditional SFI techniques [196] that relied on binary instrumentation to enforce memory access boundaries through runtime checks. Early implementations required exhaustive coverage of indirect control transfers to maintain isolation guarantees, as any uncovered path could compromise the software-only protections [133]. More recent academic efforts have introduced hardware support for SFI: HFI [143] augments the CPU with new sandbox states controlled through range registers defining permissible memory regions, along with specialized handlers for system calls and sandbox transitions. Similarly, CLPE [192] proposes ISA extensions that cryptographically bind pointers to their allocations by placing message authentication codes in the upper pointer bits and performing implicit checks on memory accesses. While these

approaches enable effective in-process isolation, they necessitate substantial modifications to the instruction set architecture.

Recent commodity hardware extensions have enabled more practical hardware-assisted isolation mechanisms. Intel's Memory Protection Keys (MPK) have been extensively investigated in academic research [113, 152, 194, 153, 169]. Systems like MemSentry [114], Hodor [89] and ERIM [194] have demonstrated MPK's utility for in-process isolation and sandboxing applications. ColorGuard [144] further illustrates how MPK-based isolation can reduce memory overhead in WebAssembly deployments for Function-as-a-Service scenarios. However, because permission keys remain configurable from userspace, MPK alone cannot prevent an isolated domain from reconfiguring its own permissions to escape isolation boundaries following an arbitrary code execution compromise. While MemSentry does not address this security gap, Hodor and ERIM address it through binary analysis to verify the absence of exploitable instruction sequences at component load time, with Hodor relying on hardware watchpoints and ERIM on binary rewriting. However, this approach requires comprehensive binary scanning that may prove prohibitively expensive in certain deployment scenarios. NoJITsu [153] applies MPK specifically to JIT-compiled code, primarily managing write permissions during compilation. To secure permission modifications, Donkey [169] proposes hardware modifications introducing a privileged CPU mode accessible only through interrupt-like mechanisms for MPK configuration.

While Intel MPK has dominated academic research in this domain, recent work has explored alternative hardware primitives. Intel Total Memory Encryption - Multi-Key (TME-MK) [97] encrypts pages based on page table entry bits and provides integrity verification through per-cacheline message authentication codes. TME-Box [193] exploits page aliasing and integrity protection to implement cache-line-granular in-process isolation, though unlike PBI, it still requires program instrumentation to verify correct page alias usage and control-flow integrity to prevent inadvertent domain escapes.

Isolation challenges have also been addressed in alternative contexts with similar requirements to protect the permission configuration. Dorami [116] operates within the RISC-V machine mode, controlling the Physical Memory Protection [103] extension to isolate the secure monitor from other firmware at the same privilege level. Meanwhile, EKC [215] tackles Linux kernel module isolation by controlling page tables and interrupt configurations to separate trusted components from other kernel-level code. Though these systems target different settings, they share with ERIM and other MPK-based approaches the characteristic that isolated components can modify their own isolation configuration, necessitating binary analysis of loaded components to verify that untrusted code cannot subvert security primitives.

ARMlock [227] utilizes Arm v7 memory domain support [12] (no longer available in Arm v8) for in-process isolation. These memory domains are mutually exclusive and linked to first-level page table entries, resulting in 1MB memory granularity with only one domain accessible at any time, contrasting with the 16 permission overlays supported by POE. PANIC [213] takes a different approach, executing untrusted code at Exception Level 1 alongside the kernel while exploiting Privileged Access Never and unprivileged load/store instructions to realize in-process isolation. To separate unprivileged code from kernel space, PANIC must intercept and emulate various instructions with privilege-level-dependent behavior. NanoZone [125], proposed in 2025, employs POE for in-process isolation within Confidential Computing Architecture [13] domains. It relies on program instrumentation combined with control-flow integrity to prevent misuse of domain transition instructions, contrasting with PBI's approach of program-counter-based permission enforcement that eliminates the need for explicit domain switching instructions. LightZone [220] implements in-process isolation on Arm by running each process in kernel mode in its own virtual machine (VM), and using privileged kernel features to enforce isolation within the process.

LMP [93] presents an alternative hardware-assisted approach that leverages Intel's (deprecated) MPX (Memory Protection Extensions) [148] to protect the shadow stack for efficient backwards-edge control-flow integrity. Unlike information hiding techniques that rely on address space randomization to conceal shadow stacks, LMP uses MPX's hardware-accelerated bounds checking to enforce that only authorized instructions can modify shadow stack memory regions. However, LMP requires instrumenting store operations with MPX boundary checks, resulting in an average overhead of 3.9%. Furthermore, LMP focuses on protecting only the shadow stack region, unlike PBI which provides protection for arbitrary domains.

Chapter 7

BLACKOUT

The contents of this chapter are based on the following publication:

Hossam ElAtali et al. "BLACKOUT: Data-Oblivious Computation with Blinded Capabilities". In: *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*. CCS '25. Taipei, Taiwan: Association for Computing Machinery, Oct. 2025. DOI: 10.1145/3719027.3765169

7.1 Introduction

As demonstrated in the preceding chapters, protecting sensitive data against remote adversaries requires addressing both direct access threats and side-channel vulnerabilities simultaneously. On one hand, BliMe provides robust confidentiality guarantees against both direct access and side-channel leakage through hardware-enforced taint tracking and data obliviousness. On the other hand, Program-Counter-Based Isolation (PBI) offers efficient memory protection that ensures data integrity and confidentiality, but lacks protection against side-channel attacks. The challenge lies in providing an efficient unified system that provides comprehensive security guarantees. This represents Research Question 4, the culmination of our systematic investigation into hardware-assisted mechanisms for data protection. Traditional approaches have so far avoided this unification problem, treating memory safety and side channels as separate, orthogonal concerns. Memory safety hardware (such as Capability Hardware Enhanced RISC Instructions (CHERI) [211]) and programming languages (such as Rust) focus on spatial and temporal memory safety violations to prevent run-time attacks that lead to direct access. Constant-time programming attempts to

eliminate side channels by avoiding any secret-dependent behavior in the code. However, while memory safety has been more successfully tackled by the aforementioned approaches (CHERI and Rust), constant-time programming has been less successful.

Writing constant-time code by hand is hard, evident from the many flaws discovered in production side-channel-resistant code [30]. Consequently, industry recommendations urge all but the most specialized developers to refrain from developing constant-time code [98] because even slight mistakes can lead to exploitable side channels. For example, central processing units (CPUs) today do not treat control flow as a secret, thus allowing secret-dependent control-flow to leak sensitive information through the timing of operations or their effects on caches in a complex, hard-to-control, but observable way. Optimizing compilers can transform code intended to be constant-time into *functionally equivalent* (in terms of inputs and output) machine instructions but, for instance, branch on a secret value [168, 77]. Similarly, hardware optimizations like speculative execution that are transparent to software can be exploited to leak secret data through existing side channels [122, 111]. Unlike memory-safety defects, which manifest as crashes or other faulty program behavior once a bug is triggered, flaws in constant-time programming *fail silently* and might be detected only after successful exploitation, if at all.

To address RQ4, we propose `BLACKOUT` (Blinded Architectural Capabilities and Kernel for Oblivious Userspace Tasks), a hardware-software extension to the CHERI architecture that demonstrates how capability-based memory protection can be enhanced with data-oblivious computation guarantees. `BLACKOUT` builds upon CHERI's proven memory safety properties while adding hardware-enforced side-channel protection, creating a unified architecture that efficiently protects both data integrity and confidentiality against remote adversaries.

`BLACKOUT` provides novel *blinded capabilities* to augment CHERI with hardware-enforced taint tracking to guarantee confidentiality of secret data against conventional and speculative side channels. Unlike prior approaches, blinded capabilities avoid the need for additional tag bits in memory (beyond those employed by the baseline CHERI design). Inside the CPU, registers are extended with a *blindedness* bit, which is set when the register is loaded with secret data. Arithmetic logic unit (ALU) operations incorporate taint tracking, propagating blindedness bits from operands to destination registers; therefore, any data derived from secret data is also marked as secret. Instructions operating on such data in registers ensure that secret data is only written to memory through blinded capabilities which have exclusive-access to "blinded" areas of memory. Crucially, any attempt to misuse secret data—such as affecting control flow or as an address in memory operations—results in a fault, effectively preventing side-channel leakage.

We also present a *data-oblivious programming model* for CHERI C which, aided by our

blinded capability-aware Clang/LLVM compiler helps programmers in writing constant-time code for BLACKOUT. Our changes to the LLVM compiler infrastructure are *non-invasive*, consisting of compiler passes that do not interfere with existing compiler optimizations, but instruct BLACKOUT hardware about blinded variables. BLACKOUT allows developers to write constant-time code with minimal additional annotations, benefit from compile-time diagnostics, and turn previously silent constant-time bugs into explicit errors reported through CHERI's exception mechanism.

To demonstrate the practicality of our approach, we realize blinded capabilities on the CHERI-RISC-V architecture on a field-programmable gate array (FPGA) softcore based on the speculative out-of-order CHERI-Toooba processor and integrate blinded capabilities into the CheriBSD operating system (OS) and software stack. We evaluate the area and performance overheads using data-oblivious benchmarks from seminal works in data-oblivious computing for comparability. BLACKOUT offers **the first unified solution to memory safety and side-channel confidentiality with minimal overhead**, addressing key limitations of existing methods. In summary, our contributions are:

- A hardware-software co-design for *blinded capabilities* that extends CHERI with the ability to carry out data-oblivious computation in userspace tasks (Section 7.5).
- BLACKOUT: a realization of blinded capabilities on a CHERI-RISC-V FPGA softcore based on the speculative out-of-order CHERI-Toooba processor IP (Section 7.6.1).
- A programming model and software stack for blinded capabilities and support for BLACKOUT in the CHERI-enabled Clang/LLVM compiler and CheriBSD OS, thus enabling a broad class of applications to benefit from BLACKOUT (Section 7.6.2).
- Evaluation of BLACKOUT using the CoreMark industry-standard performance benchmark and data-oblivious benchmarks from prior work, showing minimal performance impact on data-oblivious code compared to the baseline CHERI-RISC-V processor (1.5%), and moderate impact compared to a processor with neither CHERI's memory-safety nor BLACKOUT enforcement (23.5%) (Section 7.7).

We open-source the artifacts for BLACKOUT's complete hardware and software stack to enable reproducibility of our results and to facilitate further research by the community.

7.2 The CHERI capability architecture

CHERI is an instruction set architecture (ISA) extension that integrates a capability-based hardware-software co-design for memory protection. It extends conventional ISAs with

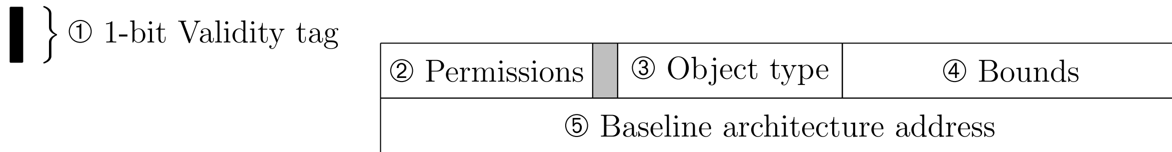


Figure 7.1: In-memory representation of CHERI capabilities adapted from Watson et al. [199]

hardware-supported capabilities to verify memory accesses via code or data pointers. The CHERI ISA specification [200] defines how capabilities are represented in registers and memory and offers capability-aware instructions to manipulate them. Current implementations of CHERI support the RISC-V and Arm8-A instruction sets with CHERI-enabled processors developed by Arm [83], Microsoft [7] and companies and universities in the RISC-V ecosystem.

Figure 7.1 illustrates the in-memory representation of a CHERI capability. Each capability is double the width of the native pointer type: 128 bits on 64-bit platforms and 64 bits on 32-bit platforms. A single-bit *validity tag* ①, stored separately, ensures integrity by invalidating capabilities manipulated by non-capability-aware instructions. Capability-aware instructions preserve tags as long as the operation on a capability is valid but prevent unauthorized manipulation and injection of arbitrary capabilities. These include the store capability via capability (`csc`) and load capability via capability (`clc`) instructions which are used to store, respectively load, capabilities to and from locations in memory identified by a capability operand. In a CHERI-enhanced architecture, address and general-purpose CPU registers are extended to store the full capability representation. For example, in CHERI-RISC-V, all general-purpose registers are 128-bit *capability registers*; the program counter (PC) and stack pointer (SP) are represented by program counter capability (`pcc`) and stack pointer capability (`csp`) registers, respectively. The capabilities themselves include several fields:

- **Permissions** (②) define allowed operations.
- **Object type** (③) enables temporary "sealing", rendering a capability unusable until it is "unsealed" by a special instruction. This enables opaque pointer types and fine-grained in-process isolation.
- **Bounds** (④) specify the accessible memory range relative to the baseline architecture address (⑤). They are stored in a compressed format [211] to reduce memory footprint, but require stricter alignment on larger object allocations.

New capabilities are always derived from an existing capability, with their lineage traceable to initial boot-time capabilities. CHERI enforces *monotonicity* ensuring newly created capabilities cannot exceed the permissions or bounds of their parent. The `candperm` instruction allows the permissions of a capability to be dropped according to a given mask, but not gained. Controlled exceptions, such as sealed capabilities for exception handling and compartmentalization, allow limited non-monotonicity.

Spatial- and temporal-safety enforcement. The bounds, stored with the virtual address, underpin CHERI's spatial memory safety. Each memory allocation is associated with a capability describing its valid address range and access permissions. This enables inherent spatial memory-safety properties. To provide temporary safety for heap-based allocations, CHERI requires a capability revocation mechanism, such as Cornucopia [63]. Cornucopia scans for capabilities pointing to freed memory, allowing such stale capabilities to be revoked. Extensions to the CHERI software and hardware have also explored sandboxing [41], and initialization safety [76, 88].

CHERI-enabled software stack. CheriBSD [50] is a modified version of the open-source FreeBSD operating system, designed to support CHERI-RISC-V both in emulation and on hardware. It is a fully functional OS prototype, demonstrating how CHERI support can be integrated into a conventional OS design. The CheriBSD kernel and userspace can be built in "*pure-capability*" CHERI C/C++ which means all conventional memory pointers are replaced by corresponding capabilities by the CHERI-LLVM compiler. In pure-capability mode, compiler-generated code derives bounded capabilities for automatic (stack-allocated) variables from the `csp`. For global variables, the run-time linker populates a capability-aware global offset table, also referred to as the *captable*. Capabilities for heap-allocated data is handled by Cornucopia which is integrated into CheriBSD via a *shim layer*, known as the `malloc` revocation shim (`malloc` revocation shim), which sits on top of the underlying BSD libcs memory allocator. Consequently, CheriBSD enforces both spatial and temporal memory safety for CHERI-enabled software.

7.3 System and Adversary Model

We assume the same system model as in CHERI. We trust the OS to prevent memory containing sensitive information from being exposed to other processes after a process exits, either normally or as a result of CHERI exceptions. We assume memory-safety properties

provided by CHERI, which prevent adversaries from tampering with the program's control flow or directly inferring memory contents beyond that which are exposed during normal program operation. We also assume a data-independent timing (DIT) [128] mode is available.

We assume the same adversary model as in CHERI. In addition, we assume the adversary has the ability to observe side channels during a program's execution, possibly from another process running on the system simultaneously. In this work, we consider side-channel threats that can be mitigated by data-oblivious software, including timing side channels, such as cache timing [149, 217, 85, 40], instruction timing [157] (with DIT) and port-contention timing [6]. We also consider transient execution attacks that leak information *loaded* by mis-speculated instructions within the same address space, such as Spectre [111, 129, 115, 91], in scope, even if mis-training can be done across address spaces [34]. Attacks with transient instructions that can load data *across* address spaces, such as Meltdown and microarchitectural data sampling (including load-store-buffers) [35], and transient capability forgery, such as Meltdown-CF [68], are out of scope, but can be prevented by using Capability Speculation Contracts (CSCs) [68]. Attacks that require physical access to the system, e.g., to measure power consumption, or those through direct memory access (DMA) peripherals are out of scope.

7.4 Goals and Challenges

In this section, we outline our overarching goals and the primary challenges associated with them.

7.4.1 Goals and Primary Challenges

Adapting hardware-enforced taint tracking to capability-based architectures. Our primary goal is integrating data-oblivious computation with the CHERI protection model. While it may seem straightforward to implement a data-oblivious ISA on top of an CHERI-enabled architecture, there are two key challenges to overcome. First, the CHERI architecture is intended to be applied onto conventional ISAs such as RISC-V and Arm. Bolting an existing data-oblivious ISA (Section 9.2) on top of CHERI will require invasive changes to CHERI and/or the underlying architecture making real-world adoption less realistic. Second, significant changes to the architecture will negatively impact performance.

To address these challenges, we propose enhancing the existing CHERI capability model with *blinded capabilities* which ensure data accessed through them is operated on exclusively in a data-oblivious manner. Memory managed by blinded capabilities is referred to as *blinded memory*. This avoids the need to propagate blindedness tags to memory thus overcoming the drawbacks of simply integrating an existing data-oblivious ISA with CHERI [59, 218].

Practical programming model for blinded capabilities. Our second goal is introducing a practical programming model for blinded capabilities. CHERI-enabled languages (CHERI C/C++ [201]) enforce memory-safety properties (Section 7.2) at run-time whereas data-oblivious ISAs enforce oblivious access to secret data (Section 7.5). Thus, our model must adhere to both CHERI and data-oblivious properties.

A particular challenge arises because the CHERI compiler occasionally allows memory accesses without explicit capabilities (e.g., certain stack accesses). To ensure such accesses do not target blinded memory (which would raise a hardware fault), our blinded capability-enhanced compiler must explicitly enforce that all accesses to blinded stack variables occurs through blinded capabilities. We describe this compiler enhancement further in Section 7.5 and Section 7.6.2.

7.4.2 Additional Challenges

Integrating blinded memory management with the CHERI protection model introduces several additional challenges.

Maintaining capability monotonicity. Introducing blinded capabilities adds new privileges that must align with CHERI's existing monotonicity properties. Our design considers "non-blinded" a distinct permission, which, when removed, permanently classifies a capability as a blinded capability. This upholds monotonicity—blinded capabilities cannot be promoted to regular, non-blinded capabilities.

Exclusive access to blinded memory. As blinded capabilities are based on CHERI capabilities, they inherently face the *revocation problem*. Deallocated memory regions may still be accessible through residual capabilities, risking use-after-free vulnerabilities and inadvertent disclosure of blinded data. Our design addresses this by ensuring new blinded capabilities obtain exclusive access to newly allocated blinded memory inside the process. Thus, blinded memory cannot be subject to use-after-free conditions from residual, non-blinded capabilities to it.

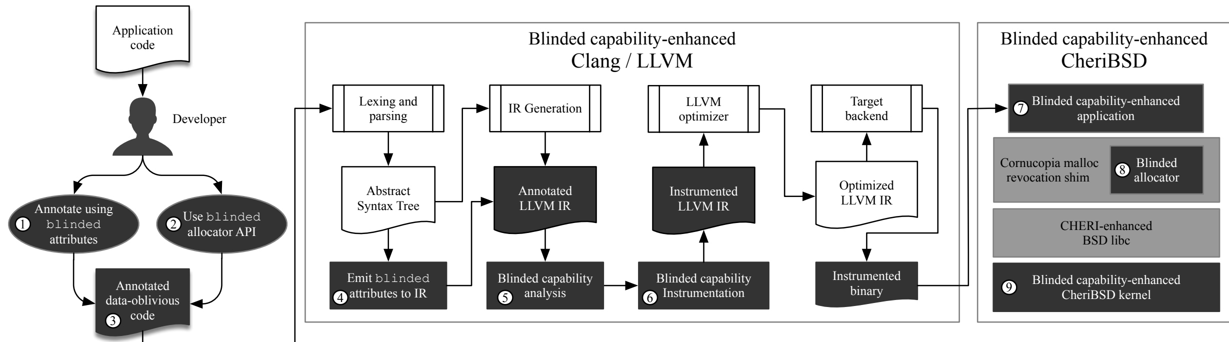


Figure 7.2: High-level overview of the components of the blinded capability software stack. Dark gray denotes additions or modifications needed to support blinded capabilities, while light grey indicates components inherited from the CHERI architecture. White denotes components without significant changes.

Securely reclaiming blinded memory. Blinded memory must be securely reclaimed, as residual sensitive data may remain upon deallocation. Since CHERI does not inherently guarantee memory initialization safety, we need to ensure secure deallocation through automatic memory clearing with blinded capability-enhanced compiler and a blinded capability-enhanced allocator (Section 7.6.2).

7.5 Blinded Capability Design

Blinded capabilities introduce a new programming model that developers must follow to protect their sensitive data. While providing a completely unrestricted programming model can seem appealing, it 1) cannot guarantee exclusive access to blinded data (Section 7.4.2) and 2) does not guide the developer towards avoiding hardware faults caused by security violations. The end result is that the developer must manually analyze and transform their code to prevent it from faulting. The goal of our programming model is therefore to provide strong security guarantees for memory safety and side-channel protection by guiding the developer using compiler warnings/errors, and, where possible, automatically applying code transformations.

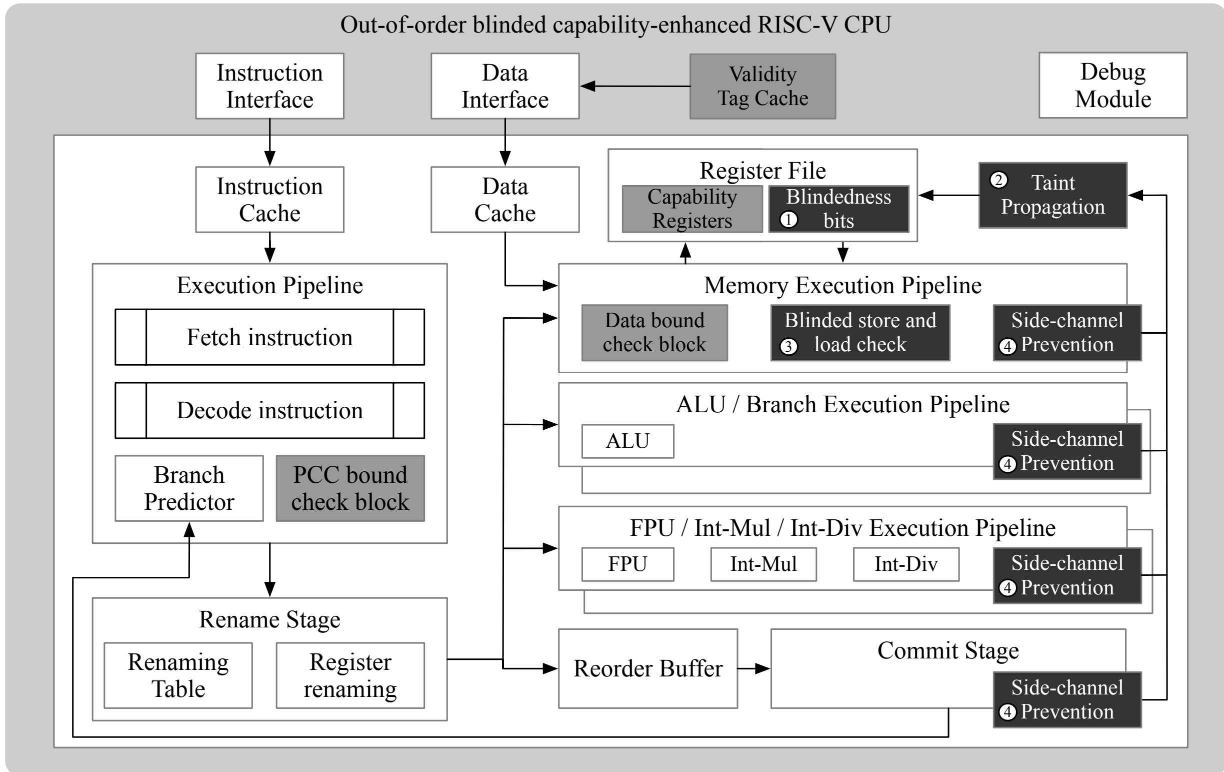


Figure 7.3: High-level overview of the blinded capability-enhanced CHERI-RISC-V CPU hardware components. Dark grey denotes additions or modifications needed to support taint tracking, while light grey indicates additions common to CHERI-enabled CPUs. White denotes components without significant changes.

7.5.1 Software Architecture

Figure 7.2 shows a high-level overview of the blinded capability software stack. Developers can indicate to the compiler that a variable is sensitive by annotating its declaration with the `blinded` attribute ①. We refer to such variables as *blinded variables*. The compiler will then ensure that any accesses to blinded variables are permitted only through blinded capabilities, guaranteeing that the values of these variables will always be blinded when loaded into registers. We refer to registers holding blinded data as *blinded registers*. Blinded memory can be dynamically allocated using the `blinded` allocator application programming interface (API) ② which returns a blinded capability. Similarly, the compiler enforces that references to blinded memory cannot be assigned to non-blinded variables. Inside the compiler, we modify the Clang front-end to emit `blinded` attributes to the LLVM intermediate

```
#define __blinded [[clang::annotate_type("blinded")]]

❶ __attribute__ ((blinded))
int data_oblivious_select(bool cond, int x, int y) {

❷ { bool __blinded c; // c declared blinded and
                      // accessed via blinded capability

  int res;          // res not declared blinded
                    // but blindedness is inferred
  ❸ Compiler infers res blinded from ❺

  c = cond;   // Uses store via blinded capability
              // (argument already in register)
  ❹ Compiler knows c is already blinded based on declaration

  res = (x * c) + (y * (!c)); // HW propagates
                              // blindedness to res
  ❺ Compiler infers res is blinded from this assignment

    return res;
}
```

Listing 7.1: Data oblivious conditional select function which returns one of the arguments `x` or `y` based on the value of `cond` demonstrating the inference capabilities of the blinded capability-enhanced compiler.

representation (IR) of the compiled program ④. These are used by the blinded capability analysis ⑤ and instrumentation passes ⑥ to produce a blinded capability-instrumented application binary. Global variables annotated with `blinded` attribute are reserved from a separate `blinded data` section by the static linker.

At run-time, the blinded capability-enhanced application ⑦ requires platform support in the form of the blinded allocator ⑧ and an OS with a CHERI-capable kernel that has been altered to ensure its internal handling of capabilities does not trigger blinded capability violations. C startup code added by the compiler to ⑦ is responsible for initializing capabilities pointing to data in the blinded data section as blinded capabilities. System software not making use of blinded capabilities does not require modification (apart from their adaptation to CHERI). We discuss the compiler enhancements and operating system changes in detail in Section 7.6.2.

7.5.2 Hardware architecture

Figure 7.3 shows the high-level overview of the hardware changes needed to support blinded capabilities (shown in dark gray) applied to an out-of-order RISC-V core, such as MIT's RISCY-OO [221] or Bluespec's Toooba [145] processor intellectual property (IP). Apart from the changes introduced by CHERI (shown in light gray) the majority of the changes

```
1  void bad_func(bool cond, int x, int *out) {
2
3  ① { int __blinded a = x;   // a declared blinded
4      int b;
5
6
7     if(cond)
8         b = a;   // HW propagates blindedness to b
9     ② Compiler undecided on whether b becomes blinded
10
11    *out = a;    // HW fault if out non-blinded (I1)
12    ③ Allowed by compiler as out might be blinded
13
14    if(a != 0)   // Violates I4
15        b = a;
16    else
17        return
18    ④ Rejected by compiler as blinded a used in control-flow decision
19
20    if (b != 0)     // HW fault if b blinded (I4)
21        *out = a;   // HW fault if out non-blinded (I1)
22 }  ⑤ Control-flow allowed by compiler as b might be non-blinded
```

Listing 7.2: Non-data-oblivious function which attempts to branch on a blinded condition and write blinded data to a possibly non-blinded output parameter. This code is rejected by the blinded capability-enhanced compiler.

necessary in the CPU are to facilitate in-CPU taint tracking. In the register file, the general-purpose capability registers are extended to hold an additional *blindedness bit* ① which tracks whether the register is blinded. The taint-propagation hardware logic ② will in turn ensure that any data derived from these blinded registers, e.g., through arithmetic operations, will also "be blinded", i.e., it sets the blindedness bit of the destination register.

All blinded data is bound by a set of invariants (I1 to I5), stemming from the need to enforce exclusive access to blinded memory (Section 7.4.2), and side-channel prevention.

Exclusive access invariants:

I1 Blinded data cannot be stored into memory using non-blinded capabilities.

I2 No capability of any kind can be stored into blinded memory.

I3 Bounds of valid blinded and non-blinded capabilities must not simultaneously overlap. However, once a region is released (i.e., is no longer accessible through a valid capability, e.g., due to heap deallocations or stack frame pops), it can be assigned to future blinded or non-blinded capabilities.

Side-channel protection invariants:

Table 7.1: Blindedness bit propagation and side-channel prevention rules enforced by `BLACKOUT` hardware. x represents a "don't care" condition, which indicates the actual signal value or values have no impact on the decision outcome.

Instruction	**Blinded capability**	**Blindedness**		**Decision**	**Result blindedness**
Arithmetic / Logic	-	**Op1**	**Op2**		
		a	b	Propagate	$a \vee b$
Branching	**in Addr Reg**	**Addr Reg**	**Condition Ops**		
	no	0	0	Allow	-
	no	1	x	Fault	-
	no	x	1	Fault	-
	yes	x	x	Fault	-
Load	**in Addr Reg**	**Addr Reg**			
	no	0		Propagate	0
	yes	0		Propagate	1
	x	1		Fault	-
Store	**in Addr Reg**	**Addr Reg**	**Data Reg**		
	no	0	0	Allow	-
	no	0	1	Fault	-
	yes	0	1	Allow	-
	yes	0	0	Allow	-
	x	1	x	Fault	-

I4 Control-flow instructions cannot use blinded operands as conditions or target addresses.

I5 Load and store instructions cannot use blinded operands as addresses.

I1 is enforced by hardware through additional *blinded store and load checks* ③ in the memory execution pipeline which is responsible for the execution of any load and store instructions, whereas I4 and I5 are enforced by *side-channel prevention* logic integrated into any pipeline which is responsible for instructions with blinded operands ④. Any attempt by software to violate I1, I4 and I5 will result in a hardware fault. Table 7.1 shows the full blindedness bit propagation and side-channel prevention rules enforced by `BLACKOUT`.

I3 cannot be efficiently enforced by hardware alone. As mentioned in Section 7.4.2, blinded memory reclaimed when blinded capabilities are destroyed must be erased to avoid leaking blinded data; this is handled by the compiler and memory allocator for stack and heap variables, respectively (Figure 7.2, Section 7.6.2). The combination of hardware, compiler and software stack guarantees data confidentiality.

Data-oblivious computation ensures that control flow and memory accesses don't depend on secret data. Outputs, however, may be non-secret (e.g., decrypted data meant for users). In practice, such output data cannot be left blinded indefinitely as some result of data-oblivious computation must eventually be extracted. In `BLACKOUT`, results can be marked non-secret by issuing non-blinded capabilities for result buffers. This reveals the final, non-secret result of blinded computation by relaxing invariant I3 in a controlled way to unblind the result. For example, since I3 is (in part) enforced by the blinded allocator, the allocator can expose a separate API for obtaining blinded results that, in contrast to, regular blinded memory returns two capabilities for a dedicated area for blinded results: one which is blinded, and used to write to the area, and the other non-blinded, but only usable for reading the result. However, the developer must use the blinded capability to the result area carefully to prevent accidental leakage of secret data. We discuss additional options to secure extraction further in Section 7.8.

7.5.3 Motivating Examples

The concrete examples in Listings 7.1 and 7.2 demonstrate how the hardware and the compiler work together to prevent blinded memory leakage. The hardware ensures that taint tracking is both accurate and precise (no under- or over-tainting occurs in the hardware), and that attempted leaks cause a fault. The compiler ensures that capabilities respect exclusive blinded memory access, and that reclaimed blinded stack memory is zeroed out.

The compile-time analysis is dependent on the developer providing *(*initial) information about which variables are expected to contain sensitive information through `blinded` attributes (`__blinded`, ❷ in Listing 7.1). Optionally, the developer can also declare functions blinded (`__attribute__((blinded))`, ❶ in Listing 7.1). This clearly identifies functions expected to return blinded data, improving developer ergonomics. The compile-time analysis is not limited to the information provided by the developer. For example, in Listing 7.1 ❸, the compiler can infer the variable `res` must be blinded due to the assignment from `c` tainting it at ❺. However, annotating functions as blinded improves the readability of data-oblivious code.

Listing 7.2 shows a non-data-oblivious function which would fault at run-time due to violating I4 (and possibly I1). This example demonstrates that the compiler can reject some non-data-oblivious programs outright. However, as discussed in Section 2.3.3, compile-time data-obliviousness analysis cannot be sound as data-obliviousness is ultimately a property at machine-code level. For example, at ②, the compiler cannot know whether `cond==1`. Thus, it *cannot prove* that `b` is always blinded and must allow the control flow at ⑤. However,

if `cond==1` at run time, the hardware will propagate the blindedness bit to `b` at ②; the subsequent branch at ⑤ will fault.

For ③, the compiler does not know whether `out` is a blinded capability and must also allow compilation. If `out` at run time is referenced by an unblinded capability, the hardware will detect the violating store and raise a fault.

The hardware would fault at ④ due to violation of I4. However, since it is detectable by the compiler, we force a compilation error to inform the developer early in the development cycle. This is especially useful for code paths that are rarely exercised with real workloads and require fuzzing techniques to uncover. More annotations allow the compiler to make more informed decisions and detect violating operations that it cannot infer on its own.

7.6 BLACKOUT Blinded Capability Implementation

We now describe Blinded Architectural Capabilities and Kernel for Oblivious Userspace Tasks (`BLACKOUT`), our realization of blinded capabilities for CHERI-RISCV, CHERI-LLVM, and the CheriBSD OS.

7.6.1 BLACKOUT CHERI-RISC-V CPU

We implement blinded capabilities in register-transfer level (RTL) on CHERI-Toooba [68], a CHERI-enabled RISC-V processor based on Bluespec's speculative out-of-order Toooba IP [145]. Unlike previous data-oblivious ISAs [218, 59], `BLACKOUT` *does not add any additional instructions to the underlying ISA*; all architectural changes are achieved by adjusting the CHERI permission model.

Non-oblivious access permission. We introduce a new "non-oblivious access" permission bit in the previously unused section of the CHERI capability representation. The CHERI-Toooba IP allocates 12 bits for hardware permissions and 4 bits for user permissions, leaving 3 bits unused. By default, the blinded capability bit in newly created capabilities is set to 1. In this initial configuration, the enforcement of blinded capabilities is turned off, allowing the capability to be used freely within its defined bounds. When the existing CHERI `candperm` instruction (Section 7.2) is invoked with the blinded capability permission as an operand, it changes the non-oblivious access bit to 0, thereby enabling the enforcement of blinded capabilities. Defining the non-oblivious access bit this way means that a capability with this permission is *allowed* to handle data in a non-oblivious manner.

Registers & taint-tracking. CPU registers are extended with an additional blindedness bit to signify whether the data stored in the register is blinded. Any load instructions executed with a blinded capability set the blindedness bit of the target register to 1. Store instructions with a blinded capability only store the data in memory, but not the blindedness bit. Data confidentiality is maintained by the exclusive access invariants (I1 to I3, Section 7.5), which ensure that blinded data stored in memory is only accessible through blinded capabilities. Register-to-register instructions, such as arithmetic and logical operations, propagate the blindedness bit: if the output depends on a blinded input, the output becomes blinded. Overwriting a register containing blinded data with non-blinded data will result in that register's blindedness bit to be unset.

Limiting impact on capability-modifying instructions. We prohibit valid capabilities from being blinded (I2, Section 7.5). The execution of any capability-modifying instruction with operands that would result in a blinded capability causes a fault. Enforcing this invariant allows us to avoid unnecessarily making capability-modifying instructions data-oblivious. Note that this enforcement does not break the CHERI programming model; "*blinded blinded capabilities*" are themselves redundant as they can never be dereferenced to access memory in order to maintain confidentiality.

Spilling registers containing blinded data. Blinded data can reside in any general-purpose capability register, including those designated as caller- or callee-saved by the RISC-V application binary interface (ABI). The CHERI-RISC-V compiler can generate code that spills blinded registers using `csc` instructions, and restore them with `clc` instructions. Normally, spilling blinded registers onto the stack via the non-blinded `csp` violates invariant I1 (Section 7.5) as spills target non-blinded memory. However, since stack memory allocated for registers spills is inaccessible by other capabilities than the `csp`, CHERI's memory-safety guarantees ensure that spilled register cannot be accessed improperly. Thus, we permit `BLACKOUT`'s `csc` and `clc` instructions to spill and restore blinded registers `csp`.

However, another challenge arises: the non-blinded `csp` generates non-blinded data when storing spilled values. This causes ambiguity when restoring spilled registers, as the CPU cannot distinguish between blinded and non-blinded register spills. To address this, `BLACKOUT`'s `csc` emits special *blinded register records (BRRs)* as result spilling blinded registers. A BRR is a 128-bit structure containing the spilled value and 64-bit marker. This marker differentiates BRRs from conventional CHERI capabilities and maintains their validity tags. When spilled capability register is restored, the validity tag signals that the value restored from memory is either a capability or a BRR. The marker indicates whether

the value is a BRR, ensuring the destination register is correctly blinded.

Transient execution. As mentioned in Section 2.3, hardware optimizations, such as speculation, can inadvertently introduce side channels when operating on secrets. To protect blinded data against such leakage, we ensure that any blinded data flowing to *decision-making* parts of the hardware (such as branch predictors) are zeroed out. This guarantees that execution is truly oblivious to blinded data, both architecturally and transiently. This is similar to approaches used in prior work [218, 59].

7.6.2 BLACKOUT Software Stack

We leverage Clang's existing `annotate_type` attribute designed for static analysis tools [31] for the `__blinded` attribute. Normally, `annotate_type` is not propagated to the LLVM IR. Thus, we extend Clang to emit `__blinded` attributes into the IR (④, Figure 7.2).

We introduce an additional analysis pass for blinded variables (⑤, Figure 7.2) in the LLVM-backend. This pass performs recursive dependency analysis to trace all uses of `__blinded` variables in data flows involving the store, load, and GetElementPtr (GEP) IR instructions. By recursively analyzing flows from blinded source instructions, e.g., loads from variables declared as `__blinded`, the pass automatically propagates the blindedness property to variables acting as sinks, blinding them at their respective point of allocation.

We additionally introduce an instrumentation pass (⑥, Figure 7.2) that transforms stack allocations (alloca IR instructions) for variables annotated with `__blinded`. The instrumentation adds the `llvm.cheri.cap.perms.and` intrinsic to each annotated alloca to unset the "non-blinded" permission control bit, thus turning the associated capabilitity into a blinded capability. Consequently, all stack memory associated with `__blinded` variables are only accessible through blinded capabilities.

Blinded capability-enhanced allocator. We extended the Cornucopia revocation mechanism to provide a `blinded_malloc` API that returns blinded capabilities to blinded heap allocations. To meet I3 (Section 7.5), such blinded heap allocations must not overlap with other, non-blinded allocation. This ensures the blinded capability returned `blinded_malloc` has exclusive access to the newly created allocation. The `blinded_malloc` API relies on Cornucopia to ensure the blinded capability does not overlap with any concurrently existing capabilities. `BLACKOUT`'s `blinded_malloc` is integrated into CheriBSD via the `malloc` revocation shim (similar to the integration of Cornucopia, discussed in Section 7.2).

Table 7.2: Area and power costs on VCU118 @ 25MHz expressed in number of LUTs and registers, and Watts respectively.

	logic	Δ(%)	memory	Δ(%)	registers	Δ(%)	power	Δ(%)
CHERI-Toooba Core	697508	–	20852	–	412493	–	6.205	–
Blinded CHERI-Toooba Core	705863	1.2	20855	0.0	412913	0.1	6.536	5.3

This makes blinded capabilities allocator-agnostic, allowing userspace processes to employ different underlying allocators as long as allocations occur via the shim.

Securely reclaiming blinded memory. Recall from Section 7.4.2 that blinded memory must be securely reclaimed to prevent information leaking from previously blinded memory regions. `BLACKOUT` implements explicit zeroing policies for blinded heap and stack allocations. For blinded heap allocation, the `free` API, implemented in the shim, inspects whether memory to be deallocated is blinded and erases the contents of blinded allocations before freeing it. For stack-allocated blinded variables, the compiler automatically inserts a memset IR instruction to zero out their memory immediately after the variable's lifetime ends. This ensures sensitive data from blinded variables is securely erased, preventing unintended reuse or leakage when stack frames are reallocated. Blinded global data persists until the process terminates and consequently does not need to be reclaimed. The contents of (physical) pages are zeroed out by the OS before they are recycled for other processes.

Integration to CheriBSD. As discussed in Section 7.6.1, any newly created capability must initially be configured with its "non-blinded" permission bit set. To support blinded capabilities in CheriBSD, we perform a thorough scan of all CheriBSD code to ensure that any derived permissions (e.g., for drivers or userspace) also unset the "non-blinded" permissions. We also extend the CheriBSD CPU exception handler to handle Blinded capability exceptions, and adapted the signal code with the newly introduced from the Toooba Core. We integrated blinded capabilities into CheriBSD 24.05, resulting in a total of 100 lines of code changes over 14 distinct files.

7.7 Evaluation

We evaluate the overheads (area and performance) and security of `BLACKOUT`. To evaluate overheads, we extended the CHERI-RISC-V Toooba FPGA softcore (RV64ACDFIMSUxCHERI),

Table 7.3: Performance cost on VCU118 @ 25MHz expressed as CoreMark test results for $5x10^3$ iterations. The CoreMark score for a processor is reported as CoreMark-iterations-per-second-per-core-MHz. The Δ is relative to CHERI-Toooba Core results.

	Total ticks	Δ		Total time (sec)	Δ		Iterations/sec	Δ		Score
CHERI-Toooba										
baseline (nocap)	927674227	–		37	–		135	–		5.4
purecap	951983228	24309000	2.62%	38	1	3%	131	4	2.3%	5.24
Blinded CHERI-Toooba										
baseline (nocap)	927729879	55652	0.01%	37	0	0%	135	0	0%	5.4
purecap	952083879	24409652	2.63%	38	1	3%	131	4	2.3%	5.24

which is based on the open-source Bluespec RISC-V 64-bit Toooba core.

We use the BESSPIN-GFE security evaluation platform [156], which has out-of-the-box support for the CHERI-Toooba softcore, replacing the standard core with our Blinded CHERI-Toooba. We synthesize the system-on-chip (SoC) at 25MHz (default for BESSPIN-GFE) targeting the Xilinx Virtex UltraScale+ VCU118 FPGA.

7.7.1 Power & resource usage

Table 7.2 shows the power and resource usage obtained from Xilinx Vivado 2019.1. The overheads are minimal ($\approx 1\%$ area and $\approx 5\%$ power) compared to the unmodified CHERI-Toooba. This is expected since our hardware additions require no additional storage in memory or caches and only a single additional bit for registers.

7.7.2 Performance

For performance measurements, we run several benchmarks on the VCU118 FPGA. First, we evaluate the impact of `BLACKOUT` CHERI-Toooba on unblinded workloads by running the EEMBC CoreMark benchmark [70] bare-metal on the GFE for 5×10^3 iterations over three separate test runs. Table 7.3 shows the mean result demonstrating that there is only a negligible effect on performance for unblinded workloads between the unmodified CHERI-Toooba core and our `BLACKOUT` CHERI-Toooba variant. We also obtain timing reports from Xilinx Vivado to show the effect of our hardware changes on the maximum clock frequency attainable. Both `BLACKOUT` and the baseline are synthesizable at 25MHz, and the worst negative slack (WNS) for `BLACKOUT` is 0.08ns compared to 0.06ns for the baseline, demonstrating no significant effect on clock frequency.

OISA benchmarks. Second, for blinded workloads, we evaluate the performance impact of BLACKOUT using five OISA benchmarks adapted from Yu et al. [218]. To adapt them to BLACKOUT, we simply blind secret inputs with the `__blinded` attribute, change dynamic allocations to use our blinded allocator API, and compile the benchmarks using our blinded capability-enhanced compiler. Porting the OISA benchmarks to BLACKOUT took 1-5 LoC changes (¡1%) per benchmark. We measured the performance on CheriBSD with BLACKOUT support in pure-capability mode with the Cornucopia revocation mechanism enabled. Our setup follows the CheriBSD benchmark guide [52], with the exception of running the BLACKOUT benchmarks in pure-capability mode (Section 7.2) with blinded capabilities, and enabling the experimental Cornucopia revocation mechanism, which is essential to guarantee exclusive access for blinded capabilities. We compiled all benchmarks with `-O3` and measured the combined user and system time using the *time* command. This measures the entire lifetime of blinded capabilities—including capability creation, algorithm execution, and memory reclamation (i.e., zeroing out blinded memory). For the `findmax`, `binary_search`, and `integer_sort` benchmarks, we used blinded input arrays containing 2^N integers. For the matrix multiplication benchmark, we multiplied two square matrices of size $2^{\frac{N}{2}} \times 2^{\frac{N}{2}}$. In the DNN example, the blinded input size was $2^{\frac{N}{2}}$, and the network consisted of two layers, each of size $2^{\frac{N}{2}} \times 2^{\frac{N}{2}}$, with a fixed output size of 2^6. We evaluated all benchmarks with $N \in \{12, 14, 16, 18, 20\}$.

The results in Figure 7.4 show the OISA benchmark run-time in nanoseconds for different input sizes and three configurations: *baseline*, with capability-enforcement disabled, *purecap*, which enforces CHERI memory-safety only, and *purecap + blinded*, which enforces all BLACKOUT invariants including CHERI memory-safety and data-obliviousness. As our hardware modifications do not add additional cycles to any instructions, all configurations can run with the same FPGA bitstream, requiring only different compilation flags. The baseline configuration uses CHERI's "hybrid" mode which allows CHERI-capable processors to run legacy, non-capability code.

The geometric mean overhead per benchmark is shown in Figure 7.4f. The overall result shows a minimal geometric mean overhead of 1.5% for BLACKOUT compared to the *purecap* configuration, which is several times lower than overheads for prior work enforcing data-oblivious computation [218, 59]. The overhead in blinded workloads is caused by several factors:

1. Clearing blinded memory at the end of a blinded capability's lifecycle. This is done by the compiler for blinded memory on the stack on function returns, and by the `blinded_malloc` for blinded memory on the heap on freeing. For instance, when N = 20, binary search operates on a 4MiB array. In this setting, memory reclamation (zeroing out) accounts for most of the overhead.

2. Additional stores and loads to set the blindedness bit in registers. When initializing blinded data at the start of the benchmarks, any data in registers must be first stored into blinded memory and then loaded back to set the register blindedness bit. This only occurs for the initial blinded data and is not required to propagate blindedness during execution. The effect on performance is therefore minimal. Nevertheless, we discuss a potential optimization to avoid this in Section 7.8.

3. Changes to instruction cache performance caused by the slight increase in code size due to compiler instrumentation.

We also evaluate the *combined* performance impact of CHERI's memory-safety and `BLACKOUT`'s data-oblivious enforcement. We measure a moderate geometric mean overhead of 23.5% compared to the unprotected baseline configuration with capability-enforcement completely disabled. We observe that the overhead for both CHERI's purecap and `BLACKOUT`'s purecap + blinded modes relative to the unprotected baseline is significantly larger in the `binary_search` benchmark, particularly for smaller input sizes, compared to the other OISA benchmarks. We investigate further the composition of the overhead in that benchmark. In consultation with the University of Cambridge Computer Laboratory Security Group we identified three possible reasons for the poor performance:

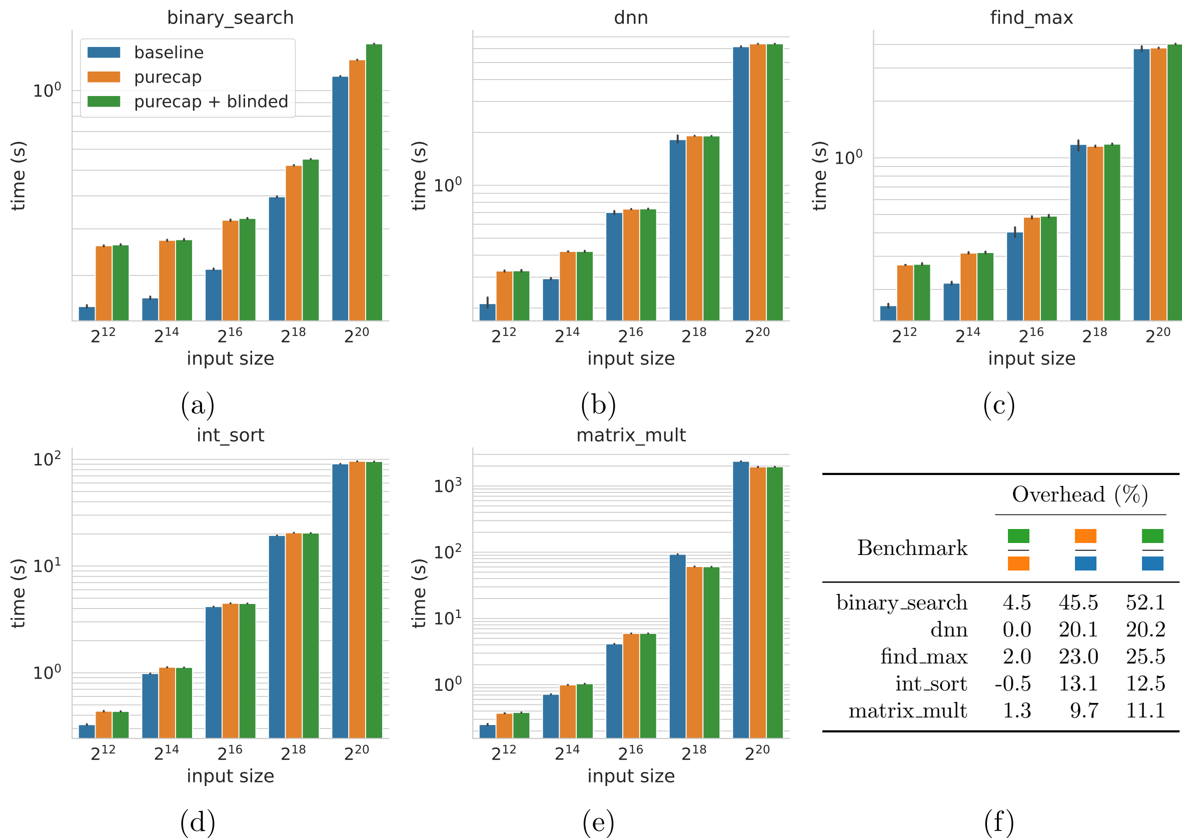
1. Inefficiencies in CHERI-RISC-V's relocation representation and processing converting function pointers into capabilities.

2. Lack of support for lazy binding of functions in CHERI-RISC-V which manifests as high initial load times at startup.

3. The current version of CHERI-LLVM currently disables most loop optimizations, making it slower for the type of code the `binary_search` benchmark relies on compared to compilation to a non-CHERI RISC-V target.

To isolate the effect of these potential root causes for the overhead that are independent of the `BLACKOUT`-related changes, we repeat the `binary_search` benchmark in two additional configurations: a) a statically linked benchmark on CheriBSD designed to mitigate the inefficiencies in relocation (1) and lazy binding (2), and b) a bare-metal version of the benchmark that runs without CheriBSD. Crucially, the bare-metal benchmark measurements include only the execution of the data-oblivious algorithm, eliminating the initial startup cost, reclaiming of memory, and system calls, thus isolating the computational cost of blinded capabilities without their impact on memory management.

Figure 7.5 shows the results of these additional `binary_search` configurations and allows us to make the following observations: First, in the statically-linked benchmarks,

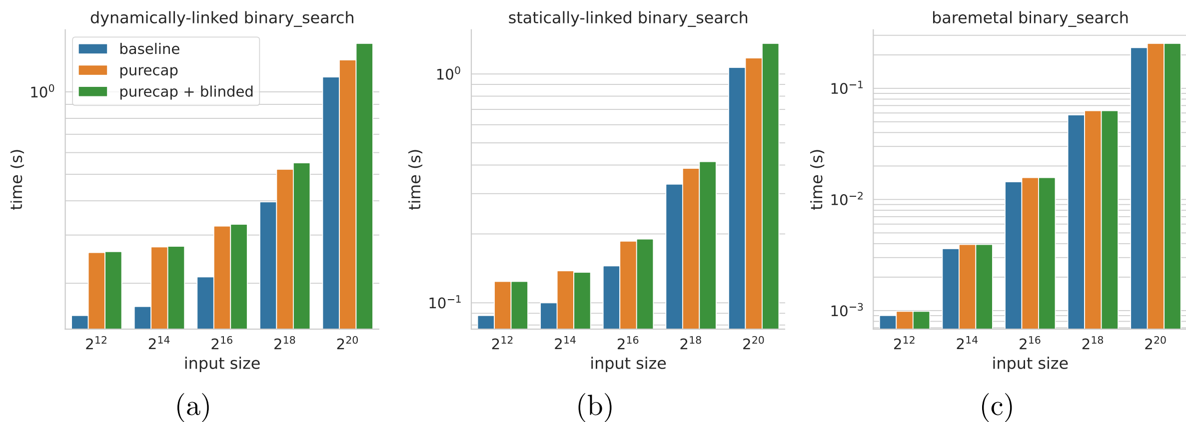


Benchmark	Overhead (%)		
binary_search	4.5	45.5	52.1
dnn	0.0	20.1	20.2
find_max	2.0	23.0	25.5
int_sort	-0.5	13.1	12.5
matrix_mult	1.3	9.7	11.1

Figure 7.4: Run-time in seconds of the OISA `binary_search` (a), `dnn` (b),`find_max` (c), `int_sort` (d), and `matrix_mult` (e) benchmarks built for the *baseline* with capability-enforcement disabled, *purecap* mode with CHERI memory-safety enforced but data-obliviousness not enforced, and *purecap+blinded* mode with both CHERI memory-safety and data-obliviousness enforced. Figure 7.4f shows the geometric mean overheads for each benchmark aggregated over all input sizes.

the overhead for smaller input sizes improve, but the result for larger input sizes is similar to the dynamically-linked benchmarks, suggesting that the one-off cost of initial load times is amortized over the longer benchmark runs in both cases. Second, the overheads for both purecap and purecap + blinded modes compared to the baseline are significantly lower when run on baremetal. This supports our hypothesis that relocation (1) and lack of lazy binding (2) are the main sources of overhead for `binary_search` in Figure 7.4.

We also note some cases in the `matrix_mult` benchmark in Figure 7.4e where the baseline performs slightly worse than the other two. We examined the generated assembly



(a) (b) (c)

Figure 7.5: Run-time in seconds of the OISA `binary_search` benchmark in dynamically-linked, statically-linked, and bare-metal configurations for the *baseline*, *purecap*, and *purecap+blinded* modes. The geometric mean overhead of *purecap+blinded* compared to *baseline* aggregated over all input sizes falls from 52.1% for the dynamically-linked configuration (a) to 32.0% for the statically-linked one (b). The corresponding geometric mean is 9.1% for the baremetal configuration (c).

for these benchmarks and discovered that the CHERI-enabled compiler does a better job at optimizing parts of the code whose effect on performance grows with input size, leading to slightly lower run-times in the purecap and purecap + blinded configurations.

SpectreGuard benchmarks. Third, we run the SpectreGuard [69] synthetic benchmark using the version adapted by Daniel et al. [49] to evaluate ProSpeCT (see Section 9.3). This version uses data-oblivious implementations of the `chacha20`, `sha2`, and `curve25519` cryptographic algorithms from HACL*[228], a formally verified cryptographic library written in F* [183]. The benchmarks represent a workload consisting of public-data computations, which gain substantial performance from speculative execution, alongside encryption routines, which are less impacted by speculation. Each benchmark varies the proportion of speculative versus cryptographic work (S/C), as shown in Table 7.4. The ProSpect benchmark version already annotates all secret values in those test cases. We adapt those annotations to `blinded` attributes and run all the test cases on `BLACKOUT` CHERI-Toooba in both *purecap* and *purecap + blinded* versions. Table 7.4 shows the results (average of 20 runs) for the `chacha20` and `sha2` test cases. Unlike for the OISA benchmark results, where `BLACKOUT`'s overhead could be attributed to zeroing out stack and heap variables, the ProSpeCT version of the SpectreGuard benchmark places all

Table 7.4: SpectreGuard benchmark performance average measured over 20 runs of each test case (maximum $\sigma = 0.08\%$).

Blinded CHERI-Toooba	25S/75C		50S/50C		75S/25C		90S/10C	
	Total Ticks	Δ (to row above)	Total Ticks	Δ (to row above)	Total Ticks	Δ (to row above)	Total Ticks	Δ (to row above)
chacha20								
nocap	25860874	–	23132682	–	25417083	–	20661078	–
purecap	22546871	-3314003 (−12.81%)	20378637	-2754045 (−11.91%)	25430015	12932 (0.05%)	20256607	-404472 (−1.96%)
purecap + blinded	22541024	-5847 (−0.03%)	20378935	298 (0.00%)	25434929	4914 (0.02%)	20256061	-546 (0.00%)
sha2								
nocap	24819790	–	21595015	–	25417599	–	22163550	–
purecap	24771185	-48604 (−0.20%)	21591164	-3851 (−0.02%)	26342237	924638 (3.64%)	22378912	215362 (0.97%)
purecap + blinded	24771730	544 (0.00%)	21593499	2335 (0.01%)	26342910	672 (0.00%)	22371464	-7448 (−0.03%)

secret variables in static variables allocated from the program's data section, similar to global variables. Consequently, BLACKOUT does not introduce any discernible overhead for encryption time or workload time. The slight speed improvement of the purecap + blinded version compared to the purecap baseline in chacha20 25S/7C and sha2 90S/10C falls within standard deviation (maximum $\sigma = 0.08\%$). For completeness, we also include results for nocap, which performs worse than purecap in most cases. We attribute this difference to improvements in the RISC-V backend for CHERI-RISC-V targets compared to plain RISC-V targets in CHERI-LLVM, resulting in more compact and efficient assembly. We confirmed this through manual investigation.

For the Curve25519 test case, we discovered a constant-time violation caused by a secret-dependent branch instruction in the compiled binary. Although this violation is not present in the formally verified source code, it is introduced by the compiler, which in our case is CHERI's fork of Clang. This violation is not reported in ProSpeCT's evaluation because it uses the GCC compiler, which likely correctly avoided generating such a secret-dependent branch. This difference in generated assembly code highlights the usefulness of BLACKOUT: different compilers and/or configurations can silently introduce constant-time violations which BLACKOUT can detect and stop even if the source does not contain such a violation.

7.7.3 Security

Empirical Spectre mitigation evaluation. CHERI-Toooba is vulnerable to Spectre-BTB [111], -RSB [129, 115] and -STL [91] (cf. test suite in [67]). We blinded the secret value in all test cases by inserting a single `candperm` instruction. As seen in Table 7.5, BLACKOUT prevents all Spectre attacks from [67] because BLACKOUT catches side-channel violations even during speculation, but suppresses faults until speculation is confirmed (and ignores them otherwise). Additionally, we have replicated ProSpeCT's [49] Spectre tests. However,

Table 7.5: Transient-execution attacks successfully prevented on CHERI-Toooba and Blinded CHERI-Toooba. ✓ indicates the attack is successfully defended against, while ✗ indicates the attack succeeds.

	Spectre variant			
	PHT	BTB	RSB	STL
CHERI-Toooba	✓	✗	✗	✗
Blinded CHERI-Toooba	✓	✓	✓	✓

as they involve dereferencing a secret data value as a pointer, they are inherently prevented by CHERI's architectural security properties, which preclude dereferencing non-capability data. Note that none of the Spectre tests in [67, 49] use transient capability forgery, and therefore do not require CSC to prevent.

Empirical non-interference evaluation. Data-oblivious software inherently provides the non-interference property [210]. We verify this empirically using the methodology from [210, 209, 29]: relevant signal traces are extracted from the VCD waveforms across different program runs to ensure that runs with different secret data values produce identical signal traces. As in prior work [210], we extract traces for signals that could leak secret data through cache-timing (addresses of cache accesses), speculative execution (branch predictor states), and port contention and instruction latencies (reorder buffer scheduling). As `BLACKOUT` enforces data-oblivious processing of blinded data, we use the data-oblivious `binary_search` program from [218] as a case study. We vary the blinded data values between two runs of the program, and verify that the extracted traces for the relevant signals are identical.

Theoretical security evaluation. Our security argument is composed of three invariants:

1. **Standard CHERI-provided memory safety**. This includes attacks based on memory safety violations, such as out-of-bounds access and use-after-free. We do not loosen any of the restrictions imposed by standard CHERI (such as proper capability bounds and lifetime). We are therefore compliant with the standard CHERI model, and as such inherit its safety guarantees against spatial and temporal memory violations. This extends to the guarantees provided by formal models [72, 54, 155].
2. **Side-channel protection for blinded data**. This is provided by the enforcement of data-oblivious computation on blinded data (I4 and I5). Any violation of this results in a

fault, as explained in Section 7.5. We inherit formal guarantees for this from the OISA and BliMe models [218, 59]. We further show its efficacy empirically by running the OISA benchmarks [218] using `BLACKOUT` and verify that introducing non-data-oblivious changes to them is either detected by our compiler or causes a run-time fault.

3. **Confidentiality of blinded data with respect to direct access**. While BliMe guarantees the confidentiality of blinded data in memory using in-memory tags, we achieve the same goal by ensuring that 1) blinded capabilities have exclusive access to blinded data throughout their lifetime (I3), 2) memory containing blinded data is cleared at the end of the corresponding blinded capability's lifetime (also I3), and 3) blinded data cannot be stored to memory using unblinded capabilities (I1). The resulting combination ensures that, as in OISA and BliMe, any data stored as blinded into memory, is loaded as blinded into registers. This holds until the data is explicitly declassified (Section 7.5).

7.8 Discussion & Limitations

Blinding existing variables and oblivious-data-race-safety. As discussed in Section 7.5, `BLACKOUT` ensures that memory designated as blinded—either annotated by the developer, or inferred as such by the compiler—is blinded at allocation time. This ensures exclusive access through corresponding blinded capabilities, satisfying invariants I1 to I3 (Section 7.5). However, some correct data-oblivious programs may initially allocate memory as non-blinded, only requiring it to be blinded after interacting with other blinded data. While `BLACKOUT` restricts this pattern, a less-restrictive variation could allow dynamically blinding previously allocated memory. This would require scanning program memory to revoke existing, non-blinded capabilities referencing to-be-blinded memory, similar to revocation strategies used by Cornucopia [204] and Cornucopia Reloaded [63]. The trade-off is increased performance overhead. Future work might explore dynamic or delayed revocation strategies to reduce this cost while ensuring eventual exclusive access.

Eventually, data-oblivious software must *unblind* memory after data-oblivious computation concludes and secret intermediates are cleared (Section 7.5). In concurrent scenarios, shared blinded memory may require revoking overlapping blinded capabilities to maintain temporal memory safety and prevent unauthorized reuse after unblinding. Currently, `BLACKOUT` does not address this *oblivious-data-race-safety*, leaving it as an open problem for future work.

Unblinding results through capability escrow. A solution for further securing the unblinding of blinded memory containing non-sensitive results is to leverage CHERI's facilities for *capability sealing* (Section 7.2). Sealed capabilities cannot be de-referenced and protect against tampering by fixing properties like permissions and bounds; any attempt to tamper with the capability will result in invalidating it. In this context, sealed capabilities enable a form of "*capability escrow*", where a non-blinded version of a blinded capability stored for safe-keeping until the data-oblivious operations complete. Software can use a blinded capability to seal its non-blinded counterpart, which cannot be de-referenced while sealed and thus poses no threat to confidentiality. After data-oblivious computation has completed, secret intermediate values are cleared, and the blinded capability is used to unseal the non-blinded counterpart, allowing access to the result. Designing software APIs for capability escrow is left for future work.

Different techniques for developer annotations In `BLACKOUT`, developers use `__blinded` annotations to indicate blinded variables (Section 7.6.2). Compiler attributes are a common mechanism for conveying semantic information without changing the underlying language. `BLACKOUT` leverages Clang's existing `annotate_type` mechanism, available since LLVM 14, avoiding intrusive compiler changes.

However, this approach has limitations—annotating positional parameters in functions can be cumbersome within the confines of existing types of annotations. An alternative would be to introduce a "blinded C" dialect that integrates blinding directly into the type system, similar to the approaches discussed for Rust in Section 7.1. For example, a `blinded` keyword akin to `const` could be used to mark variables as a blinded counterpart of their basic type. While this would improve developer ergonomics and simplify function parameter annotations, it would require intrusive changes to the language and compiler. This could hinder maintainability and extensibility, especially for evolving platforms like CHERI.

Limitations of blinded register record (BRR) structures for blinded data storage. While BRR structures effectively manage blinded register spills by clearly identifying and restoring blinded values, it is impractical to store all blinded data exclusively using BRR-like structures. The principal reason is the substantial memory overhead—BRR structures effectively double memory consumption. Such overhead would significantly degrade system performance and scalability, particularly for applications requiring large quantities of blinded data. Therefore, BRR structures should remain reserved for specific contexts, such as register spilling, while more efficient blinded capabilities handle general blinded memory storage.

ISA extensions. As we note in Section 7.6.1, BLACKOUT, unlike previous data-oblivious ISAs [218, 59], does not add any additional instructions to the underlying ISA but leverages existing CHERI functionality with only small adjustments to its permission model to facilitate necessary architectural changes to support blinded capabilities. Future work can extend the ISA by adding dedicated instructions to support blinded memory. A potential extension is an instruction to directly set the blindedness bit in a register without a load via a blinded capability. This would allow the blinded capability-enhanced compiler to optimize code where a certain variable can be kept completely in a register throughout its lifetime. Currently, BLACKOUT requires such variables to be allocated on the stack in order to load them via the corresponding blinded capability.

Relevance for CHERI standarization. BLACKOUT demonstrates a practical method for enhancing CHERI's security model without invasive architectural changes by integrating data-oblivious computation capabilities into the existing CHERI protection model. This integration provides insights into how capability-based architectures can evolve to support advanced security properties, such as data-oblivious computation, while maintaining compatibility and performance. Consequently, BLACKOUT's principles and mechanisms can inform ongoing CHERI standardization efforts [39], potentially guiding the evolution of CHERI-enabled architectures towards broader security guarantees. Above, we discussed adding custom instructions. But maintaining the CHERI ISA as we currently do in BLACKOUT provides backward-compatibility for non-BLACKOUT hardware and can ease integration into the standard.

Relationship to non-interference. Since BLACKOUT builds on BliMe, it also ensures (as shown in Section VI of [59]) the values of blinded data have no effect on the rest of the system by 1) preventing blinded values from directly flowing to insecure instructions, and 2) by ensuring no secret-dependent branches are allowed when operating on blinded data, with an exception for controlled declassification as described above and in Section 7.5.2. Given its scope, we conjecture that this guarantee satisfies the non-interference property even though it was not explicitly discussed in [59]. Providing a formal model for BLACKOUT itself and explicitly establishing non-interference are left for future work. Nevertheless, we empirically investigate non-interference for BLACKOUT in Section 7.7.3.

Chapter 8

Discussion & Future Work

In this chapter, I give a landscape of the different protection mechanisms present in my work, and show the trade-offs made by each solution within that landscape. From this, I identify potential gaps and present interesting directions for future work.

8.1 Solution comparisons

I start with a visualization of the landscape: Figure 8.1 shows how the solutions compare to each other in different aspects, represented by different radial axes. Some axes are inverted for consistency such that a larger radial value is "better" for all axes. Individually, solutions are evaluated in each aspect compared to an unmodified commercial off-the-shelf (COTS) baseline.

Confidentiality BliMe provides the highest level of confidentiality guarantees (similar to fully-homomorphic encryption (FHE)), as it protects data from both direct access and side channels, even against a malicious OS or hypervisor. BLACKOUT also covers both direct access and side channels, but does not protect against malicious OSs and hypervisors, as they could gain access to data by manipulating the virtual-to-physical address mappings. CHERI ranks higher than PBI due to its stronger memory safety guarantees, which prevent memory vulnerabilities from occurring, compared to PBI's memory protection mechanism, which does not completely prevent memory vulnerabilities, but limits their exploitability.

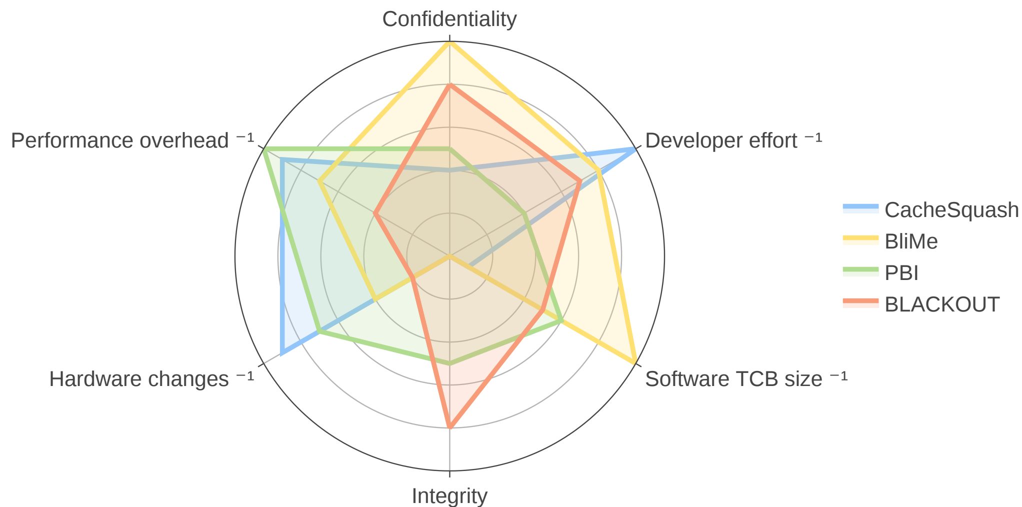


Figure 8.1: Spider chart comparing proposed solutions across confidentiality, integrity, software TCB size, required developer effort, required hardware changes, and performance overhead.

Integrity BLACKOUT provides the highest level of integrity due to its memory safety guarantees inherited from CHERI. As explained above, PBI ranks lower than BLACKOUT as its memory protection mechanism is weaker than the guarantees provided by BLACKOUT. On the other end, both Blinded Memory (BliMe) and CacheSquash provide no integrity guarantees.

Software trusted computing base (TCB) size This axis is inverted such that a smaller TCB size results in a larger radial value. BliMe ranks highest as it requires no trust in software other than what is required for remote attestation. Specifically, it requires no trust in the code processing the data or in the underlying OS and hypervisor. The remaining solutions require some level of trust in software. PBI requires that the software correctly deploys the primitive to protect memory; however, it can be used for sandboxing and isolating vulnerable parts of the code, avoiding the need to trust the *entire* codebase. In a pure-capability configuration, BLACKOUT can also provide isolation for untrusted code, e.g., using capability sealing. In a hybrid configuration, however, BLACKOUT requires trust in the entire codebase and software stack; parts of the code that use regular pointers instead of capabilities can freely access memory. CacheSquash requires the highest level of trust in software, as any malicious code can easily leak secrets through architectural (rather

than microarchitectural) side channels, bypassing CacheSquash.

Developer effort This axis is inverted such that less required developer effort results in a larger radial value. CacheSquash is software-transparent, requiring no change to software and therefore no input from the developer. BliMe requires minimal changes to software that is already data-oblivious for blinding and unblinding data at the beginning and end of the program, respectively. Transforming code to achieve data-obliviousness is not in the scope of BliMe; nevertheless, the BliMe Linter is available to assist with this task, reducing developer effort. BLACKOUT requires more involvement from the developer in order to annotate secrets throughout the program; however, compiler assistance is provided to guide the developer to the required annotations. PBI requires the developer to manually partition memory into domains, and ensure that the protection primitive is used correctly to avoid vulnerabilities. In addition, no compiler assistance is provided. While this is already common practice in software projects that deploy sandboxing and in-process isolation, e.g., V8 [81], it nevertheless requires the largest effort.

Hardware changes This axis is inverted such that fewer hardware changes results in a larger radial value. CacheSquash requires minimal changes: in the core, only the load-store unit (LSU) is modified to send cancellations on squashing, and in the caches, only the miss status holding registers (MSHRs) are modified to process cancellations. Second is PBI, which adds a few registers and requires minor changes to the translation lookaside buffers (TLBs) and memory management unit (MMU) permission check logic. BliMe is ranked lower due to its reliance on memory tagging, which modifies registers, caches and the memory controller to process tags. In addition, BliMe modifies the ALUs to implement taint-tracking and enforce data-obliviousness. BLACKOUT has the highest level of hardware changes as, in addition to the changes required by BliMe, it requires those introduced by CHERI, e.g., double-width registers, and new instructions and capability checks.

Performance overhead This axis is inverted such that a lower performance overhead results in a larger radial value. PBI ranks highest because it improves performance compared to an unmodified system that uses traditional page protections to achieve the same level of security. CacheSquash follows due to its near-zero overheads. BliMe has minimal overheads of $\approx 8\%$, which can be further reduced as shown in Chapter 5. BLACKOUT, on the other hand, has moderate overheads compared to an unmodified baseline due to the inherited underlying CHERI architecture.

8.2 Limitations

Our work is subject to several limitations that should be considered when interpreting our results and applying our solutions.

FPGA-based evaluation The power, resource usage, and maximum clock frequency overheads measured on the FPGA are only an estimate of the overheads as they would be on ASICs, which are the true targets for the processors used in our evaluations. The differences between FPGAs and ASICs can result in significant differences in overheads, which must be accounted for when adapting our changes to production hardware.

Design complexity and verification Hardware changes inherently introduce additional costs to design complexity and verification efforts. While our changes are minimalistic by design to reduce such costs, they must be considered when evaluating our solutions for industrial adoption.

Workload-specific behavior Our performance evaluations rely on geometric means of industry-standard benchmark suites, such as SPEC CPU 2017, that attempt to provide a well-rounded representative set of benchmarks. However, certain workload behavior can increase overheads (e.g., if the software results in repeated zeroing of freed blinded data in BLACKOUT). As a result, adoption of our solutions might require fine-tuning of system parameters (e.g., cache sizes, memory allocation sizes) to achieve optimal performance, especially for specialized workloads that have specific patterns in their behavior.

8.3 Future work

In previous chapters, I identified project-specific promising future work directions. In this section, I present future work with potential synergies and combinations between the solutions.

Integrity for BliMe

As previously mentioned, BliMe's lack of data integrity guarantees fueled the need for BLACKOUT, which introduces strong integrity (due to memory safety) in exchange for

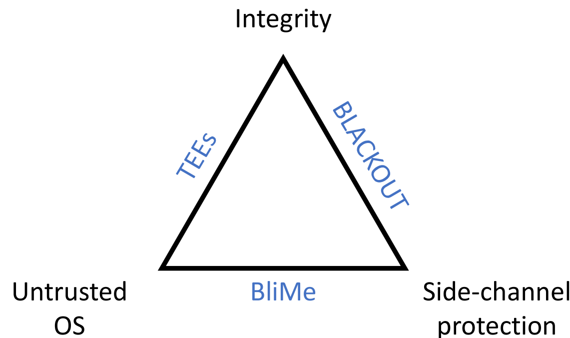


Figure 8.2: Trade-offs made by TEEs, BliMe and BLACKOUT between integrity guarantees, side-channel protection and threat model assumptions.

reduced confidentiality (due to the lack of protection against malicious OSs and hypervisors). Traditional trusted execution environments (TEEs), such as Intel SGX, provide integrity and confidentiality guarantees against malicious OSs and hypervisors by verifying enclave code before execution, but lack side-channel protection. The trade-offs made by each is shown in Figure 8.2. A promising direction for future work is the combination of all the guarantees provided by these solutions. While this is achievable by trivially combining the solutions, the resulting overhead will likely be prohibitive. The challenge is, therefore, in designing a synergistic solution that can identify overlapping protection mechanisms and subsequently reduce the overhead by eliminating redundancies.

Integrity with a small TCB

Figure 8.1 shows a gap where stronger data integrity guarantees seem to require a higher level of trust in the software stack. This is natural as complete distrust of the software (as in BliMe) can allow it to overwrite data or intentionally perform incorrect computations. Unlike confidentiality, which can be protected with hardware alone using taint-tracking, integrity is harder to achieve solely with hardware. Future work can explore this further, potentially combining cryptography with hardware enforcement. One possibility is ensuring data is only processed by verifiable code (identified by cryptographic hashes), in a similar manner to Intel SGX enclaves. Another option to ensure computation results are correct, without requiring complete trust in the software, is to record computations as they occur and verify them using remote attestation; this is similar to the techniques used by control-flow attestation [3]. Yet another alternative is for problems whose solutions can be efficiently

verified. Prior work exists to leverage this in trusted TEE applications [191], but a solution with limited trust in software remains a challenge.

Fault injection and physical attacks

Rowhammer [109] is a vulnerability in modern dynamic random-access memory (DRAM) modules that threatens the integrity of data. Due to the high proximity of DRAM cells, toggling a row of cells at a sufficiently high rate can result in bit flips in adjacent rows. Exploits of this phenomenon by a remote adversary have been continuously demonstrated despite the number of proposed defenses [141]. In this dissertation, we make the common and reasonable assumption that reading data from any address in memory retrieves the last value that was written to that address; this includes any tags in memory such as BliMe's blindedness tags or BLACKOUT's capability tags. Error-correcting code (ECC) bits in DRAM modules raise the barrier for Rowhammer attacks by correcting bit flips, but recent variants of Rowhammer have shown that they can be defeated [43]. Sullivan et al. and Lamster et al. combine memory tagging and ECC bits to reduce memory tagging overheads and make memory tagging more deployable, but their proposals do not improve the effectiveness of ECC against Rowhammer [181, 117]. Other fault-injection attacks based on dynamic voltage and frequency scaling (DVFS), such as CLKscrew [185], V0LTpwn [105] and Plundervolt [140], produce similar attack patterns and are also out of scope. While fault injection attacks have traditionally been considered an orthogonal problem to memory safety, an interesting approach could be to align the data integrity goals desired by both and attempt to derive integrity guarantees from hardware redundancy, such as that used by fault tolerant circuits [160]. In such a solution, the hardware redundancy would not only protect against fault injection attacks, but would also act as a "ground truth" for data against run-time attacks, similar to how shadow stacks are used.

In this dissertation, I focus on *remote* adversaries, those without physical access to the system. Physical access enables full read-write side-band access to memory through direct connection to the DRAM bus. It also facilitates more powerful side-channel attacks, such as those relying on power [112], electromagnetic [71, 126] or temperature measurements [94], or fault-injection attacks, such as VoltPillager [38]. Protecting against physical attacks must take economics into account: a determined attacker with sufficiently advanced equipment can strip silicon layers and observe circuit internals, but at a significant cost of time, money and effort. Furthermore, the location at which the system is deployed is a significant factor: mobile phones are much easier to gain physical access to than servers in a guarded warehouse. As a result, defenses against physical attacks cover a range of threat models. A common threat model is one where an adversary can snoop memory buses and gain direct

read and write access to memory, but is unable to observe the internals of the CPU die. Traditional TEEs make this assumption and rely on authenticated encryption to ensure data confidentiality and integrity against such as adversary. One possibility for future work is to combine the protection provided by authenticated encryption to cover both physical attacks and run-time attacks simultaneously.

Chapter 9

Related Work

9.1 Taint tracking

A large body of work exists on taint tracking, also called dynamic information flow tracking (DIFT) [179]. Hu *et al.* [92] present a survey that includes several hardware-based taint-tracking techniques with varying goals and security/performance trade-offs, and at different abstraction layers. Speculative Taint Tracking [219] applies taint tracking to the results of speculatively executed instructions to prevent them leaking information. Tiwari *et al.* [189] propose taint tracking at the gate level, and use it to create a processor that is able to track all information flows, but has a limited ISA and suffers from large overheads. Taint tracking can also be performed purely in software. Data flow integrity is a form of taint tracking that protects software against non-control data attacks by using reaching definitions analysis [36]. Pointer tainting [214] is another defense against non-control data attacks that taints user input and detects an attack when a tainted value is dereferenced as a pointer.

9.2 Data-oblivious execution

Preventing side-channel leakage requires covering several observable outputs. Several algorithms have been proposed to obfuscate data-dependent memory access patterns [79, 178, 190, 17]. However, they all come at a significant cost to performance. Other work has focused on writing and verifying data-oblivious and constant-time code [139, 22, 21, 45, 42, 9, 161, 147, 64, 175, 226, 4, 137, 167, 62, 30], resulting in numerous tools offering informal [162, 203, 206, 207, 53, 118] and formal [37, 48] guarantees. An actively maintained list of

"constant-timeness" verification tools (CT-tools) currently includes 55 such tools [131]. One work that aims to automatically transform code to make it constant-time is Constantine [30], which extends LLVM to compile code into constant-time binaries. It relies, however, on dynamic analysis to identify vulnerable code for transformation. This can be imprecise as full execution path coverage is not guaranteed, potentially leading to some vulnerable code not being transformed.

Both static analysis and formal methods face significant challenges. The principal shortcoming of static analysis approaches is that data-obliviousness can only be defined at machine-code level, rather than for high-level language constructs. Capturing microarchitectural subtleties of real-world hardware in formal models (and keeping the models up-to-date as hardware evolves) is difficult. Static analyses may not be sound and can lead to over- or under-tainting. Lastly, testing whether a program is data-oblivious remains challenging as tools built on static and formal analysis are typically not integrated into modern toolchains, have significant technical limitations including high overheads to compilation time and many false-positives, and are difficult to use [99, 73, 66].

In practice, constant-time coding practices *alone* are unreliable [158]. Compiler optimizations continue to evolve, and new compilers emerge in unexpected contexts (e.g., in-silicon just-in-time compilers in CPU hardware). Documentation gaps across the software and hardware stack further hinder efforts, especially when target platforms are not narrowly confined [157]. Thus, robust enforcement of data-obliviousness requires hardware/software codesign as in BliMe and BLACKOUT. Such designs allow programmers to annotate sensitive data, enabling hardware to enforce protection against side-channel leakage.

Architectural-Mimicry (AMi) [210] introduces new ISA primitives for more efficient control-flow-linearization, addressing how to make programs data-oblivious. AMi is complementary to data-oblivious ISAs such as BliMe and BLACKOUT, which enforce data-obliviousness through ISA contracts. However, it requires greater developer involvement as it requires developer write assembly manually (with correct usage of new primitives); incorrect usage can *silently* leak secrets.

9.3 Point solutions for side-channel attacks

The literature contains a variety of side-channel attacks that leak information through the processor's caches [217, 85, 149, 84, 86]. Defenses against these attacks rely on temporal or spatial isolation between processes; the cache is either flushed on context switches or is partitioned in such a way that each process uses a separate fixed portion of the cache [151,

198]. However, this results in unnecessary overhead when processing non-sensitive data. Other methods change the cache architecture or replacement policy but also suffer from unnecessary overheads [166, 197].

The cache attacks mentioned above are usually used as building blocks to create covert channels for more sophisticated attacks that exploit transient execution such as Meltdown and Spectre [122, 111]. In response, a range of defenses have been proposed to stop these attacks by isolating the relevant microarchitectural components, such as caches [110, 216, 165, 164, 186, 5, 107, 225, 127]. Speculative taint-tracking techniques[219, 69, 170, 49] delay instructions dependent on speculatively loaded secret data. Unlike CacheSquash, the aforementioned defenses incur significant overheads as they aim to prevent all side-channel leakage. Of recent work, ProSpeCT [49] formalizes the constant-time policy with respect to control flow and memory accesses for a broad class of speculation mechanisms. Its scope is guaranteeing that hardware does not leak secrets of *already constant-time* programs during speculation. Unlike BliMe and BLACKOUT, neither ProSpeCT nor other transient execution defenses *enforce* their guarantees on non-speculative execution.

Chapter 10

Conclusion

Modern computing systems face an increasingly complex threat landscape where sensitive data must be protected against both direct access attacks and sophisticated side-channel vulnerabilities. This dissertation has demonstrated that hardware-assisted mechanisms can indeed efficiently protect the integrity and confidentiality of sensitive data against remote adversaries, addressing threats that span from conventional run-time attacks to subtle microarchitectural side channels. Through a systematic investigation of four research questions, we present four solutions-—CacheSquash, BliMe with its extensions, PBI, and BLACKOUT-—that collectively demonstrate that hardware assistance can provide robust security guarantees while maintaining the performance characteristics essential for practical deployment.

CacheSquash provides speculation-aware cache protection with near-zero overhead (0.48% geometric mean), demonstrating that effective Spectre mitigation is achievable without performance penalties. By cancelling mis-speculated cache requests, it significantly reduces transient execution attack surfaces.

BliMe enables secure outsourced computation with 8% overhead through hardware-enforced taint tracking. It simultaneously protects against direct access and side-channel attacks without requiring trust in processing software or operating systems. The Dolma extension proves this approach scales to specialized ML accelerators with 5.6% overhead. The BliMe Linter uses static analysis to automatically detect potential BliMe violations in software, thereby easing adoption by software developers.

PBI introduces program-counter-based memory protection, eliminating permission register updates during domain transitions, improving security compared to prior mechanisms

by avoiding the presence of unprivileged permission changing instructions in the codebase and limiting available return-oriented programming (ROP) gadgets.

BLACKOUT unifies memory safety and side-channel protection by extending CHERI with blinded capabilities. With only 1.5% overhead compared to pure-capability CHERI and 23.5% compared to unprotected baselines, it proves comprehensive protection is practically achievable.

The central thesis—that hardware-assisted mechanisms can efficiently protect sensitive data against remote adversaries—is validated across multiple threat models and performance requirements. Each solution addresses specific aspects while contributing to a unified understanding of hardware-assisted security. The progression from focused protection (e.g., CacheSquash) to comprehensive solutions (BLACKOUT) demonstrates that protection against multiple threats can be achieved when properly implemented at the hardware level.

The landscape of computer security continues to evolve as adversaries develop increasingly sophisticated attack techniques. However, this dissertation demonstrates that through careful hardware-software co-design, it is possible to stay ahead of these threats while maintaining the performance characteristics that modern applications demand. The solutions presented here represent more than individual technical contributions—they establish a foundation for thinking about security as an integral part of system design rather than an afterthought. By showing that hardware assistance can provide strong security guarantees efficiently, this work opens new possibilities for building inherently secure computing systems. The journey toward comprehensive protection of sensitive data in computing systems is ongoing, but this dissertation has established that hardware-assisted mechanisms provide a viable and efficient path forward. Through continued research and development in this direction, we can build computing systems that provide strong security guarantees without sacrificing the performance and usability that users rightfully expect.

References

[1] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[2] Martín Abadi et al. "Control-flow integrity". In: *Proceedings of the 12th ACM conference on Computer and communications security*. CCS '05. New York, NY, USA: Association for Computing Machinery, Nov. 2005, pp. 340–353. ISBN: 978-1-59593-226-6. DOI: 10.1145/1102120.1102165. URL: https://dl.acm.org/doi/10.1145/1102120.1102165 (visited on 10/22/2025).

[3] Tigist Abera et al. "C-FLAT: Control-Flow Attestation for Embedded Systems Software". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 743–754. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978358. URL: https://dl.acm.org/doi/10.1145/2976749.2978358 (visited on 10/23/2025).

[4] Adil Ahmad et al. "OBLIVIATE: A Data Oblivious Filesystem for Intel SGX". In: *Proceedings of the 2018 Network and Distributed System Security Symposium*. NDSS '18. San Diego, CA: Internet Society, 2018. ISBN: 978-1-891562-49-5. DOI: 10.14722/ndss.2018.23284. URL: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_06A-2_Ahmad_paper.pdf (visited on 04/06/2025).

[5] Sam Ainsworth and Timothy M. Jones. "MuonTrap: Preventing Cross-Domain Spectre-like Attacks by Capturing Speculative State". In: *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*. ISCA '20. Virtual Event: IEEE Press, Sept. 2020, pp. 132–144. ISBN: 978-1-7281-4661-4. DOI: 10.1109/ISCA45697.2020.00022. URL: https://doi.org/10.1109/ISCA45697.2020.00022 (visited on 07/11/2025).

[6] Alejandro Cabrera Aldaya et al. “Port Contention for Fun and Profit”. In: *Proceedings of the 40th IEEE Symposium on Security and Privacy*. S&P ’19. San Francisco, CA, USA: IEEE, May 2019, pp. 870–887. DOI: 10.1109/SP.2019.00066. URL: https://ieeexplore.ieee.org/document/8835264 (visited on 07/13/2025).

[7] Saar Amar et al. “CHERIoT: Complete Memory Safety for Embedded Devices”. In: *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’23. Toronto ON Canada: ACM, Oct. 2023, pp. 641–653. ISBN: 979-8-4007-0329-4. DOI: 10.1145/3613424.3614266. URL: https://dl.acm.org/doi/10.1145/3613424.3614266 (visited on 06/28/2024).

[8] Alon Amid et al. “Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs”. In: *IEEE Micro* 40.4 (2020), pp. 10–21. ISSN: 1937-4143. DOI: 10.1109/MM.2020.2996616.

[9] Marc Andrysco et al. “On Subnormal Floating Point and Abnormal Timing”. In: *Proceedings of the 36th IEEE Symposium on Security and Privacy*. S&P ’15. San Francisco, CA, USA: IEEE, May 2015, pp. 623–639. DOI: 10.1109/SP.2015.44. URL: https://ieeexplore.ieee.org/document/7163051 (visited on 04/06/2025).

[10] Apple. *Secure Enclave*. https://support.apple.com/en-ca/guide/security/sec59b0b31ff/web. Apple Support, 2022.

[11] Arm. *AMBA CHI Architecture Specification*. 2024. URL: https://developer.arm.com/documentation/ihi0050/latest/.

[12] Arm. *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*. URL: https://developer.arm.com/documentation/ddi0406/b/System-Level-Architecture/Virtual-Memory-System-Architecture--VMSA-/Memory-access-control/Domains (visited on 10/23/2025).

[13] Arm. *Arm Confidential Compute Architecture*. en. URL: https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture (visited on 10/23/2025).

[14] Arm. *Armv8.1-M PACBTI Extensions*. 2024. URL: https://developer.arm.com/documentation/109576/0100/Branch-Target-Identification (visited on 10/23/2025).

[15] Arm. “MTE User Guide for Android OS”. en. In: (2023).

[16] Krste Asanović et al. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, Apr. 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[17] Gilad Asharov et al. "OptORAMa: optimal oblivious RAM". In: *Proceedings of the 39th international conference on the theory and applications of cryptographic techniques*. Number of pages: 30 Place: Zagreb, Croatia. Berlin, Heidelberg: Springer-Verlag, May 2020, pp. 403–432. ISBN: 978-3-030-45723-5. DOI: 10.1007/978-3-030-45724-2_14. URL: https://doi.org/10.1007/978-3-030-45724-2_14.

[18] Jean-Philippe Aumasson. *Cryptocoding*. https://github.com/veorq/cryptocoding. Github, 2022.

[19] Jonathan Bachrach et al. "Chisel: Constructing Hardware in a Scala Embedded Language". In: *Proceedings of the Design Automation Conference*. New York, NY, USA, 2012, pp. 1216–1225. ISBN: 978-1-4503-1199-1. DOI: 10.1145/2228360.2228584.

[20] Mohammad Behnia et al. "Speculative interference attacks: breaking invisible speculation schemes". In: *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2021.

[21] Daniel J. Bernstein. "Curve25519: New Diffie-Hellman Speed Records". In: *Proceedings of the 9th International Conference on Theory and Practice of Public-Key Cryptography*. PKC '06. Berlin, Heidelberg: Springer-Verlag, Apr. 2006, pp. 207–228. ISBN: 978-3-540-33851-2. DOI: 10.1007/11745853_14. URL: https://doi.org/10.1007/11745853_14 (visited on 04/06/2025).

[22] Daniel J. Bernstein. "The Poly1305-AES Message-Authentication Code". In: *Proceedings of the 12th International Conference on Fast Software Encryption*. FSE '05. Berlin, Heidelberg: Springer-Verlag, Feb. 2005, pp. 32–49. ISBN: 978-3-540-26541-2. DOI: 10.1007/11502760_3. URL: https://doi.org/10.1007/11502760_3 (visited on 04/06/2025).

[23] Daniel J. Bernstein et al. "High-Speed High-Security Signatures". In: *Journal of Cryptographic Engineering* 2.2 (2012), pp. 77–89. ISSN: 2190-8516. DOI: 10.1007/s13389-012-0027-1.

[24] Daniel J. Bernstein et al. "TweetNaCl: A Crypto Library in 100 Tweets". In: *Progress in Cryptology - LATINCRYPT 2014*. Ed. by Diego F. Aranha and Alfred Menezes. Vol. 8895. Cham, 2015, pp. 64–83. ISBN: 978-3-319-16295-9. DOI: 10.1007/978-3-319-16295-9_4.

[25] Atri Bhattacharyya et al. "SpecROP: Speculative Exploitation of ROP Chains". In: *Proceedings of the 29th USENIX Security Symposium*. USENIX Security '20. USENIX Association, 2020.

[26] Christian Bienia et al. “The PARSEC benchmark suite: characterization and architectural implications”. In: *International Conference on Parallel Architectures and Compilation Techniques*. ACM, 2008. ISBN: 978-1-60558-282-5. DOI: 10.1145/1454115.1454128.

[27] Nathan Binkert et al. “The gem5 simulator”. In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718.

[28] Tyler Bletsch et al. “Jump-oriented programming: a new class of code-reuse attack”. In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ASIACCS ’11. New York, NY, USA: Association for Computing Machinery, Mar. 2011, pp. 30–40. ISBN: 978-1-4503-0564-8. DOI: 10.1145/1966913.1966919. URL: https://doi.org/10.1145/1966913.1966919 (visited on 10/18/2025).

[29] Marton Bognar et al. “MicroProfiler: Principled Side-Channel Mitigation through Microarchitectural Profiling”. In: *Proceedings of the 8th IEEE European Symposium on Security and Privacy*. EuroS&P ’23. IEEE, July 2023, pp. 651–670. DOI: 10.1109/EuroSP57164.2023.00045. URL: https://ieeexplore.ieee.org/document/10190518 (visited on 07/29/2025).

[30] Pietro Borrello et al. “Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization”. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’21. New York, NY, USA: Association for Computing Machinery, Nov. 2021, pp. 715–733. ISBN: 978-1-4503-8454-4. DOI: 10.1145/3460120.3484583. URL: https://doi.org/10.1145/3460120.3484583 (visited on 04/01/2025).

[31] Martin Brænne and Dmitri Gribenko. *[RFC] New Attribute ‘annotate_type‘ (Iteration 2)*. LLVM Discussion Forums. (accessed 2025-03-30). Apr. 2022. URL: https://discourse.llvm.org/t/61378 (visited on 03/30/2025).

[32] Nathan Burow et al. “Control-Flow Integrity: Precision, Security, and Performance”. In: *ACM Comput. Surv.* 50.1 (Apr. 2017), 16:1–16:33. ISSN: 0360-0300. DOI: 10.1145/3054924. URL: https://dl.acm.org/doi/10.1145/3054924 (visited on 10/23/2025).

[33] Bytecode Alliance. *GHSA-hpqh-2wqx-7qp5: Memory access due to code generation flaw in Cranelift module*. https://github.com/bytecodealliance/wasmtime/security/advisories/GHSA-hpqh-2wqx-7qp5. CVE-2021-32629. 2021.

[34] Claudio Canella et al. "A Systematic Evaluation of Transient Execution Attacks and Defenses". In: *Proceedings of the 28th USENIX Security Symposium*. USENIX Security '19. Santa Clara, CA, USA: USENIX Association, Aug. 2019, pp. 249–266. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/canella.

[35] Claudio Canella et al. "Fallout: Leaking Data on Meltdown-resistant CPUs". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 769–784. ISBN: 978-1-4503-6747-9. DOI: 10.1145/3319535.3363219. URL: https://dl.acm.org/doi/10.1145/3319535.3363219 (visited on 07/12/2025).

[36] Miguel Castro, Manuel Costa, and Tim Harris. "Securing software by enforcing data-flow integrity". In: *Proceedings of the 7th symposium on operating systems design and implementation*. OSDI '06. USA: USENIX Association, 2006, pp. 147–160. ISBN: 1-931971-47-1.

[37] Sunjay Cauligi et al. "FaCT: A Flexible, Constant-Time Programming Language". In: *2017 IEEE Cybersecurity Development (SecDev)*. SecDev '17. Cambridge, MA, USA: IEEE, Sept. 2017, pp. 69–76. DOI: 10.1109/SecDev.2017.24. URL: https://ieeexplore.ieee.org/document/8077809 (visited on 04/06/2025).

[38] Zitai Chen et al. "VoltPillager: hardware-based fault injection attacks against intel SGX enclaves using the SVID voltage scaling interface". In: *Proceedings of the 30th USENIX Security Symposium*. USENIX Security '21. USENIX Association, 2021, pp. 699–716. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/chen-zitai.

[39] CheriAlliance. *CHERI Alliance – Industry-led Security Technology*. (accessed 2025-04-06). 2025. URL: https://cheri-alliance.org/ (visited on 04/06/2025).

[40] Li-Chung Chiang and Shih-Wei Li. "Reload+Reload: Exploiting Cache and Memory Contention Side Channel on AMD SEV". In: *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Vol. 2. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, Mar. 2025, pp. 1014–1027. ISBN: 979-8-4007-1079-7. DOI: 10.1145/3676641.3716017. URL: https://dl.acm.org/doi/10.1145/3676641.3716017 (visited on 04/02/2025).

[41] David Chisnall et al. "CHERI JNI: Sinking the Java Security Model into the C". In: *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '17. New York, NY,

USA: Association for Computing Machinery, Apr. 2017, pp. 569–583. ISBN: 978-1-4503-4465-4. DOI: 10.1145/3037697.3037725. URL: https://dl.acm.org/doi/10.1145/3037697.3037725 (visited on 10/31/2023).

[42] Jeroen V. Cleemput, Bart Coppens, and Bjorn De Sutter. "Compiler Mitigations for Time Attacks on Modern X86 Processors". In: *ACM Transactions on Architecture and Code Optimization* 8.4 (Jan. 2012), pp. 1–20. ISSN: 1544-3566, 1544-3973. DOI: 10.1145/2086696.2086702. URL: https://dl.acm.org/doi/10.1145/2086696.2086702 (visited on 04/06/2025).

[43] Lucian Cojocar et al. "Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks". In: *2019 IEEE Symposium on Security and Privacy (SP)*. ISSN: 2375-1207. May 2019, pp. 55–71. DOI: 10.1109/SP.2019.00089. URL: https://ieeexplore.ieee.org/document/8835222 (visited on 10/23/2025).

[44] R. Joseph Connor et al. "PKU pitfalls: attacks on PKU-based memory isolation systems". In: *Proceedings of the 29th USENIX Security Symposium*. USENIX Security '20. USA: USENIX Association, 2020. ISBN: 978-1-939133-17-5.

[45] Bart Coppens et al. "Practical Mitigations for Timing-Based Side-Channel Attacks on Modern X86 Processors". In: *Proceedings of the 30th IEEE Symposium on Security and Privacy*. S&P '09. San Francisco, CA, USA: IEEE, May 2009, pp. 45–60. ISBN: 978-0-7695-3633-0. DOI: 10.1109/SP.2009.19. URL: https://doi.org/10.1109/SP.2009.19 (visited on 04/06/2025).

[46] Mozilla Corporation. *SpiderMonkey*. en-US. URL: https://spidermonkey.dev/ (visited on 10/23/2025).

[47] Victor Costan and Srinivas Devadas. *Intel SGX explained*. 2016. URL: https://ia.cr/2016/086.

[48] Lesly-Ann Daniel, Sebastien Bardin, and Tamara Rezk. "Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level". In: *Proceedings of the 41st IEEE Symposium on Security and Privacy*. S&P '20. San Francisco, CA, USA: IEEE, May 2020, pp. 1021–1038. ISBN: 978-1-7281-3497-0. DOI: 10.1109/SP40000.2020.00074. URL: https://ieeexplore.ieee.org/document/9152766/ (visited on 04/06/2025).

[49] Lesly-Ann Daniel et al. "ProSPeCT: Provably Secure Speculation for the Constant-Time Policy". In: *Proceedings of the 32nd USENIX Security Symposium*. USENIX Security '23. Anaheim, CA, USA: USENIX Association, Aug. 2023, pp. 7161–7178. ISBN: 978-1-939133-37-3. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/daniel.

[50] Brooks Davis et al. "CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, Apr. 2019, pp. 379–393. ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304042. URL: https://dl.acm.org/doi/10.1145/3297858.3304042 (visited on 10/31/2023).

[51] Joe Devietti et al. "Hardbound: architectural support for spatial safety of the C programming language". In: *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*. ASPLOS XIII. New York, NY, USA: Association for Computing Machinery, 2008, pp. 103–114. ISBN: 978-1-59593-958-6. DOI: 10.1145/1346281.1346295. URL: https://doi.org/10.1145/1346281.1346295.

[52] Digital Security by Design. *Benchmarking Guidance - Getting Started with CheriBSD 23.11.* (accessed 2025-04-13). May 2024. URL: https://www.cheribsd.org/getting-started/23.11/benchmarking/ (visited on 04/13/2025).

[53] Craig Disselkoen. *Haybale-Pitchfork*. UCSD PLSysSec. Oct. 2024. URL: https://github.com/PLSysSec/haybale-pitchfork (visited on 03/21/2025).

[54] Anna Lena Duque Antón et al. "VeriCHERI: Exhaustive Formal Security Verification of CHERI at the RTL". In: *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*. ICCAD '24. New York, NY, USA: Association for Computing Machinery, Apr. 2025, pp. 1–9. ISBN: 979-8-4007-1077-3. URL: https://doi.org/10.1145/3676536.3676841 (visited on 07/25/2025).

[55] Hossam ElAtali and N. Asokan. "CacheSquash: Making caches speculation-aware". In: *arXiv preprint: 2406.12110* (2025). DOI: 10.48550/arXiv.2406.12110.

[56] Hossam ElAtali, Hans Liljestrand, and Jan-Erik Ekberg. "PBI: Program-Counter-Based Isolation". In: *To Be Submitted*. 2025.

[57] Hossam ElAtali et al. "BLACKOUT: Data-Oblivious Computation with Blinded Capabilities". In: *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*. CCS '25. Taipei, Taiwan: Association for Computing Machinery, Oct. 2025. DOI: 10.1145/3719027.3765169.

[58] Hossam ElAtali et al. "BliMe Linter". In: *Proceedings of the 2024 IEEE Secure Development Conference (SecDev)*. Pittsburgh, PA, USA: IEEE, Oct. 2024, pp. 46–53. DOI: 10.1109/SecDev61143.2024.00011.

[59] Hossam ElAtali et al. "BliMe: Verifiably Secure Outsourced Computation with Hardware-Enforced Taint Tracking". In: *Proceedings of the 2024 Network and Distributed System Security Symposium*. NDSS '24. San Diego, CA, USA: Internet Society, 2024. ISBN: 1-891562-93-2. DOI: 10.14722/ndss.2024.24105.

[60] Hossam ElAtali et al. "Data-Oblivious ML Accelerators using Hardware Security Extensions". In: *Proceedings of the 2024 IEEE International Symposium on Hardware Oriented Security and Trust*. Tysons Corner, VA, USA: IEEE, 2024. DOI: 10.1109/HOST55342.2024.10545398.

[61] Archibald Samuel Elliott et al. "Checked C: Making C Safe by Extension". In: *2018 IEEE Cybersecurity Development (SecDev)*. Sept. 2018, pp. 53–60. DOI: 10.1109/SecDev.2018.00015. URL: https://ieeexplore.ieee.org/abstract/document/8543387 (visited on 10/23/2025).

[62] Saba Eskandarian and Matei Zaharia. "ObliDB: Oblivious Query Processing for Secure Databases". In: *Proc. VLDB Endow.* 13.2 (Oct. 2019), pp. 169–183. ISSN: 2150-8097. DOI: 10.14778/3364324.3364331. URL: https://doi.org/10.14778/3364324.3364331 (visited on 04/06/2025).

[63] Nathaniel Wesley Filardo et al. "Cornucopia Reloaded: Load Barriers for CHERI Heap Temporal Safety". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Vol. 2. ASPLOS '24. La Jolla CA USA: ACM, Apr. 2024, pp. 251–268. ISBN: 979-8-4007-0385-0. DOI: 10.1145/3620665.3640416. URL: https://dl.acm.org/doi/10.1145/3620665.3640416 (visited on 06/28/2024).

[64] Ben Fisch et al. "IRON: Functional Encryption Using Intel SGX". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 765–782. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134106. URL: https://doi.org/10.1145/3133956.3134106 (visited on 04/06/2025).

[65] Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*. 2024. URL: https://www.agner.org/optimize/microarchitecture.pdf.

[66] Marcel Fourné et al. ""These Results Must Be False": A Usability Evaluation of Constant-Time Analysis Tools". In: *Proceedings of the 33rd USENIX Security Symposium*. USENIX Security '24. USENIX Association, 2024. ISBN: 978-1-939133-44-1. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/fourne.

[67] Franz A Fuchs et al. “Developing a Test Suite for Transient-Execution Attacks on RISC-V and CHERI-RISC-V”. In: *Fifth Workshop on Computer Architecture Research with RISC-V*. CARRV ’21. 2021, p. 7. URL: https://carrv.github.io/2021/papers/CARRV2021_paper_95_Fuchs.pdf.

[68] Franz A. Fuchs et al. “Safe Speculation for CHERI”. In: *Proceedings of the 42nd IEEE International Conference on Computer Design*. ICCD ’24. Milan, Italy: IEEE, Nov. 2024, pp. 364–372. ISBN: 979-8-3503-8040-8. DOI: 10.1109/ICCD63220.2024.00063. URL: https://ieeexplore.ieee.org/document/10818045 (visited on 01/18/2025).

[69] Jacob Fustos, Farzad Farshchi, and Heechul Yun. “SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks”. In: *Proceedings of the 56th Annual Design Automation Conference 2019*. DAC ’19. New York, NY, USA: Association for Computing Machinery, June 2019, pp. 1–6. ISBN: 978-1-4503-6725-7. DOI: 10.1145/3316781.3317914. URL: https://dl.acm.org/doi/10.1145/3316781.3317914 (visited on 07/12/2025).

[70] Shay Gal-On and Markus Levy. “Exploring coremark a benchmark maximizing simplicity and efficacy”. In: *The Embedded Microprocessor Benchmark Consortium* 6.23 (2012), p. 87.

[71] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. “Electromagnetic Analysis: Concrete Results”. In: *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*. Berlin, Heidelberg, 2001, pp. 251–261. ISBN: 3-540-42521-7.

[72] Dapeng Gao and Tom Melham. “End-to-End Formal Verification of a RISC-V Processor Extended with Capability Pointers”. In: *Proceedings of the 21st Conference on Formal Methods in Computer-Aided Design*. Ed. by Ruzica Piskac and Michael W. Whalen. FMCAD ’21. TU Wien, Oct. 2021. DOI: 10.34727/2021/ISBN.978-3-85448-046-4. URL: https://repositum.tuwien.at/handle/20.500.12708/18607 (visited on 04/09/2025).

[73] Antoine Geimer et al. “A Systematic Evaluation of Automated Tools for Side-Channel Vulnerabilities Detection in Cryptographic Libraries”. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’23. New York, NY, USA: Association for Computing Machinery, Nov. 2023, pp. 1690–1704. ISBN: 979-8-4007-0050-7. DOI: 10.1145/3576915.3623112. URL: https://doi.org/10.1145/3576915.3623112 (visited on 04/06/2025).

[74] Hasan Genc et al. "Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration". In: *Proceedings of the 58th annual design automation conference*. 2021. DOI: 10.1109/DAC18074.2021.9586216.

[75] Hasan Genc et al. "Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration". In: *Proceedings of the 58th Annual Design Automation Conference*. 2021. DOI: 10.1109/DAC18074.2021.9586216.

[76] Aïna Linn Georges et al. "Efficient and Provable Local Capability Revocation Using Uninitialized Capabilities". In: *Proceedings of the 48th ACM SIGPLAN Symposium on Principles of Programming Languages*. Vol. 5. POPL '21. Jan. 2021. DOI: 10.1145/3434287. URL: https://dl.acm.org/doi/10.1145/3434287 (visited on 10/29/2023).

[77] Lukas Gerlach, Robert Pietsch, and Michael Schwarz. "Do Compilers Break Constant-time Guarantees?" In: *Proceedings of the 29th International Conference on Financial Cryptography and Data Security*. FC '25. Miyakojima, Japan: Springer, Apr. 2025. URL: https://fc25.ifca.ai/preproceedings/13.pdf.

[78] Barbara Gigerl. "Automated Analysis of Speculation Windows in Spectre Attacks". Master's thesis. Graz University of Technology, 2019.

[79] Oded Goldreich and Rafail Ostrovsky. "Software Protection and Simulation on Oblivious RAMs". In: *Journal of the ACM* 43.3 (1996), pp. 431–473. ISSN: 0004-5411. DOI: 10.1145/233551.233553.

[80] Google. *SafeSide*. 2020. URL: https://github.com/google/safeside/tree/main.

[81] Google. *V8 JavaScript engine*. URL: https://v8.dev/ (visited on 10/23/2025).

[82] Ben Gras et al. "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks". In: *Proceedings of the 27th USENIX Security Symposium*. USENIX Security '18. USENIX Association, 2018.

[83] Richard Grisenthwaite. "Arm Morello Evaluation Platform -Validating CHERI-based Security in a High-performance System". In: *Proceedings of the IEEE Hot Chips 34 Symposium*. HCS '22. IEEE, Aug. 2022, pp. 1–22. DOI: 10.1109/HCS55958.2022.9895591. URL: https://ieeexplore.ieee.org/document/9895591 (visited on 10/31/2023).

[84] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "Cache template attacks: automating attacks on inclusive last-level caches". In: *Proceedings of the 24th USENIX Security Symposium*. USENIX Security '15. Number of pages: 16 Place: Washington, D.C. USA: USENIX Association, 2015, pp. 897–912. ISBN: 978-1-931971-23-2.

[85] Daniel Gruss et al. "Flush+Flush: A Fast and Stealthy Cache Attack". In: *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Vol. 9721. DIMVA '16. Berlin, Heidelberg: Springer-Verlag, July 2016, pp. 279–299. ISBN: 978-3-319-40666-4. DOI: 10.1007/978-3-319-40667-1_14. URL: https://doi.org/10.1007/978-3-319-40667-1_14 (visited on 03/30/2025).

[86] Daniel Gruss et al. "Prefetch side-channel attacks: bypassing SMAP and kernel ASLR". In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. CCS '16. Number of pages: 12 Place: Vienna, Austria. New York, NY, USA: Association for Computing Machinery, 2016, pp. 368–379. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978356. URL: https://doi.org/10.1145/2976749.2978356.

[87] *Guarded Control Stack support for AArch64 Linux — The Linux Kernel documentation*. URL: https://docs.kernel.org/arch/arm64/gcs.html (visited on 10/23/2025).

[88] Merve Gülmez et al. "Mon CHÉRI: Mitigating Uninitialized Memory Access with Conditional Capabilities". In: *46th IEEE Symposium on Security and Privacy*. S&P '15. San Francisco, CA, USA: IEEE Computer Society, Apr. 2025, pp. 829–847. ISBN: 979-8-3315-2236-0. DOI: 10.1109/SP61157.2025.00133. URL: https://www.computer.org/csdl/proceedings-article/sp/2025/223600a791/26hiTWeltII (visited on 06/14/2025).

[89] Mohammad Hedayati et al. "Hodor: intra-process isolation for high-throughput data plane libraries". In: *Proceedings of the USENIX Annual Technical Conference*. USENIX ATC '19. Renton, WA, USA: USENIX Association, 2019, 489–503. ISBN: 9781939133038.

[90] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN: 0128119055.

[91] Jann Horn. *Speculative Execution, Variant 4: Speculative Store Bypass*. Project Zero Issue Tracker. (accessed 2025-07-26). Feb. 2018. URL: https://project-zero.issues.chromium.org/issues/42450580 (visited on 07/26/2025).

[92] Wei Hu, Armaiti Ardeshiricham, and Ryan Kastner. "Hardware information flow tracking". In: *ACM computing surveys* 54.4 (May 2021). Number of pages: 39 Place: New York, NY, USA Publisher: Association for Computing Machinery tex.articleno: 83 tex.issue_date: May 2022, pp. 1–39. ISSN: 0360-0300. DOI: 10.1145/3447867. URL: https://doi.org/10.1145/3447867.

[93] Wei Huang et al. "LMP: light-weighted memory protection with hardware assistance". In: *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACSAC '16. New York, NY, USA: Association for Computing Machinery, Dec. 2016, pp. 460–470. ISBN: 978-1-4503-4771-6. DOI: 10.1145/2991079.2991089. URL: https://dl.acm.org/doi/10.1145/2991079.2991089 (visited on 10/22/2025).

[94] Michael Hutter and Jörn-Marc Schmidt. *The temperature side channel and heating fault attacks*. tex.howpublished: Cryptology ePrint Archive, Report 2014/190. 2014. URL: https://ia.cr/2014/190.

[95] Intel. *Intel Research on Disclosure Gadgets at Indirect Branch Targets in the Linux Kernel*. 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/news/update-to-research-on-disclosure-gadgets-in-linux.html.

[96] Intel. *Refined Speculative Execution Terminology*. 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/refined-speculative-execution-terminology.html.

[97] Intel. *Runtime Encryption of Memory with Intel® Total Memory...* en. URL: https://www.intel.com/content/www/us/en/developer/articles/news/runtime-encryption-of-memory-with-intel-tme-mk.html (visited on 10/23/2025).

[98] Intel. *Security Best Practices for Side Channel Resistance*. (accessed 2025-04-01). Mar. 2019. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/security-best-practices-side-channel-resistance.html (visited on 04/01/2025).

[99] Jan Jancar et al. ""They're Not That Hard to Mitigate": What Cryptographic Library Developers Think About Timing Attacks". In: *Proceedings of the 43rd IEEE Symposium on Security and Privacy*. S&P '22. San Francisco, CA, USA: IEEE, May 2022, pp. 632–649. ISBN: 978-1-6654-1316-9. DOI: 10.1109/SP46214.2022.9833713. URL: https://ieeexplore.ieee.org/document/9833713/ (visited on 04/06/2025).

[100] Alexandre Joannou et al. "Efficient Tagged Memory". In: *Proceedings of the 2017 IEEE International Conference on Computer Design (ICCD)*. ISSN: 1063-6404. 2017, pp. 641–648. DOI: 10.1109/ICCD.2017.112.

[101] Brian Johannesmeyer et al. "Kasper: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel". In: *Network and Distributed System Security Symposium*. The Internet Society, 2022.

[102] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *Proceedings of the Annual International Symposium on Computer Architecture*. ISCA '17. Toronto, ON, Canada: Association for Computing Machinery, 2017, 1–12. ISBN: 9781450348928. DOI: 10.1145/3079856.3080246. URL: https://doi.org/10.1145/3079856.3080246.

[103] Karthik B. K. *RISC-V Memory Protection: Diving Deep into the Complexities*. en. Nov. 2023. URL: https://medium.com/@talktokarthikbk/risc-v-memory-protection-diving-deep-into-the-complexities-9d751212be6b (visited on 10/23/2025).

[104] Sagar Karandikar et al. "FireSim: FPGA-Accelerated Cycle-Exact Scale-out System Simulation in the Public Cloud". In: *Proceedings of the International Symposium on Computer Architecture*. 2014, pp. 29–42. DOI: 10.1109/isca.2018.00014.

[105] Zijo Kenjar et al. "V0LTpwn: attacking x86 processor integrity from software". In: *Proceedings of the 29th USENIX Security Symposium*. USENIX Security '20. USENIX Association, 2020, pp. 1445–1461. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/kenjar.

[106] Khaled N. Khasawneh et al. "SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation". en. In: *Proceedings of the 56th Annual Design Automation Conference 2019*. Las Vegas NV USA: ACM, June 2019, pp. 1–6. ISBN: 978-1-4503-6725-7. DOI: 10.1145/3316781.3317903. URL: https://dl.acm.org/doi/10.1145/3316781.3317903 (visited on 06/04/2024).

[107] Sungkeun Kim et al. "ReViCe: Reusing Victim Cache to Prevent Speculative Cache Leakage". In: *Proceedings of the 2020 IEEE Secure Development Conference*. SecDev '20. IEEE, Sept. 2020, pp. 96–107. DOI: 10.1109/SecDev45635.2020.00029. URL: https://ieeexplore.ieee.org/document/9229934 (visited on 07/29/2025).

[108] Taehun Kim and Youngjoo Shin. "ThermalBleed: A practical thermal side-channel attack". In: *IEEE access: practical innovations, open solutions* 10 (2022), pp. 25718–25731. DOI: 10.1109/ACCESS.2022.3156596.

[109] Yoongu Kim et al. "Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors". In: *Proceedings of the 41st international symposium on computer architecuture*. ISCA '14. Number of pages: 12. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 361–372. ISBN: 978-1-4799-4394-4.

[110] Vladimir Kiriansky et al. “DAWG: A Defense against Cache Timing Attacks in Speculative Execution Processors”. In: *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’18. Fukuoka, Japan: IEEE Press, Oct. 2018, pp. 974–987. ISBN: 978-1-5386-6240-3. DOI: 10.1109/MICRO.2018.00083. URL: https://doi.org/10.1109/MICRO.2018.00083 (visited on 07/29/2025).

[111] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *Proceedings of the 40th IEEE Symposium on Security and Privacy*. S&P ’19. May 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002. URL: https://ieeexplore.ieee.org/document/8835233 (visited on 03/30/2025).

[112] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. “Differential power analysis”. In: *Proceedings of the 19th international cryptology conference on advances in cryptology*. CRYPTO ’99. Number of pages: 10. Berlin, Heidelberg: Springer-Verlag, 1999, pp. 388–397. ISBN: 3-540-66347-9.

[113] K. Koning. “Tricking hardware into efficiently securing software”. Undefined/Unknown. PhD-Thesis - Research and graduation internal. Vrije Universiteit Amsterdam, 2021.

[114] Koen Koning et al. “No Need to Hide: Protecting Safe Regions on Commodity Hardware”. en. In: *Proceedings of the Twelfth European Conference on Computer Systems*. Belgrade Serbia: ACM, Apr. 2017, pp. 437–452. ISBN: 978-1-4503-4938-3. DOI: 10.1145/3064176.3064217. URL: https://dl.acm.org/doi/10.1145/3064176.3064217 (visited on 10/05/2025).

[115] Esmaeil Mohammadian Koruyeh et al. “Spectre Returns! Speculation Attacks Using the Return Stack Buffer”. In: *Proceedings of the 12th USENIX Workshop on Offensive Technologies*. WOOT ’18. Baltimore, MD, USA: USENIX Association, Aug. 2018. URL: https://www.usenix.org/conference/woot18/presentation/koruyeh (visited on 07/26/2025).

[116] Mark Kuhne, Stavros Volos, and Shweta Shinde. *Dorami: Privilege Separating Security Monitor on RISC-V TEEs*. en. arXiv:2410.03653 [cs]. Oct. 2024. DOI: 10.48550/arXiv.2410.03653. URL: http://arxiv.org/abs/2410.03653 (visited on 10/23/2025).

[117] Lukas Lamster et al. “Voodoo: memory tagging, authenticated encryption, and error correction through MAGIC”. In: *Proceedings of the 33rd USENIX Security Symposium*. USENIX Security ’24. Philadelphia, PA, USA: USENIX Association, 2024. ISBN: 978-1-939133-44-1.

[118] Adam Langley. *Ctgrind*. Apr. 2025. URL: https://github.com/agl/ctgrind (visited on 04/06/2025).

[119] Hyun Bin Lee et al. "DOVE: a data-oblivious virtual environment". en. In: *Proceedings of the 2021 network and distributed system security symposium*. Virtual: Internet Society, 2021. ISBN: 978-1-891562-66-2. DOI: 10.14722/ndss.2021.23056. URL: https://www.ndss-symposium.org/wp-content/uploads/ndss2021_7B-1_23056_paper.pdf (visited on 04/20/2022).

[120] Peinan Li et al. "Conditional Speculation: An Effective Approach to Safeguard Out-of-Order Execution Against Spectre Attacks". en. In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Washington, DC, USA: IEEE, Feb. 2019, pp. 264–276. ISBN: 978-1-72811-444-6. DOI: 10.1109/HPCA.2019.00043. URL: https://ieeexplore.ieee.org/document/8675250/ (visited on 06/05/2024).

[121] Hans Liljestrand et al. "{PAC} it up: Towards Pointer Integrity using {ARM} Pointer Authentication". en. In: *Proceedings of the 28th USENIX Security Symposium*. USENIX Security '19. USENIX Association, 2019, pp. 177–194. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/liljestrand.

[122] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space". In: *Proceedings of the 27th USENIX Security Symposium*. USENIX Security '18. Baltimore, MD: USENIX Association, Aug. 2018, p. 19. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp (visited on 03/30/2025).

[123] Moritz Lipp et al. "PLATYPUS: software-based power side-channel attacks on x86". In: *Proceedings of the 2021 IEEE symposium on security and privacy*. 2021, pp. 355–371. DOI: 10.1109/SP40001.2021.00063.

[124] Fangfei Liu et al. "Last-level cache side-channel attacks are practical". In: *2015 IEEE symposium on security and privacy*. 2015, pp. 605–622. DOI: 10.1109/SP.2015.43.

[125] Shiqi Liu et al. *NanoZone: Scalable, Efficient, and Secure Memory Protection for Arm CCA*. en. arXiv:2506.07034 [cs]. June 2025. DOI: 10.48550/arXiv.2506.07034. URL: http://arxiv.org/abs/2506.07034 (visited on 10/23/2025).

[126] J. Longo et al. "SoC It to EM: ElectroMagnetic Side-Channel Attacks on a Complex System-on-Chip". In: *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*. Berlin, Heidelberg, 2015, pp. 620–640. ISBN: 978-3-662-48323-7. DOI: 10.1007/978-3-662-48324-4_31.

[127] Kevin Loughlin et al. "Dolma: Securing Speculation with the Principle of Transient Non-Observability". In: *Proceedings of the 30th USENIX Security Symposium*. USENIX Security '21. USENIX Association, Aug. 2021, pp. 1397–1414. ISBN: 978-1-939133-24-3.

[128] lowRISC. *Ibex Security Features*. Ibex Documentation. (accessed 2025-07-26). July 2025. URL: https://ibex-core.readthedocs.io/en/latest/03_reference/security.html (visited on 07/26/2025).

[129] Giorgi Maisuradze and Christian Rossow. "Ret2spec: Speculative Execution Using Return Stack Buffers". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. New York, NY, USA: Association for Computing Machinery, Oct. 2018. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243761. URL: https://doi.org/10.1145/3243734.3243761 (visited on 07/12/2025).

[130] Andrea Mambretti et al. "Speculator: a tool to analyze speculative execution attacks and mitigations". In: *Computer Security Applications Conference*. ACM, 2019.

[131] Masaryk University Centre for Research on Cryptography and Security. *CT-Tools*. (accessed 2025-04-07). 2025. URL: https://crocs-muni.github.io/ct-tools/ (visited on 04/07/2025).

[132] Ali Jose Mashtizadeh et al. "CCFI: Cryptographically Enforced Control Flow Integrity". en. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. Denver Colorado USA: ACM, Oct. 2015, pp. 941–951. ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813676. URL: https://dl.acm.org/doi/10.1145/2810103.2813676 (visited on 10/23/2025).

[133] Stephen McCamant and Greg Morrisett. "Evaluating SFI for a CISC architecture". In: *Proceedings of the 15th USENIX Security Symposium*. USENIX Security '06. USA: USENIX Association, July 2006. (Visited on 10/22/2025).

[134] Damiano Melotti, Maxime Rossi-Bellom, and Andrea Continella. "Reversing and Fuzzing the Google Titan M Chip". In: *Reversing and Offensive-Oriented Trends Symposium*. New York, NY, USA, 2021, pp. 1–10. ISBN: 978-1-4503-9602-8. DOI: 10.1145/3503921.3503922.

[135] *Memory Protection Keys — The Linux Kernel documentation*. URL: https://docs.kernel.org/core-api/protection-keys.html (visited on 10/23/2025).

[136] Microsoft. *Mitigating speculative execution side channel hardware vulnerabilities*. 2018. URL: https://msrc.microsoft.com/blog/2018/03/mitigating-speculative-execution-side-channel-hardware-vulnerabilities/.

[137] Pratyush Mishra et al. "Oblix: An Efficient Oblivious Search Index". In: *Proceedings of the 39th IEEE Symposium on Security and Privacy*. S&P 18. IEEE, May 2018, pp. 279–296. DOI: 10.1109/SP.2018.00045. URL: https://ieeexplore.ieee.org/document/8418609 (visited on 04/06/2025).

[138] MITRE. *CWE - 2024 CWE Top 25 Most Dangerous Software Weaknesses*. 2024. URL: https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html (visited on 10/23/2025).

[139] David Molnar et al. "The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks". In: *Proceedings of the 8th International Conference on Information Security and Cryptology*. ICISC '05. Berlin, Heidelberg: Springer-Verlag, Dec. 2005, pp. 156–168. ISBN: 978-3-540-33354-8. DOI: 10.1007/11734727_14. URL: https://doi.org/10.1007/11734727_14 (visited on 04/06/2025).

[140] Kit Murdock et al. "Plundervolt: software-based fault injection attacks against intel SGX". In: *Proceedings of the 2020 IEEE symposium on security and privacy*. 2020.

[141] Onur Mutlu and Jeremie S. Kim. "RowHammer: A retrospective". In: *IEEE transactions on computer-aided design of integrated circuits and systems* 39.8 (Aug. 2020). Number of pages: 17 Publisher: IEEE Press tex.issue_date: Aug. 2020, pp. 1555–1571. ISSN: 0278-0070. DOI: 10.1109/TCAD.2019.2915318. URL: https://doi.org/10.1109/TCAD.2019.2915318.

[142] Santosh Nagarakatte et al. "SoftBound: highly compatible and complete spatial memory safety for c". In: *Proceedings of the 30th ACM Conference on Programming Language Design and Implementation*. PLDI '09. Dublin, Ireland: Association for Computing Machinery, 2009, 245–258. ISBN: 9781605583921. DOI: 10.1145/1542476.1542504. URL: https://doi.org/10.1145/1542476.1542504.

[143] Shravan Narayan et al. "Going beyond the Limits of SFI: Flexible and Secure Hardware-Assisted In-Process Isolation with HFI". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2023. Vancouver, BC, Canada: Association for Computing Machinery, 2023, 266–281. ISBN: 9781450399180. DOI: 10.1145/3582016.3582023. URL: https://doi.org/10.1145/3582016.3582023.

[144] Shravan Narayan et al. "Segue & ColorGuard: Optimizing SFI Performance and Scalability on Modern Architectures". en. In: *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Rotterdam Netherlands: ACM, Mar. 2025, pp. 987–1002. ISBN:

979-8-4007-0698-1. DOI: 10.1145/3669940.3707249. URL: https://dl.acm.org/doi/10.1145/3669940.3707249.

[145] Rishiyur S. Nikhil. *Riscy-OOO*. CSAIL CSG. Mar. 2025. URL: https://github.com/csail-csg/riscy-OOO (visited on 04/06/2025).

[146] NVIDIA. *NVIDIA H100 Tensor Core GPU Architecture*. 2023. URL: https://resources.nvidia.com/en-us-tensor-core (visited on 12/20/2023).

[147] Olga Ohrimenko et al. "Oblivious Multi-Party Machine Learning on Trusted Processors". In: *Proceedings of the 25th USENIX Security Symposium*. USENIX Security '16. USA: USENIX Association, Aug. 2016, pp. 619–636. ISBN: 978-1-931971-32-4. (Visited on 04/06/2025).

[148] Oleksii Oleksenko et al. *Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches*. arXiv:1702.00719 [cs]. June 2017. DOI: 10.48550/arXiv.1702.00719. URL: http://arxiv.org/abs/1702.00719 (visited on 07/13/2023).

[149] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: The Case of AES". In: *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*. CT-RSA '06. Berlin, Heidelberg: Springer-Verlag, Feb. 2006, pp. 1–20. ISBN: 978-3-540-31033-4. DOI: 10.1007/11605805_1. URL: https://doi.org/10.1007/11605805_1 (visited on 03/30/2025).

[150] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. "Lord of the ring(s): side channel attacks on the CPU on-chip ring interconnect are practical". In: *Proceedings of the 30th USENIX Security Symposium*. USENIX Security '21. USENIX Association, 2021, pp. 645–662. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/paccagnella.

[151] Daniel Page. *Partitioned cache architecture as a side-channel defence mechanism*. tex.howpublished: Cryptology ePrint Archive, Report 2005/280. 2005. URL: https://ia.cr/2005/280.

[152] Soyeon Park et al. *libmpk: Software Abstraction for Intel Memory Protection Keys*. en. arXiv:1811.07276 [cs]. Nov. 2018. DOI: 10.48550/arXiv.1811.07276. URL: http://arxiv.org/abs/1811.07276 (visited on 09/14/2025).

[153] Taemin Park et al. "NoJITsu: Locking Down JavaScript Engines". en. In: *Proceedings 2020 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2020. ISBN: 978-1-891562-61-7. DOI: 10.14722/ndss.2020.24262. URL: https://www.ndss-symposium.org/wp-content/uploads/2020/02/24262.pdf (visited on 05/18/2025).

[154] Sandro Pinto and Nuno Santos. "Demystifying Arm TrustZone: A comprehensive survey". In: *ACM computing surveys* 51.6 (Jan. 2019), pp. 1–36. ISSN: 0360-0300. DOI: 10.1145/3291047. URL: https://doi.org/10.1145/3291047.

[155] Louis-Emile Ploix et al. *Comprehensive Formal Verification of Observational Correctness for the CHERIoT-Ibex Processor*. Feb. 2025. DOI: 10.48550/arXiv.2502.04738. arXiv: 2502.04738 [cs]. URL: http://arxiv.org/abs/2502.04738 (visited on 04/09/2025).

[156] Michal Podhradsky, Ramy Tadros, and Austin Roach. *BESSPIN-GFE*. Galois, Inc. Oct. 2022. URL: https://github.com/GaloisInc/BESSPIN-GFE (visited on 07/03/2024).

[157] Thomas Pornin. *BearSSL - Constant-Time Mul*. (accessed 2025-04-04). 2018. URL: https://bearssl.org/ctmul.html (visited on 04/04/2025).

[158] Thomas Pornin. *Constant-Time Code: The Pessimist Case*. 2025. URL: https://eprint.iacr.org/2025/435 (visited on 04/07/2025).

[159] *QEMU*. URL: https://www.qemu.org/ (visited on 10/23/2025).

[160] Martin Radetzki et al. "Methods for fault tolerance in networks-on-chip". In: *ACM Comput. Surv.* 46.1 (July 2013), 8:1–8:38. ISSN: 0360-0300. DOI: 10.1145/2522968.2522976. URL: https://dl.acm.org/doi/10.1145/2522968.2522976 (visited on 10/23/2025).

[161] Ashay Rane, Calvin Lin, and Mohit Tiwari. "Raccoon: Closing Digital Side-Channels through Obfuscated Execution". In: *Proceedings of the 24th USENIX Security Symposium*. USENIX Security '15. USA: USENIX Association, Aug. 2015, pp. 431–446. ISBN: 978-1-931971-23-2. (Visited on 04/06/2025).

[162] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. "Dude, Is My Code Constant Time?" In: *Proceedings of the Conference on Design, Automation & Test in Europe*. DATE '17. Leuven, BEL: European Design and Automation Association, Mar. 2017, pp. 1701–1706. (Visited on 04/06/2025).

[163] RISC-V International. *Spike RISC-V ISA Simulator*. https://github.com/riscv-software-src/riscv-isa-sim. RISC-V Software, 2022.

[164] Gururaj Saileshwar and Moinuddin K. Qureshi. "CleanupSpec: An "Undo" Approach to Safe Speculation". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '19. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 73–86. ISBN: 978-1-4503-6938-1. DOI: 10.1145/3352460.3358314. URL: https://doi.org/10.1145/3352460.3358314 (visited on 07/29/2025).

[165] Christos Sakalis et al. "Efficient Invisible Speculative Execution through Selective Delay and Value Prediction". In: *Proceedings of the 46th International Symposium on Computer Architecture*. ISCA '19. New York, NY, USA: Association for Computing Machinery, June 2019, pp. 723–735. ISBN: 978-1-4503-6669-4. DOI: 10.1145/3307650.3322216. URL: https://doi.org/10.1145/3307650.3322216 (visited on 07/29/2025).

[166] Daniel Sanchez and Christos Kozyrakis. "The ZCache: decoupling ways and associativity". In: *Proceedings of the 43rd IEEE/ACM international symposium on microarchitecture*. MICRO '43. Number of pages: 12. USA: IEEE Computer Society, 2010, pp. 187–198. ISBN: 978-0-7695-4299-7. DOI: 10.1109/MICRO.2010.20. URL: https://doi.org/10.1109/MICRO.2010.20.

[167] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. "ZeroTrace : Oblivious Memory Primitives from Intel SGX". In: *Proceedings 2018 Network and Distributed System Security Symposium*. NDSS '18. San Diego, CA: Internet Society, 2018. ISBN: 978-1-891562-49-5. DOI: 10.14722/ndss.2018.23239. URL: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_02B-4_Sasy_paper.pdf (visited on 04/06/2025).

[168] Moritz Schneider et al. *Breaking Bad: How Compilers Break Constant-Time~Implementations*. Oct. 2024. DOI: 10.48550/arXiv.2410.13489. arXiv: 2410.13489 [cs]. URL: http://arxiv.org/abs/2410.13489 (visited on 03/30/2025).

[169] David Schrammel et al. "Donky: domain keys – efficient in-process isolation for RISC-V and x86". In: *Proceedings of the 29th USENIX Security Symposium*. USENIX Security '20. USA: USENIX Association, 2020. ISBN: 978-1-939133-17-5.

[170] Michael Schwarz et al. "ConTExT: A Generic Approach for Mitigating Spectre". In: *Proceedings 2020 Network and Distributed System Security Symposium*. NDSS '20. San Diego, CA: Internet Society, 2020. DOI: 10.14722/ndss.2020.24271. URL: https://www.ndss-symposium.org/wp-content/uploads/2020/02/24271.pdf (visited on 07/26/2025).

[171] Konstantin Serebryany et al. "AddressSanitizer: a fast address sanity checker". In: *Proceedings of the USENIX Annual Technical Conference*. USENIX ATC'12. USA: USENIX Association, June 2012, p. 28. (Visited on 07/12/2023).

[172] Hovav Shacham. "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)". In: *Proceedings of the 14th ACM conference on Computer and communications security*. CCS '07. New York, NY, USA: Association for Computing Machinery, 2007, pp. 552–561. ISBN: 978-1-59593-703-2. DOI: 10.1145/1315245.1315313. URL: https://doi.org/10.1145/1315245.1315313.

[173] *ShadowCallStack — Clang 22.0.0git documentation*. URL: https://clang.llvm.org/docs/ShadowCallStack.html (visited on 10/23/2025).

[174] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. "Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity". In: *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '19. New York, NY, USA: Association for Computing Machinery, June 2019, pp. 1–11. ISBN: 978-1-4503-7226-8. DOI: 10.1145/3337167.3337175. URL: https://dl.acm.org/doi/10.1145/3337167.3337175 (visited on 10/22/2025).

[175] Fahad Shaon et al. "SGX-BigMatrix: A Practical Encrypted Data Analytic Framework With Trusted Processors". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 1211–1228. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134095. URL: https://dl.acm.org/doi/10.1145/3133956.3134095 (visited on 04/06/2025).

[176] SiFive. *SiFive TileLink Specification*. 2017. URL: https://static.dev.sifive.com/docs/tilelink/tilelink-spec-1.7-draft.pdf.

[177] Standard Performance Evaluation Corporation. *SPEC CPU 2017 benchmark*. 2017. URL: https://www.spec.org/cpu2017/.

[178] Emil Stefanov et al. "Path ORAM: an extremely simple oblivious RAM protocol". In: *Journal of the ACM* 65.4 (Apr. 2018). Number of pages: 26 Place: New York, NY, USA Publisher: Association for Computing Machinery tex.articleno: 18 tex.issue_date: August 2018, pp. 1–26. ISSN: 0004-5411. DOI: 10.1145/3177872. URL: https://doi.org/10.1145/3177872.

[179] G. Edward Suh et al. "Secure program execution via dynamic information flow tracking". In: *Proceedings of the 11th international conference on architectural support for programming languages and operating systems*. ASPLOS XI. Number of pages: 12 Place: Boston, MA, USA. New York, NY, USA: Association for Computing Machinery, 2004, pp. 85–96. ISBN: 1-58113-804-0. DOI: 10.1145/1024393.1024404. URL: https://doi.org/10.1145/1024393.1024404.

[180] Yulei Sui and Jingling Xue. "SVF: Interprocedural Static Value-Flow Analysis in LLVM". In: *Proceedings of the 25th International Conference on Compiler Construction*. New York, NY, USA, 2016, pp. 265–266. DOI: 10.1145/2892208.2892235. URL: https://doi.org/10.1145/2892208.2892235.

[181] Michael B. Sullivan et al. “Implicit Memory Tagging: No-Overhead Memory Safety Using Alias-Free Tagged ECC”. In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. ISCA ’23. New York, NY, USA: Association for Computing Machinery, June 2023, pp. 1–13. ISBN: 9798400700958. DOI: 10.1145/3579371.3589102. URL: https://dl.acm.org/doi/10.1145/3579371.3589102 (visited on 09/07/2023).

[182] Nikhil Swamy et al. “Dependent Types and Multi-Monadic Effects in F*”. In: *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA, 2016, pp. 256–270. ISBN: 978-1-4503-3549-2. DOI: 10.1145/2837614.2837655.

[183] Nikhil Swamy et al. “Dependent Types and Multi-Monadic Effects in F*”. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’16. New York, NY, USA: Association for Computing Machinery, Jan. 2016, pp. 256–270. ISBN: 978-1-4503-3549-2. DOI: 10.1145/2837614.2837655. URL: https://doi.org/10.1145/2837614.2837655 (visited on 07/26/2025).

[184] Vivienne Sze et al. “Efficient Processing of Deep Neural Networks: A Tutorial and Survey”. In: *Proceedings of the IEEE* 105 (2017), pp. 2295–2329. URL: https://api.semanticscholar.org/CorpusID:3273340.

[185] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. “CLKSCREW: exposing the perils of security-oblivious energy management”. In: *Proceedings of the 26th USENIX Security Symposium*. USENIX Security ’17. Number of pages: 18 Place: Vancouver, BC, Canada. USA: USENIX Association, 2017, pp. 1057–1074. ISBN: 978-1-931971-40-9.

[186] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. “Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization”. In: *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’19. New York, NY, USA: Association for Computing Machinery, Apr. 2019, pp. 395–410. ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304060. URL: https://dl.acm.org/doi/10.1145/3297858.3304060 (visited on 07/29/2025).

[187] Mohammadkazem Taram et al. “SecSMT: Securing SMT Processors against Contention-Based Covert Channels”. In: *Proceedings of the 31st USENIX Security Symposium*. USENIX Security ’22. USENIX Association, 2022.

[188] Mohit Tiwari et al. “A small cache of large ranges: Hardware methods for efficiently searching, storing, and updating big dataflow tags”. In: *Proceedings of the 41st IEEE/ACM international symposium on microarchitecture*. ISSN: 2379-3155. 2008, pp. 94–105. DOI: 10.1109/MICRO.2008.4771782.

[189] Mohit Tiwari et al. “Complete Information Flow Tracking from the Gates Up”. In: *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA, 2009, pp. 109–120. ISBN: 978-1-60558-406-5. DOI: 10.1145/1508244.1508258.

[190] Shruti Tople et al. *PermuteRam: optimizing oblivious computation for efficiency*. tex.howpublished: Cryptology ePrint Archive, Report 2017/885. 2017. URL: https://ia.cr/2017/885.

[191] Florian Tramèr and Dan Boneh. *Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware*. 2019. arXiv: 1806.03287 [stat.ML]. URL: https://arxiv.org/abs/1806.03287.

[192] Martin Unterguggenberger et al. “Cryptographic Least Privilege Enforcement for Scalable Memory Isolation”. In: *IEEE International Symposium on Hardware Oriented Security and Trust 2025: HOST 2025*. 2025.

[193] Martin Unterguggenberger et al. “TME-Box: Scalable In-Process Isolation through Intel TME-MK Memory Encryption”. en. In: *Proceedings 2025 Network and Distributed System Security Symposium*. San Diego, CA, USA: Internet Society, 2025. ISBN: 979-8-9894372-8-3. DOI: 10.14722/ndss.2025.240277. URL: https://www.ndss-symposium.org/wp-content/uploads/2025-277-paper.pdf (visited on 10/23/2025).

[194] Anjo Vahldiek-Oberwagner et al. “ERIM: secure, efficient in-process isolation with protection keys (MPK)”. In: *Proceedings of the 28th USENIX Security Symposium*. USENIX Security ’19. Santa Clara, CA, USA: USENIX Association, 2019, 1221–1238. ISBN: 9781939133069.

[195] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. “SoK: Fully Homomorphic Encryption Compilers”. In: *Proceedings of the IEEE Symposium on Security and Privacy*. 2021, pp. 1092–1108. DOI: 10.1109/sp40001.2021.00068.

[196] Robert Wahbe et al. “Efficient software-based fault isolation”. In: *Proceedings of the 14th ACM Symposium on Operating Systems Principles*. SOSP ’93. Asheville, North Carolina, USA: Association for Computing Machinery, 1993, 203–216. ISBN: 0897916328. DOI: 10.1145/168619.168635. URL: https://doi.org/10.1145/168619.168635.

[197] Zhenghong Wang and Ruby B. Lee. “A novel cache architecture with enhanced performance and security”. In: *Proceedings of the 41st IEEE/ACM international symposium on microarchitecture*. MICRO 41. Number of pages: 11. USA: IEEE Computer Society, 2008, pp. 83–93. ISBN: 978-1-4244-2836-6. DOI: 10.1109/MICRO.2008.4771781. URL: https://doi.org/10.1109/MICRO.2008.4771781.

[198] Zhenghong Wang and Ruby B. Lee. “New cache designs for thwarting software cache-based side channel attacks”. In: *Proceedings of the 34th international symposium on computer architecture*. ISCA ’07. Number of pages: 12 Place: San Diego, California, USA. New York, NY, USA: Association for Computing Machinery, 2007, pp. 494–505. ISBN: 978-1-59593-706-3. DOI: 10.1145/1250662.1250723. URL: https://doi.org/10.1145/1250662.1250723.

[199] Robert N M Watson et al. *An Introduction to CHERI*. Technical Report UCAM-CL-TR-941. 15 JJ Thomson Avenue Cambridge CB3 0FD United Kingdom: Computer Laboratory, University of Cambridge, Sept. 2019, 43 pages. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf (visited on 11/03/2023).

[200] Robert N. M. Watson et al. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9)*. Technical Report UCAM-CL-TR-987. 15 JJ Thomson Avenue Cambridge CB3 0FD United Kingdom: Computer Laboratory, University of Cambridge, Sept. 2023, 523 pages. DOI: 10.48456/TR-987. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-987.html (visited on 11/03/2023).

[201] Robert N M Watson et al. *CHERI C/C++ Programming Guide*. Technical Report UCAM-CL-TR-947. 15 JJ Thomson Avenue Cambridge CB3 0FD United Kingdom: Computer Laboratory, University of Cambridge, June 2020, p. 33. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-947.pdf.

[202] Martin Weidmann. *Arm A-Profile Architecture Developments 2022 - Architectures and Processors blog - Arm Community blogs - Arm Community*. en. Sept. 2022. URL: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/arm-a-profile-architecture-2022 (visited on 10/23/2025).

[203] Samuel Weiser et al. “DATA–differential Address Trace Analysis: Finding Address-Based Side-Channels in Binaries”. In: *Proceedings of the 27th USENIX Security Symposium*. USENIX Security ’18. USA: USENIX Association, Aug. 2018, pp. 603–620. ISBN: 978-1-931971-46-1. (Visited on 04/06/2025).

[204] Nathaniel Wesley Filardo et al. "Cornucopia: Temporal Safety for CHERI Heaps". In: *Proceedings of the 41st IEEE Symposium on Security and Privacy*. S&P '20. IEEE, May 2020, pp. 608–625. DOI: 10.1109/SP40000.2020.00098. URL: https://ieeexplore.ieee.org/document/9152640 (visited on 10/31/2023).

[205] *Whole program LLVM*. URL: https://github.com/travitch/whole-program-llvm (visited on 05/13/2024).

[206] Jan Wichelmann et al. "MicroWalk: A Framework for Finding Side Channels in Binaries". In: *Proceedings of the 34th Annual Computer Security Applications Conference*. ACSAC '18. New York, NY, USA: Association for Computing Machinery, Dec. 2018, pp. 161–173. ISBN: 978-1-4503-6569-7. DOI: 10.1145/3274694.3274741. URL: https://doi.org/10.1145/3274694.3274741 (visited on 04/06/2025).

[207] Jan Wichelmann et al. "Microwalk-CI: Practical Side-Channel Analysis for JavaScript Applications". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS '22. New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 2915–2929. ISBN: 978-1-4503-9450-5. DOI: 10.1145/3548606.3560654. URL: https://doi.org/10.1145/3548606.3560654 (visited on 04/06/2025).

[208] Sander Wiebing et al. "InSpectre Gadget: Inspecting the Residual Attack Surface of Cross-privilege Spectre v2". In: *Proceedings of the 33th USENIX Security Symposium*. USENIX Security '24. USENIX Association, 2024.

[209] Hans Winderix et al. "Architectural Mimicry: Innovative Instructions to Efficiently Address Control-Flow Leakage in Data-Oblivious Programs". In: *Proceedings of the 45th IEEE Symposium on Security and Privacy*. S&P '24. San Francisco, CA, USA: IEEE, May 2024, pp. 3697–3715. ISBN: 979-8-3503-3130-1. DOI: 10.1109/SP54263.2024.00047. URL: https://ieeexplore.ieee.org/document/10646702/ (visited on 07/29/2025).

[210] Hans Winderix et al. "Libra: Architectural Support For Principled, Secure And Efficient Balanced Execution On High-End Processors". In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. CCS '24. New York, NY, USA: Association for Computing Machinery, Dec. 2024, pp. 19–33. ISBN: 979-8-4007-0636-3. DOI: 10.1145/3658644.3690319. URL: https://doi.org/10.1145/3658644.3690319 (visited on 07/25/2025).

[211] Jonathan Woodruff et al. "CHERI Concentrate: Practical Compressed Capabilities". In: *IEEE Transactions on Computers* 68.10 (Oct. 2019), pp. 1455–1469. ISSN: 1557-9956. DOI: 10.1109/TC.2019.2914037. URL: https://ieeexplore.ieee.org/document/8703061 (visited on 10/31/2023).

[212] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. "SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities". In: *Network and Distributed System Security Symposium*. The Internet Society, 2020.

[213] Jiali Xu et al. "PANIC: PAN-assisted Intra-process Memory Isolation on ARM". en. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. Copenhagen Denmark: ACM, Nov. 2023, pp. 919–933. ISBN: 979-8-4007-0050-7. DOI: 10.1145/3576915.3623206. URL: https://dl.acm.org/doi/10.1145/3576915.3623206 (visited on 10/23/2025).

[214] Jun Xu and Nithin Nakka. "Defeating memory corruption attacks via pointer taintedness detection". In: *Proceedings of the 2005 international conference on dependable systems and networks*. DSN '05. Number of pages: 10. USA: IEEE Computer Society, 2005, pp. 378–387. ISBN: 0-7695-2282-3. DOI: 10.1109/DSN.2005.36. URL: https://doi.org/10.1109/DSN.2005.36.

[215] Jiaqin Yan et al. "EKC: a portable and extensible kernel compartment for deprivileging commodity OS". In: *Proceedings of the 34th USENIX Security Symposium*. USENIX Security '25. Seattle, WA, USA: USENIX Association, 2025. ISBN: 978-1-939133-52-6.

[216] Mengjia Yan et al. "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy". In: *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '18. IEEE, Oct. 2018, pp. 428–441. DOI: 10.1109/MICRO.2018.00042. URL: https://ieeexplore.ieee.org/document/8574559 (visited on 07/11/2025).

[217] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *Proceedings of the 23rd USENIX Security Symposium*. USENIX Security '24. USA: USENIX Association, Aug. 2014, pp. 719–732. ISBN: 978-1-931971-15-7. (Visited on 03/30/2025).

[218] Jiyong Yu et al. "Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing". In: *Proceedings 2019 Network and Distributed System Security Symposium*. NDSS '19. San Diego, CA: Internet Society, 2019. ISBN: 978-1-891562-55-6. DOI: 10.14722/ndss.2019.23061. URL: https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_05B-4_Yu_paper.pdf (visited on 01/18/2025).

[219] Jiyong Yu et al. "Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. New York, NY, USA:

Association for Computing Machinery, Oct. 2019, pp. 954–968. ISBN: 978-1-4503-6938-1. DOI: 10.1145/3352460.3358274. URL: https://dl.acm.org/doi/10.1145/3352460.3358274 (visited on 07/11/2025).

[220] Ziqi Yuan et al. "LightZone: Lightweight Hardware-Assisted In-Process Isolation for ARM64". In: *Proceedings of the 25th International Middleware Conference*. Middleware '24. New York, NY, USA: Association for Computing Machinery, Dec. 2024, pp. 467–480. ISBN: 979-8-4007-0623-3. DOI: 10.1145/3652892.3700786. URL: https://doi.org/10.1145/3652892.3700786 (visited on 09/06/2025).

[221] Sizhuo Zhang. *Toooba*. Bluespec, Inc. Mar. 2025. URL: https://github.com/bluespec/Toooba (visited on 04/06/2025).

[222] Jerry Zhao et al. "SonicBOOM: The 3rd Generation Berkeley out-of-Order Machine". In: *Proceedings of the Workshop on Computer Architecture Research with RISC-V* (2020). URL: https://carrv.github.io/2020/papers/CARRV2020_paper_15_Zhao.pdf.

[223] Lianying Zhao et al. "A Survey of Hardware Improvements to Secure Program Execution". en. In: *ACM Computing Surveys* (June 2024). ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3672392. URL: https://dl.acm.org/doi/10.1145/3672392 (visited on 06/27/2024).

[224] Zirui Neil Zhao et al. "Binoculars: Contention-Based Side-Channel Attacks Exploiting the Page Walker". In: *Proceedings of the 31st USENIX Security Symposium*. USENIX Security '22. USENIX Association, 2022.

[225] Zirui Neil Zhao et al. "Speculation Invariance (InvarSpec): Faster Safe Execution Through Program Analysis". In: *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '20. IEEE, Oct. 2020, pp. 1138–1152. DOI: 10.1109/MICRO50266.2020.00094. URL: https://ieeexplore.ieee.org/document/9251941 (visited on 07/11/2025).

[226] Wenting Zheng et al. "Opaque: An Oblivious and Encrypted Distributed Analytics Platform". In: *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*. NSDI'17. USA: USENIX Association, Mar. 2017, pp. 283–298. ISBN: 978-1-931971-37-9. (Visited on 04/06/2025).

[227] Yajin Zhou et al. "ARMlock: Hardware-based Fault Isolation for ARM". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. New York, NY, USA: Association for Computing Machinery, Nov. 2014, pp. 558–569. ISBN: 978-1-4503-2957-6. DOI: 10.1145/2660267.2660344. URL: https://doi.org/10.1145/2660267.2660344 (visited on 10/22/2025).

[228] Jean-Karim Zinzindohoué et al. "HACL*: A Verified Modern Cryptographic Library". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 1789–1806. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134043. URL: https://dl.acm.org/doi/10.1145/3133956.3134043 (visited on 07/26/2025).