Enhancing Side-channel Security:

Detection, Mitigation and Verification

Ke Jiang

College of Computing and Data Science

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

2024

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

06/10/2024

Date

Ke Jiang

Ke Jiang

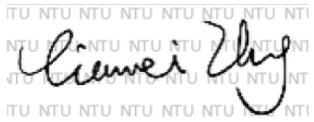
Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

06/10/2024

.....

Date



.....

Assoc. Prof. Tianwei Zhang

Authorship Attribution Statement

This thesis contains material from 2 papers accepted at conferences and 1 paper being reviewed by the conference in which I am listed as an author.

Chapter 2 is published as Ke Jiang, Yuyan Bao, Shuai Wang, Zhibo Liu, and Tianwei Zhang. 2022. Cache Refinement Type for Side-Channel Detection of Cryptographic Software. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22). Los Angeles, L.A., USA, 1583–1597. https://doi.org/10.1145/3548606.3560672

The contributions of the co-authors are as follows:

- I proposed the idea, developed the system and conducted all experiments.
- All data, including locating the vulnerabilities in cryptography libraries was analyzed by me.
- I prepared the manuscript drafts. The manuscript was revised by Asst/Prof Yuyan Bao, Assoc/Prof Shuai Wang and Assoc/Prof Tianwei Zhang.
- Asst/Prof Yuyan Bao assisted in the collation of the refinement type rules.
- Assoc/Prof Shuai Wang assisted with the rebuttal period and polished the revision.
- Dr. Zhibo Liu provided technical support on disassembly.
- Assoc/Prof Tianwei Zhang put forward constructive suggestions throughout the project.

Chapter 3 is being reviewed as Ke Jiang, Sen Deng, Yinshuai Li, Shuai Wang, Tianwei Zhang, and Yinqian Zhang. 2024. CipherGuard: Compiler-aided Mitigation against Ciphertext Side-channel Attacks.

The contributions of the co-authors are as follows:

- I proposed the idea, developed the system and conducted all experiments.
- I analyzed the data and wrote the draft of the manuscript. The draft was revised by Assoc/Prof Shuai Wang, Assoc/Prof Tianwei Zhang and Prof Yinqian Zhang.
- Sen Deng participated in the discussion of important project nodes and made valuable comments on the manuscript.

- Yinshuwai Li provided technical support on program debugging.
- Assoc/Prof Tianwei Zhang and Prof Yinqian Zhang guided the direction of the project.
- Assoc/Prof Shuai Wang put forward constructive suggestions throughout the project.

Chapter 4 is published as Ke Jiang, Tianwei Zhang, David Sanán, Yongwang Zhao, and Yang Liu. 2022. A Formal Methodology for Verifying Side-channel Vulnerabilities in Cache Architectures. In Proceedings of the 23rd International Conference on Formal Engineering Methods (ICFEM' 22), Madrid, Spain, 190-208. https://doi.org/10.1007/978-3-031-17244-1_12

The contributions of the co-authors are as follows:

- I proposed the idea, developed the verification framework and conducted all proofs.
- I analyzed the proof results, specifically connecting the results to all cache designs. I prepared the manuscript.
- The manuscript was revised by Assoc/Prof Tianwei Zhang and Asst/Prof David Sanán.
- Assoc/Prof Tianwei Zhang and Prof Yongwang Zhao participated in the discussion of important project nodes.
- Asst/Prof David Sanán provided technical support on formal verification.
- Prof Yang Liu put forward constructive suggestions throughout the project.

06/10/2024

Date

Ke Jiang

Acknowledgements

It is time to bid farewell to the Ph.D. journey, an unforgettable experience that has imparted invaluable lessons and fostered profound personal growth. The challenges and triumphs encountered along the way have not only deepened my knowledge and honed my research skills but also strengthened my resilience and broadened my perspective.

I would like to express my gratitude to my supervisor, Associate Professor Tianwei Zhang, for providing me the opportunity to pursue a Ph.D., as well as for offering valuable guidance in establishing the overall research direction. His mentorship fostered my capacity for independent scientific research and cultivated a nuanced approach to critical thinking. By granting me the freedom to explore and discover my research interests, he encouraged a spirit of inquiry and innovation, while consistently providing clear guidance and substantial help whenever needed. His exemplary conduct elucidated the qualities of a seasoned scientist, shaping my understanding of the caliber and demeanor expected in the pursuit of scholarly endeavors. His strict requirements and high standards challenged me to strive for excellence and significantly enhanced the quality of my work. For all these reasons, I am deeply thankful for his support and mentorship throughout my Ph.D. journey.

I express profound gratitude to Professor Yang Liu and Professor Yongwang Zhao for ushering me into the realm of academic pursuit during a transformative trip to Singapore. This marked the inception of my doctoral journey, altering the trajectory of my life significantly. Witnessing their dedication and exploration at the forefront of scientific frontiers served as a compelling inspiration. Their commitment to conducting research with calmness and rigor became a guiding beacon for me.

I would like to acknowledge Professor Yinqian Zhang, Associate Professor Shuai Wang, Assistant Professor David Sanán, and Assistant Professor Yuyan Bao, for

their collaborative efforts, which have contributed to the successful completion of various aspects of my doctoral journey. Their dedication and expertise have been indispensable, and I am grateful for the shared intellectual journey.

A heartfelt thank you to my friends, Kangjie Chen, Shuxin Li, Xiaoxuan Lou, Tianlin Li, Chengwei Liu, Meng Hao, Hanxiao Chen, Anran Li, Wenjun Long, Dr. Wenhao Fu, Gelei Deng, Dr. Jianwen Sun, Dr. Hongxu Chen, Dr. Fei Yang, Dr. Feng Zhang, Jinyi Xian and others who have provided emotional support, encouragement, and moments of respite during the challenging phases of this journey.

To my wife, my parents, and other family members, I am deeply indebted for their unwavering love, encouragement, and understanding throughout this journey. Their belief in my abilities have been a constant source of motivation. Their collective support has not only fortified my academic endeavors but has also been the bedrock of my personal growth. I am profoundly grateful for the invaluable contributions of my family, without whom this journey would not have been possible. Lastly, I am overjoyed to share that my wife has bestowed upon me the most precious gift of all, our daughter. Her arrival has filled our lives with immeasurable happiness and has added a new, profound dimension to our journey as a family.

To my dear family

Abstract

In the realm of computer security, side-channel attacks pose a significant threat by exploiting subtle information leaks during the operation of a system. These attacks, which include cache side-channel attacks, a particularly insidious subset, enable adversaries to extract sensitive data by exploiting cache memory behaviors. To defeat cache side-channel attacks, current research has made strides in developing software and hardware-level countermeasures, but challenges persist in achieving comprehensive and scalable methods. Furthermore, constant-time coding practices, effective against traditional cache-based side channels, face limitations with emerging threats like ciphertext side channels. Additionally, new micro-architectural designs, e.g., cache designs, lack formal guarantees of effectiveness, highlighting the need for ongoing evaluation and adaptability. This thesis addresses these challenges, with the aim of improving cache side-channel defense methods, addressing emerging ciphertext side channels, and improving the security of novel hardware designs through formal verification.

To be more specific, this thesis focuses on three primary objectives. First, it aims to advance cache side-channel detection by developing comprehensive methods to identify and mitigate vulnerabilities in cryptography software. Second, it addresses the emerging threat of ciphertext side channels by proposing a holistic approach that combines constant-time coding practices with compiler-aided mitigation mechanisms. Finally, the research seeks to enhance the security of new cache designs by providing a means for formal verification, acknowledging the challenges posed by their complexity and potential vulnerabilities. Through these endeavors, this thesis contributes to fortifying the security of modern computing systems against the evolving and increasingly sophisticated landscape of side-channel threats.

We present three significant contributions addressing side-channel vulnerabilities. Firstly, we introduce CaType, a groundbreaking refinement type-based tool for cache side-channel detection in cryptographic software. CaType analyzes x86

assembly code using refinement types, offering enhancements like bit-level granularity tracking, precise type inferences, and high scalability. It is the first static analyzer for cryptography libraries considering blinding-based defenses and utilizes cache layouts to suppress false positives, demonstrating superior performance in identifying vulnerabilities. Secondly, we propose CipherGuard, a compiler-aided mitigation tool addressing ciphertext side channels by recognizing secret-dependent store instructions and providing multiple strategies to protect these instructions. For the first time, CipherGuard employs various precise and flexible mitigation variants based on the LLVM ecosystem, achieving security without hardware modification and displaying enhanced efficiency in cryptography implementations compared to existing defense tools. Lastly, we introduce a formal methodology for security verification of cache architectures, employing an entropy-based noninterference reasoning framework to assess information leakage. The methodology, applied to eight cache architectures, demonstrates reliability and flexibility, contributing to the advancement of cache side-channel security.

Contents

List of Figures

List of Tables

Symbols and Acronyms

Symbols

H	the high security sensitivity of data
L	the low security sensitivity of data
T	the basic type
P	the predicate associated with the basic type
$\sharp$	the concatenation operation of two expressions
$[n_1 : n_2]/\cdot$	the extraction operation of the designated position of a bitvector expression
$\bowtie$	the logic operations
$\mathrm{Vec}\langle n\rangle$	the bitvector of n bits
$\sqcup$	the operation of taking the least upper bound of two types
$\|\|x\|\|_t$	the operation of inferring a bitvector's type from the types of its constituent bits based on a notion of structural priority
K	the abbreviation of SDD type
I	the abbreviation of SID type
W	the abbreviation of WRA type
U	the abbreviation of URA type
$\{\cdot\}^n$	the n-time repetitions of a bit
$\mathcal{X}$	the probability distribution of input
$\mathcal{Y}$	the probability distribution of output
$\mathbb{P}(\cdot \times \cdot)$	the power-set of two types
$\mathcal{I}$	the input content
$\mathcal{O}$	the output content
$\mathcal{P}$	the probability in real type
ψ	the event-state transition function

ϖ	the output function
$\mathcal{S}$	the state space
$\mathcal{E}$	the set of event labels
$\mathcal{M}$	the abstract state machine
Cpt	the conditional probability transition function
$\mathcal{W}$	the conditional probability matrix
$\mathcal{J}$	the joint distribution

Acronyms

IoT	Internet of Things
TEE	Trusted Execution Environment
TLB	Translation Lookaside Buffer
DRAM	Dynamic Random Access Memory
PTE	Page Table Entry
SPA	Simple Power Analysis
DPA	Differential Power Analysis
DES	Data Encryption Standard
SEMA	Simple Electromagnetic Analysis
DEMA	Differential Electromagnetic Analysis
LLC	Last Level Cache
CAT	Cache Allocation Technology
VM	Virtual Machine
ORAM	Oblivious Random Access Memory
TSX	Transactional Synchronization Extensions
SSA	Static Single Assignment
SDBC	Secret-Dependent Branch Condition
SDMA	Secret-Dependent Memory Access
SAT	Satisfiability
UNSAT	Unsatisfiability
SDD	Secret-Dependent Distribution
URA	Uniformly Random Distribution
SID	Secret-Independent Distribution
WRA	Weakly Random Distribution

CST	Constant
MA	Memory Access
BC	Branch Condition
TP	True Positive
FP	False Positive
TN	True Negative
SEV	Secure Encryption Virtualization
ES	Encrypted State
SNP	Secure Nested Paging
SGX	Software Guard Extension
TDX	Trust Domain Extension
CCA	Confidential Compute Architecture
NPT	Nested Page Table
ASID	Address Space identifier
DFSan	Data Flow Sanitizer
XEX	XOR-Encrypt-XOR
MBB	Machine Basic Block
SIMD	Single Instruction Multiple Data
CFG	Control Flow Graph
ORAM	Oblivious RAM
ASLR	Address Space Layout Randomization
HDL	Hardware Description Language
MMU	Memory Management Unit
VA	Virtual Address
PA	Physical Address
SA	Set Associative
RP	Random Permutation
RF	Random Fill
PL	Partition Locked
NO	No Observation
CO	Constant Observation

Chapter 1

Introduction

1.1 Side-channel Attacks

Side-channel attacks present a serious risk to computer security, leveraging subtle information leaks that occur during routine system operations. By exploiting unintended side effects such as power usage, electromagnetic emissions, and variations in execution times, malicious actors can deduce critical data such as encryption keys, passwords, and other sensitive information. These attacks are difficult to detect because they operate outside traditional intrusion methods, making them particularly stealthy. Currently, side-channel attacks have proven effective against both software and hardware, affecting a wide array of devices, from smartphones and Internet of Things (IoT) devices to highly secure servers.

Side-channel attacks encompass micro-architectural attacks, power attacks, electromagnetic attacks, fault attacks, and transient execution attacks. Notably, the attack and defense research on micro-architectural side-channels has advanced considerably in recent years. Moreover, with the rise of trusted computing and the application of Trusted Execution Environments (TEEs), research targeting side-channel attacks on TEEs has also emerged. Below, we offer a concise overview of side-channel attacks in different contexts.

1.1.1 Micro-architectural Side-channels

Micro-architectural side-channel attacks represent a significant and particularly insidious subset of side-channel attacks that exploit the behavior of a computer's hardware optimizations [1].

Conventional attacks. Initially, timing side-channel attacks were first utilized to compromise cryptography software by observing the execution time of operations involving private keys [2]. Over time, attention shifted to cache side-channel attacks, where attackers meticulously measure the access time to specific cache locations to deduce patterns and infer activities within the targeted system [3–8]. These attacks can expose sensitive details such as secret-dependent control-flow patterns and memory access patterns [9–14]. Cache side-channel attacks are noted for their ability to operate stealthily at a low level, posing significant challenges for detection and defense.

Subsequently, more optimizations in modern processors and operating systems have been exploited to obtain side-channel information. For instance, the branch predictor unit, which forecasts branch outcomes and pre-processes subsequent code, does not clear its values during context switches, potentially exposing sensitive data [15–17]. Another critical optimization unit is the Translation Lookaside Buffer (TLB), which stores mappings from virtual addresses to physical addresses. Attackers can exploit the TLB to ascertain if a specific page is recently accessed, thus inferring sensitive data [18].

Furthermore, additional optimizations including the memory management unit [19, 20], floating-point units [21], CPU ring interconnect [22], CPU ports [23, 24], and the random number generator [25] introduce variations in the program execution time, thereby exposing confidential information.

Fault attacks. Fault attacks exploit physical phenomena and are considered active attacks, where attackers manipulate hardware components beyond their intended limits using software. These exploits capitalize on vulnerabilities in the physical design of computer hardware, such as Dynamic Random Access Memory (DRAM), to compromise system security. One well-known fault attack is Rowhammer, which targets DRAM [26–41]. Specifically, in DRAM, each cell stores binary data (0 or 1) using a capacitor and transistor. Through repeated access to these cells, electrical charge can leak, causing unintended interactions with nearby cells. This

phenomenon induces bit flips in adjacent memory cells without direct access, potentially altering critical data and compromising system integrity. Rowhammer exemplifies how subtle manipulations of hardware at the physical level can lead to significant security risks, highlighting the intricate nature and stealthiness of fault attacks in computing environments.

Transient execution attacks. Transient execution attacks exploit instructions that are not permanently committed in the architectural state due to speculative execution mis-predictions or faults. Despite pipeline flushes ensuring functional correctness by discarding any architectural effects, traces persist in the micro-architecture [42]. Techniques for extracting information from these traces through side channels have been significantly refined over the past decade. Transient execution attacks are classified into two types: Meltdown and Spectre attacks [43, 44]. Meltdown attacks involve exploiting transient reads from L1 data caches or fill buffers to retrieve secrets, while Spectre attacks use branch predictors and the return stack buffer to induce branch mis-predictions. These attacks pose greater risks compared to conventional side channels and fault attacks, capable of leaking sensitive data such as kernel memory and passwords.

1.1.2 Trusted Computing Side-channels

Major processor vendors introduce a hardware-based technology known as TEE, which provides an isolated environment with memory encryption to fortify the integrity and confidentiality of Virtual Machines (VMs) against privileged attackers, such as malicious hypervisors or host OS. Micro-architectural side-channel attacks within TEE represent a prominent area of security research. Similar to traditional side-channels, this domain focuses on various micro-architectural components in TEE.

For instance, Dessouky et al. [45] demonstrated that timing analysis on caches can uncover sensitive data within TEEs. Building on previous work, the Prime+Count cache side-channel attack was introduced against ARM TrustZone [46]. For TLB component, Gras et al. [18] identified that attackers could exploit the TLB to determine if a specific page was recently accessed in a TEE, thereby obtaining sensitive data, with TLSBleed serving as an example. Additionally, Lee et al. [47] utilized the branch target buffer to execute the Branch Shadowing attack, while

Evtyushkin et al. [17] exploited directional branch predictors to infer sensitive information by causing branch conflicts. Moreover, page-fault side-channel attacks allowed attackers to deduce control and data flows from page faults [48]. These attacks are classified based on the Page Table Entry (PTE) flag being monitored, such as the present, accessed, dirty, and reserved bits [49]. DRAM side-channel attacks exploited timing variations during memory operations, enabling attackers to discern memory access patterns in TEEs [50]. Meltdown attacks leveraged transient reads from L1 data caches or fill buffers to extract TEE secrets, with notable examples including the Foreshadow attack [51], Cache Line Freezing attack [52], and ZombieLoad attack [53]. Spectre attacks leveraged branch mis-predictions, including SgxPectre [54], which used branch predictors, and SpectreRSB [55], which employed the return stack buffer.

Notably, researchers have discovered a new side-channel called ciphertext side-channel attack due to the deterministic memory encryption used in AMD's TEE. This attack allows an attacker to deduce relationships between successive plaintexts or identify specific plaintext patterns by observing changes in ciphertexts. The concept of ciphertext side channels was first introduced in Li et al's work [56] and further explored by systematically examining the applicability of this vulnerability [57]. While the initial research focused on AMD's TEE [58], it is important to note that this vulnerability also affects other TEE architectures based on deterministic encryption, provided that attackers can access ciphertext (through software access [56] or memory bus snooping [59]).

1.1.3 Physical Side-channels

Power attacks and electromagnetic attacks are both non-invasive side-channel attacks, meaning the attacker measures various energy consumption or electromagnetic emissions of cryptography hardware devices. These side-channel attacks are effective because bit flips in data cause changes in energy consumption or electromagnetic emissions, allowing the attacker to infer the key.

Power attacks are mainly divided into two types: Simple Power Analysis (SPA) [60, 61] and Differential Power Analysis (DPA) [62, 63]. SPA examines the waveform of current or power consumption over time, using the values of energy consumption to

infer related confidential information. Different instructions executed in a microprocessor generate different currents and power consumption, enabling attackers to deduce the control-flow information of the running program through power supply analysis. Thus, control-flows in cryptography systems that depend on the key are prime targets for attackers. For example, in the RSA cryptosystem, the sequence of square and multiply operations depends on the bit values of the key, making it a focal point for side-channel attacks. DPA is a more advanced power analysis technique that performs statistical analysis on data collected from multiple encryption and decryption operations to calculate intermediate values during computation. DPA relies on the assumption that in a cryptography system, there is an intermediate variable where part of the key's bit values determine whether two inputs will result in the same value for this variable. DPA does not focus on individual power consumption information, thus making it less susceptible to noise. By observing the S-box operations in the Data Encryption Standard (DES) cryptosystem, DPA can recover several key bit values.

Similar to power attacks, electromagnetic attacks derive sensitive information from cryptography systems by measuring the electromagnetic emissions of hardware devices. Electromagnetic attacks are also categorized into Simple Electromagnetic Analysis (SEMA) and Differential Electromagnetic Analysis (DEMA) [64, 65], following the same principles as power analysis counterparts.

Fault injection side-channel attacks involve altering the normal logic of a device through external interference, causing the cryptography system to enter an exploitable error state, thereby revealing internal states, sensitive information, or even the key. These attacks manipulate the operation of cryptography hardware devices by changing the temperature, voltage, clock frequency, magnetic field strength, or by emitting electromagnetic pulses and lasers, thus disrupting the processor's operation and causing it to produce incorrect results. For example, during encryption or decryption, random errors in the hardware device's registers or memory can cause a bit flip. By comparing the correct ciphertext with the erroneous ciphertext or observing the system's state, attackers can theoretically analyze and crack the sensitive information of the cryptography system. Studies have shown that fault injection attacks can successfully break DES and triple DES keys by flipping just 200 bits [66].

1.2 Side-channel Mitigation

1.2.1 Secure Architectures and System Designs

A number of works aim to defeat side channels from both software and hardware perspectives. We describe their mechanisms with cache side-channel attacks as an example. In short, their purposes are divided into two categories based on the strategies, namely limiting attacker's abilities (thus limiting attacker's observation) and obfuscating attacker's observation.

Limiting attacker's abilities. Specifically, designing partition-based cache circuits strongly guaranteed hardware isolation [67, 68], whereas there is a long way to go before adopting these new cache designs to commercial CPUs. Software-level cache partition is then much more practical to achieve. For instance, employing the page coloring mechanism [69, 70], where the same memory page color can be mapped into the same cache set, partitioned the use of the Last-level-Cache (LLC) for different security domains, thus excluding any possible cache conflicts. A similar approach to PLcache [67] was implemented by stealth memory technique [71] through offering hypercalls to lock stealth pages, which are never evicted out of the cache. Besides, the Cache Allocation Technology (CAT) [72] was employed to divide the LLC into secure and non-secure partitions [73]. The non-secure partition is managed as a normal cache while the secure partition only stores secure pages of a user-level program, hence cache line conflicts to secure pages are invisible to attackers. Similarly, Zhou et al. [74] managed the number of lines per cache set that an attacker may probe, weakening the attacker's ability to control the cache. Although the partition mechanism strongly confines an attacker's observation, it is at the cost of cache utilization and performance degradation.

Another approach to limiting attackers' abilities is to restrict their preemption or occupation to caches through a scheduler mechanism, especially in the Cross-VM context. Godfrey et al. [75] and Sprabery et al. [76] relied on scheduler operations to flush cache contents when switching a VM in the cloud, achieving temporal partition of caches. Another work from Varadarajan et al. [77] leveraged the hypervisor scheduler to limit the frequency of cache preemption, obfuscating predictive state to attackers. Nevertheless, relying on the scheduling approach to achieve temporal

isolation of cache is still problematic because latency-sensitive workloads may be delayed.

Obfuscating attacker's observation. An attacker fails to deduce secrets when he obtains invalid observations. Motivated by this idea, novel randomization-based cache components were proposed to randomize the resident points of data in the cache, achieving a high entropy for each data unit [67, 78–80]. However, recent studies showed that these randomization-based caches are still vulnerable to cache side-channel attacks [81, 82].

By increasing difficulty to the adversary's measurements and resulting in a fixed or random observation, it becomes infeasible for the adversary to obtain accurate timing information. For example, Aviram et al. [83] provided enforced deterministic execution to eliminate timing channels. Another two works [84, 85] modified the *rdtsc* instruction to offer coarse-grained clocks and added a small randomized offset to fuzz the guest operating system's measurement, respectively. Extremely, Li et al. [86] introduced three replication of a VM and normalizes the timing to be observed by the median of replicas collectively determined. The above three works [84–86] disabled precise timing measurement for mitigating timing-channel attacks, yet severely limiting the workloads that need accurate timing and may incur overheads for network workloads resident in the cloud. In contrast, Zhang et al. [87] periodically injected noise through flushing the L1 cache among the adversary's waiting period to obfuscate his timing measurement.

Software diversity aimed to prevent the adversary's observation from deducing a predictable cache state [26, 74, 88–90]. Precisely, Liu et al. [88] leveraged compiler transformation to guarantee memory access traces are the same no matter which control flow path is taken by the program. Later, Rane et al. [89] made improvements by arranging the exact same instructions for both paths of a branch, preventing a wider range of side channels. Without modifying the source codes, Grane et al. [90] dynamically inserted a mass of redundant operations (e.g., *nop* and memory load) to generate diversified replicas, thus randomizing the control flow of programs. Disabling and selectively enabling memory page sharing were proposed to separate the touched points in the cache [26, 74]. As memory deduplication drives the access requests to the shared libraries from two distinct programs into the same cache line, the attacker cannot deduce whether a certain path is executed by the victim program from his observation.

1.2.2 Trusted Computing Hardening

Since TEEs are built upon CPU hardware security technologies, vulnerabilities at the micro-architectural level and flaws in the architectural design can severely compromise the security of TEEs. Researchers have extensively explored architectural security enhancements, driving the iteration of TEE architectures.

To defend against micro-architectural side-channel vulnerabilities, the primary mechanisms are implemented through software, focusing on detecting the occurrence of side-channel activities and disrupting the acquisition of side-channel information. Given that micro-architectural side-channel attacks often involve frequent interrupts, detecting these interference signals can help identify ongoing attacks [91, 92]. Additionally, since frequent interrupts result in increased latency for TEE operations, some approaches used this latency as an indicator of an attack [93]. Providing an uninterrupted execution environment for TEEs is also a crucial defensive measure against side-channel attacks. For example, Chen et al. [94] ensured an uninterrupted environment through verifiable execution contracts. To prevent attackers from obtaining effective side-channel information, current defenses utilized Oblivious RAM (ORAM) [95, 96], CPU hardware features like Transactional Synchronization Extensions (TSX) [97–99], and the introduction of randomization techniques [100].

Transient execution vulnerabilities can be further divided into those caused by implementation flaws and those resulting from design flaws. The former is due to improper implementation of the CPU's exception handling mechanisms and has been addressed by hardware patches and upgrades in mainstream processors. The latter is related to branch prediction units and represent design flaws, requiring longer deployment cycles for hardware-based fixes [101]. The current mainstream solution involves compiler-assisted software hardening, although this often incurred significant performance overhead [102].

1.2.3 Constant-time Principles

Constant-time is a critical concept to resist side-channel leakage, which inherently requires the memory access and condition branch to be independent of the secrets.

Constant-time verification. Prior works develop analysis to formally reason the constant-time designs, with more and more precise leakage models. In the early stages, researchers applied a policy called program counter [103–108], where it only takes into account the critical control flow and balances the timing behavior of both branches. Considering memory accesses, subsequent works abided by the most common leakage model that exploits time variations and covers cache-based attacks. Specifically, VirtualCert [109] and constant-time MEE-CBC [110] performed typing analysis in CompCert [111] on the x86 assembly, which enforced the notion of noninterference to verify the classic observation equivalent. Similarly, FlowTracker [112] analyzed the Static Single Assignment (SSA) form for LLVM programs. Later, variable-time operations are also taken into account by Dehesa-Azuara et al. [113]. They relaxed the conventional noninterference that measures the value of a specific variable into a resource-ware noninterference, where the observation can be the execution time or cache side channels.

In addition to noninterference, the reduction-based methods were applied, where security was reduced to safety through the self-composition of programs or formulas [114–120]. Separately, Almeida et al. [114] performed deductive verification based on self-composition on C implementations, while ct-verif [115] targeted the optimized LLVM implementations. A variety of methods, such as Themis [116], Blazer [117], and IFC-CEGAR/BMC [118], were proposed to improve the self-composition through taint analysis or complexity analysis. Different from ct-verif [115] that utilized solvers to prove safety implying constant-time property, Blazy et al. [119] developed an analyzer based on abstract interpretation to compute the approximation of the execution of the analyzed program in an instrumented semantics with tainting. Recently, considering the constant-time properties were generally not preserved by the compilation, Binsec/rel [120] proposed a relational symbolic execution-based analysis tool for verifying constant-time implementations at the binary level. It self-composed the formulas of the program execution and maximized sharing the equal expressions in both executions for improving the performance of reduction.

Constant-time construction. The main target of this category is to formally construct high-assurance cryptography libraries that fundamentally resolve the constant-time issues. However, this is only restricted to the verified code level instead of the runtime. For example, F* [121] and HACL* [122] (also bases on F*

language [123]) both supplied verified cryptography libraries at the source level. Differently, the former [121] enforced a coding discipline for mitigating side channels by typing checking, while HACL* [122] proved freedom of timing channels through manual pre-postconditions that at large rely on experts. In such a way, they suffer from limited scalability. Moreover, their verification toolchain stops at the source code, thereby failing to promise secret independence properties delivered to lower code when using mainstream compilers. Vale [124] impelled the progress of constructing side-channel freedom of a cryptography program to an assembly-like code, i.e., Vale language. The Vale leakage verifier employed a taint-analysis engine and the self-composition method to verify the program's freedom from side-channel vulnerabilities. However, Vale shared the same drawback that thinking of the compiler as trusted thus not verifying it. Jasmin [125] and Fact [126] were compiler-based formal frameworks that respectively transform Jasmin programs into assembly code, and timing-sensitive FaCT code into constant-time LLVM IR.

Constant-time transformation. Transforming programs into constant-time equivalents or variations also plays a significant role. Johan Agat [103] and Kopf et al. [106] equalized the timing characteristics of secret-dependent conditional branches by inserting dummy statements that do not update global variables, while Barthe et al. [105] transformed two branches into the form of transaction. Using source-to-source transformation, Molnar et al. [104] consecutively executed secret-dependent branches and leverages bit-masking to commit the correct results. Similarly, Coppens et al. [127] executed the dummy path with local, temporary variables and submitted results with conditional instructions. Nevertheless, all these works only consider branches that are under a simple side-channel attack model, i.e., program counter security [104]. This means the branches of a balanced time property may still leak the secrets with more-capable attackers such as cache-based attacks. In addition, Barthe et al. [128, 129] proved the observational noninterference of a compilation process, guaranteeing the constant-time property for cryptography. Concretely, CompCert [111] was modified to enable it to capture constant-time during the compilation [129]. Recently, SC-Eliminator [130] removed secret-dependent branches by both executing real and decoy paths. However, its way to preloading the lookup table cannot completely eliminate the cache side channels because the table may be evicted by the attacker. Considering the unmodified addresses in the decoy path that may result in out-of-bounds memory accesses in SC-Eliminator, Soares et al. [131] guaranteed memory-safe accesses in the decoy paths. For the

decoy iterations, they both inferred a bounded loop number from the compilation and unroll the loop, hence the code size increases [130, 131]. Constantine [132], like the most radical method in constant-time, proposed the linearization of control-flow and data-flow, meaning to execute all possible code/data memory accesses. It improved the linearization of loops by just-in-time the number that the loop should execute. Nevertheless, Constantine only guaranteed the transformed code is constant-time and required subsequent compilation not to add branches, which may break the constant-time.

1.3 Research Scope and Overview

Researchers have made significant strides in defeating side-channels by developing innovative hardware-based defenses and implementing software-level countermeasures, thus enhancing the security of modern computing systems.

While there is a growing body of work dedicated to the detection of cache side-channels, many of these efforts grapple with significant challenges in terms of comprehensiveness and scalability. Detecting cache side-channels is inherently complex due to the intricate interplay between hardware, software, and micro-architectural intricacies. This complexity often results in a lack of comprehensiveness of the underlying mechanisms and vulnerabilities, which can hinder the effectiveness of detection methods. For example, CacheD [133] employed symbolic execution on tainted assembly code to construct formulas, which were then sent to a solver to check whether a memory address maps to different cache locations, depending on secret values. However, CacheD primarily focused on secret-dependent memory accesses and overlooked branches in its analysis. Additionally, the scalability of cache side-channel detection approaches is a pervasive concern. Cryptography systems become increasingly intricate and larger in scale, making it challenging to apply existing detection techniques comprehensively. One example is that modern cryptography libraries extensively use randomization techniques, such as blinding, to mitigate side-channel attacks. However, the effectiveness of these techniques (and any remaining information leaks) has not been analyzed by previous tools like CacheS [134], which is considered one of the most scalable static analysis tools

in this field. Consequently, addressing these issues in cache side-channel detection is paramount, as it ensures that security efforts remain effective in the face of evolving threats and complex computing infrastructures.

It is important to highlight that while constant-time coding practices have traditionally been an effective means of mitigating cache side-channels, their efficacy wanes when dealing with new challenges posed by ciphertext side channels. In short, ciphertext side-channel attacks exploit the deterministic memory encryption employed by TEEs, where the same physical address consistently encrypts into the same ciphertext block. This inherent determinism means that constant-time coding practices, which traditionally neutralize timing variations in cache side-channels, are ineffective against this type of attack due to leaving footprints of execution. Addressing ciphertext side channels necessitates a more holistic approach that combines constant-time coding practices with the attack mechanism. For example, CipherFix [135] employed dynamic taint analysis to identify the offsets of sensitive memory store instructions in a program. Using static instrumentation, it transformed each tainted memory store instruction into a copy that incorporated masking operations in the instrumentation section. A direct jump instruction was then inserted at the original program point, redirecting execution to the newly created copy. However, CipherFix mitigated ciphertext side-channel leakage at the cost of significant performance overhead, ranging from 2.4× to 16.8× in its most efficient variant, due to the use of static binary instrumentation.

Simultaneously, while efforts are being made to develop new cache designs aimed at countering cache side-channels, a notable challenge arises from the lack of formal guarantees regarding their effectiveness. Novel cache designs that focus on mitigating side channels often introduce complex hardware operations, for example, the randomization process [67, 78–80]. This intricacy makes it difficult to provide rigorous mathematical or formal proofs of their effectiveness. Moreover, the ever-evolving landscape of side-channel attacks means that new vulnerabilities may emerge over time, challenging the robustness of these designs. Thus, despite the innovative solutions being proposed [136–140], the absence of formal guarantees underscores the need for continuous evaluation, and adaptability to ensure that these cache designs effectively thwart emerging side-channel threats in a real-world context.

The challenges and limitations in defending against side-channel attacks discussed earlier provide the foundational motivation for this thesis. The primary objectives of this thesis are threefold:

- Advancing the field of cache side-channel detection by developing more robust and comprehensive methods for identifying and fixing these vulnerabilities.
- Addressing the emerging issue related to the constant-time principle by mitigating ciphertext side channels, which present new and formidable threats in the realm of TEEs.
- Enhancing the security of new cache designs by providing a reasoning framework for formally verifying their effectiveness in countering cache side-channel attacks.

By addressing these critical areas, this thesis seeks to contribute to the ongoing efforts to strengthen the security of modern computing systems in the face of evolving and increasingly sophisticated side-channel threats.

1.4 Major Contributions

This thesis makes the following three contributions.

First, we propose CaType, a novel refinement type-based tool for detecting cache side channels in cryptography software. Compared to previous works, CaType provides the following advantages: For the first time CaType analyzes cache side channels using refinement type over x86 assembly code. It reveals several significant and effective enhancements with refined types, including bit-level granularity tracking, distinguishing different effects of variables, precise type inferences, and high scalability. CaType is the first static analyzer for cryptography libraries in consideration of blinding-based defenses. From the perspective of implementation, CaType uses cache layouts of potentially vulnerable control-flow branches rather than cache states to suppress false positives. We evaluate CaType in identifying side-channel vulnerabilities in real-world cryptography software, including RSA, ElGamal, and (EC)DSA from OpenSSL and Libgcrypt. CaType captures all known defects in our dataset (Section 2.6.3), detects previously unknown vulnerabilities

(Section 2.6.4), and reveals several false positives of previous tools (Section 2.6.6). In terms of performance, CaType is 16× faster than CacheD [133] and 131× faster than CacheS [134] when analyzing the same libraries. These evaluation results confirm the capability of CaType in identifying side channel defects with great precision, efficiency, and scalability.

Second, we design CipherGuard, a compiler-aided mitigation methodology to counteract ciphertext side channels with high efficiency and security. For the first time, we introduce a compiler-aided strategy to address ciphertext side channels. Through the exploration and implementation of multiple mitigation variants, e.g., software-based probabilistic encryption and secret-aware register allocation, our compiler-aided strategy has been demonstrated to provide a more efficient and flexible solution in comparison to the instrumentation strategy in CipherFix [135]. CipherGuard is an LLVM-based compiler-aided tool that harnesses dynamic taint analysis and deploys vulnerability mitigation variants at the compilation stage. CipherGuard excels in generating more efficient mitigated code through in-place mitigation code insertion, precise buffer management for random nonces, and flexible register allocation. CipherGuard is evaluated in mitigating ciphertext side-channel vulnerabilities among real-world cryptography software by hardening all the sensitive memory access instructions. With efficient management of random nonce buffers and flexible register allocation, CipherGuard demonstrates a satisfactory performance overhead, averaging 1.76-3.10× across various evaluations of cryptography software. This is a significant performance improvement over CipherFix, highlighting the efficacy of the compiler-aided strategy in enhancing the security against ciphertext side channels.

Third, we propose a comprehensive methodology based on formal methods for security verification of cache architectures. Specifically, we design an entropy-based noninterference reasoning framework with two unwinding conditions to assess the information leakage of the cache designs. The reasoning framework quantifies the dependency relationships by the mutual information between the distributions of input and output of side channels. Given a cache design, we formalize its behavior specification along with the cache layouts into an abstract state machine, to instantiate the parameterized reasoning framework that discloses any potential vulnerabilities. We use our methodology to assess eight state-of-the-art cache architectures to demonstrate reliability as well as flexibility.

1.5 Outline of the Thesis

The thesis is organized as follows.

Chapter 1 briefly introduces the background of side-channel attacks in various contexts and existing methods for their detection and mitigation. The chapter then defines the scope of the research, delineating the specific areas of focus, such as side-channel detection and mitigation, as well as the verification of novel cache architectures. It also outlines the primary contributions of the research and offers an overview of the thesis's structure.

Chapter 2 reveals the limitations of current cache side-channel detection, providing the motivation of CaType's methodology. It discusses CaType's innovative features, specifically the refinement type system, and its comprehensive and scalable side-channel vulnerability detection to cryptography libraries. The evaluation results demonstrate CaType's precision and efficiency in identifying side-channel vulnerabilities, concluding with a summary of its contributions.

Chapter 3 starts by introducing the emerging threat of ciphertext side channels and CipherGuard's purpose. It explains CipherGuard's automatic recognition of secret-dependent store instructions and multiple precise mitigation variants embedded in the LLVM ecosystem, emphasizing compatibility with TEEs. Evaluation results underline CipherGuard's efficiency and security compared to prior methods, ending with a summary of contributions.

Chapter 4 begins with the importance of formal methods for cache architecture security verification and provides background on the methodology. It details the methodology's application to assess cache architectures, highlighting reliability and flexibility. The chapter concludes with a summary of its contributions and the significance of using formal methods for security verification.

Chapter 5 summarizes the thesis, highlighting contributions like novel cache side-channel detection tools, while also outlining future research directions, such as quantification of cache side-channel leakage.

Chapter 2

Refinement type-based Detection of Side-channels in Cryptography Software

2.1 Introduction

Cache-based side channels have demonstrated serious threats to cryptography algorithms, such as the symmetric cipher AES [5, 7], the asymmetric cipher RSA [11, 13, 141], and the digital signature (EC)DSA [142–144]. The essence of these cache attacks is the interference of program memory accesses toward cache units, where secret-dependent memory accesses or program branches leave distinguishable footprints in cache units. Thus, identifying and removing cache interference can eliminate side-channel leakage.

Designing novel security-aware cache architectures may eliminate adversarial interference. Prior research relies mostly on two strategies, namely partitioning-based and randomization-based approaches. Strong isolation is achieved in partition isolated caches [67, 68] by physically partitioning the shared cache into multiple zones for applications of various security levels. In contrast, randomization-based cache designs [67, 78–80] obscured adversary observations by randomizing the cache states. Although it is envisaged that these architectures will eliminate interference and secure programs that run on top of them, recent studies showed that these randomization-based caches may be still vulnerable to cache side channels [81, 82].

Also, these new cache designs achieve security promise at the expense of performance. Besides, they are not yet ready for commercial use due to extra costs in chip circuit manufacturing.

Software-based mitigation of cache side channels appears increasingly viable. However, manually detecting vulnerable cryptography code takes specialized knowledge, which drastically restricts normal developers from analyzing and patching their cryptography software. With the fast development of more efficient cryptography software under various usage scenarios, launching timely side channel analysis becomes even more challenging. With this regard, developing a general, automated, and efficient analytic tool for detecting cache side channels is receiving broad attention from both academics and industry. Recent works [133, 134, 145–148] served as examples of this. In general, these works constructed constraints through symbolic modeling of program states and cache accesses. Then, constraint-solving techniques (e.g., Z3 solver [149]) were employed to check the satisfiability of constraints and decide whether the program is vulnerable to cache side channels. While these automated methods have made concrete progress in discovering cache side channels in real-world cryptosystems, they still face a number of obstacles.

Challenge 1: *Software-based analysis needs to address precision issues and be scalable to production cryptography libraries.* CacheAudit [145] and its extension [146] calculated the upper bound of information leakage by counting all possible final cache states via abstract interpretation [150]. However, estimating the worst-case leakage bound may not reflect reality. Moreover, CacheAudit cannot pinpoint what/where the vulnerability is, prohibiting the debugging/fixing of analyzed code. Using symbolic execution, CaSym [148] distinguished two different cache states resulting from secret variants. Though CaSym covered multiple paths, it suffered from path explosion and is less scalable. CacheS [134], likely the most scalable static tool in this field, also used abstract interpretation. It achieved higher scalability due to modeling secret/non-secret semantics with symbolic formulas of different granularity. Dynamic approaches, in contrast, analyze concrete execution traces to track program states and pinpoint side channels. CacheD [133] detected secret-dependent memory accesses via symbolic execution, while not considering secret-dependent branches. DATA [147] considered both memory access leaks and branch leaks through differentiating address traces. Existing dynamic methods,

though manifest relatively improved scalability, may still be slow to analyze production cryptography libraries (due to the usage of constraint solving) or require many well-chosen inputs to induce distinct observations.

Challenge 2: *Cache models adopted by software analyzers have an effect on the scalability and detection granularity.* Relying on concrete cache replacement policies (e.g., LRU, FIFO, and PLRU), CacheAudit precisely described a program executed on the expected architecture, at the cost of scalability due to architectural complexity. CaSym used high-level abstract cache models (i.e., infinite and age models) to achieve higher analysis scalability. It used the array index to compute the accessed cache locations. However, these abstract models have granularity issues: there is a gap between the array index and the cache location in realistic architectures. At the other extreme, a much-simplified cache model was shared by works [133, 134, 146, 147, 151], where an architectural-independent model was used to detect cache side channels. Though this model is realistic and efficient, performing analysis at such granularity results in false positives, as will be discussed in this chapter.

Challenge 3: *Supporting a comprehensive analysis of cryptography software rather than some specific defects in sensitive code fragments.* For instance, CacheD omitted the analysis of secret-dependent program branches. Moreover, modern cryptography libraries extensively use randomization schemes like blinding to mitigate side channels, whose effectiveness (and remaining leaks) have not been analyzed by previous tools. Supporting randomization is inherently hard for previous static (abstract interpretation-based) tools [134, 145, 146], requiring new abstract domains, new abstract operators, and soundness proofs. Meanwhile, modeling randomization is also costly for approaches that use constraint solvers, as it demands to iterate blinding quantifiers [133, 134, 148]. Weiser et al. [147] conceptually differentiated traces derived from blinding-involved computations, but it overlooked the complex computations involving blinding in production cryptosystems, which may contain new attack vectors.

The aforementioned obstacles incentivize the design of CaType, an automated, precise, and efficient cache side-channel analysis tool. CaType is scalable and capable of analyzing large-scale, complex cryptography software. CaType follows tools [133, 147] to log execution traces of cryptography software and performs trace-based type inference on the logged traces. It features a novel refinement type

system that enables tracking program variables in the bit-level representation. Different from previous constraint-solving-based approaches that are inherently costly, our sound type system guarantees fine-grained secret tracking and side channel detection with largely improved efficiency. Lastly, CaType comprehensively models randomization-based mitigation schemes adopted in modern cryptography software. It allocates specific refined types for differentiating the responsibilities of (secret or randomized) variables, enabling precise information flow tracking under the presence of randomization. In sum, we make the following contributions:

- Conceptually, for the first time, cache side channels are analyzed using refinement type techniques. We establish our novel refinement type system directly over x86 assembly code and formulate cache side channels over refined types.
- Technically, CaType features several important and effective enhancements compared with prior tools on the basis of the refinement type system, including bit-level granularity tracking, distinguishing different effects of variables, precise type inferences, and much higher scalability. CaType takes into account randomization-based defenses using specific refined types, and uses novel cache layouts to suppress potential false positives.
- Empirically, we evaluate CaType to uncover side channel vulnerabilities among real-world cryptography libraries. CaType captures all known design flaws, identifies unknown flaws, and reveals several false positives in existing tools. CaType is 16× faster than CacheD and 131× faster than CacheS, demonstrating its high applicability toward production cryptography software.

Responsible disclosure. When publishing CaType, we identified newly discovered side-channel vulnerabilities from Libgcrypt and OpenSSL. These vulnerabilities were thoroughly investigated, and the findings were reported to relevant community developers in an effort to improve the security of these libraries. Unfortunately, no responses were received, and the issues were not assigned CVEs. We documented these new findings in detail in Section 2.6.4, with a focus on detecting such vulnerabilities rather than exploiting them for side-channel attacks, because exploiting these vulnerabilities for malicious purposes is not within the scope of this thesis. We believe that the detection techniques outlined here may serve as

useful resources for community developers who seek to understand and address these types of vulnerabilities in cryptographic implementations.

For the other parts, we present the preliminary knowledge in Section 2.2, and methodology overview in Section 2.3. Section 2.4 and Section 2.5 demonstrate more details of the design and implementation of CaType. The evaluation of CaType is conducted in Section 2.6. Section 2.7 gives a further discussion on our tool. Section 2.8 discusses the related work and Section 2.9 concludes this chapter.

2.2 Background

2.2.1 Refinement Type Systems

A type system is a well-established formal system comprising a set of rules that assigns types to terms in a programming language [152, 153]. For example, C language contains a basic type system, where types (e.g., `int`, `double`, and `int*`) give *meaning* to data in the memory or registers. Modern C compilers can feature basic type-checking rules to detect invalid operations, e.g., when a variable of double is used as `int*` (for pointer dereference), an error is thrown at the compilation time.

Type systems are widely used in language-based security research [154] like tracking secure information flow. In those systems, the types of variables and expressions are attached with annotations that specify confidentiality policies enforcing the use of the typed data. For instance, two type annotations H and L are used to denote high and low security sensitivity of data. To detect the violation of confidentiality policy, a set of type rules is defined to check if the two classified sets of data interfere with each other.

Refinement types [155] extend standard type annotations with predicates that confine the use of the values described by the type. Typically, a variable x's refinement type can be defined in the form of $x : T\{v : P\}$, where T is a basic type and P is the associated predicate. For example, a non-negative integer variable x is represented as $x : int\{v : 0 \leq v\}$, where predicate $0 \leq v$ refines the basic type int by specifying that the integer must be greater than or equal to zero. With well-defined predicates, the refinement types can provide stronger guarantees. For example, the zero-division errors can be alerted at the compilation time when the predicate

$N \geq 0$ indicates that the divisor may be zero. Meanwhile, one can elaborately specify security policies over the refinement types to verify software security vulnerabilities. Works [156–159] are successful examples of adopting refinement type systems in high-level languages (e.g., F*) to provide security guarantees in cryptography infrastructures. To our best knowledge, CaType is the first to employ refinement types over assembly code and for cache side channel detection.

2.2.2 Cache Hierarchy and Cache Side-channels

Caches are incorporated into CPUs to accelerate process execution due to the locality principle. In modern CPUs, each core (i.e., a processing unit on a CPU chip) monopolizes a L1 cache and a L2 cache. All cores share a megabyte-size Last-Level Cache (LLC). The access time for a cache hit is around tens of cycles. In contrast, the latency will become much higher (usually hundreds of cycles) when a cache miss occurs and the main memory has to be accessed. Modern CPUs use a W-way set-associative cache. Different memory blocks may reside on the same cache set, and each cache set is further divided into W cache lines. Given an N-bit memory address, S-set cache with L byte-size cache line, the lowest $log_2 L$ bits of the address represent the offset since continuous memory blocks are cached together within one `load` instruction. The middle $log_2 S$ bits starting from bit $log_2 L$ are used to locate the cache set index. The upper part represents cache hit/miss tag bits.

However, cache poses threats of secret leakage, as program cache accesses may be leveraged by adversaries to reconstruct confidential information. In this chapter, we introduce two representative vulnerable code patterns, *secret-dependent branch condition (SDBC)* and *secret-dependent memory access (SDMA)*, via classic examples in RSA.

$$
\begin{array}{ll}
1: & x \leftarrow 1 \\
2: & \textbf{for } i \leftarrow |e| - 1 \textit{ downto } 0 \\
3: & \quad x \leftarrow x^2 \textit{ mod } m \\
4: & \quad \textbf{if } e_i = 1 \textbf{ then} \\
5: & \quad\quad x \leftarrow x \cdot b \textit{ mod } m \\
6: & \textbf{return } x
\end{array}
$$

Figure 2.1: Square-and-Multiply Exponentiation.

Secret-Dependent Branch Condition (SDBC). Figure 2.1 shows a simplified view of the square-and-multiply implementation of modular exponentiation in RSA. e_i (line 4) denotes a private key and decides if line 5 is executed. By monitoring the L1 instruction cache (I-cache), attackers are aware of the execution of line 5, and further reconstruct e using well-established cache attacks [11, 13].

$$
\begin{array}{ll}
1: & g[0] \leftarrow b \ mod \ m \\
2: & \textbf{for} \ j \leftarrow 1 \ to \ 2^{S-1} - 1 \\
3: & \quad g[j] \leftarrow b^{2j+1} \ mod \ m \\
4: & x \leftarrow g[(w_{n-1} - 1)/2] \ mod \ m \\
5: & \textbf{for} \ i \leftarrow n - 2 \ downto \ 0 \\
6: & \quad x \leftarrow x^{2^{L(w_i)}} mod \ m \\
7: & \quad \textbf{if} \ w_i \neq 0 \ \textbf{then} \\
8: & \quad\quad x \leftarrow x \cdot g[(w_i - 1)/2] \ mod \ m \\
9: & \textbf{return} \ x
\end{array}
$$

FIGURE 2.2: Sliding-window Exponentiation.

Secret-Dependent Memory Access (SDMA). Besides SDBC, SDMA also leads to exploitation. Consider Figure 2.2, where the sliding window modular exponentiation algorithm initializes a precomputed array $g[i]$ (lines 1–3) to accelerate the computation. When performing decryption, a window size key w_i (line 8) is used as the index to query the precomputed table $g[i]$. For each for-loop (line 8), monitoring the accessed data cache (D-cache) line can reveal certain bits in w_i and gradually reconstruct the private key [11].

2.2.3 Cache Side Channel Mitigation

Lou et al. [160] surveyed software-level countermeasures of cache side channels. Overall, two code patterns can remove secret-dependent cache access patterns: *AlwaysAccess-BitwiseSelect* permits programs to access secret-dependent data within each loop iteration in a constant manner while deciding whether or not to accept it via bitwise operations. Moreover, if the calculation is inexpensive and free of secret-dependent branches, *On-the-fly Calculation* avoids using lookup tables, which eliminates leakage shown in Figure 2.2. Similarly, to remove secret-dependent branches, *AlwaysExecute-ConditionalSelect* enables covering all branches regardless of the `if` conditions. *AlwaysExecute-BitwiseSelect* eliminates secret-dependent branches by selecting correct results through bitwise operations.

The aforementioned code patterns can frequently introduce high overhead. They are thus less frequently used to only secure several core code fragments, which may miss subtle usage of secrets [133, 146]. *Blinding* introduces extra randomness in cryptography computations to obscure the inference of secrets. Depending on the blinding target, there are two distinct usages of blinding masks.

Key Blinding. With this scheme enabled, the attacker obtains blinded secrets without knowing the blinding mask r. As r is randomly generated before each cipher process, the attacker cannot exploit the cryptosystem. For example, exponent blinding in RSA adds a random multiple of Euler's ϕ function, i.e., $r \cdot \phi(n)$, to the secret exponent. Then, RSA decryption performs $c^{d+r \cdot \phi(n)}\ mod\ n$, which equals $c^d\ mod\ n$. Though some known attacks [161] exploit this scheme, the exponent blinding still impedes the attacker at large.

Plaintext/Ciphertext Blinding. Blinding can also be applied to plaintext/-ciphertext. For instance, when enforcing blinding, RSA converts the ciphertext m into $m \cdot r^e$, where r is the random factor. The original result $m^d\ mod\ n$ can be obtained by multiplying the new result $(m \cdot r^e)^d\ mod\ n$ by r^{-1} due to $r^{ed} \cdot r^{-1} mod\ n \equiv 1\ mod\ n$. The plaintext/ciphertext blinding defeats known-input attacks that leverage timing side channels.

Blinding can usually provide more comprehensive protection as once key/ciphertext is blinded, all their follow-up usages and their (subtle) influence on other variables should be protected. However, their effectiveness in mitigating cache side channels is not yet comprehensively analyzed, given the difficulty of modeling them automatically in previous methods (noted in **Challenge 3** in Section 2.1).

2.3 Research Overview

2.3.1 Assumptions

Threat Model. CaType follows an identical threat model as most current cache side-channel detectors [133, 134, 148, 151, 162]. We assume that an adversary

shares the same hardware platform as the victim, a typical and practical assumption in cloud computing systems. Thus, while the adversary cannot directly monitor the victim's memory accesses, he can probe the shared cache states to determine if certain cache lines have been visited by the victim software. This threat model covers the majority of cache side-channel attacks in the literature. For example, adversaries infer cache accesses by measuring the latency of the victim program in EVICT-TIME attack [5], or the latency of the attacker program in PRIME-PROBE [5, 7, 11], FLUSH-RELOAD [13], and FLUSH-FLUSH attacks [163].

Existing works [145, 148] commonly refer to the attackers in our threat model as "trace-based attackers" since they are able to probe the cache state after the execution of each program statement in the victim software. It is also worth noting that the attackers can distinguish cache layouts of instructions inside the program branches of shared libraries. This is due to the fact that modern OSes adopt aggressive memory deduplication techniques, allowing shared libraries to be mapped to copy-on-write pages. As a result, the probing granularity of attackers is precisely reduced to cache lines.

Main Audience. Consistent with previous works [120, 133, 134, 145, 147, 148, 151, 162, 164, 165], CaType is primarily designed for cryptography software developers who have sufficient knowledge about their own software. Before release, CaType serves as a "vulnerability debugger" for the developers to detect attack vectors in their software. CaType provides fully automated and speedy analysis to flag program points that leak secrets via cache side channels. Developers can accordingly patch CaType's findings to mitigate leakage. Nevertheless, we clarify that CaType is *not* an attack tool; the exploitability of its findings (e.g., whether RSA private keys can be reconstructed via CaType's findings) is beyond the scope of this thesis.

2.3.2 Methodology Overview

This section illustrates the high-level methodology overview and compares it with existing efforts in Figure 2.3. Recall that we have introduced two typical cache side channel patterns in Section 2.2.2: SDMA and SDBC. Figure 2.3(a) presents a

sample code that is vulnerable to SDMA (line 6) whereas the condition at line 9 is *not* vulnerable to SDBC, given that the `else` branch will always be executed.

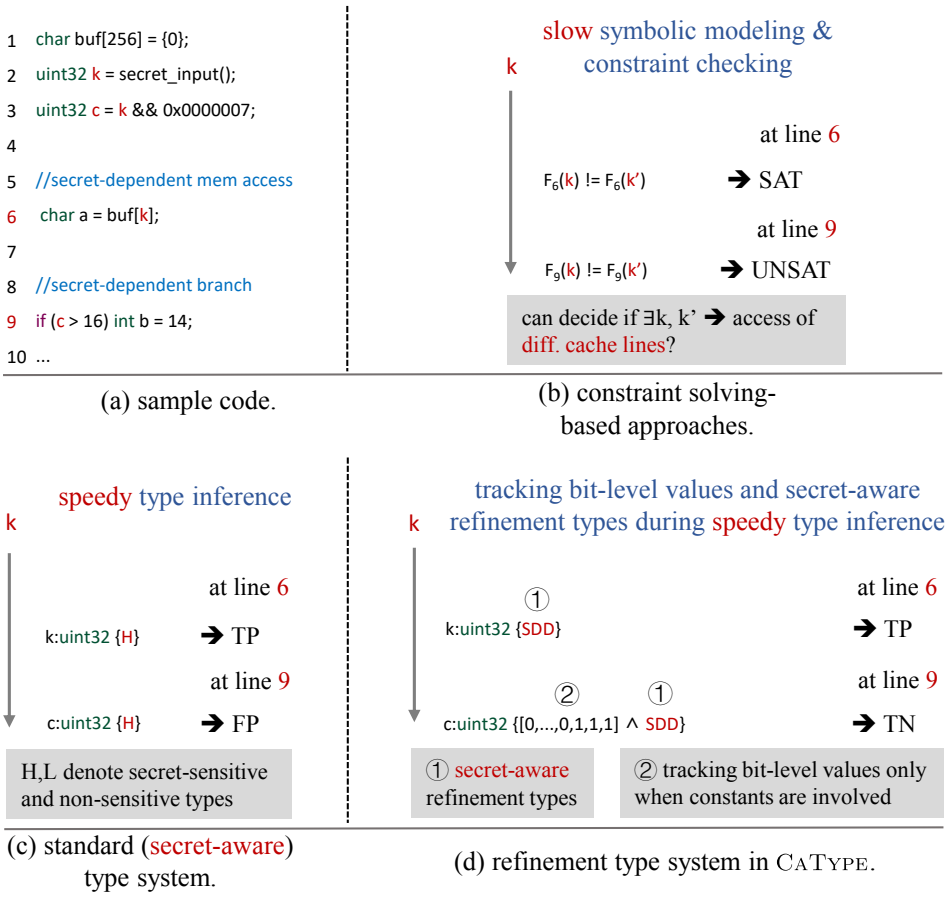


FIGURE 2.3: Comparison of constraint solving-based techniques (b), type inference-based approach (c), and CaType (d). TP, FP, and TN denote true positive, false positive, and true negative, respectively.

Symbolic Execution-Based Approaches. De facto side channel detectors perform heavyweight symbolic execution, where program (secret-related) data facts are modeled using symbolic formulas. Then, at each memory access and branch condition, they check if different secrets can lead to the access of different cache lines using constraint solving. For instance, let symbol k represent the secret read in line 2 of Figure 2.3(a), existing side channel detectors [133, 134, 148, 151] primarily check the following constraint to decide SDMA/SDBC:

$$\exists k \neq k',\ F(k) \neq F(k') \tag{2.1}$$

where F denotes the memory access constraint formed at line 6, or branch condition constraint formed at line 9. The symbolic engine forms $F(k) = b + k \times 4$ at line 6, where b is the base address of `buf`. Figure 2.3(b) illustrates the constraint solving process. The satisfiability (SAT) of Constraint 2.1 checks the existence of two secrets that lead to the access of different cache lines, such that a certain amount of secrets will be leaked to the attacker. Moreover, the symbolic engine will track computations using symbolic formulas, and at line 9, the constraint solver yields unsatisfiable (UNSAT) for Constraint 2.1, thereby proving the safety of line 9.

The primary obscurity of such detectors is *scalability*. Overall, existing symbolic execution (or abstract interpretation)-based side channel detectors need to maintain complex symbolic states for each program statement to encode program semantics. As symbolic execution continues, the symbolic constraints (encoding program states) will steadily accumulate and grow in size, filling a vast amount of memory. Even worse, existing tools need to perform constraint solving for each suspicious memory access and conditional branch instruction, and constraint solving is generally slow. With this regard, we notice that existing static analysis tools are often limited to analyzing small programs, or fail to consider the effect of side channel mitigation techniques like blinding.

Conventional Type-Based Analysis. Section 2.2.1 has introduced basic mechanisms of type systems and the extensions to track high/low secret-sensitive data with type annotations H and L. As illustrated in Figure 2.3(c), performing type inference can easily establish that the types of `k` and `c` (in lines 6 and 9, respectively) are `uint32`. Moreover, by assigning a high-security sensitivity type H to `k` at line 2, the type system identifies two usages of sensitive data at line 6 and line 9. These two statements are deemed as "vulnerable", leading to secret-dependent memory access and branch condition. Nevertheless, we underlie that while the statement at line 6 is a true positive (TP) finding, the statement at line 9 is a false positive (FP), as `c` can never exceed 7 (see line 3 in Figure 2.3(a)). Overall, conventional type-based analysis delivers speedy tracking of (secret-related) data through type annotations. They, however, lack tracking values and are less expressive than constraint-solving-based methods. Indeed, Section 2.7 compares taint analysis, *conceptually similar* to type systems enforcing information-flow security (e.g., work from Sabelfeld et al. [166]), with refinement type system implemented

in CaType. We show that taint analysis yields considerably more false positives than CaType.

Refinement Type System in CaType. Recall the refinement type of a variable x can be expressed as $x : T\{v : P\}$ (Section 2.2.1), where T and P are basic types and predicates, respectively. Figure 2.3(d) illustrates the usage of the refinement type system in CaType, where the refinement formalizes the concerned (secret-related) program properties as predicates. In particular, we use type SDD to denote secret-dependent values, and the refinement type system infers that in line 6, `k` is of type uint32$\{v : \text{SDD}\}$, revealing a potential SDMA case. Similarly, the refinement type of `c` in line 9 also has type SDD, revealing a potential SDBC case (which is *not* vulnerable; see below for clarification). CaType defines in total five predicates, systematically considering secret-dependent, secret-independent, as well as blinding operations. In this way, CaType can benefit from refinement type techniques to keep track of secret propagations and identify SDMA/SDBC in a speedy manner while correctly considering randomization mechanisms like blinding (see **Blinding** later this chapter for further discussion).

Moreover, CaType explores an important improvement, by tracking bit values directly in refinement types, in the form of value predicates. A value predicate is defined as $v = b$, where b is either 0 or 1. CaType is carefully designed to deliver a "mild tracking" of bit-level values. That is, only the refinement types of constants are initialized to comprise bit-level predicates. Then, CaType tracks the bit-level predicates via type inference in a correct yet conservative manner. For instance, when a constant, 0x0000007, is used as the mask over the secret (line 3), the type of the output means that it is a bitvector with all secret bits (except the three least significant bits) set to 0. Note that value predicates in refinement types can be absent, indicating that the precise bit-level values are unknown.

By tracking bit values from constants, CaType can exclude the majority, if not all, cases where different secret values at a suspicious SDMA/SDBC case result in visiting the *same* cache line (i.e., a safe program site). For instance, when `k` is masked by 0x0000007 before being used in the `if` condition at line 9 of Figure 2.3(a), the refinement type of `c` has all bits set to 0 except the lowest three bits, and CaType can simply decide that the branch condition will always be evaluated as "false" with an arithmetic comparison over two bitvectors. Therefore,

when analyzing the statement at line 9 of Figure 2.3(a), CaType yields a true negative (TN) finding, as shown in Figure 2.3(d). Overall, we view that the refinement type system designed in CaType manifests comparable capability with constraint-solving-based methods to analyze cache side channels. Moreover, CaType avoids the use of constraint solving, and is therefore dramatically faster; see Table 2.4 in Section 2.6.2.

Potential False Positives. We clarify that the refinement type system in CaType may not always know the precise bit values: the absence of value predicates means the value could be 0 or 1. Overall, CaType tracks the bit values introduced by constants using refinement types at "its best effort". Thus, we may encounter false positives, e.g., due to constants that are however not tracked by CaType. Nevertheless, cache side channels are rare in practice, and we confirm that all findings of CaType over production cryptosystems are true positives. Also, the refinement type system is sound without introducing false negatives, as benchmarked in Section 2.7.

Blinding. As introduced in Section 2.2.3, modern cryptosystems use randomness mechanisms like blinding to impede side channels. To capture the security property of blinding, our refinement type system facilitates a smooth and accurate modeling of blinding, by adding specific predicates in type refinement to denote uniformly random data (i.e., the blinding mask). We also define type inference rules and propagation rules for blinding involved computations, so that we can capture sufficient information used to infer potential leaks. For example, uniformly random factors can perfectly mask the result through logic xor operation, eliminating the effects of a secret if it is a source operand. See details in Section 2.4.2 and Section 2.4.3.

In contrast, adding support for blinding presumably increases the search space of constraint-solving-based methods to a great extent. Consequently, finding a SAT solution for Constraint 2.1 is highly expensive, especially when both secrets and blinding masks are present. Though an "optimal solution" is not yet clear, inspired by relevant research in perfect masking analysis [167–169], we expect to fix two different secrets k, k' and then iterate the quantifiers of all involved masks $r_1, \ldots, r_n$ to count the ranges under k, k'. This process may take a dramatically longer time or timeout.

2.4 Design

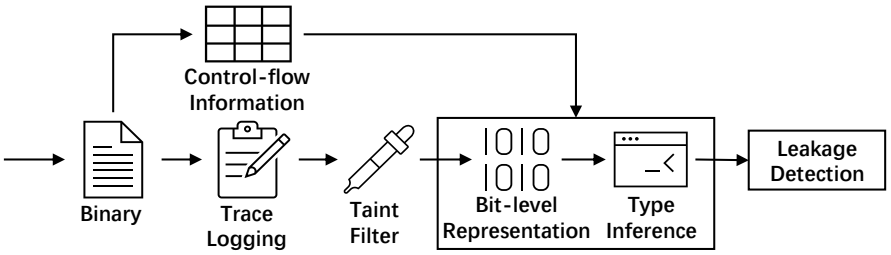


FIGURE 2.4: Workflow of CATYPE.

Overview. Figure 2.4 depicts the workflow of CATYPE. Given the cryptography software in executable format, we first run the executable using Intel Pin [170] to perform concerned cryptography computation (e.g., RSA decryption) and log an execution trace. Then, we require users of CATYPE to mark the program secrets and random factors on the execution trace, and perform taint analysis by tainting those secrets/randomness, and extract a tainted sub-trace depicting how tainted variables are propagated and used. Meanwhile, we also disassemble the executable and extract control flow information into a lookup table from the disassembled assembly code, which will be used later in checking SDBC (see Section 2.4.4).

CATYPE then performs type inference over the tainted sub-trace, by first annotating variables with bit-level types of initialized refinements (Section 2.4.1). It tracks the propagation and usage of secure-sensitive values in refined types during type inference (Section 2.4.2 and Section 2.4.3). When encountering memory accesses or branch conditions, CATYPE uses the refined types of involved variables to check if SDBC/SDMA exists (Section 2.4.4). Once a side channel flaw is discovered, it reports the detected instruction's address to users for confirmation, debugging, and patching. We now discuss each step in detail.

Design Consideration: Binary vs. Source. CATYPE is designed to directly analyze x86 binary code compiled from cryptography software. Thus, the refinement type system is defined over x86 assembly code, and CATYPE's analysis depends on the specific memory layout. Overall, side channels are sensitive to the low-level architecture and system details. We clarify that prior works in this field are consistently analyzing software in executable format. This enables the analysis of legacy code and third-party libraries without accessing source code. More importantly, by analyzing low-level assembly instructions, it is possible to take into

account low-level details, such as memory allocation. Recent work [171] has shown that compiler optimizations could introduce extra side channel opportunities that are not visible at the high-level code representation level.

Design Consideration: Information Flow Tracking. When illustrating cache side channels in Figure 2.1, Figure 2.2 and Figure 2.3, we depict how the use of secrets results in side channels. Nevertheless, in addition to side channels induced via the *direct* usage of secrets, it is crucial to treat data derived from the secrets as "sensitive". CaType *tracks both explicit and implicit information flows propagated from secrets.* When a variable x is of SDD type, and the data is loaded from a memory address formed by x, the destination variable has type SDD. Similarly, when x is used to form branch conditions, the result type is SDD as well. By modeling information flows, CaType comprehensively uncovers the attack surface of cryptosystems.

2.4.1 Bit-level Representation and Types

$$
\begin{array}{rrcl}
\text{Expr} & e & ::= & b \mid x \mid [b, \cdots, b] \mid \neg e \mid e_1 \bowtie e_2 \\
 & & \mid & e \ ?\ e_1 : e_2 \mid e_1 \sharp e_2 \mid [n_1 : n_2]/e \\
\text{Stmt} & s & ::= & x \leftarrow e \mid x \leftarrow e_1[e_2] \mid e_1[e_2] \leftarrow x \mid s_1; s_2 \\
\text{Basic Types} & \mathrm{T} & ::= & \mathrm{B} \mid \mathrm{Vec}\langle n \rangle \\
\text{Security Types} & \tau & ::= & \mathrm{SDD} \mid \mathrm{URA} \mid \mathrm{SID} \mid \mathrm{WRA} \mid \mathrm{CST} \\
\text{Refinements} & \mathrm{P} & ::= & v : \tau \mid v = b \wedge v : \tau \\
\text{Type} & \rho & ::= & \{v : \mathrm{T} \mid \mathrm{P}\} \\
\text{Type Env} & \Gamma & ::= & \emptyset \mid \Gamma, x : \rho
\end{array}
$$

Figure 2.5: Syntax of bit-level representation.

We first clarify that in analyzing x86 assembly code, registers, CPU flags, and memory cells are all considered as *variables* in CaType. We use bit-level representation for variables encountered on the execution trace, allowing us to track variables with fine-grained precision. Considering the instruction syntax in Figure 2.5, where an expression e can be a constant bit b, a variable x, a constant bitvector $[b, \cdots, b]$, or computations over expressions. Concatenation $e_1 \sharp\ e_2$ uses e_1 and e_2 to form the highest and lowest several bits, respectively. Extracting several bits from the designated position of a bitvector expression produces a fragment, dubbed as $[n_1 : n_2]/e$.

Other operations include negation ($\neg$), arithmetic and logic operations ($\bowtie$) over two expressions, and the conditional expression with three operands (the syntax mimics conditional selection in the C language). A statement s is an assignment, a memory load/store, or a sequence of statements. We clarify that execution trace forms a typical straight-line code of instructions, omitting branch merges.

Types and Hierarchy. As introduced in Section 2.3.2, a type ρ has the form of $\{v : T \mid P\}$, where T is a basic type and predicate P is the refinement. We define basic type T as primitive types of bit representations, i.e., one bit B or a bitvector of n bits $\text{Vec}\langle n\rangle$. A refinement type P is either a security type predicate τ or a conjunction with a value predicate. A security type predicate τ can be any of the five types, i.e., SDD, URA, SID, WRA and CST, denoting *secret-dependent*, *uniformly random*, *secret-independent*, *weakly random*, and *constant* values. A value predicate is termed as $v = b$ (where b is 1 or 0), meaning that v has value b. The expression typing judgment, $\Gamma \vdash e : \rho$, states that expression e has type ρ, where Γ is the typing environment mapping from variables to types.

The hierarchy of security types τ is CST $\leq$: URA $\leq$: WRA $\leq$: SID $\leq$: SDD. We clarify that among the five refined types, only SDD is related to secrets. We use WRA to denote a data of weakly random distribution, meaning it is not uniformly random (in other words, not perfect and secure blinding). URA means uniformly random data, representing perfect and secure masking. The join operator $\sqcup$ takes the least upper bound of two types; for instance, SID $\sqcup$ SDD = SDD, as SDD sits higher in the hierarchy.

Types Annotation. Before launching type inference, we first annotate variables with security types. Secrets, random factors, and constants are marked as SDD, URA, and CST, respectively. We mark other variables using SID, and type WRA may be generated during type inference. Given that we perform bit-level type annotation and inference, if variable x hosts a 32-bit secret, it is annotated as $\{v : \text{Vec}\langle 32\rangle \mid v : \text{SDD}\}$. This vector type implies that each bit in the vector has type SDD, i.e., $\forall b_i \in x.\ b_i : \{v : B \mid v : \text{SDD}\}$. For constants, we also explicitly annotate each bit (whether it equals 0 or 1) in the value predicate. Thus, each bit of a constant c is in the form of $b_i \in c.\ b_i : \{v : B \mid v = b \wedge v : \text{CST}\}$, where b is 0 or 1, depending on the value of c. Recall as noted in Section 2.3.2, our refinement type-based inference conducts a "best-effort" tracking of bit-level values derived from constants. The bit-level tracking updates value predicates during type

inference. Nevertheless, when a bit value becomes unknown (could be either 0 or 1), we conservatively omit its value predicate and only retain the security type predicate.

2.4.2 Type Inference for Bitvectors

$$\|[b_{n-1}, \cdots, b_0]\|_t = \begin{cases} \text{SDD} & \exists b_i.\ b_i : \{v : B \mid v : \text{SDD}\} \\ \text{URA} & (\nexists b_i.\ b_i : \{v : B \mid v : \text{SDD}\}) \wedge \\ & (\exists b_i.\ b_i : \{v : B \mid v : \text{URA}\}) \\ \text{SID} & (\nexists b_i.\ b_i : \{v : B \mid v : \text{SDD}\}) \wedge \\ & (\nexists b_i.\ b_i : \{v : B \mid v : \text{URA}\}) \wedge \\ & (\exists b_i.\ b_i : \{v : B \mid v : \text{SID}\}) \\ \text{WRA} & (\nexists b_i.\ b_i : \{v : B \mid v : \text{SDD}\}) \wedge \\ & (\nexists b_i.\ b_i : \{v : B \mid v : \text{URA}\}) \wedge \\ & (\nexists b_i.\ b_i : \{v : B \mid v : \text{SID}\}) \wedge \\ & (\exists b_i.\ b_i : \{v : B \mid v : \text{WRA}\}) \\ \text{CST} & \forall b_i.\ b_i : \{v : B \mid v : \text{CST}\} \end{cases}$$

FIGURE 2.6: Type propagation from single-bit to bitvector.

Different bits in a bitvector may have varying security types. Consider register `eax`, which stores a 32-bit data, where the upper 16 bits are URA and the lower 16 bits are SID. Intuitively, the bitvector's type can be inferred by simply taking the least upper bound of all the constituent bits' types, i.e., SID in this case. However, the high 16 bits are URA, meaning that each bit has equal possibility of being 0 or 1. Thus, the intuitive approach would lose the information of randomness, leading to inaccuracy in subsequent analyses.

To precisely track bit-level security propagation, we define function $\|x\|_t$ in Figure 2.6 to infer a bitvector's type from the types of its constituent bits based on a notion of structural priority. We give type SDD the highest priority, meaning that a bitvector is of type SDD if it contains at least one bit of type SDD. In the absence of SDD type, type URA is structurally preceding, i.e., if there is a bit in a vector whose type is URA, then the vector itself is URA. As seen in Figure 2.6, SID is structurally superior to WRA and CST, whereas WRA is structurally superior to CST.

From a holistic view, sensitive data (specified in refinements) are "propagated" from single-bit to whole bitvector following type rules in Figure 2.6. Therefore, information flow analysis is performed here to determine how sensitive data are propagated and influence program execution. To clarify, in addition to type rules, CaType also conducts taint analysis over the Pin-logged trace and collects a list of tainted instructions. This is a classic optimization to reduce trace length, also adopted in previous works [133, 134, 151]. Our type inference is performed on the tainted trace, as illustrated in Figure 2.4.

2.4.3 Type Inference Rules

CaType implements a comprehensive set of type inference rules over each encountered x86 assembly instruction to track the propagation of secure-sensitive types and check cache side channels.

Prim-I
$$\Gamma \vdash 0 : \{v : B \mid v = 0 \wedge v : \text{CST}\}$$

Prim-II
$$\Gamma \vdash 1 : \{v : B \mid v = 1 \wedge v : \text{CST}\}$$

Const-Conj.I
$$\frac{\Gamma \vdash e_1 : \{v : B \mid v : \tau_1\} \quad \Gamma \vdash e_2 : \{v : B \mid v = 0 \wedge v : \text{CST}\}}{\Gamma \vdash e_1 \wedge e_2 : \{v : B \mid v = 0 \wedge v : \text{CST}\}}$$

Const-Conj.II
$$\frac{\Gamma \vdash e_1 : \{v : B \mid v : \tau_1\} \quad \Gamma \vdash e_2 : \{v : B \mid v = 1 \wedge v : \text{CST}\}}{\Gamma \vdash e_1 \wedge e_2 : \{v : B \mid v : \tau_1\}}$$

Const-Disj.I
$$\frac{\Gamma \vdash e_1 : \{v : B \mid v : \tau_1\} \quad \Gamma \vdash e_2 : \{v : B \mid v = 0 \wedge v : \text{CST}\}}{\Gamma \vdash e_1 \vee e_2 : \{v : B \mid v : \tau_1\}}$$

Const-Disj.II
$$\frac{\Gamma \vdash e_1 : \{v : B \mid \tau_1\} \quad \Gamma \vdash e_2 : \{v : B \mid v = 1 \wedge v : \tau_2\}}{\Gamma \vdash e_1 \vee e_2 : \{v : B \mid v = 1 \wedge v : \text{CST}\}}$$

XOR.III
$$\frac{\Gamma \vdash e_1 : \{v : B \mid v : \text{CST}\} \quad \Gamma \vdash e_2 : \{v : B \mid v : \text{CST}\} \quad e_1 \neq e_2}{\Gamma \vdash e_1 \oplus e_2 : \{v : B \mid v = 1 \wedge v : \text{CST}\}}$$

XOR.IV
$$\frac{\Gamma \vdash e : \{v : B \mid v : \text{CST}\}}{\Gamma \vdash e \oplus e : \{v : B \mid v = 0 \wedge v : \text{CST}\}}$$

Neg.II
$$\frac{\Gamma \vdash e : \{v : B \mid v = 0 \wedge v : \text{CST}\}}{\Gamma \vdash \neg e : \{v : B \mid v = 1 \wedge v : \text{CST}\}}$$

Neg.III
$$\frac{\Gamma \vdash e : \{v : B \mid v = 1 \wedge v : \text{CST}\}}{\Gamma \vdash \neg e : \{v : B \mid v = 0 \wedge v : \text{CST}\}}$$

Figure 2.7: One-bitvector Constant Type Rules.

$$\textsc{Conj\&Disj.I}\quad \frac{\Gamma \vdash e_1 : \{v : B \mid v : \tau_1\} \qquad \Gamma \vdash e_2 : \{v : B \mid v : \tau_2\} \qquad \tau_1 \neq \text{CST} \qquad \tau_2 \neq \text{CST} \qquad \neg(\tau_1 = \text{URA} \wedge \tau_2 = \text{URA}) \qquad \bowtie \in \{\wedge, \vee\}}{\Gamma \vdash e_1 \bowtie e_2 : \{v : B \mid v : \tau_1 \sqcup \tau_2\}}$$

$$\textsc{Conj\&Disj.II}\quad \frac{\Gamma \vdash e_1 : \{v : B \mid v : \text{URA}\} \qquad \Gamma \vdash e_2 : \{v : B \mid v : \text{URA}\} \qquad \bowtie \in \{\wedge, \vee\}}{\Gamma \vdash e_1 \bowtie e_2 : \{v : B \mid v : \text{WRA}\}}$$

$$\text{XOR.I}\quad \frac{\Gamma \vdash e_1 : \{v : B \mid v : \tau_1\} \qquad \Gamma \vdash e_2 : \{v : B \mid v : \tau_2\} \qquad \tau_1 \neq \text{URA} \qquad \tau_2 \neq \text{URA} \qquad \neg(\tau_1 = \text{CST} \wedge \tau_2 = \text{CST})}{\Gamma \vdash e_1 \oplus e_2 : \{v : B \mid v : \tau_1 \sqcup \tau_2\}}$$

$$\text{XOR.II}\quad \frac{\Gamma \vdash e_1 : \{v : B \mid v : \text{URA}\} \qquad \Gamma \vdash e_2 : \{v : B \mid v : \tau\}}{\Gamma \vdash e_1 \oplus e_2 : \{v : B \mid v : \text{URA}\}} \qquad \textsc{Neg.I}\quad \frac{\Gamma \vdash e : \{v : B \mid v : \tau\}}{\Gamma \vdash \neg e : \{v : B \mid v : \tau\}}$$

FIGURE 2.8: One bit B type rules for logical operations.

Type Rules for One Bit Logical Operations. First, type rules that involve CST type are designed to propagate CST in a straightforward way. Figure 2.7 shows the one-bitvector type rules involving the CST type. Rule CONST-CONJ and CONS-DISJ rules handle the situation when the refined type of one operand expression is CST. These four rules are straightforward. Rule XOR.III and XOR.IV describe the refined type CST cases, which are consistent with basic cognition, and the value predicates are given. Rules NEG.II and NEG.III keep security types unchanged while tracking values precisely in types.

Figure 2.8 presents a representative list of type rules for one bit logical operations. Rule CONJ&DISJ.I states that if two operands are not both CST or URA, then the result type is the least upper bound of the two operands' types, which enables the tracking of secure-sensitive values in types. Rule CONJ&DISJ.II handles the circumstance in which both operands are URA. Since the value of the result is no longer distributed uniform-randomly under logic *AND* and *OR*, the result type is lifted on the type hierarchy to WRA. Rule XOR.I is similar to rule CONJ& DISJ.I, where the result type is the least upper bound of the two operands' types, provided that neither bit expression is URA or CST simultaneously. Rule XOR.II states that if one of the operands is of type URA, the result type is URA. This refers

to the fact that random factors can uniformly blind the results through exclusive or ($\oplus$) operations. Rule NEG.I keeps security types unchanged in front of the negation operation.

CONCAT.I

$$\frac{\Gamma \vdash e_1 : \{v : \mathrm{Vec}\langle n_1\rangle \mid v : \tau_1\} \quad \tau_1 \neq \mathrm{URA} \qquad \Gamma \vdash e_2 : \{v : \mathrm{Vec}\langle n_2\rangle \mid v : \tau_2\} \quad \tau_2 \neq \mathrm{URA}}{\Gamma \vdash e_1 \sharp e_2 : \{v : \mathrm{Vec}\langle n_1 + n_2\rangle \mid v : \tau_1 \sqcup \tau_2\}}$$

CONCAT.II-1

$$\frac{\Gamma \vdash e_1 : \{v : \mathrm{Vec}\langle n_1\rangle \mid v : \mathrm{URA}\} \qquad \Gamma \vdash e_2 : \{v : \mathrm{Vec}\langle n_2\rangle \mid v : \tau_2\} \quad \tau_2 \neq \mathrm{SDD}}{\Gamma \vdash e_1 \sharp e_2 : \{v : \mathrm{Vec}\langle n_1 + n_2\rangle \mid v : \mathrm{URA}\}}$$

CONCAT.II-2

$$\frac{\Gamma \vdash e_1 : \{v : \mathrm{Vec}\langle n_1\rangle \mid v : \mathrm{URA}\} \qquad \Gamma \vdash e_2 : \{v : \mathrm{Vec}\langle n_2\rangle \mid v : \mathrm{SDD}\}}{\Gamma \vdash e_1 \sharp e_2 : \{v : \mathrm{Vec}\langle n_1 + n_2\rangle \mid v : \mathrm{SDD}\}}$$

EXTRACTION

$$\frac{\Gamma \vdash e : \{v : \mathrm{Vec}\langle n\rangle \mid v : \tau_e\} \qquad m_1 \leq m_2 \qquad \|[m_1 : m_2]/e\|_t = \tau}{\Gamma \vdash [m_1 : m_2]/e : \{v : \mathrm{Vec}\langle m_2 - m_1 + 1\rangle \mid v : \tau\}}$$

LOGIC.I

$$\frac{\Gamma \vdash e_1 : \{v : \mathrm{Vec}\langle n\rangle \mid v : \tau_1\} \qquad \Gamma \vdash e_2 : \{v : \mathrm{Vec}\langle n\rangle \mid v : \tau_2\} \qquad \bowtie \in \{\wedge, \vee, \oplus\} \qquad \|e_1 \bowtie e_2\|_t = \tau}{\Gamma \vdash e_1 \bowtie e_2 : \{v : \mathrm{Vec}\langle n\rangle \mid v : \tau\}}$$

LOGIC.II

$$\frac{\Gamma \vdash e : \{v : \mathrm{Vec}\langle n\rangle \mid v : \tau\}}{\Gamma \vdash \neg e : \{v : \mathrm{Vec}\langle n\rangle \mid v : \tau\}}$$

ARITH.I

$$\frac{\Gamma \vdash e_1 : \{v : \mathrm{Vec}\langle n\rangle \mid v : \tau_1\} \qquad \Gamma \vdash e_2 : \{v : \mathrm{Vec}\langle n\rangle \mid v : \tau_2\} \qquad \tau_1 \neq \mathrm{URA} \qquad \tau_2 \neq \mathrm{URA} \qquad \neg(\tau_1 = \mathrm{CST} \wedge \tau_2 = \mathrm{CST}) \qquad \bowtie \in \{+, -, \times, \div\}}{\Gamma \vdash e_1 \bowtie e_2 : \{v : \mathrm{Vec}\langle n\rangle \mid v : \tau_1 \sqcup \tau_2\}}$$

ARITH.II-1

$$\frac{\Gamma \vdash e_1 : \{v : \mathrm{Vec}\langle n\rangle \mid v : \mathrm{URA}\} \qquad \Gamma \vdash e_2 : \{v : \mathrm{Vec}\langle n\rangle \mid v : \tau_2\} \qquad \tau_2 \neq \mathrm{SDD} \qquad \bowtie \in \{+, -, \times, \div\}}{\Gamma \vdash e_1 \bowtie e_2 : \{v : \mathrm{Vec}\langle n\rangle \mid v : \mathrm{URA}\}}$$

ARITH.II-2

$$\frac{\Gamma \vdash e_1 : \{v : \mathrm{Vec}\langle n\rangle \mid v : \mathrm{URA}\} \qquad \Gamma \vdash e_2 : \{v : \mathrm{Vec}\langle n\rangle \mid v : \mathrm{SDD}\} \qquad \bowtie \in \{+, -, \times, \div\}}{\Gamma \vdash e_1 \bowtie e_2 : \{v : \mathrm{Vec}\langle n\rangle \mid v : \mathrm{SDD}\}}$$

COMP

$$\frac{\Gamma \vdash e_1 : \{v : \mathrm{Vec}\langle n\rangle \mid v : \tau_1\} \qquad \Gamma \vdash e_2 : \{v : \mathrm{Vec}\langle n\rangle \mid v : \tau_2\} \qquad \neg(\tau_1 = \mathrm{CST} \wedge \tau_2 = \mathrm{CST}) \qquad \bowtie \in \{<, \leq, >, \geq, =, \neq\}}{\Gamma \vdash e_1 \bowtie e_2 : \{v : \mathrm{Vec}\langle 1\rangle \mid v : \tau_1 \sqcup \tau_2\}}$$

COND.I

$$\frac{\Gamma \vdash e : \{v : \mathrm{Vec}\langle 1\rangle \mid v : \mathrm{SDD}\} \qquad \Gamma \vdash e_1 : \{v : \mathrm{Vec}\langle n\rangle \mid v : \tau_1\} \qquad \Gamma \vdash e_2 : \{v : \mathrm{Vec}\langle n\rangle \mid v : \tau_2\}}{\Gamma \vdash e \ ?\ e_1 : e_2 : \{v : \mathrm{Vec}\langle n\rangle \mid v : \mathrm{SDD}\}}$$

COND.II

$$\frac{\Gamma \vdash e : \{v : \mathrm{Vec}\langle 1\rangle \mid v : \tau\} \qquad \tau \neq \mathrm{SDD} \qquad \Gamma \vdash e_1 : \{v : \mathrm{Vec}\langle n\rangle \mid v : \tau_1\} \qquad \Gamma \vdash e_2 : \{v : \mathrm{Vec}\langle n\rangle \mid v : \tau_2\} \qquad \neg(\tau_1 = \mathrm{CST} \wedge \tau_2 = \mathrm{CST})}{\Gamma \vdash e \ ?\ e_1 : e_2 : \{v : \mathrm{Vec}\langle n\rangle \mid v : \tau_1 \sqcup \tau_2\}}$$

FIGURE 2.9: Type rules for expressions involving bitvector $\mathrm{Vec}\langle n\rangle$.

Type Rules for Bitvector Operations. Figure 2.9 depicts the type rules for operations with bitvectors $\text{Vec}\langle n \rangle$. There are three rules applicable to concatenation expressions. Rule CONCAT.I states that the resultant's type takes the least upper bound of the two vectors' type, if both vectors are not URA. Rule CONCAT.II-1 states that type URA is structurally prior to other secret-free types, and CONCAT.II-2 specifies that a bitvector exhibits SDD type if at least one bit in expression e_2 is SDD. Rule EXTRACTION is a well-demonstrated example that leverages function $\|x\|_t$ to determine the refined type of the segment extracted from the source operand. Note that shift operations do not have their own rules as they can be implemented by combining concatenation and extraction operations. Rule LOGIC.I infers a vector type from the types of its constituent bits, i.e., the type of the result is inferred by applying the structural priority defined in Figure 2.6. Rule LOGIC.II is similar to Rule NEG.I.

For the arithmetic operations of two bitvectors, one difference lies in performing the calculation at the whole bitvector level as opposite to each bit. Specifically, we determine the security type of the result, and propagate it to each bit; this offers a sound estimation of each bit's security type. Similar to CONCAT rules, ARITH rules conform to the security type propagation in bitvector structures.

As specified in x86 assembly code, the comparison operation only produces one-bit bitvector $\text{Vec}\langle 1 \rangle$ to the result (i.e., the affected CPU flags). Rule COMP specifies that the resultant's type is the least upper bound of the two operands' types. We omit the case where two operands are both CST as it is straightforward. The last two rules are designed for conditional expressions. We specify two rules according to whether the condition expression e is related to the secret. Rule COND.I states that if the refined security type of the condition expression e is SDD, the result type is SDD regardless of the type of two branch expressions. We clarify that this rule allows CATYPE to keep track of implicit information flow propagated from secret-dependent branch conditions to the instructions. Thus, it facilitates detecting potential cache side channels derived from implicit information flow. In contrast, Rule COND.II takes the least upper bound of two branch expressions' types.

Statement type rules are standard, and we emphasize that CATYPE tracks secrets propagation through both explicit and implicit information flows. We extend the type environment Γ (defined in Section 2.4.1) to track the value and security type

ASSIGN-I

$$\frac{\Gamma \vdash x : \{v : B \mid v : \tau_x\} \quad \Gamma \vdash e : \{v : B \mid v = b \wedge v : \tau_e\} \quad \Gamma' = \Gamma[x \mapsto \{v : B \mid v = b \wedge v : \tau_e\}]}{\Gamma \vdash x \leftarrow e \dashv \Gamma'}$$

ASSIGN-II

$$\frac{\Gamma \vdash x : \{v : B \mid v : \tau_x\} \quad \Gamma \vdash e : \{v : B \mid v : \tau_e\} \quad \Gamma' = \Gamma[x \mapsto \{v : B \mid v : \tau_e\}]}{\Gamma \vdash x \leftarrow e \dashv \Gamma'}$$

LOAD-I

$$\frac{\Gamma \vdash x : \{v : B \mid v : \tau_x\} \quad \Gamma \vdash e_1[e_2] : \{v : B \mid v = b \wedge v : \tau_v\} \quad \Gamma' = \Gamma[x \mapsto \{v : B \mid v = b \wedge v : \tau_v\}]}{\Gamma \vdash x \leftarrow e_1[e_2] \dashv \Gamma'}$$

LOAD-II

$$\frac{\Gamma \vdash x : \{v : B \mid v : \tau_x\} \quad \Gamma \vdash e_1[e_2] : \{v : B \mid v : \tau_v\} \quad \Gamma' = \Gamma[x \mapsto \{v : B \mid v : \tau_v\}]}{\Gamma \vdash x \leftarrow e_1[e_2] \dashv \Gamma'}$$

LOAD-III

$$\frac{\Gamma \vdash x : \{v : B \mid v : \tau_x\} \quad \Gamma \vdash e_1 : \{v : \mathrm{Vec}\langle n\rangle \mid v : \tau_1\} \quad \Gamma \vdash e_2 : \{v : \mathrm{Vec}\langle n\rangle \mid v : \tau_2\} \quad \Gamma \not\vdash e_1[e_2] \quad \tau_r = \tau_1 \sqcup \tau_2 \quad \Gamma' = \Gamma[x \mapsto \{v : B \mid v : \tau_r\}]}{\Gamma \vdash x \leftarrow e_1[e_2] \dashv \Gamma'}$$

STORE-I

$$\frac{\Gamma \vdash e_1 : \{v : \mathrm{Vec}\langle n\rangle \mid v : \tau_1\} \quad \Gamma \vdash e_2 : \{v : \mathrm{Vec}\langle n\rangle \mid v : \tau_2\} \quad \Gamma \vdash x : \{v : B \mid v = b \wedge v : \tau_x\} \quad \Gamma' = \Gamma[e_1[e_2] : \{v : B \mid v = b \wedge v : \tau_x\}]}{\Gamma \vdash e_1[e_2] \leftarrow x \dashv \Gamma'}$$

STORE-II

$$\frac{\Gamma \vdash e_1 : \{v : \mathrm{Vec}\langle n\rangle \mid v : \tau_1\} \quad \Gamma \vdash e_2 : \{v : \mathrm{Vec}\langle n\rangle \mid v : \tau_2\} \quad \Gamma \vdash x : \{v : B \mid v : \tau_x\} \quad \Gamma' = \Gamma[e_1[e_2] : \{v : B \mid v : \tau_x\}]}{\Gamma \vdash e_1[e_2] \leftarrow x \dashv \Gamma'}$$

SEQ

$$\frac{\Gamma \vdash s_1 \dashv \Gamma'' \quad \Gamma'' \vdash s_2 \dashv \Gamma'}{\Gamma \vdash s_1; s_2 \dashv \Gamma'}$$

FIGURE 2.10: Type Rules for Statements.

of each element in a vector, i.e., $\Gamma ::= \emptyset \mid \Gamma, x : \rho \mid \Gamma, e_1[e_2] : \rho$. We use the following rule to derive the type of vector indexing expression:

$$\text{T-Vec-Index}\ \frac{e_1[e_2] \in \Gamma}{\Gamma \vdash e_1[e_2] : \rho}$$

Figure 2.10 shows the type rules for statements. It tracks values in a flow-sensitive way. The type judgment is in the form $\Gamma \vdash S \dashv \Gamma'$, meaning that statement S is type-checked under Γ, and produces a new type environment Γ'. The notation $\Gamma[x \mapsto \rho]$ overrides x's type with ρ in Γ if x is in Γ; otherwise extends Γ with $[x \mapsto \rho]$. Rules Assign-I and Assign-II are for assignments, with and without the presence of a value predicate respectively. Rule Assign-I updates variable x's value predicate with the one on the right-hand side, enabling precise tracking values in types. Rules Load and Store are used for reading from and writing into memories, where memory access safety is assumed. Rules Load-I and Load-II retrieve the type of the vector e_1 at index e_2 and update x's type with it if $e_1[e_2]$ is already tracked in Γ. Otherwise, x's type is updated with $\tau_1 \sqcup \tau_2$ (rule Load-III), which is capable of tracking implicit information flow. Rules Store-I and Store-II update the type of the element $e_1[e_2]$ with x's type. Rule Seq checks the first instruction S_1 under Γ and produces a new type environment Γ'', under which, instruction S_2 is checked.

Proposition 2.1. *Our type system guarantees security-safety statically: if an expression e is given the type $\{v : T \mid v : \tau\}$, then the type of its runtime value will be at least at level τ on the type hierarchy.*

That is, the type system in CaType is sound, and it does not make any false negatives in its analysis; see further discussions and empirical results about type system correctness in Section 2.7.

2.4.4 Cache Side-channel Detection

Section 2.2.2 has illustrated two representative forms of cache side channels, i.e., SDMA and SDBC. When performing type inference, CaType will check each

encountered memory access or conditional jump instruction to see if cache side channels exist. Specifically, to check if a memory access leads to SDMA, we right shift the variable holding memory address by L bits, and decide if the resulting variable is of SDD type. Following a common setup [133, 134, 151], L equals 6, standing for 64-byte (2^6) cache line size on modern CPUs.

For SDBC, previous research [148, 151] merely checks if different secrets induce distinct executing branches. In contrast, CaType checks if the conditional expression is of SDD type, and further assures two branches are not within identical cache lines. Recall as shown in Figure 2.4, we disassemble the cryptography software executable and recover the control flow structure. At this step, we compute the covered cache units of two branches: a SDBC is confirmed, in case the condition is of SDD type, and two branches are placed within distinguishable (at least one non-overlapping) cache lines.

An Illustrative Example. We use an example from the OpenSSL library to visually demonstrate the type inference and detection of side channels. With respect to code in Figure 2.11, we present the corresponding (simplified) type inference procedure launched by CaType in Table 2.1. The first and second columns report the applied type inference rules and the refinement types of relevant variables. The last column reports the relevant cache line layout: MA(a) represents a secret-dependent memory access, and we also report the accessed cache line. BC(a, b, c) indicates that for a conditional control transfer the `if` branch starts at virtual address a (ends at address b), whereas the `else` branch starts at b and ends at c. We also report the accessed cache lines in the last column ("c-line").

Before analysis, users mark `eax` as "secrets" (type SDD). With type inference applied, CaType identifies one SDMA and two SDBC (marked in red). As shown in the last column, for the memory address of the SDMA, CaType checks that the refinement type of highest $32 - L$ bits is of SDD type. As for those two SDBC cases, in addition to checking the branch condition's type is SDD, CaType further checks whether the `if` and `else` branches are located within distinguishable cache lines. CaType confirms all three cases as vulnerable to cache side channels, whose findings are aligned with [133, 134].

TABLE 2.1: Type inference over sample assembly code. To ease reading, we use K, I, W, and U to term refinement type predicates, corresponding to SDD, SID, WRA, and URA types. $\{K\}^{32}$ means bit K repeats 32 times, while $\{1\}^{16}$ means bit 1 repeats 16 times. "c-line" stands for cache line.

Involved refinment types	Applied rules	Control-flow & cache lines
$eax = \{K\}^{32} : SDD$		
$eax = \{K\}^{16}\{0\}^{16} : SDD,\ r_0 = \{1\}^{16}\{0\}^{16} : CST$	LOGIC.I, CONJ&DISJ.I, CONST-CONJ.I&II	
$eax = \{K\}^{16}\{0\}^{16} : SDD,\ r_0 = \{K\}^{16}\{0\}^{16} : SDD,$ $zf = \{K\} : SDD$	LOGIC.I, CONJ&DISJ.I, CONST-CONJ.I	
`je` condition $(zf) \longrightarrow$ secret-dependent		BC(8049629,8049661,804968c) true branch $\longrightarrow$ c-line 201258 false branch $\longrightarrow$ c-line 201259 20125a
$eax = \{K\}^{32} : SDD$		
$eax = \{K\}^{8}\{0\}^{24} : SDD,\ r_0 = \{1\}^{8}\{0\}^{24} : CST$	LOGIC.I, CONJ&DISJ.I, CONST-CONJ.I&II	
$eax = \{K\}^{8}\{0\}^{24} : SDD,\ r_0 = \{K\}^{8}\{0\}^{24} : SDD,$ $zf = \{K\} : SDD$	LOGIC.I, CONJ&DISJ.I, CONST-CONJ.I	
`je` condition $(zf) \longrightarrow$ secret-dependent		BC(8049635,804964b,804965f) true branch $\longrightarrow$ c-line 201258 false branch $\longrightarrow$ c-line 201259
$eax = \{K\}^{32} : SDD$		
$eax = \{0\}^{24}\{K\}^{8} : SDD,\ r_0 = 24 : CST$	EXTRACTION, CONCAT.I	
$eax = \{0\}^{24}\{K\}^{8} : SDD,\ r_0 = 135332960 : CST,$ $r_1 = \{0\}^{4}\{1\}\{0\}^{6}\{1\}\{0\}^{3}\{1\}\{0\}^{5}\{1\}\{0\}^{2}\{K\}^{8} : SDD,$ memory address $(r_1) \longrightarrow$ secret-dependent	ARITH.I, CONCAT.I	MA(804963b) destination $\longrightarrow$ c-line 0x201258$\cdots$
$eax = \{0\}^{24}\{K\}^{8} : SDD,\ r_0 = \{0\}^{24}\{1\}^{8} : CST$	LOGIC.I, CONJ&DISJ.I, CONST-CONJ.I&II	
$eax = \{0\}^{24}\{K\}^{8} : SDD,\ r_0 = 24 : CST$	ARITH.I, CONCAT.I	
		BR(8049649,8049691)

† r_0 and r_1 represent temporary variables. ‡ zf represents Zero Flag register.

```
804961d: mov eax, ptr [ebp+0x8]
8049620: and eax, 0xffff0000
8049625: test eax, eax
// secret-dependent condition
8049627: je 8049661
8049629: mov eax, ptr [ebp+0x8]
804962c: and eax, 0xff000000
8049631: test eax, eax
// secret-dependent condition
8049633: je 804964b
8049635: mov eax, ptr [ebp+0x8]
8049638: shr eax, 0x18
// secret-dependent mem access
804963b: mov al, ptr [eax+0x8110460]
8049641: and eax, 0xff
8049646: add eax, 0x18
8049649: jmp 8049691
```

FIGURE 2.11: BN_num_bits_word.

2.5 Implementation

CATYPE is implemented in Scala, and presently performs analysis on cryptography software executables compiled on 32-bit x86 platforms. However, extending CATYPE to other platforms, e.g., 64-bit x86, is not complex. See discussion in Section 2.7. As a common practice for trace-based analysis, we use Pin [170] to log each covered instruction and its associated execution context, including all values in CPU registers. These logged contexts are used to compute the concrete values of pointers in the follow-up static analysis phase. In other words, our type inference phase employs a practical and common memory model [133, 172], such that we decide the addresses stored in a pointer using their concrete values logged on the trace.

We use *objdump* to disassemble executable files of cryptography software, and recover the control flow graph over the disassembled assembly code. Currently, when encountering an indirect jump, we conservatively consider that it can jump to any legitimate control transfer destinations in the disassembled assembly code. For each conditional jump, we collect the memory address ranges of its `if/else` branches from the disassembled code. We build a lookup table over these control

transfer information when checking if executing secret-dependent branches can visit different cache lines.

Usage of CaType. To use CaType, users need to manually identify the secrets and random factors like blinding in assembly code of cryptography software. As noted in Section 2.3.1, CaType is designed primarily for cryptography software developers, who have detailed knowledge of their own code. Note that the knowledge of sensitive data in cryptography binary code is generally assumed by previous side channel detectors, as most of them analyze binary code [133, 134, 147, 151].

We clarify that, as existing works [133, 134, 151], flagging secret (e.g., RSA private key) only requires mundane reverse engineering of cryptography executable and marking memory buffers that store keys. To date, disassemblers are mature for processing cryptography executables. Moreover, to ease the localization of secrets/random factors in assembly code, we recommend developers to compile cryptography software with debug information attached. We observe that it takes less than 30 minutes to flag the secrets for each of our evaluated cryptography software. Other than manually localizing secrets, all follow-up analyses are done automatically by CaType, whose outputs would be localized vulnerable points in assembly code, as illustrated in Table 2.1. Then, developers will need to map those leakage assembly instructions to source code for diagnosis and patching. To ease mapping assembly instructions to source code, it is also suggested to compile binary code with debug information attached, thereby encoding source code line number into assembly instructions.

In addition, we do not particularly mark certain one-way functions on the execution trace, e.g., functions applying key blinding over secrets. Instead, we assign refined types (URA) to random data before the analysis, and whenever keys are used together with blinding, refined types for secrets and blinding will naturally fit their corresponding type inference rules (as defined in Figure 2.8 and Figure 2.9). Therefore, we should not miss any one-way function provided that random data has been marked correctly before the analysis.

2.6 Evaluation

2.6.1 Evaluation Setup

We evaluate CaType on production cryptosystems. Evaluations are conducted in Ubuntu 16.04 with Intel Xeon 3.50GHz CPU, 32GiB RAM. We collect execution traces of algorithms including RSA, ElGamal, and (EC)DSA from OpenSSL and Libgcrypt (see Table 2.2). $*$ represents using random factor on plaintext/ciphertext and $\star$ indicates using random factor on secrets. Besides, we evaluate the effectiveness of CaType on a constant-time dataset offered in Binsec/Rel [120]. This will validate the correctness of our methodology to a reasonable extent.

Table 2.2: Cryptosystems analyzed by CaType.

Algorithms	Implementations	Versions
RSA	OpenSSL	1.0.2f^*, 1.1.0g^*, 1.1.0h^* 1.1.1n^*, 3.0.2*
	Libgcrypt	1.6.1*, 1.7.3*, 1.9.4**
ElGamal	Libgcrypt	1.6.1, 1.7.3*, 1.9.4**
(EC)DSA	OpenSSL	1.0.1e, 1.1.0g, 1.1.0$i^\star$ 1.1.1$n^\star$, 3.0.2*

The RSA/ElGamal algorithms from both libraries leverage the built-in secret generation function for generating 2048-bit secrets. The ECDSA algorithm adopts OpenSSL *sect571r1* curve. We initiate the plaintext or the message to be signed as "hello world". We use Intel Pin to log the execution traces when executing the cryptography software for standard decryption/signature procedures, including the majority of asymmetric encryption functions such as modular exponentiation in RSA/ElGamal and point multiplication in the signature procedure of ECDSA.

2.6.2 Results Overview

Vulnerability Detection. We present the positives reported by CaType in Table 2.3. We report that CaType confirms all cache side channel vulnerabilities that have been found by CacheD/CacheS. Moreover, it identifies new defects that were neglected in previous analyses of the same cryptography software. CaType detects precisely 485 information leakage sites, including 440 known sites and 45

TABLE 2.3: Identified Information Leakage Sites/Units by CATYPE. We compare the results with recent works, including CacheD, CacheS and DATA.

Cryptography setup	Leakage Sites (known/unknown)	Leakage Units (known/unknown)	CacheD reported [133]	CacheS reported [134]	DATA reported [147, 173]
			Leakage Sites/Units†	Leakage Sites/Units†	Leakage Units‡
RSA-OpenSSL 1.0.2f	30/0	6/0	2/2	6/3	4
RSA-OpenSSL 1.1.0g	30/4	8/1	-	-	5
RSA-OpenSSL 1.1.0h	22/0	5/0	-	-	5
RSA-OpenSSL 1.1.1n	9/0	5/0	-	-	3
RSA-OpenSSL 3.0.2	9/4	4/2	-	-	2
RSA-Libgcrypt 1.6.1	31/4	9/1	22/5	40/11	-
RSA-Libgcrypt 1.7.3	24/4	8/1	0/0	0/0	-
RSA-Libgcrypt 1.9.4	4/5	2/3	-	-	-
ElGamal-Libgcrypt 1.6.1	31/4	9/1	22/5	40/11	-
ElGamal-Libgcrypt 1.7.3	24/4	8/1	0/0	0/0	-
ElGamal-Libgcrypt 1.9.4	3/0	1/0	-	-	-
ECDSA-OpenSSL 1.0.1e	98/0	9/0	-	-	9
ECDSA-OpenSSL 1.1.0g	49/0	6/0	-	-	6
ECDSA-OpenSSL 1.1.0i	13/0	3/0	-	-	3
ECDSA-OpenSSL 1.1.1n	14/0	2/0	-	-	2
ECDSA-OpenSSL 3.0.2	14/0	2/0	-	-	3
DSA-OpenSSL 1.1.0i♮	0/4	0/1	-	-	-
DSA(swapped)-OpenSSL 1.1.0i♮	9/4	4/1	-	-	-
DSA-OpenSSL 1.1.1n	13/4	3/1	-	-	3
DSA-OpenSSL 3.0.2	13/4	3/1	-	-	3
total	440/45	97/14	46/12	86/25	48

† The RSA and Elgamal from Libgcrypt library are counted together in CacheD [133] and CacheS [134].
‡ We collect all leaky functions reported in DATA [147, 173] and locate whether these leaky functions appear in the corresponding OpenSSL version.
♮ DSA (OpenSSL-1.1.0i) and its swapped patch are only evaluated for the key blinding part.

newly found sites. To better characterize findings, we adhere to CacheD/CacheS to group adjacent leakage sites (assembly instructions) into a unit and eliminate duplicated units. This way, 97 known units are confirmed and 14 unknown units are discovered. Works [147, 173] only report leakage units, which are compared here. We elaborate on the findings of CaType in the following two subsections.

Also, for the constant-time dataset offered by Binsec/rel [120], CaType has *no* positive findings, meaning that CaType (over this dataset) does not produce false positives or false negatives. We notice that constant-time computations in this dataset (e.g., comparison and conditional selection) extensively use bitwise operations. Since CaType performs bit-level type inference, CaType manifests high accuracy without treating safe bitwise operations as vulnerable. Note that constant-time operations provided in this dataset are frequently used in modern cryptography libraries; thus, experiments on this dataset verify the correctness of CaType to a reasonable extent.

Analysis Against Randomization. CaType is evaluated against blinding over plaintext/ciphertext and keys. CaType confirms that the secret leakage exists in OpenSSL-1.0.2f and Libgcrypt-1.6.1/1.7.3, notwithstanding the introduction of plaintext/ciphertext blinding. Note that secrets are still exposed to side channels without blinding in these cases. In contrast, key blinding mitigates most leakage sites. For instance, evaluations of RSA/ElGamal in Libgcrypt-1.9.4 reveal that secrets are now labeled as random data (with type URA) by CaType. However, this protection is at the cost of introducing extra (potentially vulnerable) procedures to perform blinding. CaType discovers five new leakage sites in RSA/Libgcrypt-1.9.4. These leakage units cover both the private key d and the prime p (recall in RSA, d and p are secrets). Therefore, we show that though key blinding obscures secrets, it introduces new leakage sites due to extra calculations. In sum, by considering random factors with specific refined types, CaType can analyze side channel mitigation techniques implemented in modern cryptography software.

Performance Evaluation. We compare CaType with CacheD and CacheS by using the same cryptography implementations, and report the comparison results in Table 2.4 (first five rows). For cryptography libraries evaluated by CacheD/CacheS (with a total of 4.4M instructions), CaType finishes the analysis with around 120 CPU seconds, and exhibits promising speed across all evaluation settings with no timeout cases. To compare with CacheD/CacheS, we use the processing time per

10 thousand lines as an indicator. CaType handles per 10 thousand lines in 0.27 seconds on average, while CacheD and CacheS require 4.42 and 35.41 CPU seconds, respectively. We also report performance statistics of other RSA evaluation settings in the next rows of Table 2.4. Their trace lengths range between thousands and millions. Figure 2.12 illustrates the approximately linear correlations between trace length and time. Considering the complexity of analyzing real-world cryptosystems, CaType displays a highly promising performance and scalability.

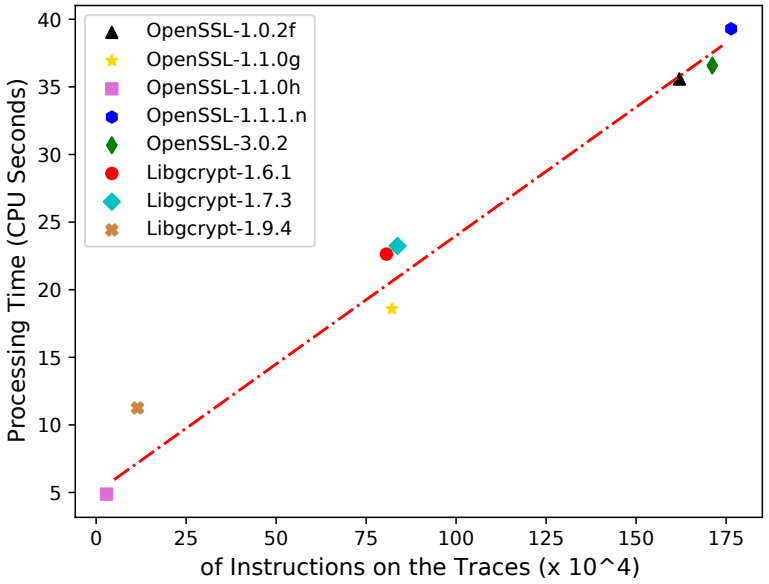


FIGURE 2.12: Trace lengths/processing time towards the analysis of RSA implementations.

The performance comparison results (Table 2.4) demonstrate the superiority of type inference as opposed to existing works (e.g., [133, 134, 148, 151]) that use the constraint solver to decide the satisfiability of side channel constraints. Holistically, those works suffer from the accumulation of complex constraints when performing symbolic execution along the trace. In contrast, type inference ensures each deduction step has a straightforward result without huge search space. Overall, without using constraint solving, CaType maintains a comparable analysis capability as those of CacheD/CacheS. As noted in Section 2.4.4, by using bit-level secret tracking (SDD), deciding if secret-dependent memory access leads to cache side channels is recast to essentially recognize SDD in refined types. This pattern match operation is very efficient without undermining soundness.

TABLE 2.4: Performance comparison with CacheD/CacheS. We also list the analysis of eight RSA implementations for scalability assessment.

Cryptography setup	Instructions on the Traces	Processing Time (CPU Seconds)	Time of Per 10^4 Lines	CacheD	CacheS
				Per 10^4 Lines	Per 10^4 Lines
RSA & Elgamal OpenSSL-1.0.2f	1,620,404	35.58	0.22	3.49	21.16
RSA & Elgamal Libgcrypt-1.6.1	1,379,652	36.00	0.26	4.93	45.36
RSA & Elgamal Libgcrypt-1.7.3	1,411,081	48.40	0.34	3.92	54.57
total (first three rows)	4,411,137	119.98	0.27	4.42	35.41
RSA-OpenSSL 1.0.2f	1,620,404	35.58	0.22	-	-
RSA-OpenSSL 1.1.0g	822,151	18.58	0.22	-	-
RSA-OpenSSL 1.1.0h	28,874	4.88	1.69	-	-
RSA-OpenSSL 1.1.1n	1,763,970	39.29	0.22	-	-
RSA-OpenSSL 3.0.2	1,711,746	36.57	0.21	-	-
RSA-Libgcrypt 1.6.1	806,410	22.63	0.28	-	-
RSA-Libgcrypt 1.7.3	837,215	23.23	0.27	-	-
RSA-Libgcrypt 1.9.4	114,733	11.25	0.98	-	-

2.6.3 Discussion of Known Vulnerabilities

RSA/ElGamal-Libgcrypt. CaType confirms all vulnerabilities reported by prior works CacheD/CacheS in the RSA/ElGamal implementations from Libgcrypt-1.6.1, which adopts pre-computation tables for the sliding-window exponentiation.

```
1 void _gcry_mpi_powm(gcry_mpi_t res, gcry_mpi_t base,
2   gcry_mpi_t expo, gcry_mpi_t mod){
3   ···
4   e = ep[i];
5   count_leading_zeros(c, e);
6   e = (e ≪ c) ≪ 1;
7   ···
8   e0 = (e ≫ (BITS_PER_MPI_LIMB − W));
9   count_trailing_zeros(c0, e0);
10   e0 = (e0 ≫ c0) ≫ 1;
11   ···
12   base_u = b_2i3[e0 − 1];
13   base_u_size = b_2i3size[e0 − 1];
14   ···
15 }
```

Figure 2.13: RSA/ElGamal information leaks found in Libgcrypt-1.6.1.

Figure 2.13 demonstrates the sliding-window implementation. Before the sliding-window algorithm, two lookup tables are constructed. The first table stores the modular exponentiation values of various bases and the second one stores the length of the corresponding value. In the main loop of modular exponentiation, symbol `e` represents the element of secret array and `e0` represents each sliding-window of `e`. Then `e0` is used to access the pre-computation tables. Intuitively, with a PRIME-PROBE attack, different cache sets are observed accessed under different sliding-window values, eventually leaking the secret `e`.

Although Libgcrypt-1.7.3 employs a direct computation scheme rather than using pre-computation tables, CaType still finds 24 leakage sites that leak the secret length, which also exist in Libgcrypt-1.6.1. However, no leaks are reported in CacheD/CacheS about Libgcrypt-1.7.3. In the CacheS paper, they admit these leak points are false negatives of their tool. As Libgcrypt-1.9.4 adopts a new algorithm (i.e., left-to-right exponentiation), CaType reports a known secret length leakage in function _gcry_mpih_add_n, whereas prior leak operations are discontinued.

RSA-OpenSSL. Concerning OpenSSL, CaType first confirms the existence of CVE-2018-0737, where RSA private key is leaked during key generation, in functions BN_gcd and BN_mod_inverse from OpenSSL-1.1.0g/1.1.0h. When analyzing modular inverse, CaType detects a new vulnerability in the function BN_rshift1 that discloses the length of the secret (see Section 2.6.4). A recently found vulnerability comes from function BN_num_bits_word, reported in CacheD/CacheS.

```
1 int BN_num_bits(const BIGNUM *a){
2   int i = a->top - 1;
3   bn_check_top(a);
4   if (BN_is_zero(a)) return 0;
5   return ((i * BN_BITS2) + BN_num_bits_word(a->d[i]));
6 }
7 int BN_num_bits_word(BN_ULONG l){
8   static const char bits[256] = {
9     0,1,2,2,3,3,3,3,4,4,4,4,4,4,4,4,
10      ...
11    8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,
12  };
13  if (l & 0xffff0000L){
14    if (l & 0xff000000L) return bits[l >> 24] + 24;
15    else return bits[l >> 16] + 16;
16  }
17  else
18    if (l & 0xff00L) return bits[l ≫ 8] + 8;
19    else return bits[l];
20  }
21 }
```

Figure 2.14: RSA information leaks found in OpenSSL-1.0.2f.

```
1 BN_window_bits_for_ctime_exponent_size(b) \
2       ((b) > 937 ? 6  :  \
3       (b) > 306 ? 5  :  \
4       (b) > 89 ? 4  :  \
5       (b) > 22 ? 3  : 1)
```

Figure 2.15: Window size of modular exponentiation.

The BN_num_bits_word is called by BN_num_bits that counts the number of bits of a secret (see Figure 2.14). The secret is stored in a BIGNUM struct, where the key value is stored in a byte array `a->d` of 32-bit element, and the length of the array is stored in `a->top`. The number of bits of the last element requires

determined separately because its valid bits may be less than 32 bits. Therefore, function BN_num_bits_word refers to a lookup table to determine the number of bits of the last element of the secret array. Different last elements lead to different entries of the lookup table being accessed. Meanwhile, the branches further narrow down the secret length. Thus, it is a combined vulnerability of memory access and branch. CaType performs the type deduction process in Table 2.1. The issue exists in OpenSSL-1.0.2f/1.1.0g/1.1.0h and has been fixed [174], hence disappears in the latest OpenSSL versions (OpenSSL-1.1.1n/3.0.2).

CaType confirms a secret length leakage in function BN_window_bits_for_ctime_exponent_size in all analyzed OpenSSL versions, shown in Figure 2.15. The issue is also reported in CacheS, but is not fixed in the latest OpenSSL. CaType also detects a vulnerability reported in DATA, where constant-time flags of RSA secret primes p and q are not propagated to the temporary copies inside the function BN_MONT_CTX_set during the Montgomery initialization for modular inverse. This issue exists in OpenSSL-1.0.2f, but the other four OpenSSL libraries resolve it.

ECDSA-OpenSSL. When evaluating the (EC)DSA implementations, we mark the `nonce` used in Montgomery ladder as a secret. This is because the leaky nonce can result in the Hidden Number Problem (HNP) [175, 176], where collecting enough leaky nonce contributes to the recovery of private keys through constructing lattice [177–179]. CaType confirms a direct leakage of the nonce in the Montgomery ladder implementation from OpenSSL-1.0.1e. This vulnerability was reported in [142], and this flaw (CVE-2014-0076) has been fixed by the developers and implemented in a non-branch commit [180, 181]. Recently, Ryan [143] reports a vulnerability located in modular reduction of (EC)DSA implementations in OpenSSL that uses an early abort condition to estimate the range of private keys. CaType confirms this vulnerability comes from function BN_ucmp and BN_usub inside function BN_mod_add_quick.

ECDSA performs the second step of signature as $s = (r \cdot priv\_key + m)\ mod\ order$ (see Figure 2.16), where $r \cdot priv\_key$ represents the result of the first step that multiplies part of the signature r with the private key $priv\_key$. Before the addition operation of the second step, the value of $r \cdot priv\_key$ ensures to be reduced into the range [0, $order$-1]. By observing whether a reduction behaves after the addition

operation inside the second step (function BN_mod_add_quick), an attacker can deduce the range information of the private key *priv_key*.

```
1 ECDSA_SIG *ossl_ecdsa_sign_sig(···){
2   ···
3   do{
4     ···
5     if (!BN_mod_mul(tmp, priv_key, ret->r, order, ctx)){
6       ECerr(EC_F_OSSL_ECDSA_SIGN_SIG, ERR_R_BN_LIB);
7       goto err;
8     }
9     if (!BN_mod_add_quick(s, tmp, m, order)){
10      ECerr(EC_F_OSSL_ECDSA_SIGN_SIG, ERR_R_BN_LIB);
11      goto err;
12    }
13    ···
14  }
15  while(1);
16  ···
17 }
18 int BN_mod_add_quick(BIGNUM *r,
19   const BIGNUM *a, const BIGNUM *b,
20   const BIGNUM *m){
21   if (!BN_uadd(r, a, b))
22     return 0;
23   if (!BN_ucmp(r, m) >= 0)
24     return BN_usub(r, r, m);
25   return 1;
26 }
```

FIGURE 2.16: ECDSA information leaks found in OpenSSL-1.1.0g.

```
1 if (!BN_add(r, k, order)
2   || !BN_add(X, r, order)
3   || !BN_copy(k, BN_num_bits(r) > order_bits ? r : X))
4   goto err;
```

FIGURE 2.17: Bignumber resize.

CATYPE is also evaluated on analyzing the lifetime of a nonce, including the generation, scalar multiplication, modular inversion, and main signing process. The leakage sites identified by CATYPE fully cover the findings reported in Weiser et al.'s paper [173]. For example, by distinguishing whether an extra limb is used to expand the representation of nonce in BN_add, CATYPE confirms the padding

resize vulnerabilities about the nonce reported in CVE-2018-0734 for DSA and CVE-2018-0735 for ECDSA, as shown in Figure 2.17. The vulnerability states that the result buffer resizes one more limb to hold the result. By distinguishing the resize operations, attackers can learn the range information of the nonce. Other known leakage sites of the nonce (e.g., skipping leading zero limbs through bn_correct_top, performing an early stop in BN_cmp, and conditional branches in BN_mul) are identified by CaType; they still exist in the latest versions. Individually, CaType reports non-constant-time vulnerabilities in OpenSSL-1.0.1e when performing ECDSA nonce modular inverse. This is because the constant-time flag was not set to the nonce. OpenSSL-1.1.0g/1.1.0i, on the other hand, implement Fermat's little theorem via constant-time modular exponentiation. Benefit to the cache layout checking, CaType finds four new leakage sites that reveal the secret key size through a series of `else/if` branches in DSA from OpenSSL-1.1.0i/1.1.1n/3.0.2 (see Section 2.6.4). Contrary to our expectations, CaType does not mark cases in the `switch` statement of BN_copy as vulnerable. Through rechecking the source code and its disassembly code, we confirm that CaType performs a correct inference because the trace on the cache cannot be distinguished (see Section 2.6.6).

2.6.4 Unknown Vulnerabilities

CaType finds new vulnerable program points in Libgcrypt-1.6.1/1.7.3 that have been analyzed by existing tools. It finds that the size of secret exponentiation is leaked through the `if/else` statements at the beginning of function _gcry_mpi_powm, as shown in Figure 2.18. The sliding-window size `W` is determined by the size of secret exponent `esize`. Different execution traces of the `if/else` statements can be differentiated because it occupies multiple cache lines. However, we admit that the `if/else` statements are a moderate leakage because only line 1 and line 5 can be distinguished directly. CaType cannot distinguish execution between line 2 and line 4.

We find a new vulnerability in the OpenSSL function BN_rshift1 which performs Greatest Common Divisor (GCD) using the Euclid algorithm. Figure 2.19 presents the source code from version 1.1.0g. We first demonstrate how this function leaks the length of the one-shifted-right operand. Function BN_rshift1 performs shifting

```
1 if (esize * BITS_PER_MPI_LIMB > 512) W = 5;
2 else if (esize * BITS_PER_MPI_LIMB > 256) W = 4;
3 else if (esize * BITS_PER_MPI_LIMB > 128) W = 3;
4 else if (esize * BITS_PER_MPI_LIMB > 64) W = 2;
5 else W = 1;
```

FIGURE 2.18: Window size selection.

```
1 int BN_rshift1(BIGNUM * r, const BIGNUM * a){
2    ···
3    i = a− > top;
4    ap = a− > d;
5    ···
6    rp = r− > d;
7    t = ap[− − i];
8    c = (t&1)? BN_TBIT : 0;
9    if (t ≫= 1)
10       rp[i] = t;
11    while (i > 0){
12       t = ap[− − i];
13       rp[i] = ((t ≫ 1)&BN_MASK2) | c;
14       c = (t&1)? BN_TBIT : 0;
15    }
16    ···
17 }
```

FIGURE 2.19: BN_rshift1 information leaks found in OpenSSL-1.1.0g.

to the right one-bit for each element of the BN_ULONG structure. The length of the source operand (i.e., `a->top`) is used as the while loop's condition. CaType confirms it as a secret-dependent branch, where the judgment of the while loops and part instructions inside the while loops (lines 11–13) are stored in one cache line and the subsequent instructions until the end of function BN_rshift1 (lines 14–17) are stored in another cache line. Therefore, the trace of the while loops can be distinguished. By probing the while loop condition, the value of `a->top` is inferred as one increment to the number of while loops.

Weiser et al. [182] proposed a page-level attack to recover RSA primes p and q when performing prime testing using BN_gcd. CaType confirms the vulnerability in which four different branches are identified because BN_rshift1 of each branch is at different cache lines. Meanwhile, we argue the length information of the source operand leaked by BN_rshift1 accelerates the recovery in this work [182]. For

example, between two adjacent loop operations ($a_{i+1} = a_i/2,\ a_{i+1} = (a_i - b_i)/2$ or $a_{i+1} = (a_i - b_i)/2,\ a_{i+1} = a_i/2$), one decrement in the latter a_{i+1}'s length indicates that the topmost bit of the former a_{i+1} is one. This deduction helps to reduce the range of intermediate results for each Euclid loop. In addition to BN_rshift1, CaType finds similar leakage in BN_lshift1 from OpenSSL-3.0.2.

```
1 int bn_mul_normal(BN_ULONG * r,
2   BN_ULONG * a, int na, BN_ULONG * b, int nb){
3   ...
4   for(;;){
5     if (--nb <= 0) return;
6     rr[1] = bn_mul_add_words(&(r[1]), a, na, b[1]);
7     if (--nb <= 0) return;
8     rr[2] = bn_mul_add_words(&(r[2]), a, na, b[2]);
9     if (--nb <= 0) return;
10      rr[3] = bn_mul_add_words(&(r[3]), a, na, b[3]);
11      if (--nb <= 0) return;
12      rr[4] = bn_mul_add_words(&(r[4]), a, na, b[4]);
13      ...
14   }
15 }
```

Figure 2.20: bn_mul_normal information leaks found in OpenSSL-1.1.0i.

CaType also finds another vulnerability in the OpenSSL-1.1.0i implementation of DSA (see Figure 2.20). The key blinding mechanism of DSA first multiplies the random factor `blind` and the DSA secret key `dsa->priv_key` by calling the function BN_mul, which calls the function bn_mul_normal to perform a classic multiplication if the length of both operands is less than BN_MULL_SIZE_NORMAL. Figure 2.20 presents a for-loop, where four elements of the secret key as a group are multiplied by `blind`. Then, four branches, where `nb` is the length of the secret key, control whether to end the loop. When `nb` equals zero, the multiplication is complete and the function bn_mul_normal returns. CaType confirms that the secret-dependent branches leak the length of the secret key. By probing different if-conditions present in distinct cache lines, the value of the secret length can be recovered. Such vulnerable operations are found in the latest OpenSSL-1.1.1n/3.0.2.

2.6.5 Discussion about Blinding

As stated in Section 2.6.2, CaType shows that the plaintext/ciphertext blinding cannot eliminate cache side channels, given that secrets themselves are still exposed (e.g., secret-dependent memory accesses and branches in modular exponentiation from Libgcrypt-1.6.1). However, key blinding impedes nearly all leakage. For example, CaType reports no vulnerability in the modular exponentiation from Libgcrypt-1.9.4. By inspecting the type inference outputs, we find that the secret exponent is marked as a random number (URA) through a series of blinding operations before conducting modular exponentiation. However, CaType finds new leakage sites in the blinding process. Considering key blinding in RSA/Libgcrypt-1.9.4, which uses $d\_blind = (d \ mod \ (p-1)) + (p-1) * r$ to mask the secret exponent d before performing modular exponentiation. Here, p represents one RSA prime number and r is the random factor. CaType newly discovers five leakage sites in the subtraction and division operations. They leak the length of the prime number p and secret exponent d. For instance, the function _gcry_mpi_sub_ui is invoked to perform $p-1$ on p. It leaks the length of p whenever the resize operation is performed on the result operand, as well as at other length-related branches.

Apart from the key blinding in Libgcrypt-1.9.4, CaType also explores the effect of different key blinding positions on mitigating cache side channels. For instance, DSA implementation from OpenSSL-1.1.0i applies key blinding b to avoid leaking the private key x as follows:

$$s = (bm + bxr) \ mod \ q \tag{2.2}$$

$$s = s \cdot k^{-1} \ mod \ q \tag{2.3}$$

$$s = s \cdot b^{-1} \ mod \ q \tag{2.4}$$

where statements 2.2, 2.3, and 2.4 are executed sequentially. Swapping statements 2.3 and 2.4 results in different key blinding use, which is applied in a LibreSSL patch [183]. CaType compares the original patch with the swapped one (we manually swap statements 2.3 and 2.4 in OpenSSL-1.1.0i DSA). We find nine additional leakage sites related to the length of the inverse nonce `kinv` in

the swapped patch (see Table 2.3), although the statement 2.3 also leaks the inverse nonce length in the original patch. We argue when executing statement 2.4 first, s does not possess the property of randomization anymore due to $b(m + xr)b^{-1}\ mod\ q \equiv (m + xr)\ mod\ q$. Hence, the nonce inverse `kinv` is exposed to the attacker. The swapped practice is fix in a LibreSSL patch [184].

2.6.6 Reducing False Positives

We explain how CaType reduces false positives by using cache layouts rather than cache states to detect side channels. Considering Figure 2.21, function BN_copy is used by RSA and (EC)DSA. Take (EC)DSA as an example, whose secret nonce is copied from `b` to `a` via BN_copy. In particular, a `switch` statement at line 8 helps skipping the copy of leading zero in `b`. By manually reviewing this function, we would anticipate that certain information about the nonce is leaked by discriminating executed switch cases. However, CaType deems this case as safe.

```
1 BIGNUM *BN_copy
2   (BIGNUM *a,
3   BIGNUM *b){
4   ...
5   /* assign values in groups of 4 */
6   ...
7   switch (b->top&3){
8   case 3 : A[2] = B[2];
9   case 2 : A[1] = B[1];
10   case 1 : A[0] = B[0];
11   case 0 :; }
12   ...}
```

Figure 2.21: BN_copy from the OpenSSL Library.

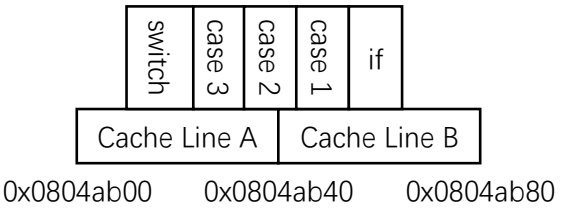


Figure 2.22: Cache layout from OpenSSL-1.1.0g

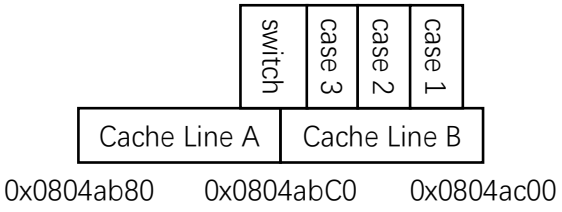


FIGURE 2.23: Cache layout from OpenSSL-1.1.0h

We analyze the result released by CATYPE from the perspectives of cache side-channel attacks, i.e., the FLUSH-RELOAD and PRIME-PROBE attacks. We depict the cache layouts of BN_copy from OpenSSL-1.1.0g and OpenSSL-1.1.0h in Figure 2.22 and Figure 2.23. In these two libraries, the `switch` statement occupies two separate cache lines. Thus, the first cache line must be visited. Meanwhile, instructions after the statement are loaded into the second cache line and are also visited; in an extreme case, the whole `switch` statement is loaded into one cache line. In sum, different switch cases are not distinguishable (e.g., for the FLUSH-RELOAD attack). We further consider whether a PRIME-PROBE attack can distinguish the difference in cache layouts. First, the base addresses are loaded into the cache regardless of whether they correspond to the source array (`A[]`) or the destination array (`B[]`). Second, the largest offset for the element among the last group (both destination and source) is 8 bytes. In that sense, the address of any element is mapped to the same cache line (address $\gg 6$ for 64-byte cache lines). Therefore, PRIME-PROBE cannot collect a distinguishable observation and fails to extract secrets. However, CacheD/CacheS simply treats BN_copy as vulnerable, given that a secret-dependent branch condition (line 8) is (inaccurately) treated as "vulnerable" in the view of their cache state-based vulnerability pattern. However, it is indeed a false positive.

Robustness of Using Cache Layouts. The above experiments are conducted using OpenSSL's default compilation setting. The `switch` statement may be vulnerable, when the code chunk of each switch case occupies distinct cache lines. Overall, we anticipate that different optimization settings could result in placing instructions into different cache lines. To benchmark the robustness of using cache layouts instead of using cache state-based threat models, we measure how compiler optimizations may influence the results of CATYPE, whose results are given in Table 2.5. At this step, we only measure side channels due to SDBC, because we use the cache layout model to check SDBC. Also, given that we need to manually

confirm and compare each finding across different optimizations, we only select a crypto library when its SDBC-related source code has visible changes across different versions. For instance, while we evaluate Libgcrypt 1.6.1, 1.7.3, and 1.9.4 in Table 2.3, we only evaluate versions 1.7.3 and 1.9.4, since version 1.6.1 appears to be identical with 1.7.3 in terms of those SDBC cases flagged by CaType.

Table 2.5 shows that optimizations affect the analysis results, as heavy optimizations tend to "condense" code into fewer cache lines. Similar to Table 2.3, we provide the discovered leakage sites as well as grouped leakage units. CaType can accurately capture the subtle leakage (without making false positives) with its employed cache layout threat model. With manual efforts, we confirm that *all* cases are true positives. Indeed, we report that all `-O2` findings are subsumed by those of `-O0`, and all `-O3` findings are subsumed by `-O2` findings. In contrast, we report that CacheD/CacheS yields *identical* findings across different optimization settings, meaning that they have a considerable number of false positives under `-O2` and `-O3`.

Table 2.5: Branch vulnerabilities identified by CaType under gcc `-O0`, `-O2`, and `-O3` optimization settings.

Cryptography setup	**gcc-5.4**		
	-O0	**-O2**	**-O3**
RSA-OpenSSL 1.1.0g	27/9	24/9	24/9
RSA-OpenSSL 1.1.0h	20/5	18/5	18/5
RSA/Elgamal-Libgcrypt 1.7.3	17/7	14/7	14/7
RSA/Elgamal-Libgcrypt 1.9.4	6/4	6/4	6/4
ECDSA-OpenSSL 1.1.0g	38/6	22/6	19/6
ECDSA-OpenSSL 1.1.0i	10/3	7/3	7/3
ECDSA-OpenSSL 3.0.2	9/2	9/2	9/2
DSA-OpenSSL 1.1.0i	4/1	3/1	3/1
DSA-OpenSSL 1.1.1n	14/4	12/4	12/4
total	145/41	115/41	112/41

2.7 Discussion and Limitation

Type System Benchmarking. Scientifically, it would be ideal to benchmark our refinement type system against some "synthetic datasets" to determine their algorithmic effectiveness and efficiency before evaluating side channel detections,

which is a "downstream" application of our type system. Nevertheless, it is practically hard to find a proper (synthetic) dataset to solely evaluate the type system, and using downstream applications to reflect the effectiveness of a type system is a common evaluation plan used by relevant works [185–187]. To avoid potential confusion, we revisit the effectiveness and efficiency of our type system as follows.

First, our type system is sound (per Proposition 2.1). All typing rules are intuitive, and there are no "tricky" ones implemented in CaType. Thus, the soundness is at ease. Second, in terms of efficiency, our implementation manifests approximately $O(n)$ complexity, where n is the number of instructions in a given trace. CaType is empirically very efficient. As demonstrated in Figure 2.12, CaType manifests a mostly linear growth in terms of the trace length and processing time. Overall, the end-to-end evaluation on side channel analysis illustrates the accuracy of CaType, thereby reflecting the effectiveness of its underlying type systems at large.

Further to the above discussion, we empirically evaluate the type system by comparing it with taint analysis to check correctly-tagged variables. In general, taint analysis offers a holistic modelling of how secrets propagate through the program, while our type system is *more precise*. Most taint analysis implementation is performed at the syntax level (whose cost and accuracy is conceptually similar to conventional, syntax-level type inference). In contrast, as shown in Section 2.3.2 and Figure 2.3, CaType's type system tracks bit-level values/secrets uniformly using refined types; thus, the type system captures stronger semantics properties, e.g., it models how blinding obscures secrets. Therefore, properly masked secrets are not treated as secrets in CaType (i.e., they do not have an SDD type), but taint analysis will "over-taint" them.

Recall CaType first conducts taint analysis over the Pin-logged trace before performing type inference. Thus, we compare the number of tainted registers/memory cells with the number of variables of type SDD over the same trace. Table 2.6 reports the evaluation results. As clarified above and observed in Table 2.6, the number of variables of SDD type is less than the number of tainted variables, as expected. Also, we confirm that *all* variables of type SDD exist in the tainted set, i.e., our type inference phase has no false negatives (when using tainted variables as the baseline). More importantly, we also manually study every "over-tainted" variable that does not have type SDD. As shown in the 6th column of Table 2.6, taint analysis finds considerably more tainted variables than type inference. Given

TABLE 2.6: Checking the correctness of refinement type system in CATYPE by comparing with taint analysis. "FPs" denotes false positives of taint analysis. We randomly select 100 cases for each setting for confirmation except ElGamal/Libgcrypt 1.9.4.

Cryptography setup	Instructions on the traces	Tainted instructions	Tainted registers and memory cells	Registers and memory cells with SDD types	"Over-tainted"	FPs in 100 "over-tainted" cases
RSA-OpenSSL 1.0.2f	1,620,404	4,127	3,271	3,165	106	100
RSA-OpenSSL 1.1.0g	822,151	4,092	3,568	3,452	116	100
RSA-OpenSSL 1.1.0h	28,874	9,672	7,939	5,977	1,962	100
RSA-OpenSSL 1.1.1n	1,763,970	51,933	34,518	30,276	4,242	100
RSA-OpenSSL 3.0.2	1,711,746	56,218	39,799	37,507	2,292	100
RSA-Libgcrypt 1.6.1	806,410	130,141	125,648	100,493	25,155	100
RSA-Libgcrypt 1.7.3	837,215	140,478	133,105	107,731	25,374	100
RSA-Libgcrypt 1.9.4	114,733	102,132	104,536	103,676	860	100
ElGamal-Libgcrypt 1.6.1	573,242	334,024	286,095	184,976	101,119	100
ElGamal-Libgcrypt 1.7.3	573,866	334,194	286,213	185,297	100,916	100
ElGamal-Libgcrypt 1.9.4	4,676	1,274	1,145	1,140	5	5
ECDSA-OpenSSL 1.0.1e	2,277,459	258,261	241,326	237,381	3,945	100
ECDSA-OpenSSL 1.1.0g	415,415	140,596	124,907	110,488	14,419	100
ECDSA-OpenSSL 1.1.0i	298,463	81,988	72,090	55,810	16,280	100
ECDSA-OpenSSL 1.1.1n	182,745	315	220	120	100	100
ECDSA-OpenSSL 3.0.2	164,613	315	220	120	100	100
DSA-OpenSSL 1.1.0i♮	18,516	4,766	3,970	3,371	599	100
DSA(swapped)-OpenSSL 1.1.0i♮	18,608	4,698	3,899	3,230	669	100
DSA-OpenSSL 1.1.1n	1678	578	435	302	133	100
DSA-OpenSSL 3.0.2	1678	578	435	302	133	100
total	12,236,462	1,660,380	1,473,339	1,174,814	298,525	1,905

♮ DSA (OpenSSL-1.1.0i) and its swapped patch are only evaluated for the key blinding part.

the difficulty of manual inspection, for each evaluation setting, we randomly select 100 cases (if there are more than 100 cases). For each case, we comprehend the causality of how variable is tainted, and decide if this is a true positive (meaning that the tainted variable is carrying secrets correctly) or not.

We show the manual inspection results in the last column of Table 2.6. We find that all the "over-tainted" variables are *false positives* of the taint analysis. It is thus correct for our type system to neglect them. Among in total 1,905 randomly selected cases, the "over-tainted" variables belong to the following categories: ① variables of SDD type that have been appropriately masked with blinding, while they are still tainted, ② variables that are further tainted by variables belonging to ①, ③ variables of SDD type that have been zeroized by constants, whereas taint analysis retains the taint label over those variables, and ④ the base address of a secret buffer is deemed as a taint source, such that whenever loading from the base address, the output will be tainted. While ①, ②, and ③ are due to the inherent limitation of standard taint analysis technique, ④ is due to the "clumsy" implementation of our adopted taint analysis tool. We use the taint analysis tool provided by CacheD. Note that ④ eases the implementation of a taint engine, but overestimates secrets. Secrets (and their associated non-secret data) are often stored in a BIGNUM struct. By treating the base address of this struct as the taint source, non-secret data in the struct are all tainted due to ④. Out of 1,905 manually checked cases, we find that about 52% cases fall in ④, whereas the remaining 48% cases are due to ①, ②, or ③. Thus, we estimate that around 143K ($298,525 \times 48\%$) false positives are due to the inherent limitation of taint analysis, which are correctly eliminated by our refinement type system.

Extension. We discuss the extension of CaType from both architectural and analysis target perspectives. First, the current implementation of CaType supports to analyze 32-bit x86 binaries. Given that the closely-related works (e.g., CacheD, CacheS, and CacheAudit) only support 32-bit x86 binaries, supporting the same binary format enables an "apple-to-apple" comparison. Moreover, CaType can be extended to 64-bit binaries with no extra research challenge. We expect to convert each refinement type, currently a 32-bit vector, to a 64-bit vector. We also need to handle new instructions. Nevertheless, these are engineering endeavors rather than open-ended research problems. We leave it as one future work to support other architectures including 64-bit x86.

Also, from the analysis target perspective, side channel analyzers in this field require to flag program secrets (or other sensitive data) specified by users, and then start to analyze their influence on cache. Detectors (including CaType) are *not* limited to crypto software. Analyzing crypto software targeted by previous analyzers, however, makes it easier to compare CaType with them. Given the scalability of CaType, it should be feasible to extend CaType to analyze production software running in TEEs and detect their side channel leaks [96, 188, 189].

2.8 Related Work

Perfect masking analysis conducted on power side channels is highly relevant to our work [190, 191]. In such analysis, all intermediate computation outputs are statistically examined for independence between secret data and power side channels. Recent efforts employ a type-based technique to deduce potentially leakage of program intermediate variables. Specifically, works [192–194] used a syntactic type system that primarily relies on the variable structural information. Works [195, 196] extended the syntax-based approach to a semantic-based type system that refines inference rules for boolean masking scheme analysis. Two improvements [197, 198] added rules for additive and multiplicative masking. These works inspire the design of our refinement type system. However, crucial gaps exist in applying these rules to detect cache side channels. First, perfect masking analysis of software power side channel countermeasures targets specific masked programs (often bitwise operations), whose computation is usually straightforward (calculating and then assigning). Cache side channel analysis targets complicated production cryptosystems. Type systems proposed in prior works are primarily for bitvector logical operations, not general x86 assembly semantics. Second, our tentative exploration shows that earlier typing rules were often incomplete; they may need to use constraint solving when typing rules cannot be applied. Their performance is therefore downgraded. In contrast, CaType's type inference rules completely infer refined types for variables.

2.9 Conclusion

Detecting cache side channels in production cryptographic software is still an open problem. This chapter presents CaType, a refinement type-based tool to deliver highly efficient and accurate analysis of cache side channels over x86 binary code. Evaluation over real-world cryptographic software shows that CaType identifies side channels with high precision, efficiency, and scalability.

Chapter 3

Compiler-aided Mitigation against Side-channels in Trusted Execution Environment

3.1 Introduction

Data security in cloud computing platforms has become a major concern, hindering many data owners from fully leveraging Virtual Machines (VMs) from public service providers. To ensure the confidentiality of operations and data in untrusted cloud environments, major processor vendors have introduced a hardware-based technology known as Trusted Execution Environment (TEE). TEE provides an isolated environment with memory encryption to fortify the integrity and confidentiality of VMs against potential threats posed by privileged attackers, such as malicious hypervisors or host OS. Over the years, many TEE-enabled processors have been released, including AMD Secure Encrypted Virtualization (SEV) [58] and its iterations SEV Encrypted States (SEV-ES) [199] and SEV Secure Nested Paging (SEV-SNP) [200], Intel Software Guard Extensions (SGX) [201, 202], Intel Trust Domain Extensions (TDX) [203] and ARM Confidential Compute Architecture (CCA) [204].

Admittedly, existing TEE processors still exhibit security weaknesses upon deeper analysis by security researchers. For instance, SEV-SNP is designed to mitigate

various known attacks that could undermine the prior versions of SEV, including but not limited to unauthenticated encryption [205–207], Nested Page Table (NPT) remapping [208–210], unprotected I/O [211], and unauthorized Address Space Identifier (ASID) [212]. However, recent works disclosed a new vulnerability in SEV-SNP, i.e., ciphertext side-channel attack [56, 57], enabling a malicious hypervisor to extract the secret keys used in cryptography programs from their encrypted memory. In essence, the ciphertext side-channel attack leverages the deterministic nature of memory encryption in SEV-SNP: a consistent plaintext block at the same physical address is always encrypted into the same ciphertext block. Therefore, the adversary can construct a one-to-one mapping between the plaintext and ciphertext by continuously monitoring changes in the ciphertext of the encrypted guest memory. Researchers have demonstrated successful attacks against mainstream cryptography libraries, such as OpenSSL and WolfSSL, even when they are equipped with constant-time practices. More seriously, this ciphertext side-channel attack is not limited to SEV-SNP, but can threaten any TEE processors utilizing deterministic memory encryption with the memory bus snooping technique [59].

Several approaches have been designed for the detection and mitigation of ciphertext side-channel attacks. CipherH [213] combined dynamic taint tracking and static symbolic execution to identify ciphertext side-channel vulnerabilities in cryptography software. It utilized DataFlowSanitizer (DFSan) [214] to efficiently mark sensitive memory store instructions. Within functions involving these marked instructions, CipherH conducted symbolic execution and constructed formulas checked by a constraint solver to identify potential vulnerabilities. It has successfully detected ciphertext side-channel vulnerabilities in many popular cryptography libraries, including WolfSSL, OpenSSL, and MbedTLS. CipherFix [135] employed dynamic taint analysis to acquire the offset of sensitive memory store instructions in a program. With static instrumentation, it transformed each tainted memory store instruction into a code snippet with masked store in the instrumentation section. Besides, a direct jump instruction was inserted at the original program point, redirecting to the newly created code snippet. CipherFix mitigated ciphertext side-channel leakage at the cost of an average performance overhead ranging from 2.4× to 16.8× in its most efficient variant. Obelix [215], the latest side-channel mitigation tool, leveraged Oblivious RAM (ORAM) to execute and access

enforced union code blocks, making execution flows indistinguishable to attackers, thus protecting both executed code and accessed data. Nevertheless, Obelix suffered from considerable performance overhead inherent to ORAM itself (e.g., hundreds of times), presenting a significant deployment issue.

For the mitigation tool capable of protecting existing cryptography libraries, i.e., CipherFix, the substantial overhead introduced can be attributed to two main factors: the insertion mechanism it employs and the code it inserts. Given that CipherFix conducts static instrumentation at the binary level, it necessitates frequent jumps to the instrumentation code unless the original reference is altered. This results in a large number of missed branch predictions, causing a performance loss to some extent. Moreover, the platforms with inconsistent instruction lengths, where jumps rely on interrupt procedures (e.g., `int3`), exacerbate the performance degradation. Regarding the inserted code, CipherFix requires the placement of initialization code during the program startup. For instance, it monitors explicit allocations of sensitive heap memory through standard library functions like `malloc`. It then generates and zero-initializes another heap area of the same size for storing the mask. Executing this part of the code can be time-consuming, especially if it is called multiple times.

It is essential to explore alternative approaches for enhancing program security with higher efficiency. Instead of relying on code instrumentation as CipherFix or dynamic obfuscation as Obelix, we turn to a different strategy: mitigating side-channel vulnerabilities *at the compilation stage*, which offers several advantages. Firstly, it incorporates security measures into the compilation process in a more integrated and streamlined manner, fixing the potential vulnerabilities directly in the code generation phase. In contrast, instrumentation typically addresses vulnerabilities in a compiled executable, lacking the same level of insight and control over the code generation process. Secondly, the compiler has full analysis capabilities over the entire program, while instrumentation is often constrained to specific locations where probes are inserted, potentially limiting the scope of examination to isolated points within the program. Thirdly, compilers can still utilize various resources and optimizations for generating efficient code during program repair. The instrumentation method, however, can only be applied to fixed executable files, facing constraints in layout adjustment and resource utilization, such as registers.

Thus, in this research, we focus on revisiting ciphertext side-channel vulnerabilities through the compiler itself, including the detection of sensitive memory access instructions and the mitigation of vulnerable points by the compilation process.

To this end, we introduce CipherGuard, an LLVM-based compiler-aided methodology for automated and efficient mitigation against ciphertext side channels. Integrated into the LLVM compilation flow, CipherGuard conducts dynamic taint analysis at the last optimized LLVM-IR level, capturing all sensitive memory access instructions and their precise memory references, including virtual symbols of stack and heap memories. This tainted information aids the LLVM backend in implementing multiple vulnerability mitigation variants. CipherGuard then performs vulnerability mitigation at the LLVM Machine-IR level, employing variants such as software-based probabilistic encryption, which introduces a random nonce to secret data upon memory writes, and secret-aware register allocation, allowing the retention of secret-related variables in registers. The resulted fixed Machine-IR is compiled into a secure executable that can run directly in existing TEEs. With efficient management of random nonce buffers and flexible register allocation, CipherGuard demonstrates a satisfactory performance overhead, averaging 1.76-3.10× across various evaluations of cryptography software. This is a significant performance improvement over CipherFix, highlighting the efficacy of the compiler-aided strategy in enhancing the security against ciphertext side channels. In sum, we make the following contributions:

- For the first time, we introduce a compiler-aided strategy to address ciphertext side channels. Through the exploration and implementation of multiple mitigation variants, e.g., software-based probabilistic encryption and secret-aware register allocation, our compiler-aided strategy has been demonstrated to provide a more efficient and flexible solution in comparison to the instrumentation strategy.
- It presents CipherGuard, an LLVM-based compiler-aided tool that harnesses dynamic taint analysis and deploys vulnerability mitigation variants at the compilation stage. CipherGuard excels in generating more efficient mitigated code through in-place mitigation code insertion, precise buffer management for random nonces, and flexible register allocation.

- It evaluates CipherGuard in mitigating ciphertext side-channel vulnerabilities among real-world cryptography software. CipherGuard hardens all the sensitive memory access instructions with superior performance compared to CipherFix. This demonstrates the high applicability and practicality toward the protection of cryptography software in a compiler-aided manner.

In the following, we present the preliminary knowledge in Section 3.2, and methodology overview in Section 3.3. We demonstrate more details of the design and implementation in Section 3.4 and Section 3.5. We set the experiment and evaluate CipherGuard in Section 3.6. Section 3.7 gives a further discussion on our tool. Finally, Section 3.8 concludes this chapter.

3.2 Background

3.2.1 Ciphertext Side-channel Attacks

The concept of the ciphertext side channel was initially introduced in work [56], which was further expanded upon in study [57] to systematically explore the applicability of this vulnerability. Briefly, the attacker observes the changes in ciphertexts to deduce relationships between successive plaintexts or identify patterns of specific plaintext changes.

The ciphertext side channel originates from the deterministic memory encryption mode implemented in SEV-SNP. The encrypted memory is calculated by an XOR-Encrypt-XOR (XEX) mode, expressed as: $c = ENC(m \oplus T(P_m)) \oplus T(P_m)$, where the plaintext m undergoes the XOR operations before and after AES-128 encryption with a tweak value $T(P_m)$ that incorporates the physical address P_m. The absence of freshness in the encryption process results in the identical ciphertext for the encryption of the same plaintext at a given physical address each time. While the initial examination of the ciphertext side channel primarily targeted AMD's TEE [58], it is crucial to acknowledge that this vulnerability extends to other deterministic encryption-based TEE architectures as long as attackers have read accesses to ciphertext (via software access [56] or memory bus snooping [59]).

Two attack schemes were introduced in work [57]. The *Dictionary* attack involves the continuous monitoring of the ciphertext of a secret variable at a fixed memory

```
1: pbit ← 1;
2: for i ← cardinality_bit - 1 downto 0{
3:     kbit ← BN_is_bit_set(k, i) ∧ pbit;
4:     EC_POINT_CSWAP(kbit, r, s, ...);
5:     ...
6:     pbit ← pbit ∧ kbit;}
```

FIGURE 3.1: ossl_ec_scalar_mul_ladder.

address to construct a dictionary containing mappings of ciphertext-plaintext pairs. Consider the code snippet shown in Figure 3.1, extracted from the ECDSA Montgomery ladder algorithm implemented in OpenSSL-3.0.2. In each loop iteration, the `BN_is_bit_set` function is utilized to obtain one bit of the secret k, determining the conditional swap in the subsequent line. Following this, the *kbit* variable is computed through an XOR operation with the value in *pbit*, which is then written back to *pbit*. Given the dual XOR operations in lines 3 and 6, *pbit* ultimately stores each bit of the secret k. The attacker records consecutive ciphertext pairs (*pbit-kbit*) both before and after the `BN_is_bit_set` function, aiming to deduce k_i in each iteration based on the changes observed in ciphertext pairs. Ultimately, the attacker recovers the whole secret k.

```
1: for i ← 0 to nwords - 1{
2:     t ← (a.d[i] ∧ b.d[i])
3:         & condition;
4:     a.d[i] ← a.d[i] ∧ t;
5:     b.d[i] ← b.d[i] ∧ t;}
```

FIGURE 3.2: BN_constant_swap.

In contrast, the *Collision* attack focuses on identifying repetitions or alterations in certain ciphertexts to break the constant-time mechanism. Figure 3.2 shows one example of such attack against the constant-time-swap function `BN_constant_swap`. This function takes two variables a and b, along with a secret decision *condition* (e.g., *kbit* in line 4 of Figure 3.1. If *condition* is set to 1, the values of a and b are exchanged, leading to observable changes in ciphertext. Conversely, if *condition* is 0, the ciphertext remains unaltered. In this way, the *Collision* attack undermines the constant-time swap functions by recovering the secret decision *condition*.

Currently, many well-known cryptography applications are vulnerable to this attack, including RSA and ECDSA (such as *secp256k1* and *secp384r1*) equipped

with constant-time algorithms, ECDSA from WolfSSL-5.3.0, ECDSA and RSA from MbedTLS-3.1.0, as well as EdDSA (*Ed25519*) from OpenSSH adopted by Ubuntu LTS 20.04 [56, 57]. Given the widespread impact and severity of these vulnerabilities, there is an urgent need to mitigate ciphertext side-channel attacks.

3.2.2 Countermeasures to Ciphertext Side-channels

While hardware-based countermeasures offer stronger security guarantees by fundamentally eliminating ciphertext side channels, they require thorough security validation before chip circuit manufacturing for commercial use. Therefore, we opt to mitigate attacks from the software perspective, which allows for more immediate implementation and deployment without the need for hardware modifications.

Unfortunately, existing countermeasures to cache and timing side channels [5, 7, 11, 13, 141–144] cannot mitigate ciphertext side channels [56, 57, 213]. Specifically, the most prominent countermeasure to these traditional side-channel threats is the concept of constant-time cryptography. Previous efforts adhering to this concept can be categorized into three classes. 1) Researchers verified whether a cryptography program satisfies the constant-time criterion using various approaches, including the program counter model [103–108], observation equivalence-based noninterference [109, 110, 112, 113], and self-composition-based noninterference [114–120]. 2) Conceptually, formally constructing high-assurance cryptography libraries shall fundamentally resolve the constant-time issues, leveraging formal languages like F* [121], HACL* [122], Vale [124], Jasmin [125] and Fact [126]. 3) Transforming existing programs into constant-time equivalents also significantly contributes to resisting side channels. For instance, some approaches [130, 131] executed both real and decoy paths; Constantine [132] leveraged the linearization of control-flow and data-flow.

Without detailed implementation, AMD's whitepaper [216] and Li et al. [57] proposed the following countermeasures to ciphertext side-channels. However, they all have limitations.

- Preserving secret-related variables in registers throughout their lifecycle [57], thereby preventing them from being stored in memory. While limiting the

observation of ciphertexts, this strategy faces some practical challenges for implementation due to finite register resources.

- Avoiding the reuse of fixed memory addresses to ensure the generation of fresh ciphertext and confuse the observation of valid ciphertext [57, 216]. However, this requires additional memory allocation and deallocation, as well as precise runtime data reference management, which could introduce significant performance overhead.

- Introducing a random nonce to the secret data with each memory write operation to enhance the unpredictability of the resulting ciphertext [57], including masking and padding strategies proposed by AMD [216]. Due to the need for extended data structures to store padding values, existing state-of-the-art defense effort, namely CipherFix [135], implements masking through a binary instrumentation approach. However, this method suffers from limitations inherent to instrumentation itself.

To summarize, there is no single software-based scheme that is perfectly suited in both methodology and implementation. Therefore, exploring the optimization of each mitigation method and the combination of various approaches, achieved through different implementations such as the compiler-aided approach, provides valuable insights for the research field of mitigating ciphertext side-channel attacks.

3.3 Methodology Overview

3.3.1 Threat Model

We share the same threat model with prior works [56, 57] where a privileged attacker performs the ciphertext side-channel attack to steal the secrets of the cryptography program running inside a confidential VM, e.g., AMD SEV-SNP. The attacker has full system privileges and is capable of reading the ciphertext of the entire encrypted memory [57]. We also account for single-step attack [217], which aids in selecting optimal observation points in the control flow by controlling procedures within confidential VMs and pausing after each instruction. In this work, we only consider the attack vector of ciphertext side-channels, and assume that

the confidential VM hardware and the entities running inside it (i.e., guest OS, applications) are trusted. It is worth noting that the content of registers could be leaked via ciphertext side channels during the kernel context switch [57]. However, the authors also acknowledged that this could be easily fixed with a kernel patch. Hence, in this work, we assume the registers are secure to protect the secrets.

3.3.2 A Motivating Example

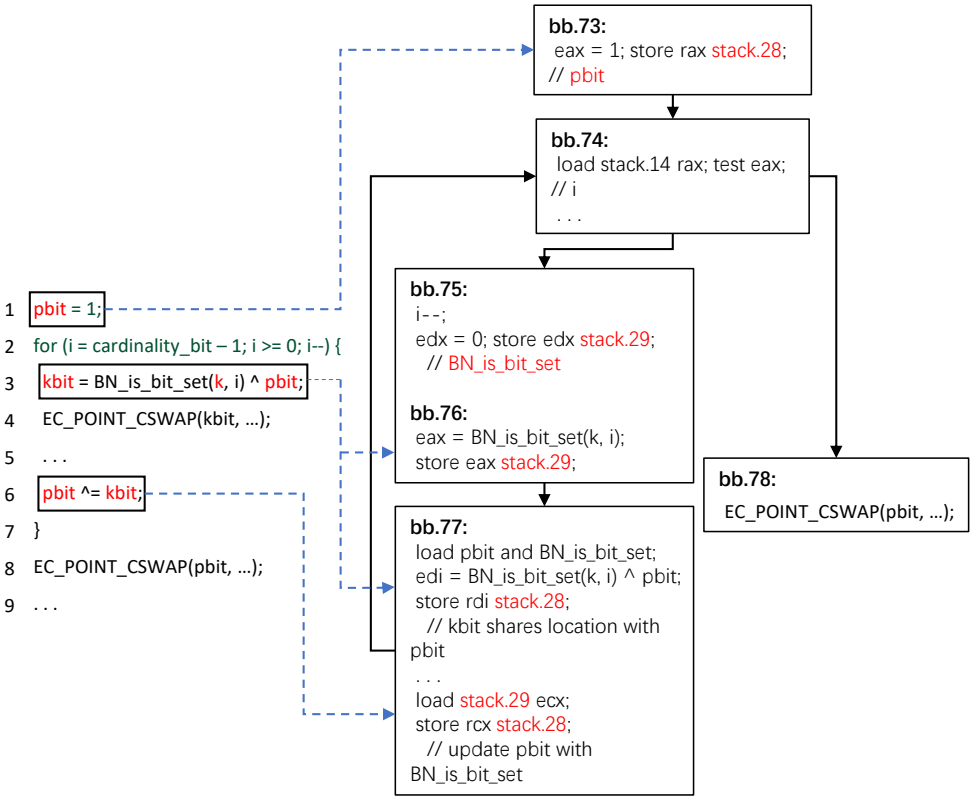


FIGURE 3.3: `ossl_ec_scalar_mul_ladder` and its Machine Basic Block.

We revisit the process of compiling a program using LLVM to illustrate how ciphertext side channels are identified. Figure 3.3 presents the Machine Basic Blocks (MBBs) of the code snippet from the function `ossl_ec_scalar_mul_ladder`, which is handled by the register allocation pass in LLVM. We simplify the representation of these MBBs by only demonstrating the ciphertext side-channel relevant variables. Additionally, we format the first operand as the source and the second as the destination. In *bb.*73 of Figure 3.3, the register *eax* is allocated to hold the variable *pbit*. Since *pbit* is not used immediately or in succession, the compiler optimization

inserts a memory store instruction to write it back to *stack*.28. Subsequently, the compiler inserts a reload instruction for the spilled variable *pbit* once it is involved in operations, as shown in *bb*.77 where *pbit* is reloaded. This gives us the first source of ciphertext side-channel occurrence: the **spilling-reloading** mechanism during compilation generates additional memory writes related to secret-dependent variables, resulting in involuntary leakage. A similar situation arises in *stack*.29, as demonstrated in *bb*.75 and *bb*.76, where the variable directly holds the secret returned from `BN_is_bit_set`. If an attacker gains knowledge of the compilation process, he can directly observe the ciphertext changes of *stack*.29 and potentially recover the entire private key.

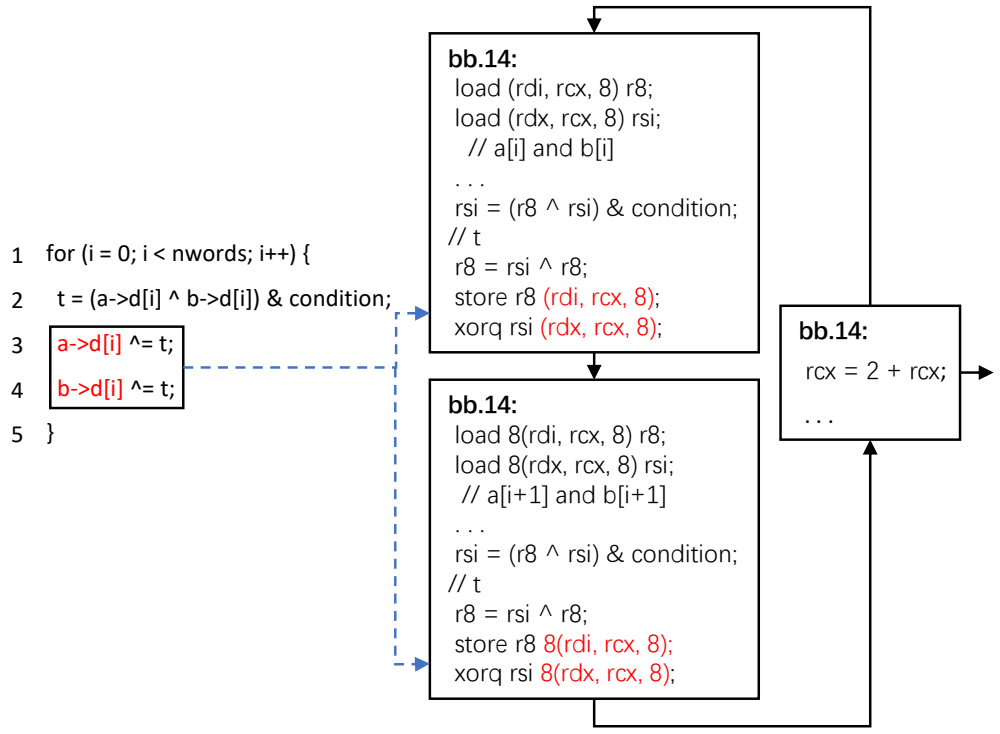


FIGURE 3.4: `BN_constant_swap` and its Machine Basic Block.

Next, we observe another vulnerable pattern, wherein intermediate values are written back to allocated heap areas, as demonstrated by the function `BN_constant_swap` in Figure 3.4. The compiler directly generates store instructions that point to heap memories, using $(rdi, rcx, 8)$ for $a \rightarrow d[i]$ and $(rdx, rcx, 8)$ for $b \rightarrow d[i]$. Naturally, if these data are stored in registers during their usage period until the final results are written to memory, the difficulty of the attack will be significantly increased as the attacker can only obtain the final observation. Unfortunately, even with

the vector registers in Single Instruction Multiple Data (SIMD), big numbers in cryptography software cannot be continuously held in registers. Thus, this gives us the second source of ciphertext side-channel occurrence: the compiler pervasively resorts to using stack and heap memory writes to store intermediate values due to **insufficient register resources**, thereby resulting in large attack surfaces for secret leakage.

3.3.3 Motivations of Compiler-aided Mitigation

After understanding how ciphertext side-channel leaks occur during compilation, the compiler-aided strategy emerges as a natural and effective solution to defeat such attacks. Compared to the binary instrumentation strategy, the compiler-aided strategy offer several advantages.

In-place Code Insertion. The compiler-aided method repairs the program alongside the compilation process in an integrated and streamlined manner. For instance, it inserts the mitigation code adjacent to any sensitive memory access instructions related to *stack*.28 and *stack*.29 in Figure 3.3 as well as instructions highlighted in red in Figure 3.4. Thus, it ensures the fixed programs maintain their execution flow as much as possible, eliminating the need for frequent jumps to the instrumentation code that affect branch predictions, as seen in the binary instrumentation approach of CipherFix.

Smooth Management for Random Nonces. It is feasible to introduce and manage a random nonce for the secret data with each memory write operation during compilation. We denote this method as *software-based probabilistic encryption*. For example, the compiler creates two additional slots directly in the stack area to hold the nonces for *stack*.28 and *stack*.29 in Figure 3.3. Afterwards, it precisely links sensitive memories to their corresponding nonce locations through explicit virtual symbols in LLVM-IR, i.e., *stack*.28 and *stack*.29. In contrast, it is challenging to allocate extra space for sensitive variables in the binary instrumentation method due to the constraints imposed by fixed binaries.

Flexible Register Allocation. It is a significant advantage for the compiler to preserve secret-related variables within registers throughout their lifecycle, by leveraging register allocation strategies before the final binary is generated. We

denote this method as *secret-aware register allocation*. For the binary instrumentation method, it is challenging to flexibly allocate the available registers. There are some possible solutions to address this, including performing liveness analysis on a fixed binary and preserving register values before allocating them to sensitive variables, or reserving a certain number of registers at the beginning of compilation. However, these solutions can inevitably lead to large performance degradation.

Accurate Variable Length. The LLVM-IR shown in Figure 3.3 and Figure 3.4 also serves as a valuable tool in identifying variable lengths of different memory regions that require protection. Notably, the length of the stack and heap is crucial for subsequent vulnerability mitigation processes, as it aids in determining the duration for which memory needs to be protected. The compiler-aided approach enables the protection of each memory unit independently, rather than the entire memory, making the process more direct and efficient. Oppositely, the binary instrumentation approach, focuses on protecting a slice of heap memory, primarily because it can only track explicit memory allocations and deallocations, such as `malloc`, `realloc`, `calloc`, and `free`. This overly cautious memory protection introduces unnecessary computations as parts of them do not contain sensitive information that requires safeguarding, potentially diminishing the performance of the system.

Compensatory Dynamic Taint Analysis. Resorting to dynamic taint analysis offers the advantage of rapidly identifying secret-related variables in the execution path. However, this comes at the cost of sacrificing some degree of coverage. The binary instrumentation method tracks sensitive stacks and heaps only in the executed paths, leaving variables in untouched paths unprotected. While employing a similar practice, the virtual symbols of LLVM-IR shown in Figure 3.3 and Figure 3.4 offer a form of compensation to the dynamic taint analysis. To be specific, the virtual symbols representing sensitive stacks and heaps are initially tainted in the executed paths. Subsequently, they are comprehensively recognized in all MBBs of a tainted function due to the streamlined compilation process. While not achieving the same level of coverage as static analysis, this still offers a more comprehensive protection compared to the binary instrumentation method.

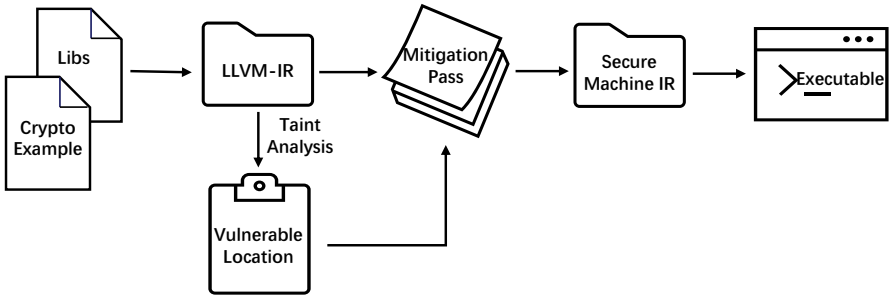


FIGURE 3.5: Workflow of CIPHERGUARD.

3.3.4 Architecture Overview of CipherGuard

Figure 3.5 depicts the workflow of CIPHERGUARD, which comprises a dynamic taint analysis phase (flagging sensitive memory writes) and a static rewriting phase (mitigating vulnerable points). Both phases are integrated in the LLVM ecosystem. To apply CIPHERGUARD on a cryptography application, the developer needs to first mark the secrets (e.g., private keys). The application and its shared dependencies are then compiled and linked into standalone bitcode, after which subsequent operations are performed on this bitcode.

Specifically, the dynamic taint analysis phase performs taint tracking on LLVM-IR to identify sensitive memory accesses (Section 3.4.1). Then the mitigation pass phase employs multiple variants within the realm of software-based probabilistic encryption and secret-aware register allocation to protect the tainted instructions in LLVM Machine-IR. Additionally, precise buffer management for random nonces is carried out for all variants (Section 3.4.4). Lastly, the patched LLVM Machine-IR is subsequently compiled into a hardened binary for deployment. The descriptions of 3 variants are as below.

- Variant 1 employs the `rdrand` instruction to generate a random nonce when encountering sensitive memory store instructions for both stack and heap areas. It optimizes the compiling process by utilizing a pre-generated random nonce buffer, reducing the performance bottleneck associated with generating it on-the-fly (Section 3.4.2).
- Variant 2 employs the `vaesenc` instruction from AES-NI and a shift/rotate-based linear transformation from XorShift128+ [218] to fulfill the requirement of generating a random nonce on-the-fly (Chapter 3.4.2).

- Building upon Variant 1, Variant 3 seeks to safeguard secrets in sensitive stack areas by retaining them within vector registers, such as SSE registers, throughout their lifecycle, thereby avoiding any spillage to memory (Chapter 3.4.3).

Application Scope. The primary audience for CipherGuard consists of developers aiming to deploy cryptography software on modern TEEs. CipherGuard provides security assurances for existing crypto systems utilized in TEEs because the hardware instructions required for implementing its three variants are mainstream and universal.

3.3.5 Technical Challenges

IR Level. Existing taint analysis tools based on LLVM, such as DFSan, adeptly track the information flow of tainted variables at the LLVM-IR level with high accuracy and efficiency. However, as analyzed in Section 3.3.2, optimizations in the LLVM backend, such as the register allocation pass, may inadvertently undermine the efforts of mitigating ciphertext side-channel attacks. On the other hand, the lower Machine-IR level could also be utilized for the mitigation, although it would require additional efforts such as adapting the granularity of taint analysis, maintaining the usage of physical registers, and manually allocating the mask buffer as implemented in CipherFix. Therefore, ensuring the efficiency and effectiveness of the program repair requires fixing the program at the LLVM Machine-IR level with the register allocation pass.

This consideration presents a challenge of aligning the tainted results obtained at the LLVM-IR level with the LLVM Machine-IR level. This can be addressed by the transformation of LLVM-IR to LLVM Machine-IR through instruction selection and linear sequence instruction emitting. Although replaced with target instructions, the semantic difference between these two levels is minimal, allowing for the tainted results from LLVM-IR to be located within LLVM Machine-IR through simple matching. As displayed in Figure 3.3 and Figure 3.4, the presentation of the program as basic blocks in LLVM Machine-IR can mirror the structure in LLVM-IR.

Vulnerability Detection. Ideally, performing mitigation only on true vulnerable points can minimally impact the execution of the protected program. *However, it is challenging to achieve precise detection of ciphertext side-channel leakage.* For example, CipherH, a relatively accurate detection tool at present, may still introduce false positives in analyzing ciphertext side channels. Hence, we opt to protect all sensitive memory accesses flagged by taint analysis, following the practice in CipherFix. It is important to clarify that CipherGuard can be integrated with any ciphertext side-channel detector for the subsequent mitigation.

In Section 3.2.1, we demonstrate how secrets are tracked through direct memory writes. However, in addition to ciphertext side channels induced via the direct usage of secrets, *it is challenging to consider data derived from the secrets as "sensitive" as well.* Therefore, as a common practice in this line of research, CipherGuard tracks both explicit and implicit information flows propagated from the secret. In this manner, CipherGuard comprehensively uncovers attack surfaces of the cryptography application.

Random Nonce Management and Register Allocation. While the benefits of smooth random nonce management and flexible register allocation are evident, *implementing these strategies in the compilation process poses a challenge in terms of performance.* CipherGuard automatically allocates and manages stack slots, saving random nonces for sensitive stack areas. However, for sensitive heap areas, optimizing the performance requires preemptively allocating space to store these random nonces. Therefore, CipherGuard allocates this space beforehand and references it for the corresponding heap areas in an efficient way. Next, to optimize the generation of random nonces, it is a good choice to pre-generate a buffer containing enough random numbers. This also requires to determine the indexing location inside the random number buffer for each sensitive stack and heap. Lastly, due to insufficient register resources, CipherGuard heuristically decides which registers to use and how to assign them to sensitive variables based on the demand-centric strategy.

3.4 Detailed System Design

CipherGuard comprises the following pivotal components with three variants. The taint analysis technique (Section 3.4.1) is employed to precisely identify sensitive memory access instructions across all variants. The software-based probabilistic encryption (Section 3.4.2) transforms these sensitive memory instructions into masking-hardened snippets with random nonces in all variants. The secret-aware register allocation (Section 3.4.3) is applied in Variant 3 to retain sensitive stack variables in registers. The buffer management (Section 3.4.4) assists all variants in efficiently accessing random nonces.

3.4.1 Tainting Secret Locations

CipherGuard initiates taint tracking on the LLVM-IR code derived from the given cryptography program. At present, multiple taint analysis schemes may be employed to track secret propagation at different compilation stages of a program. For instance, to detect cache side-channel leakage, Wang et al. [133] applied tainting directly on the binary of the cryptography program, resulting in a series of logged instructions. However, this approach suffered from speed limitations, and the tracking of implicit information flow may not be comprehensive. Additionally, Weiser et al. [147] performed taint analysis on a logged execution trace, but a single trace may yield false negatives. Moreover, tainting at the compiler level is a common practice in program vulnerability detection. For example, some works [132, 213] utilized the DFSan tool to obtain a set of secret-related LLVM-IR instructions. Considering the subsequent program repair on LLVM Machine-IR and the need for comprehensive and readily available implicit information flow tracking, we also adopt the compiler-level taint analysis scheme with DFSan.

Since our objective is to protect sensitive memory writes against ciphertext side channels, any store instruction that writes secret variables into memory must be tainted. These store instructions encompass both direct memory move instructions and indirect memory operation instructions, e.g., arithmetic instructions whose destination operand is a memory unit. To aid in locating places where sensitive variables are read, CipherGuard also taints all secret-related read operations, including direct memory move and calculation instructions containing a memory

unit as an operand. Furthermore, CipherGuard tracks implicit information flows for comprehensive and conservative modeling of secret propagation. A secret may appear in control-flow branches as a condition or in memory accesses as the index. Then variables guarded by a tainted condition are tainted as secret-related; variables assigned through memory access are tainted once the index of the buffer is tainted.

CipherGuard tracks the propagation of taint labels and identifies tainted memory accesses over a single execution trace of the cryptography program. On one hand, varying inputs, such as different secret or public input lengths, may result in divergent execution traces, leading to the discovery of different tainted memory accesses. On the other hand, in practice, cryptography applications often follow a relatively "fixed" execution flow, where the execution traces are similar for different inputs, with some loop iterations executed more times depending on the input length. This observation aligns with the findings reported in CipherH [213]. Therefore, we adopt the same strategy in CipherFix to log execution traces multiple times for each cryptography program with varied secrets and inputs. This practice serves as another form of compensation to the dynamic taint analysis.

3.4.2 Software-based Probabilistic Encryption

The mitigation phase targets the tainted instructions collected from the aforementioned taint analysis phase and performs program transformations (patching) on these memory instructions in the form of LLVM Machine-IR. We intercept each tainted memory instruction during the register allocation phase of the LLVM backend. The key of our software-based probabilistic encryption is to introduce a random nonce into secret variables before they are written back into the memory, thereby achieving unpredictable ciphertexts. To accomplish this, we develop two variants in CipherGuard. Variant 1 relies on the `rdrand` instruction to generate random nonces, while Variant 2 utilizes the AES-NI and XorShift128+ schemes to generate random nonces.

Mitigation Code Insertion. Figure 3.6 illustrates the general scenario for inserting the mitigation code, where MEM_{key} represents a memory cell of a sensitive memory access instruction that may locate in the stack or heap. The inserted mitigation code includes the following operations: 1) loading the masked plaintext and

1:	load	MEM_{key}	REG_{key}	// sensitive load
2:	load	MEM_{mask}	REG_{mask}	// load mask
3:	save	EFLAGS		
4:	xor	REG_{mask}	REG_{key}	
5:	restore	EFLAGS		
6:	operate	REG_{key}		
7:	save	EFLAGS		
8:	update	REG_{mask}		// generate new mask
9:	xor	REG_{mask}	REG_{key}	
10:	restore	EFLAGS		
11:	store	REG_{mask}	MEM_{mask}	// store mask
12:	store	REG_{key}	MEM_{key}	// sensitive store

FIGURE 3.6: In-place code insertion.

random nonce from the source memory and its corresponding nonce buffer (lines 1–2); 2) XORing the nonce with the masked plaintext to obtain original plaintext (lines 3–5); 3) performing the actual calculation on the secret variable (line 6); 4) updating the new random nonce and XORing it with the secret result to obtain a new masked plaintext (lines 7–10); and 5) storing the masked plaintext and its new random nonce to their respective storage locations (lines 11–12).

To insert the code, we need to determine the memory cells that hold the random nonce. We employ two different strategies for stack and heap areas, respectively. For the stack area, since the mitigation code is inserted along with the register allocation phase of the LLVM backend, we can freely allocate and organize a stack slot for holding the nonce. For instance, we allocate a new stack slot MEM_{mask} to store the new random nonce within the stack area (line 11 of Figure 3.6), and associate MEM_{mask} with the source memory MEM_{key} (line 1). Subsequently, when loading the nonce for MEM_{key}, we directly insert a load instruction that references MEM_{mask}.

For the heap area, even though the LLVM backend can intercept each `malloc`, `realloc`, `calloc`, and `free` call to dynamically allocate memory for nonce buffers, it introduces large overhead due to the invocation of system calls and initialization procedures. We devise a hash mechanism to circumvent this issue. We leverage the heap address of source memory MEM_{key} at runtime to calculate the index at which the corresponding nonce is stored in the `.bss` section (see Section 3.4.4 for more details). Then, the newly generated random nonce is stored in `.bss` referenced

by this index. With this method, we mechanically insert the same code for each encountered tainted heap instruction.

Random Nonce Generation. To enhance the security protection, we update the random nonce for each sensitive store instruction. In Variant 1, we use `rdrand` to pre-generate a nonce buffer in the `.bss` section with 1K random numbers during the initialization of the cryptography program. For an unmasked stack area containing secrets, Variant 1 selects a new random number from the `.bss` section at the corresponding location as the initial nonce. Subsequently, the nonce is updated by incrementing three when the stack area at the same location needs to store new secrets (see Section 3.6.5 for security analysis). Variant 2, on the other hand, generates a random nonce on-the-fly using AES-NI and XorShift128+ schemes. The AES-NI scheme involves inserting a single instruction with two 128-bit vector registers, such as `xmm14` and `xmm15` in CipherGuard. Conversely, the XorShift128+ scheme requires 11 instructions with three 128-bit vector registers, from `xmm13` to `xmm15`.

3.4.3 Secret-aware Register Allocation

The inherent locality of the stack area usage underscores the importance of preserving secret contents within registers, thereby preventing their inadvertent exposure through memory spills and thwarting potential ciphertext observations by attackers. Additionally, utilizing registers to store secrets reduces memory access consumption, thereby contributing to performance improvements. Building upon Variant 1, the masking scheme for the stack area can be further enhanced through register allocation.

Feasibility Analysis. However, it is challenging to achieve this scheme due to the limited register resources. To evaluate this, we conduct a heuristic investigation as follows: 1) We determine the maximum number of stack slots involved in sensitive memory access instructions among all tainted functions extracted from different cryptography programs, as shown in the second column of Table 3.1. Notably, the EdDSA implementation in libsodium has the largest number of sensitive stack slots (583). As discussed in Section 3.3.5, it is difficult to accommodate such a requirement of vector registers in SIMD. This challenge prompts us to explore a secret-aware register allocation approach, which involves tracking the liveness of

registers and performing timely deallocation for stacks that occur more frequently in tainted functions. 2) We then determine the number of vector registers required for this scheme. By examining the disassembled cryptography programs in Table 3.1, we note that the first 8 vector registers (`xmm0` - `xmm7`) are often allocated by the LLVM backend to optimize data movement for consecutive addresses. Therefore, we heuristically preserve the last 8 SSE vector registers (`xmm8` - `xmm15`) and assess their performance impact on the protected programs, as shown in the last two columns of Table 3.1. On average, the performance impact is 7%, with a minimum observed impact of 1% in SHA512 of libsodium and a maximum impact of 20% in AES of mbedTLS. This suggests that preserving the last 8 SSE vector registers for sensitive stack slots is a viable and practical approach.

TABLE 3.1: The maximum numbers of sensitive stack slots among tainted functions.

Implementation	**Stack Slots**	**Impact on performance**	
		Cycles	**Factor**
libsodium-EdDSA	583	198331	1.04
libsodium-SHA512	27	43043	1.01
mbedTLS-AES	7	571968	1.20
mbedTLS-Base64	25	11575	1.03
mbedTLS-ChaCha20	4	609942	1.02
mbedTLS-ECDH	52	4586425	1.08
mbedTLS-ECDSA	52	4095496	1.04
mbedTLS-RSA	52	1943880	1.03
OpenSSL-ECDH	157	1171024	1.19
OpenSSL-ECDSA	79	18180467	1.08
WolfSSL-AES	43	689316	1.08
WolfSSL-ChaCha20	5	512287	1.06
WolfSSL-ECDH	100	362967	1.04
WolfSSL-ECDSA	30	4559759	1.08
WolfSSL-EdDSA	159	426239	1.02
WolfSSL-RSA	28	543835	1.11
Average	-	-	1.07

Register Allocation. When analyzing the tainted functions within the cryptography program, our primary focus is on stack slots that occur frequently. This consideration arises from the understanding that recurring stack slots are more likely to be repeatedly overwritten, exhibiting sequential change patterns that become the targets of ciphertext side-channel attacks. Moreover, preserving these

stack slots within registers reduces the overhead associated with the masking operations on them. Consequently, we construct two structures in the LLVM backend: `StackUsage`, which presents mappings from all sensitive stack slots to their respective MBB locations in the current tainted functions, and `StackOpt`, which selects mappings from `StackUsage` based on the quantity of stack slot's MBB locations until its capacity satisfies the number of available vector registers. With these two structures, secret-aware register allocation relies on two core operations:

• *Allocation:* when encountering a sensitive stack store instruction whose stack slot falls within the mapping of `StackOpt`, Variant 3 allocates an available vector register location to hold the secrets. At the same time, its MBB location in the mapping of `StackOpt` is accordingly deleted to assist in tracking the liveness of the allocated vector register.

• *Deallocation:* Once all MBB locations of the sensitive stack slot in `StackOpt` are deleted, its associated vector register reaches the end of the liveness, and becomes available for allocation to other sensitive stack slots. Subsequently, by selecting another stack slot from `StackUsage` to replace the sensitive stack slot based on the quantity of MBB locations, Variant 3 allocates available vector registers to the remaining stack slots in `StackOpt`.

```
42:  entry, if.end19, if.end24, if.end30,...
38:  entry, if.end24, if.end30,...
39:  entry, while.body, if.end30,...
52:  if.end30,...
49:  if.end30,...
46:  while.body, if.then10, if.end12, if.then18, if.end19,...
43:  while.body, if.end19, if.then22, if.then28, if.end30,...
40:  entry, while.cond,...
50:  if.end30,...
44:  while.body, if.end19, if.end24, if.end30,...
```

FIGURE 3.7: Sensitive stack slots contained in MBBs.

We illustrate the register allocation process for sensitive stack slots using the function `bn_mul_add_words` of OpenSSL-ECDSA as an example. Figure 3.7 describes a sorted `StackOpt` structure, where the first element represents the sensitive stack slots followed by their positions of appearance in MBBs. We simulate the register allocation process until the `if.end30` MBB and omit the subsequent MBBs for brevity. Figure 3.8 visualizes the allocation process in Figure 3.7. Each stack slot

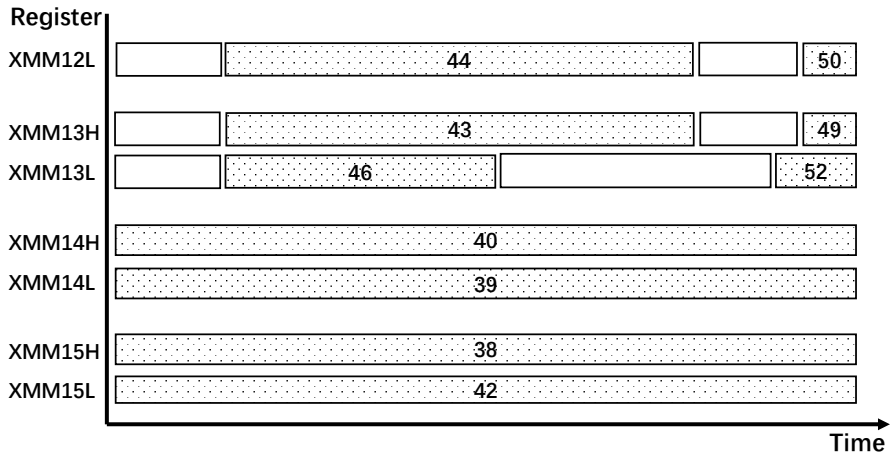


FIGURE 3.8: An example of register allocation from the function `bn_mul_add_words` of OpenSSL-ECDSA. In the visualization, the white and shaded blocks represent the liveness of stacks, with shaded blocks containing numbers that denote registers holding the sensitive stack slots.

is formatted with a size of 8 bytes, resulting in 16 independent storage cells allocated from `xmm8` to `xmm15`. `H` and `L` represent the high and low 64-bits of a vector register, respectively. Notably, registers `xmm13L`, `xmm13H` and `xmm12L` are recycled and reallocated to stack slots 52, 49 and 50, respectively, showing how liveness tracking enables a more optimized storage capacity.

3.4.4 Managing Nonce Buffers

In CIPHERGUARD, all the inserted mitigation codes involving the masking protection scheme must consider the management of random nonces, including selecting the initial random nonce from the random number buffer (Variants 1 and 3) and storing the random nonce currently in use for the corresponding stack and heap areas (all variants).

Random Nonce Buffer. As briefly introduced in Section 3.4.2, CIPHERGUARD pre-generates a buffer in `.bss` containing 1K random numbers using `rdrand` during the program initialization. For unmasked stack or heap areas in Variants 1 and 3, which require an initial random nonce, CIPHERGUARD leverages the memory address to calculate an index for selecting the corresponding random number from `.bss`. Assuming the memory address in the sensitive memory access instruction

is *addr*, the index is calculated as: $index = addr \ \& \ 0x3FF$. Then, the random number in the `.bss` section is obtained using $randomArray(, index, 8)$, where $randomArray$ represents the starting address of the random number buffer in the `.bss` section.

While it is a fact that using memory addresses over 8K may result in the same index and hence the same random number for masking, it is important to note that the final ciphertext will still differ due to the different plaintexts in these memory locations. Another approach to enhance the security is to periodically call the `rdrand` instruction to generate new random numbers and overwrite the buffer, making it hard for the attacker to guess.

Currently Used Nonces. Two distinct strategies for storing the random nonces currently in use are employed for the sensitive stack and heap areas. For the stack, the compiler automatically allocates slots to hold the random nonces and associates each slot with the corresponding source memory, as demonstrated in Section 3.4.2. Accessing these nonces is easily facilitated by adding an offset to the `rsp` register, which is generated by the compiler for subsequent mask loading. Additionally, we initialize all random buffers to zero at the beginning of the function, specifically in the entry block of the function. For the reliability of the fixed functions, this strategy ensures that the nonces are isolated from stacks in other locations by leveraging the `rbp` register. In contrast, CipherFix adds a constant distance to the current memory block to locate its corresponding nonce buffer. Unfortunately, this poses a potential risk as the nonce buffers are not positioned within the runtime stack area, which may result in overlap with other stack areas, ultimately leading to program malfunctions.

For the heap, instead of applying a constant offset for the nonce buffer like in CipherFix, we place the nonce buffer in the `.bss` section as well, pre-allocating sufficient space in advance. A heuristic hash function maps heap addresses to buffer indices for retrieving nonces: $((addr \ \& \ 0xFFFFF) * 648056) \gg 22$. In this setting, 128K consecutive 8-byte entries are mapped independently into the 1M nonce buffer without collisions since all variants of CipherGuard align masking operations to 8-byte addresses. However, the maximum index generated by the hash function is 162,012, which exceeds 128K. To accommodate this, a 256K-entry buffer is required, resulting in a 2M nonce buffer with around 50% utilization in

our configuration. Through experimentation, we found that a nonce buffer space of 2MB is sufficient for cryptographic programs without collisions.

CipherGuard offers several advantages for managing mask buffers in the heap areas compared to CipherFix. Firstly, it resolves the overlapping risk present in CipherFix, where the fixed distance between the heap area and its corresponding mask buffer may become insufficient if the program dynamically allocates a large amount of memory. CipherGuard addresses this by employing an indexing method to locate the mask buffer, ensuring that each heap area has a dedicated mask buffer without the risk of overlap. Secondly, CipherFix requires tracking each `malloc`, `realloc`, `calloc`, and `free` call for dynamic (de)allocation of mask buffers, which introduces overhead and impacts the runtime performance. In contrast, CipherGuard achieves direct access to the mask buffer by calculating the index, thereby eliminating the need for system calls and reducing the overhead. Furthermore, similar to the mask buffer for the stack areas, the mask buffer for the heap areas also needs to be initialized to zero. Unlike CipherFix, which requires special routine code to assign each mask buffer with zero, CipherGuard simplifies this process by initializing `.bss` during program linking.

To handle potential hash collisions in nonce buffers for heap areas, such as in server environments, we propose dynamically expanding the buffer by allocating multiple groups of entries for each index (10 groups per index, requiring a 20M nonce buffer). If two different heap addresses generate the same index, the first 8 bytes are used to match the heap address, allowing us to locate the corresponding nonce in the subsequent 8 bytes. We evaluated this method by simulating collisions across all 10×128K entries. The results showed only a slight increase of time to initialize the expanded nonce buffer and handle each collision ($\approx 8ms$), indicating minimal performance impact.

3.5 Implementation

CipherGuard is developed within the LLVM framework (LLVM-9 version), comprising a total of 4.6K lines of C++ code. However, it is important to note that **CipherGuard is not reliant on specific features of LLVM**. Further discussion on the portability of CipherGuard to other compiler frameworks is provided in

Section 3.7. CipherGuard consists of a taint analysis component utilizing DFSan APIs and a mitigation component seamlessly integrated with the register allocation pass in the LLVM backend. To facilitate the 3 variants in CipherGuard, we assemble 41 instruction blocks to perform various mitigation operations. These operations include data moving, generating random numbers using three different schemes, calculating hash indices, and other relevant tasks.

Usage of CipherGuard. To use CipherGuard, developers need to manually identify the secrets within the cryptography program and mark them using the DFSan API. This labeling process is user-friendly. For example, it is simple to insert `dfsan_set_label(label, key, sizeof(key))` into any cryptography program to record both the pointer and size of the secret. Subsequently, the taint analysis and LLVM backend mitigation processes automatically identify the tainted memory access instructions and execute the mitigation without requiring human intervention. Ultimately, the resulting security-hardened binary can be seamlessly executed on the TEE processor.

3.6 Evaluation

3.6.1 Experiment Setup

We evaluate 3 variants of CipherGuard on a variety of cryptography libraries, including libsodium-1.0.18, mbedTLS-3.3.0, OpenSSL-3.0.2, and WolfSSL-5.3.0. All experiments are conducted on an AMD EPYC 7513 CPU with SEV-SNP enabled, running each cryptography library on the Ubuntu 20.04 platform. To compare with CipherFix, we use the same snapshots of cryptography libraries evaluated in CipherFix. To be specific, EdDSA (*Ed25519*), ECDSA (*secp256r1*), ECDH (*X25519*) and RSA signature schemes are demonstrated to be vulnerable to ciphertext side channels in their implementations. Simultaneously, we incorporate symmetric primitives such as AES-GCM and ChaCha20-Poly1305 (AES and ChaCha20 for short, respectively), the hash function SHA512, and the decoding function Base64. For our evaluations, we utilize the default optimization levels, specifically `-O3` for OpenSSL and `-O2` for the other libraries.

3.6.2 Comparison between Variants

Performance Overhead. We employ the `rdtsc` instruction to precisely measure the execution cycles of original and patched cryptography libraries. Each cryptography library undergoes 200 iterations, and we measure the average execution overhead. Table 3.2 provides a comprehensive overview of the performance statistics for patched cryptography libraries using 3 variants of CipherGuard. In essence, all 3 variants exhibit an acceptable performance cost while effectively eliminating the ciphertext side-channel leakage.

Specifically, Variant 1 introduces an average of 2.53× overhead, with a minimal of 1.09× in repaired ChaCha20 of WolfSSL and a maximum of 6.17× in ECDH of OpenSSL. This result is attributed to the utilization of a pre-generated nonce buffer, a strategy that reduces the performance bottleneck associated with generating random nonces on-the-fly when encountering sensitive memory store instructions, particularly highlighting the impact of the `rdrand` instruction on patched cryptography software. This improvement emphasizes the effectiveness of CipherGuard in achieving a balance between performance and security.

The requirement of on-the-fly nonce generation introduces two approaches in Variant 2: the use of AES-NI and XorShift128+ schemes. These two approaches achieve an average overhead of 2.78× and 3.10×, respectively, resulting in a marginal performance reduction compared to the optimized `rdrand` scheme in Variant 1. Notably, the execution of XorShift128+ takes slightly longer due to its requirement for multiple instructions compared to the `vaesenc` instruction from AES-NI. The nuanced performance differences provide valuable insights in selecting specific strategies for on-the-fly nonce generation in cryptography software.

Building upon the foundation of Variant 1, Variant 3 seeks to safeguard secrets in sensitive memory access instructions by retaining them within registers throughout their lifecycle, avoiding any spillage to memory. Although this scheme demonstrates a better average performance improvement over Variant 1, with an overhead of 1.76× as opposed to 2.53×, it is worth noting that its performance superiority cannot be achieved in all cryptography libraries. This is attributed to the trade-off introduced by the utilization of SSE registers. While this reduces the cost of inserting additional mask instructions to protect sensitive memory access instructions, it impacts the availability of SSE registers in other parts of the program.

TABLE 3.2: Performance statistics towards mitigated cryptography software with 3 variants of CIPHERGUARD. Results are obtained by measuring the average clock cycles using the `rdtsc` instruction. XS+ is short for XorShift128+.

Implementation	**Orig**	**Variant 1**		**Variant 2**				**Variant 3**	
		RDRAND-OPT	**Factor**	**AES**	**Factor**	**XS+**	**Factor**	**REGISTER**	**Factor**
libsodium-EdDSA	191618	648861	3.39	651930	3.40	682159	3.56	304067	1.59
libsodium-SHA512	42482	101088	2.38	137298	3.23	152227	3.58	105061	2.47
mbedTLS-AES	474926	574695	1.21	815542	1.72	866192	1.82	993255	2.09
mbedTLS-Base64	11220	16877	1.50	18360	1.64	19642	1.75	15493	1.38
mbedTLS-ChaCha20	597000	752978	1.26	1236573	2.07	1385737	2.32	1251150	2.10
mbedTLS-ECDH	4253887	7680051	1.81	8819828	2.07	9626046	2.26	6211978	1.46
mbedTLS-ECDSA	3942140	7310694	1.85	8133126	2.06	8717678	2.21	5448579	1.38
mbedTLS-RSA	1884568	4314759	2.29	4658550	2.47	5504460	2.92	2861580	1.52
OpenSSL-ECDH	986228	6081644	6.17	6132829	6.22	6997656	7.10	1519590	1.54
OpenSSL-ECDSA	16854915	66816768	3.96	65915307	3.91	76290783	4.53	34374077	2.04
WolfSSL-AES	639055	1341064	2.10	2126681	3.33	2340311	3.66	1897460	2.97
WolfSSL-ChaCha20	481650	526643	1.09	507535	1.05	517214	1.07	537044	1.12
WolfSSL-ECDH	348690	737935	2.12	731250	2.10	761804	2.18	522330	1.50
WolfSSL-ECDSA	4226940	20976600	4.96	20141736	4.77	25013569	5.92	9132870	2.16
WolfSSL-EdDSA	419507	1317104	3.14	1296690	3.09	1350810	3.22	639840	1.53
WolfSSL-RSA	491970	608190	1.24	645980	1.31	737520	1.50	631806	1.28
Average	-	-	2.53	-	2.78	-	3.10	-	1.76

The Worst Case. Among Variant 1 and Variant 2, mitigating ciphertext side channels in ECDH from OpenSSL incurs the highest overhead, ranging from 6.17× to 7.10×. We manually examine the range of tainted MBBs in the function where the mitigation code is inserted. The observation reveals that a large number of MBBs contained secret values within loops, which inevitably introduce additional cycles due to the multiple memory reads/writes involved in the masking protection implemented in both Variant 1 and Variant 2. This finding motivates the adoption of Variant 3, which retains secret variables in SSE registers rather than performing masking. In particular, this approach significantly reduces overhead in the context of loops. As shown in Table 3.4, Variant 3 improves the worst-case overhead for ECDH in OpenSSL.

`rdrand` as the Bottleneck. Employing the on-the-fly generation of random numbers as nonces provides a robust defense mechanism. However, relying on `rdrand` as the source of random numbers in this mechanism can incur a significant time cost. This is evident in the implementation of the on-the-fly method in the CipherFix-Fast variant, leading to an average overhead of 16.8× and a maximum of 40×. Consequently, we opt for the pre-generation of a nonce buffer to reduce the performance impact.

To assess the performance enhancement introduced by Variant 1 of CipherGuard in the context of the on-the-fly method, we focus on asymmetric cryptography libraries (RSA and ECDSA), and symmetric cryptography libraries (AES and ChaCha20). The results in Table 3.3 demonstrate a notable performance improvement for asymmetric cryptography libraries, albeit only marginal improvement for symmetric cryptography libraries, achieved by Variant 1. This enhancement is particularly evident in programs involving blinding and nonce, with increased numbers of tainted functions and instructions. Thus, the advantage of pre-generating a nonce buffer reaffirms the effectiveness of Variant 1 in optimizing the performance of on-the-fly methods.

Alternative Nonce. Unlike Variant 1, which selects a random number as a nonce and increments it for each sensitive memory write, the methods in Variant 2 directly generate a new random nonce when an updated nonce is required. AES-NI accomplishes this with a single `vaesenc` instruction, while XorShift128+ requires multiple instructions, resulting in a slight larger overhead compared to AES-NI.

TABLE 3.3: Performance improvement by Variant 1 over the on-the-fly `rdrand` method. CC20 is short for ChaCha20.

Implementation	**Variant 1 Factor**	**On-the-fly `rdrand` Factor**	**Improvement**
ECDSA-mbedTLS	1.85	10.87	5.87
ECDSA-OpenSSL	3.96	18.92	4.77
ECDSA-WolfSSL	4.96	18.60	3.75
RSA-mbedTLS	2.29	14.91	6.51
RSA-WolfSSL	1.24	16.21	13.07
AES-mbedTLS	1.21	2.38	1.96
AES-WolfSSL	2.10	3.10	1.47
CC20-mbedTLS	1.26	1.30	1.03
CC20-WolfSSL	1.09	1.24	1.13
Average	2.21	9.72	4.39

Apart from the differences in nonce generation, Variant 2 also mandates the separate reservation of part of SSE vector registers for its use, i.e., two and three for the two methods respectively. Such impact is exemplified in Table 3.2. Reserving a small number of SSE vector registers could impact other parts of the cryptography library, which amplify the burden on the repaired program compared to Variant 1.

The Profit of Registers. The intuitive advantage of using registers to store intermediate results is a reduction in memory access consumption. However, as discussed previously, the advantage of Variant 3 over Variant 1 does not hold for all cryptography applications. A preliminary hypothesis suggests that Variant 3 concurrently impacts the availability of SSE registers in other parts of the program. To test this hypothesis, we implement a new strategy "RSV+RDRAND-OPT", which adopts the method from Variant 1 while reserving 8 SSE vector registers unused. This allows the compiler to freely allocate these 8 SSE vector registers like Variant 1, or reserve these SSE vector registers for holding secrets of sensitive memory access instructions like Variant 3.

Table 3.4 shows the results of this new strategy, revealing an average overhead of 2.98×, which is higher than the overheads of both Variant 1 and Variant 3. Specifically, some cryptographic functions (SHA512 from libsodium, AES and ChaCha20 from mbedTLS, AES, ChaCha20 and RSA from WolfSSL) experience worse performance when SSE vector registers are utilized to protect sensitive memory access instructions, due to their negative impact on other parts of the program. For those functions that enjoy a positive performance profit, we observe a notable disparity,

TABLE 3.4: Profit analysis of variant 3.

Implementation	Variant 1	RSV+RDRAND-OPT		Variant 3
	Factor	Cycles	Factor	Factor
libsodium-EdDSA	3.39	680723	3.55	1.59
libsodium-SHA512	2.38	121020	2.85	2.47
mbedTLS-AES	1.21	1135500	2.39	2.09
mbedTLS-Base64	1.50	20820	1.86	1.38
mbedTLS-CC20	1.26	1329720	2.23	2.10
mbedTLS-ECDH	1.81	7981500	1.88	1.46
mbedTLS-ECDSA	1.85	10137960	2.57	1.38
mbedTLS-RSA	2.29	5459100	2.90	1.52
OpenSSL-ECDH	6.17	6121680	6.21	1.54
OpenSSL-ECDSA	3.96	73640040	4.37	2.04
WolfSSL-AES	2.10	1916760	3.00	2.97
WolfSSL-CC20	1.09	614670	1.28	1.12
WolfSSL-ECDH	2.12	849300	2.44	1.50
WolfSSL-ECDSA	4.96	21068010	4.98	2.16
WolfSSL-EdDSA	3.14	1428990	3.41	1.53
WolfSSL-RSA	1.24	902452	1.83	1.28
Average	2.53	-	2.98	1.76

particularly in ECDH from OpenSSL, where Variant 3 proves to be particularly profitable.

We delve deeper into the allocation of SSE vector registers to sensitive memory access instructions in ECDH from OpenSSL. We follow the design principles of secret-aware register allocation outlined in Section 3.4.3, and check the range of MBBs in a function where these SSE vector registers are allocated to protect sensitive memory access instructions. The observed pattern reveals a notable advantage when registers are used to hold stack memory within loops, effectively saving extra cycles that would be incurred by employing masks for protection, as implemented in Variant 1. Conversely, if the protection extends to insufficient sensitive stack memory within a loop — such as using SSE vector registers to safeguard sequential sensitive memory access instructions — the benefit may not be sufficient to offset the impact of not allocating these registers to other parts of the cryptography program. This highlights the importance of the context and specifics of the cryptographic algorithm in determining the efficacy of secret-aware register allocation.

Variant Selection. With all variants providing sufficient security guarantees (see Section 3.6.5), selection of an appropriate variant is based on the performance

and usage requirements. For Variant 1, pre-generated random buffers serve as an optimization compared to on-the-fly random nonce generation. Variant 2, on the other hand, is preferred when there is a need to generate a random nonce on-the-fly, although it requires the use of SSE registers, which may affect cryptography software with frequent data movement for consecutive addresses to some extent. For less performance overhead, Variant 3 is worth considering, even though it occupies more SSE registers than Variant 2. In short, developers proficient in deploying cryptography software in TEEs are recommended to use Variants 2 and 3 due to their better understanding of the context and specifics of the software. For general users, Variant 1 is recommended.

3.6.3 Comparison with CipherFix

Dynamic Taint Analysis. The initial step of using CipherGuard is to conduct dynamic taint analysis for identifying sensitive memory access instructions. Subsequently, the compilation process with DFSan and the generation of tainted LLVM-IR instructions are automated. Remarkably, the identification of sensitive memory access instructions for each cryptography library requires less than 20 minutes. In particular, the taint analysis for symmetric primitives like AES and ChaCha20 concludes within 2 to 3 minutes. Despite the utilization of blinding and nonces in asymmetric primitives such as RSA and ECDSA, which extends the execution process, the taint analysis still completes in about 10 minutes, representing a reasonably acceptable duration.

In the evaluation presented in Table 3.5, we meticulously tally the numbers of functions and instructions for each cryptography library identified by the taint analysis, as indicated in the fourth and fifth columns. The results showcase a broad spectrum, with a minimum of 6 sub-functions in SHA512 from libsodium to a maximum of 796 sub-functions in ECDSA from OpenSSL, with an average of 120 functions across the assessed cryptographic implementations. The range of tainted instructions within these tagged functions spans from 275 in ChaCha20 of mbedTLS to a substantial 17,834 instructions in ECDSA of OpenSSL. Compared to CipherFix, as evidenced by the second and third columns in Table 3.5, our approach exhibits a more detailed and extensive tracking of secrets.

TABLE 3.5: Performance comparison with CIPHERFIX based on the same number of tainted functions. The replication of CIPHERFIX is conducted on its FAST version.

Implementation	**CipherFix**		**CipherGuard**		**CipherFix-Fast**				**CipherGuard-Variant 2**			
	FUN	**INS**	**FUN**	**INS**	**Orig**	**AES**	**Factor**	**Consume**	**INS**	**AES**	**Factor**	**Consume**
libsodium-EdDSA	14	616	17	1311	131790	779580	5.92	1052	630	271557	1.42	127
libsodium-SHA512	6	155	6	586	60060	103920	1.73	283	211	84139	1.98	197
mbedTLS-AES	19	96	17	326	312990	1165470	3.72	8880	318	780317	1.64	960
mbedTLS-Base64	5	25	9	386	10230	16440	1.61	248	26	14897	1.33	141
mbedTLS-ChaCha20	15	234	14	275	397080	922800	2.32	2247	267	1193730	2.00	2235
mbedTLS-ECDH	20	65	51	1063	3314160	10658700	3.22	112993	171	7760267	1.82	20505
mbedTLS-ECDSA	51	448	69	1446	1385790	10117620	7.30	19491	1156	7179808	1.82	2801
mbedTLS-RSA	35	300	44	1238	2893260	11347680	3.92	28181	673	3640590	1.93	2609
OpenSSL-ECDH	11	117	721	12876	373800	583680	1.56	1794	123	1280850	1.30	2395
OpenSSL-ECDSA	91	653	796	17834	3236490	6735210	2.08	5358	991	22131750	1.31	5325
WolfSSL-AES	5	204	6	499	401550	725040	1.81	1586	192	809066	1.27	885
WolfSSL-ChaCha20	9	182	14	404	706740	1022880	1.45	1737	343	505800	1.05	70
WolfSSL-ECDH	10	291	20	543	191280	385440	2.02	667	147	701408	2.01	2399
WolfSSL-ECDSA	40	328	59	932	3143010	8558370	2.72	16510	242	8321686	1.97	16920
WolfSSL-EdDSA	31	835	28	715	312360	746250	2.39	520	616	893040	2.13	769
WolfSSL-RSA	48	696	43	704	537750	1200660	2.23	952	529	615510	1.25	234
Average	26	328	120	2571	-	-	2.87	12656	415	-	1.64	3661

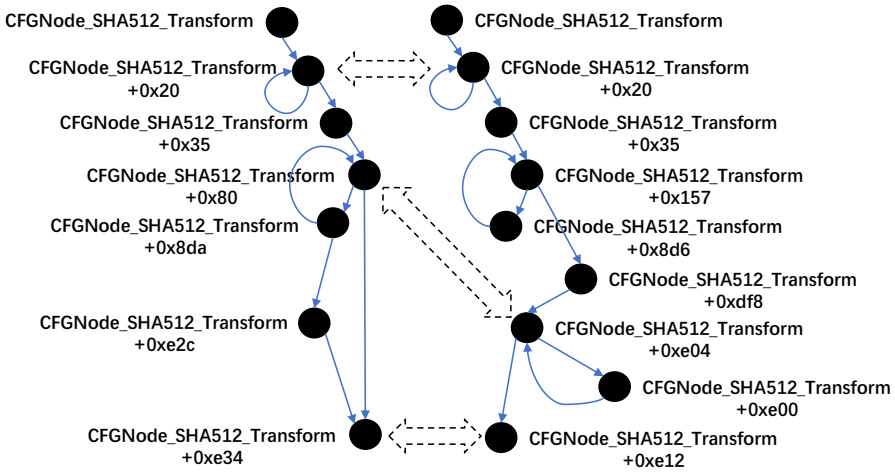


FIGURE 3.9: Function `SHA512_Transform` from the libsodium-SHA512 serves as an example to illustrate the construction of CFGs and identify critical nodes.

Mitigation Coverage Comparison. Accurately identifying tainted instructions is crucial as they serve as candidates for the subsequent mitigation process. Despite that CIPHERGUARD and CIPHERFIX employ different compilers, i.e., Clang and GCC, we recognize that the fundamental control-flow remains consistent across both compilers subjected to the same optimization settings. Leveraging this similarity, we extract binaries containing instrumentation information utilized during the taint analysis stage. Subsequently, we analyze the basic blocks of tainted functions in these binaries by constructing the control-flow graphs (CFGs). Within each basic block, we identify and label the tainted instructions, facilitating statistical analysis. Figure 3.9 depicts a comparison example of the function `SHA512_Transform` from libsodium-SHA512. The left sub-figure represents the CFG generated by CIPHERGUARD, while the right sub-figure corresponds to the CFG generated by CIPHERFIX. As observed, the CFG structures generated by both CIPHERGUARD and CIPHERFIX exhibit similarities. Three crucial nodes have been manually identified to further partition the control flow. We further conduct a line-by-line comparison of the code within the two binaries, and observe very small divergence, with an average difference of only 2%. We conclude the variance in compilers does not significantly influence the disparity in the number of tainted instructions identified by CIPHERGUARD and CIPHERFIX in Table 3.5.

We observe that the cryptography libraries listed in Table 3.5 are monitored by a similar number of tainted functions, with the exception of two implementations from OpenSSL libraries. Therefore, our initial step is to examine the variance

in taint analysis among these cryptography libraries by randomly selecting five tainted functions for comparison. Upon closer examination of the additional instructions marked by CipherGuard, we discover that they fall into several categories. Firstly, CipherGuard conservatively tags the parameters of temporarily assigned variables utilized to store intermediate calculation results. For instance, the size of the temporarily assigned variable serves as a loop variable involved in the calculation of sensitive variables. Meanwhile, CipherFix omits the intermediate or resultant variables in operations with sensitive variables, focusing more on the direct involvement of the secret key. Therefore, when turning to the OpenSSL libraries, CipherGuard marks the addition parameters of temporarily assigned variables, as well as intermediate and resultant variables. Consequently, more relevant functions and instructions are identified compared to CipherFix, as demonstrated in Table 3.5.

We highlight that the comparison of taint coverage serves merely to demonstrate the superior performance of CipherGuard which covers a significantly larger number of instructions than CipherFix. The focus of this paper is not on the taint analysis itself. Hence, we refrain from delving into the detailed implementation of the taint analysis component of both CipherGuard and CipherFix.

Fair Performance Comparison. To facilitate a fair performance comparison between CipherGuard and CipherFix, we evaluate them based on as many tainted instructions as possible within the same cryptography library. However, it is challenging to achieve this due to the inherent variability in the number of tainted functions and instructions across these tools. For example, protecting the tainted instructions in the loops causes an execution with more cycles. To address this, we manually select the tainted functions that are common to both tools – CipherGuard applies the mitigation process to functions that CipherFix has flagged. This ensures a more aligned execution flow between the two tools. Despite these efforts, we acknowledge that CipherGuard still identifies more instructions than CipherFix, albeit as close as possible.

The numbers of selected tainted instructions are detailed in the fourth to last columns of Table 3.5. Both CipherGuard and CipherFix insert masking instructions with the `vaesenc` instruction from AES-NI, effectively excluding the effects of any optimization specific to CipherGuard. The results reveal that CipherGuard incurs an average overhead of 1.64×, while CipherFix exhibits a

higher overhead of 2.87×. An indicator labeled "consume" is introduced to quantify the additional average cycles incurred by the patched cryptography library. It highlights an average consumption of 3,661 cycles in CipherGuard and a substantial 3.46× increase in CipherFix.

In addition to the differences in the execution flow, the notable variation between the two tools are attributed to specific aspects of CipherFix. These aspects include frequent jumps to the instrumentation code, monitoring of `malloc` to allocate heap memory for mask buffers, and the necessary initialization process for the mask cache during runtime. Conversely, CipherGuard adopts a compiler-aided method that optimizes these time-consuming tasks by sequentially inserting mask instructions and efficiently managing nonce buffers in the `.bss` section. The observed performance difference underscores the effectiveness of CipherGuard's design in streamlining the mitigation process and minimizing the associated overhead.

3.6.4 Comparison with Obelix

Ciphertext Freshness. In contrast to CipherFix or CipherGuard, which utilize masking techniques to ensure ciphertext freshness for each sensitive memory write, Obelix employs address rotation to achieve this goal. During compilation, the program is divided into structurally uniform code blocks, designed to be indistinguishable to side-channel attackers, and data is partitioned into equally sized blocks, with this division occurring transparently at runtime. By leveraging ORAM, the program's execution is obfuscated through access to multiple code and data blocks, including dummy blocks. To counteract ciphertext side-channel vulnerabilities, especially for large amounts of data, additional strategies are applied: they are protected using padding, where each encryption block consists of a data chunk and a counter.

Obelix offers several advantages over CipherGuard such as reducing additional memory consumption for random nonce storage and avoiding complex indexing to locate random nonces, thereby preventing hash collisions and ensuring more reliable mitigation. However, it also faces certain drawbacks. First, it is subject to the inherent limitations of ORAM, which introduces numerous dummy operations and data read/write cycles. Meanwhile, rotating addresses of large memory objects is more costly, making it suitable primarily for small regions like the data scratchpad.

In addition, ORAM implementation requires splitting large data into chunks of half the encryption block size, such as dividing 16 bytes of ciphertext into two 8-byte ciphertexts. In summary, while Obelix provides advantages in memory usage and mitigation reliability, it also introduces performance overhead due to the complexities of ORAM and the costs associated with rotating large memory objects.

Performance Comparison. The performance comparison between Obelix and CipherGuard, as shown in Table 3.6, highlights the overhead introduced by Obelix across different cryptographic algorithms in mbedTLS. Obelix incurs significant performance overheads, with the highest factor observed in the mbedTLS-ECDH implementation (88,852), followed by mbedTLS-RSA (78,750). Other implementations like mbedTLS-AES and mbedTLS-ChaCha20 still show unsatisfiable overheads of 1,607 and 4,403, respectively. On average, Obelix introduces an overhead factor of 34,831.

In contrast, the worst-case performance overhead for CipherGuard remains substantially lower, with the highest overhead in mbedTLS-RSA at 2.92 and other implementations showing overheads ranging from 1.75 to 2.32. This comparison indicates that Obelix introduces higher performance overheads across all tested cryptographic functions compared to the worst-case scenario for CipherGuard. Overall, Obelix provides enhanced security by mitigating ciphertext side-channel vulnerabilities, but at the cost of significantly higher performance overhead compared to CipherGuard.

Table 3.6: Performance comparison between Obelix and CipherGuard. The factor data comes from Obelix paper and Table 3.2.

Implementation	**Obelix Factor**	**Worse-case in CipherGuard Factor**
mbedTLS-AES	1607	2.09
mbedTLS-Base64	541	1.75
mbedTLS-ChaCha20	4403	2.32
mbedTLS-ECDH	88852	2.26
mbedTLS-RSA	78750	2.92
Average	34831	-

3.6.5 Security Analysis

Locations of Nonce Buffers. In CipherGuard, the nonce buffers contain both pre-generated random numbers and currently used nonces, stored in the `.bss` section. The security of masking operations across all variants relies on the integrity and confidentiality of these nonce buffers to prevent leakage. While accessing these buffers may create execution traces that could be exploited, their security is protected by Address Space Layout Randomization (ASLR), which loads the `.bss` section into different memory locations each time. Attackers would need to exploit memory vulnerabilities in the software, which is not the main focus of CipherGuard's protections. Thus, CipherGuard provides similar security guarantees as CipherFix for nonce buffers.

Furthermore, even if ASLR is bypassed [219], the security of the nonce buffers remains robust due to SEV memory encryption, which ensures that these buffers are always encrypted. The pre-generated nonce buffer stays consistent, leading to no variation in its ciphertext, while the currently used nonce is updated with each use during masking operations, resulting in unpredictable ciphertext.

Quality of Nonce. In CipherGuard, the secret-aware register allocation plays a crucial role in fundamentally eliminating ciphertext side-channel attacks by preserving secrets within registers. However, the effectiveness of our software-based probabilistic encryption scheme ultimately hinges on the quality of random number generation. Therefore, to provide robust security guarantees, it is imperative to ensure the reliability and randomness of the generated nonces.

We use the ECDSA Montgomery ladder algorithm from OpenSSL (Figure 3.3) as an example to explore how secrets are safeguarded by masking operations in Variant 1 and Variant 2 of CipherGuard, with three different random number generation techniques. Without any protection, the vulnerability point, where $pbit \leftarrow pbit \wedge kbit$ spills the secret *pbit* into memory, can be exploited by attackers to construct mappings of ciphertext-plaintext pairs and then infer each bit of the secret. We aim to calculate the distribution changes of *pbit* without and with the software-based probabilistic encryption method. To achieve this, we first disassemble the binary to determine the entry address of the vulnerability point. Then we employ hooks inserted by Pintool [170] to print the context when the instruction at this point is executed. By running the cryptography library twice with different ECDSA

private keys, we track the writing of *pbit* 512 times within the loop. To quantify the distribution changes of the secrets, we measure the entropy of the collected secret sequence as $H(X) = -\sum_{x \in \mathcal{X}} p(x) log_2 p(x)$. The results under different variants are presented in Table 3.7.

TABLE 3.7: Entropy of the secret distribution under different variants. For each variant, we run the cryptography library twice to ensure that the original secrets are different.

Secret	Orig	RDRAND	AES	XS+
secret1	0.99946	9.0	-	-
secret2	0.99911	9.0	-	-
secret3	0.99972	-	9.0	-
secret4	0.99910	-	9.0	-
secret5	0.99841	-	-	9.0
secret6	0.99814	-	-	9.0

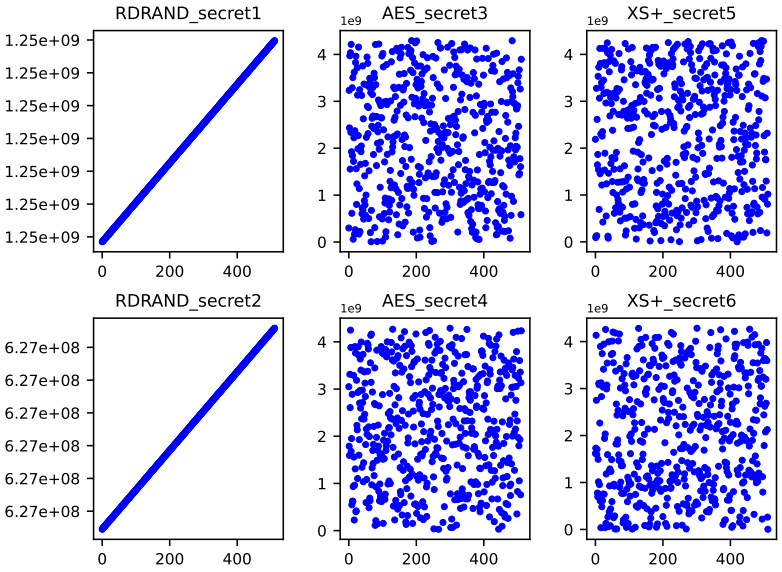


FIGURE 3.10: Scatter distribution of masked *pbit* under different variants. Each secret sequence comprises 512 values.

For the six different unprotected secrets, the entropy is similar, hovering around 0.99, as each bit can only be either 0 or 1. This implies that the entropy of these unprotected secret sequences leaks a significant amount of information. Then the attacker can easily infer the pattern of the secret sequences from the ciphertext side channel. After applying Variant 1 and Variant 2, the entropy of the secret sequences reaches 9.0, which is close to the "upper-bound" information entropy score

of 512 completely random numbers. Each iteration of the secret becomes random, and it becomes exceedingly difficult for the adversary to deduce any meaningful pattern from the masked secret sequences. The scatter distribution of the masked secret sequences, as illustrated in Figure 3.10, showcases a completely random and patternless distribution, further reinforcing the security of the masking approach in CipherGuard.

Re-use of Nonces. In CipherGuard, the nonce selection is based on the address of the target secret, leading to nonce reuse for secrets at the same address. When a function repeatedly calls the same callee, the stack frames share the same address space. However, CipherGuard maintains security through two key mechanisms: varying parameters that keep ciphertext unpredictable under SEV and a masking process that safeguards the callee against information leakage. As a result, an attacker can only deduce that the same secret is being used by noticing identical ciphertext sequences.

Increment-by-Three in Nonces. Updating nonces in Variant 1 is essential for ensuring CipherGuard's security. When consecutive keys differ significantly, masked secrets produce unpredictable ciphertexts. However, in cases where the key changes by only one bit, a weak nonce update (e.g., incrementing by one) can expose this change through identical masked plaintexts. For example, as illustrated in Figure 3.11, observing masked plaintexts starting from even and odd nonces can leak up to four key bits, weakening security.

Keys	1	1	0	1	0
Even Nonce	0x4A7C3D94	0x4A7C3D95	0x4A7C3D96	0x4A7C3D97	0x4A7C3D98
Observe 1	0x4A7C3D95	0x4A7C3D94	0x4A7C3D96	0x4A7C3D96	0x4A7C3D98
Odd Nonce	0x25647b21	0x25647b22	0x25647b23	0x25647b24	0x25647b25
Observe 2	0x25647b20	0x25647b23	0x25647b23	0x25647b25	0x25647b25

Figure 3.11: A corner case arises when a one-bit change in the secret can be revealed by observing identical masked plaintext across multiple masking.

This weak nonce update scheme can be resolved by incrementing the nonce by three, ensuring that at least two bits of the masked plaintext differ between updates. This approach blocks identical ciphertexts from revealing bit changes while maintaining a similar performance.

3.7 Discussion

Limitations. CipherGuard relies on the dynamic taint analysis to detect potential sensitive memory access instructions. To ensure comprehensiveness of protection, we employ a conservative dynamic taint analysis method across multiple secrets and inputs. While this method yields comprehensive taint results, it may trigger unnecessary protection over the memory writes that do not lead to actual ciphertext side-channel leakage, thus incurring additional runtime overhead. Eliminating this overhead necessitates a static, whole-program taint analysis capable of precisely identifying all sensitive memory access instructions that lead to true leakage. However, this remains an open problem, as contemporary research can only statically analyze a small subset of the program [134, 148].

Potential Extensions. CipherGuard supports a comprehensive set of most commonly used x86 instructions, including instructions for data movement, arithmetic and logical calculations involving sensitive memory locations, comparison instructions referencing sensitive memory cells, and zero or flag extension instructions, as well as frequently used SSE/AVX vector instructions. This coverage is already sufficient for mitigating ciphertext side-channel vulnerabilities in common cryptography applications. Nevertheless, CipherGuard offers a flexible interface to accommodate additional instructions not currently handled by the tool. This underscores its high degree of extensibility for future enhancements and adaptations to evolving security requirements and architectural changes.

Compiler Compatibility. CipherGuard is general and stands independent of specific features provided by any compiler framework. For instance, to implement CipherGuard with GCC, one can replace the syntax specific to LLVM IR with that of GCC IR while maintaining consistent taint propagation rules. Tainted instructions identified during the taint analysis phase serve as targets for protection within the lower-level GCC IR. Integrating the mitigation process into the register allocation pass of GCC is straightforward, aligning with CipherGuard's design philosophy of seamless integration into compiler pipelines.

3.8 Conclusion

This chapter presents CipherGuard, a compiler-aided tool evaluated to efficiently and accurately mitigate ciphertext side channels on real-world cryptography applications, demonstrating high precision, efficiency, and scalability.

Chapter 4

Noninterference-based Verification of Side-channels in Microarchitectural Designs

4.1 Introduction

Micro-architectural side-channel attacks have incurred serious threats to computer security over the past decades [160]. These side-channel attacks mainly exploit the timing observations from hardware components (e.g., CPU cache [5, 7], Translation Look-aside Buffer (TLB) [18, 220]) to infer confidential information. The essence of these attacks is the interference [221] from the memory accesses between different programs or even inside one program. Such interference leaves regular footprints on certain hardware components, which can be captured by an adversary to recover confidential information about the victim program. Past works have demonstrated successful attacks to steal cryptographic keys (symmetric ciphers [3], asymmetric ciphers [11, 13, 222], signature algorithms [142–144], post-quantum ciphers [223]), keystrokes [224], visited websites [225], and system configurations [226].

To mitigate cache side-channel attacks, a variety of defense solutions have been proposed. One promising direction is to design security-aware hardware components to reduce or prevent side-channel information leakage. These designs mainly follow two kinds of strategies. The partitioning-based solutions [67] physically partitioned the shared cache components into multiple zones for different domain applications

to achieve strong isolation. The randomization-based solutions [67, 78–80] obfuscated the adversary's observations by randomizing the cache states. These architectures exhibit great generalization and efficiency in protecting the programs running atop them. Although these architectures have been thoroughly considered and evaluated by researchers during the design phase, *it is still important to check whether there are any potential security vulnerabilities in these sophisticated cache systems before fabricating the actual chips.*

Over the past years, various methods have been proposed to evaluate cache side-channel vulnerabilities in hardware components. Unfortunately, they suffer from certain limitations, making it hard to apply them for practical and comprehensive verification. Specifically:

- Some works [67, 78–80] simulated the mechanisms of the newly designed caches against different types of side-channel attacks and empirically evaluated their effectiveness. Due to the lack of formal verification, they are not comprehensive, and can possibly miss some side-channel vulnerabilities. It also takes a lot of time to perform the cycle-accurate simulations in order to obtain convincing evaluation results.

- A couple of approaches [136–138] abstractly described the cache behaviors and define the execution paths that are treated as suspicious behaviors under a side-channel attack. Thereafter, they exhaustively search whether these suspicious behaviors are hidden in the cache behavior combinations. However, the modeling process is not formally guaranteed. Besides, the analysis is based on the exhaustive exploration of the execution traces, which can easily suffer from the combinatorial explosion issue.

- Another challenge in verifying cache architectures is their probabilistic behaviors. To handle this issue, some works introduce methods based on statistics and entropy for security analysis. Zhang et al. [139] formally constructed a cache state transition simulation through model checking techniques to get stable probability matrices within finite steps. To quantify information leakage, they calculated the mutual information between the input distribution and observable outputs. He et al. [140] established a probabilistic information flow graph to model the interaction between the victim and attacker programs in the CPU cache. They defined the concept of security-critical

paths as the union of an attacker's observation and a victim's information flow. Equal probability of each node throughout the security-critical paths means the attacker cannot distinguish the victim's cache-accessed information. These methods face the balance issue between probability accuracy and state explosion. Besides, manual analysis and associating the probability to hardware behaviors can lead to incomprehensive conclusions.

- New hardware description languages (HDL) were introduced to design secure hardware circuits [227–229] with formal proof. These solutions are not comprehensive for side-channel analysis: they can only be applied to the partitioning-based caches while failing to evaluate the randomization-based designs with stochastic behaviors. Besides, they are not user-friendly and need manual work for attaching security labels and defining security policies.

To overcome the limitations of assessing the security of cache designs, this chapter introduces a novel methodology based on formal methods by theorem proving to comprehensively verify the security of cache architectures against side-channel attacks. First, we formalize the specifications of cache designs in an event-state machine way. We offer functional correctness proofs to guarantee the consistency between the specifications and designs, which is ignored in prior works. Second, we design a noninterference reasoning framework to verify the side-channel vulnerability resident in the cache specifications. It adopts the concept of entropy [230, 231] as the theoretical basis to assess the information leakage. We propose two unwinding conditions to unify and evaluate different types of secure caches (e.g., partitioning-based, randomization-based), making our solution comprehensive. Third, we implement our framework in Isabelle/HOL [232], and adopt it to verify eight state-of-the-art cache designs. In summary, we make the following contributions:

- We implement a noninterference reasoning framework based on information entropy that unifies both the deterministic and non-deterministic event models. We define nonleakage as security property by mutual information and derive two general unwinding conditions. We design interfaces for this framework to offer verification services.

- We formally specify each cache design in an event-state machine way on top of general set-associative cache layouts, forming a complete cache specification. We prove the cache specification is an instantiation of the reasoning framework, hence can be efficiently verified security properties.
- We evaluate our entropy-based noninterference reasoning framework on eight state-of-the-art cache designs. The verification practice shows that our methodology possesses high theoretical reliability and flexibility.

We present the background about cache side-channel attacks and mutual information in Section 4.2. We give the threat model and briefly describe the methodology in Section 4.3. The design of reasoning framework is shown in Section 4.4. Section 4.5 presents a case-study, and Section 4.6 analyzes the verification results of eight state-of-the-art cache architectures. Section 4.7 concludes this chapter.

4.2 Background

4.2.1 Cache Side-channel Attacks

Modern computer architectures adopt various optimizations to accelerate the execution, at the cost of confidentiality threats. One typical example is the caching technique: a small hardware unit is introduced to store the recently accessed data, which are expected to be used again due to the locality principle. Obtaining data directly from this unit is much faster. However, such timing differences enable an adversary program to recover the victim program's access traces, when these programs share the same hardware component. Such attacks have been realized in different levels of CPU caches and TLBs.

Cache hierarchy. Most CPU caches are organized in a n-way set-associative way. A n-way set-associative cache can be treated as a two-dimensional data array. Each row is called a *cache set*, which is further divided into n *cache lines*. Each memory block is mapped to one cache set indexed by its memory address. This block can be stored in any cache lines in this set, determined by a replacement policy. When a CPU core wants to access a memory block, if it resides in the cache, the CPU can directly obtain it, resulting in a cache hit with a fast access speed. The CPU

has to fetch the data from the main memory to the cache, otherwise. This results in a cache miss with a much slower access speed. Particularly, a cache with only one way in each set (i.e., $n = 1$) is a direct-mapped cache, while a cache with only one set is called fully-associative.

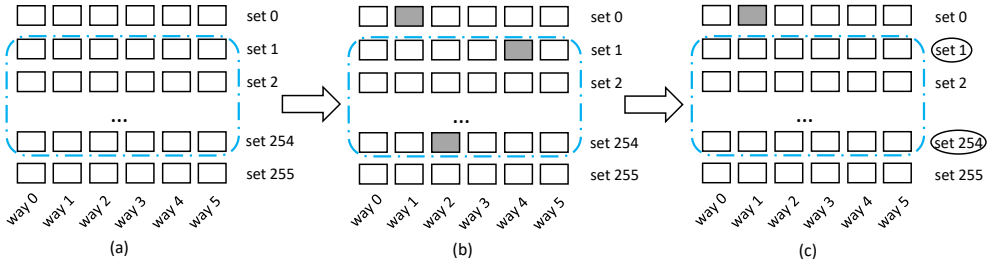


FIGURE 4.1: Side-channel attack scheme. Sub-figure (a) represents the preparation phase, (b) the waiting phase, and (c) the observation phase.

Cache attacks. The first cache attacks deduce cryptographic secrets by observing the whole execution time [2, 3, 6, 8]. In recent days, cache side-channel attack techniques narrow down to a smaller granularity. The timing difference between a cache hit and a cache miss can reveal information about the program's access traces. A cache side-channel attack typically involves three steps (see Figure 4.1). (1) Preparation: the adversary manipulates the states of certain cache lines with its own address space [233]. For instance, PRIME-PROBE attack [5] fills up the entire critical cache sets, while a FLUSH-RELOAD attack [13] and a FLUSH-FLUSH attack [163] evict certain cache lines through the *clflush* instruction. The area controlled by the adversary is shared with the victim. And, the adversary has the knowledge that the victim's access pattern in this area is related to its confidential information. (2) Waiting: the adversary does nothing until the victim finishes several execution circles. The victim may load its data blocks into the cache and replace the cache lines occupied by the adversary. (3) Observation: the adversary collects the footprints left by the victim program. For example, the PRIME-PROBE attack re-accesses the critical cache set to check if certain blocks were evicted by the victim. The FLUSH-RELOAD attack reloads the target cache lines to determine if it has been touched by the victim. The FLUSH-FLUSH attack re-flushes the target cache lines to check whether data is loaded into these lines by the victim. The victim's cache access pattern is thus leaked to the attacker.

TLB attacks. Modern memory systems utilize a Memory Management Unit (MMU) to translate the virtual addresses (VA) issued from CPU to physical addresses (PA) for accessing main memory. The VA-PA mapping is stored in the page table for translation. A small hardware unit, TLB, is implemented to store the latest used mapping to accelerate the translation. Similar as the CPU cache, a TLB is also organized by the *set* and *way* structure. Therefore, the side-channel techniques from cache can be extended to attack the TLB to extract the memory accesses [18, 220].

4.2.2 Mutual Information

Intuitively, the concept of noninterference tells that one domain (denoted as victims) does not affect the observation of another domain (denoted as adversaries). The concept is consistent with the cache side-channel attack schemes because an adversary can deduce the victim's memory access patterns that are associated with secrets when its observation depends on the victim's behaviors. Furthermore, from the perspective of the probability distribution, information leakage of the side-channel is equivalent to the existence of a dependency relationship between the input (victim's behaviors) and output (adversary's observation). And, this kind of dependency relationship can be calculated by mutual information. This is the motivation of this chapter where we interpret the cache side-channel attack schemes by noninterference and measure the information flow of this noninterference through mutual information of Shannon Theory [230, 231].

We denote a victim's behaviors as the uncertain information that an attacker wishes to explore by side-channel attacks. This information is viewed as input X and has probability distribution $\mathcal{X}$. First, entropy defines the uncertainty of the information itself, of $H(X) = -\sum_{x\in\mathcal{X}} p(x)log_2p(x)$. Second, conditional entropy measures the uncertainty about X when the attacker has the knowledge of output Y. It is defined as $H(X|Y) = -\sum_{y\in\mathcal{Y}} p(y)\sum_{x\in\mathcal{X}} p(x|y)log_2p(x|y)$. Lastly, mutual information between X and Y measures the information that an adversary can learn about X if he gains the knowledge through output Y, defined as $I(X;Y) = H(X)-H(X|Y)$. One property of mutual information is that it is symmetry: $I(X;Y) = I(Y;X)$. It can be calculated through a joint probability matrix, as shown in equation 4.1.

$$I(X;\,Y) = \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y) log_2 \frac{p(x,y)}{p(x)p(y)} \tag{4.1}$$

$$= \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x)p(y|x) log_2 \frac{p(y|x)}{p(y)} \tag{4.2}$$

4.2.3 Isabelle/HOL

Isabelle/HOL [232] is a higher-order logic theorem prover. It offers common types (e.g., naturals (*nat*), integers (*int*) and booleans (*bool*)). The keyword *datatype* is used to define an inductive data type. Composed data types include tuple, record, list, and set. Projection functions *fst* and *snd* return elements t_1 and t_2 of a tuple $(t_1 \times t_2)$. Isabelle/HOL offers record type to include multiple elements of different data types. Assignment symbol = is used to initialize the contents of a record, while := is used to update it. Lists are defined by an empty list denoted as [ ], and a concatenation constructor represented as #. The ith component of a list xs is accessed by $xs!i$. The cardinality of a set s (i.e., $|s|$) is denoted as $card\ s$, returning zero when set s is infinite. Isabelle/HOL provides *definition* command to specify a non-recursive function, while *primrec* command for primitive recursions.

Isabelle/HOL supports parametric theories with the keyword *locale*. A *locale* includes a series of parameter declarations (with keyword *fixes*) and assumptions (with keyword *assumes*). Isabelle/HOL users can instantiate a locale through *interpretation* command, where concrete data is assigned to parameters and declarations are added to the current context. We construct a parametric noninterference reasoning framework through *locale* command and instantiate it in different cache architecture scenarios through *interpretation* command.

4.3 Methodology Overview

4.3.1 Threat Model

We consider the cache architectures to be verified are involved in the following threat model. The victim and the attacker share the same cache environment and

a cross-core/VM attack allows the attacker and the victim to execute in parallel on different cores/VMs. The attacker cannot directly observe the memory content from the CPU, probing cache states to check whether the victim's data is resident in the cache indirectly instead. This model captures most cache side-channel attacks in the literature. For example, EVICT-TIME attack [5] measures the latency of victim's program, PRIME-PROBE attack [5, 7, 11], FLUSH-RELOAD attack [13] and FLUSH-FLUSH attack [163] measure the latency of attacker's program.

We also mention that the attacker accurately monitors both cache set and cache line granularities. This is because modern OS adopts the page sharing technique that removes the duplication of shared libraries, enabling probing the shared libraries narrow to a cache line.

4.3.2 Architecture

In a micro-architectural side-channel attack, an adversary can deduce the memory access pattern when its observation is dependent on the victim's behaviors. From the perspective of the joint probability distribution, side-channel leakage is equivalent to the existence of a dependency relationship between the input and output of the channel. This is the motivation to employ joint probability distribution for side-channel quantification and verification.

The workflow of our proposed methodology is shown in Figure 4.2. It includes two components. (1) A reasoning framework is designed to quantify the information leakage of the target system. The essence of the framework is to interpret the noninterference property through mutual information. (2) A complete cache specification includes the cache behavior formalization and the general cache layouts. The cache behavior is described as an event-state transition. Its formalization is first proved to meet the consistency with its design. The cache specification instantiates the interface layer offered by the reasoning framework. Therefore, for verifying whether a cache specification satisfies the security properties, we only need to verify whether it satisfies two unwinding conditions. Violations of both conditions indicate the cache design is vulnerable to side-channel attacks. In this chapter, we mainly focus on the fundamentals of information leakage, while skipping the analysis of adversarial strategies for extracting the secrets from the footprints of

the victim program. As shown in Figure 4.2, the reasoning framework contains the following components.

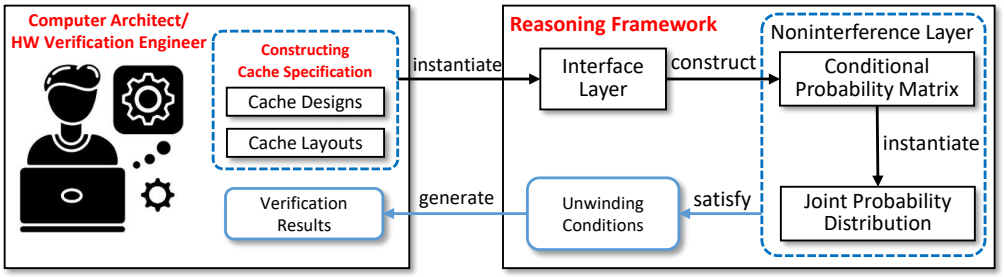


FIGURE 4.2: Workflow of our proposed approach.

Interface layer: this layer is used to connect the given cache specification to the noninterference layer. The interface layer offers an event-state transition function and output function to parse cache behaviors into probabilistic representations.

Noninterference layer: this is the core of our reasoning framework. It introduces a parameterized joint probability distribution between the victim's information X and attacker's observation Y. We calculate the mutual information $I(X;Y)$ as information flow security property to quantify how much information the attacker can learn about X from Y. The joint probability distribution can be constructed by multiplying a series of probabilistic inputs and a conditional probability matrix. The conditional probability matrix models the relationships between the input and the output of cache designs.

Unwinding conditions: this defines the conditions to satisfy information flow security property, meaning to make the mutual information zero according to equation 4.2. We deduce two unwinding conditions as shown in equation 4.3, when we stipulate that each input probability is greater than zero. The first condition $C1$ indicates that the attacker learns nothing when there is no observation. The second condition $C2$ shows the attacker's observation is constant and independent of the victim's behaviors.

$$\begin{aligned} C1 &: \forall x\ y.\ p(y|x) = 0 \longrightarrow I(X;Y) = 0 \\ C2 &: \forall x\ y.\ p(y|x) = p(y) \longrightarrow I(X;Y) = 0 \end{aligned} \tag{4.3}$$

4.3.3 Available Proving Technique

Separation logic [234] is a powerful tool for reasoning about memory and can prove properties such as the independence between different regions of memory or different parts of a system. From this perspective, separation logic can indeed be used to prove cache side-channel leaks, where changes to one part of the cache do not affect the rest of the system. Furthermore, separation logic provides a more granular analysis of cache usage, allowing for precise reasoning about the transitions between different cache states.

However, in the context of side-channel verification, specifically proving the dependency relationship between the input and output of the channel, it is not necessary to prove that a specific cache entry remains unchanged during cache state transitions. This is because randomization-based cache designs may affect any cache entry when an insecure cache miss occurs, making it difficult to guarantee that a cache entry stays the same throughout state transitions. In this scenario, using separation logic could potentially complicate the proof of side-channel leaks.

We focus on verifying whether changes in cache entries result in detectable information leakage, and our framework provides the precision needed to ensure the accurate detection of side-channel leaks. While separation logic could theoretically assist in proving isolation between cache entries, our formalization relies on mutual information in Isabelle/HOL to verify whether the leakage caused by changes in specific cache entries is significant enough to be detected. This approach enables us to prove the properties of information flow and leakage, rather than concentrating on low-level memory separation. While memory separation can be seen as complementary to our framework, it is not the primary focus of this analysis.

4.4 Design of Reasoning Framework

In this section, we provide more details about our reasoning framework.

4.4.1 An Abstract State Machine

A cache specification implements a state machine through instantiating the interface layer. We construct the interface layer for the purpose of re-usability because verification of any cache architecture only requires instantiating the interface functions.

First, we model the input distribution space as $\mathbb{P}(\mathcal{I} \times \mathcal{P})$, which is the power-set of type $\mathcal{I} \times \mathcal{P}$. Label $\mathcal{I}$ describes the input content and $\mathcal{P}$ is of real type describing probabilities. We further stipulate any valid input distribution is neither empty nor infinite (we follow the Isabelle/HOL definition where the cardinality of infinite sets is zero). We omit input elements with zero-probability because they will not result in any outputs. We also guide that all inputs are different and the sum of the probabilities of all inputs equals one. We use the operator . to denote an attribute of a input, e.g., $x.i$ and $x.p$ represent the content and probability of input x, respectively.

Definition 4.1 (Valid Input Distribution Sets)**.**

$$makeInput \triangleq \{\mathcal{X}.\ \mid \mathcal{X} \mid > 0 \wedge (\forall m\ n \in \mathcal{X}.\ m \neq n \longrightarrow m.i \neq n.i) \wedge (\forall d \in \mathcal{X}.\ d.p > 0) \wedge \sum_{d \in \mathcal{X}} d.p = 1\}$$

Next, we formalize the event-state transition function as ψ, which describes a non-deterministic event model. It is a single step execution triggered by the event label and the input, of type $\mathcal{E} \times \mathcal{X} \rightarrow \mathbb{P}(\mathcal{S} \times \mathcal{S})$. Label $\mathcal{S}$ represents the state space and $\mathcal{E}$ is the set of event labels. We use $\psi(e, x)/s$ to represent that all event-state transitions happen when we execute the event e on the state s with input x.

Definition 4.2 (Event-state Transition Function from State s)**.**

$$\psi(e, x)/s \triangleq \{t.\ t \in \psi(e, x) \wedge (\exists s'.\ t = (s, s'))\}$$

Then, we define an abstract output function that extracts the output from each transition tuple (s, s'), of type $\varpi : (\mathcal{S} \times \mathcal{S}) \rightarrow \mathcal{O}$. Label $\mathcal{O}$ describes the output content. With these functions, we construct an abstract state machine in the interface layer.

Definition 4.3. The interface layer implements an abstract state machine $\mathcal{M}$ as tuple $\langle \mathcal{S}, \mathcal{E}, \mathcal{X}, \mathcal{O}, \psi, \varpi \rangle$, where $\mathcal{S}$ is the state space, $\mathcal{E}$ is the set of event labels, $\mathcal{X}$ and $\mathcal{O}$ are the valid input distribution and output content respectively, ψ is the event-state transition function, and ϖ is the output function.

4.4.2 Noninterference

The concept of noninterference indicates that the behaviors of one domain do not affect the observation of another domain [221]. In our reasoning framework, these two domains correspond to the victim's input and the attacker's observation in the side channel. To describe such a side-channel mechanism, we construct a joint probability distribution through the functions from the interface layer step by step. We quantify the effect of the interaction between the victim and attacker by calculating mutual information from the joint probability distribution.

A joint probability can be written as $P(X)P(Y|X)$. Therefore, to instantiate a joint probability distribution is to construct the input distribution $P(X)$ and a conditional probability matrix $P(Y|X)$. The input distribution can be directly inherited from the interface layer. For example, it is any set that satisfies $\mathcal{X} \in makeInput$.

To construct a conditional probability matrix, we first introduce a conditional probability transition function Cpt. It first applies output function to each state transition tuple to get all possible outputs, shown as $\mathcal{O} = \{d.\ \exists t \in \psi(e,x)/s.\ d = \varpi(t)\}$. Then, for each output $o \in \mathcal{O}$, it counts all the transitions that produce the output o to get the probability, which is the proportion of these transitions in the total transitions. The definition of Cpt is as follows.

Definition 4.4 (Conditional Probability Transition Function)**.**

$$
\begin{aligned}
Cpt(e,x)/s \triangleq \{y.\ \exists o \in \mathcal{O}.\ & \\
& \mathcal{T}_{sub} = \{t.\ t \in \psi(e,x)/s \wedge \varpi(t) = o\} \wedge \\
& y.o = o \wedge y.p = \frac{\mid \mathcal{T}_{sub} \mid}{\mid \psi(e,x)/s \mid}\} \\
where\ \mathcal{O} = \{d.\ \exists t \in \psi(e,x)/s.\ d = \varpi(t)\} &
\end{aligned}
$$

The result of this function is the output distribution $\mathcal{Y}$, of type $\mathbb{P}(\mathcal{O} \times \mathcal{P})$. We can easily prove the properties the output distribution satisfies as follows.

Lemma 4.1 (Output Distribution)**.**

$$\mathcal{Y} = Cpt(e, x)/s \; and \; it \; satisfies \;\; | \, \mathcal{Y} \, | > 0 \wedge$$
$$(\forall m \; n \in \mathcal{Y}.\; m \neq n \longrightarrow m.o \neq n.o) \wedge (\forall d \in \mathcal{Y}.\; d.p > 0) \wedge \sum_{d \in \mathcal{Y}} d.p = 1$$

The function Cpt only takes one input while the valid input distribution $\mathcal{X}$ contains limited input contents. Therefore, the next step is to apply the function Cpt to each input in $\mathcal{X}$. The result of this process is a conditional probability matrix $\mathcal{W}$. Each row of the matrix ($w[y_1|x_i]$,$w[y_2|x_i]$ $\ldots$ $w[y_{|Y|}|x_i]$) can be viewed as the representation of the conditional probability distribution of output y_1, $y_2 \ldots y_{|Y|}$ under the input x_i.

Now it is time to build the joint probability distribution. The following function $makeJoint$ matches each input x that belongs to the input distribution $\mathcal{X}$ with any conditional probability y that is part of $Cpt(e, x)/s$. Then the joint probability is the product of the corresponding probabilities of these two elements. Joint distribution $\mathcal{J}$ is defined as $\mathbb{P}((\mathcal{I} \times \mathcal{O}) \times \mathcal{P})$.

Definition 4.5 (Joint Probability Distribution)**.**

$$makeJoint \triangleq \{j.\; \exists x \in \mathcal{X}.\; \exists y \in Cpt(e, x)/s.$$
$$j.i = x.i \wedge j.o = y.o \;\wedge j.p = x.p * y.p\}$$

The computation of mutual information from equation 4.1 requires two marginal probability distributions. We take the marginal probability of the input x as an example: we first collect the subset $\mathcal{J}_{sub} = \{j.\; j \in makeJoint.\; j.i = x.i\}$ that takes all elements whose input dimension is equal $x.i$ from the joint probability distribution. Then the marginal probability is the sum of the probabilities of all such elements. Its definition is shown below. We omit the definition of $margOutput$.

Definition 4.6 (Input Marginal Probability Distribution)**.**

$$margInput \triangleq \{mi.\ \exists x \in \mathcal{X}.\ \mathcal{J}_{sub} = \{j.\ j \in makeJoint.\ j.i = x.i\} \wedge mi.i = x.i \wedge mi.p = \sum_{d \in \mathcal{J}_{sub}} d.p\}$$

Now we give the definition of mutual information based on equation 4.1. Function $mutualInfo$ takes each element $j \in makeJoint$ from the joint probability distribution, and then calculates the marginal probabilities of the input (mi) and output (mo) respectively. Afterwards, the value of the mutual information is the accumulation of $j.p * log_2 \frac{j.p}{mi.p * mo.p}$, when iterating the element j.

Definition 4.7 (Mutual Information)**.**

$$mutualInfo \triangleq \sum_{j \in makeJoint} j.p * log_2 \frac{j.p}{mi.p * mo.p}$$

4.4.3 Unwinding Conditions

With mutual information calculated above, we assess the information leakage of noninterference by the following definition.

Definition 4.8 (Information Leakage)**.**

$$nonleakage \triangleq \forall e\ \mathcal{X}\ s.\ mutualInfo = 0$$

According to equation 4.3, two unwinding conditions imply that the mutual information equals zero. We give the definitions of these two unwinding conditions and prove the implication relationships.

Theorem 4.2 (Condition 1: No Observation)**.**

$$\forall e\ s.\ \forall x \in \mathcal{X}.\ \forall y \in Cpt(e, x)/s.\ y.p = 0 \longrightarrow mutualInfo = 0$$

Proof. When the conditional probability $y \in Cpt(e, x)/s$ equals zero, the corresponding joint probability $j.p = x.p * y.p$ also equals zero. Then unfolding

the definition of $mutualInfo$ and substituting $j.p$ as 0, the accumulated result $0 * log_2 \frac{0}{mi.p * mo.p}$ is zero. In the end, the mutual information is zero. □

Theorem 4.2 gives the advice that if the attacker cannot observe anything from the footprints released by the victim, then the cache dose not leak any information. This condition can be used in some partitioning-based designs [67].

Theorem 4.3 (Condition 2: Constant Observation)**.**

$$\begin{aligned} &\forall e\ s.\ \forall x \in \mathcal{X}.\ \forall y \in Cpt(e,x)/s. \\ &\qquad y.p = \sum_{d \in \mathcal{J}_{sub}} d.p \ \longrightarrow mutualInfo = 0 \\ &\qquad where\ \mathcal{J}_{sub} = \{j.\ j \in makeJoint.\ j.o = y.o\} \end{aligned}$$

Proof. For any joint probability $j \in makeJoint$, its corresponding marginal probability of the input $mi.p$ equals its input probability $x.p$ because the sum of the probabilities of all elements in $Cpt(e,x)/s$ equals one. Also, the joint probability $j.p$ can be calculated by $x.p * y.p$. We have $y.p = \sum_{d \in \mathcal{J}_{sub}} d.p$, where $\mathcal{J}_{sub} = \{j.\ j \in makeJoint.\ j.o = y.o\}$ according to the condition 2 above. The accumulated result in the definition of $mutualInfo$, $j.p * log_2 \frac{j.p}{mi.p * mo.p}$ of equation 4.1, is then folded and substituted as $x.p * \sum_{d \in \mathcal{J}_{sub}} d.p * log_2 \frac{x.p * \sum_{d \in \mathcal{J}_{sub}} d.p}{x.p * \sum_{d \in \mathcal{J}_{sub}} d.p}$. Its value equals zero, leading the mutual information to be zero as well. □

Theorem 4.3 describes a scenario where the conditional probability distribution is constant for any input. Therefore, the footprints caused by any input are the same. Note that Theorem 4.3 only requires the values of each column in the matrix $\mathcal{W}$ to be the same. When this condition is applied into cache designs, we can find that some randomization-based strategies further manipulate that the values of $w[y_1|x_i],\ w[y_2|x_i] \ldots w[y_{|Y|}|x_i]$ are in the same probability, which is one special case of the above condition.

In the end, either of these two unwinding conditions can imply no information leakage, as shown in the following theorem.

Theorem 4.4 (Unwinding Conditions Reasoning)**.**

$$(condition1 \vee condition2) \Longrightarrow nonleakage$$

4.5 Application of Our Methodology

In this section, we demonstrate how to verify three existing micro-architectural designs: conventional Set-Associative (SA) cache, security-aware Random-Permutation (RP) cache [67] and Random-Fill (RF) TLB [235] with our framework.

4.5.1 Verifying Cache Designs

Modeling Cache Architecture. We start with the specification of the general cache layouts. A cache line is the smallest unit among cache layouts, which is defined as a **record** $ca\_line = ca\_set,\ ca\_way,\ ca\_tag,\ valid,\ lock,\ owned$. The first three fields directly represent the cache index, cache way and cache tag. The following fields denote whether the cache line is used, whether its content is protected, and which process is occupying it. We define the cache structure and the specification it needs to satisfy as follows: the parameterized cache layouts are constructed by a list whose length is **M**, where each element of the list represents a cache set with **W** cache lines (i.e., W-ways). In a cache set, the *ca_set* identifier of each cache line equals its cache set index, and all cache lines in one cache set have different *ca_way*.

Definition 4.9 (The Cache Structure)**.**

$$
\begin{aligned}
&Cache :: \text{“}(ca\_line\ set)\ list\text{”}\ and\ it\ satisfies : \mid Cache \mid = M \wedge \\
&(\forall l < M.\ \mid Cache\ !\ l \mid = W) \wedge (\forall l < M.\ \forall e \in (Cache\ !\ l).\ e.ca\_set = l) \wedge \\
&(\forall l < M.\ \forall e_i\ e_j.\ e_i \in (Cache\ !\ l) \wedge e_j \in (Cache\ !\ l). \\
&\qquad e_i \neq e_j \longrightarrow e_i.ca\_way \neq e_j.ca\_way)
\end{aligned}
$$

With the cache layouts definition, we formalize the memory request that acts as the input from the victim and is performed by the corresponding cache design on the cache layouts. A memory request is denoted as a **record** $mem\_req = tagbits,\ setbits,\ protected,\ process$. Label *tagbits* is used to compare the tag field of a cache line, and *setbits* is sent to the cache mapping to get the actual cache index. Label *protected* denotes whether the memory data is protected by its owner,

and *process* represents the owner of this memory block in two values: **H** and **L** denote the confidential and non-confidential processes respectively. Last, we design the system state that includes the cache structure and the mapping structure from a memory request to the cache set index. The structure is formally defined as **record** $state = Cache,\ Mapping$.

Specifying Cache behaviors. With these definitions, we construct behavior specifications for two cache designs. First, a SA cache follows the traditional view of a cache design, and we omit the explanation here. Its behavior specification is defined below.

```
definition sa_read ::
  "state ⇒ memory_request ⇒ state set"
  where "sa_read s mr ≡
    let Mapping = s->maps; Cache = s->sram;
        redir_index = Mapping -> (mr->index); // find real set index
        redir_set = Cache ! redir_index in // locate cache set
    if probe_line redir_set mr
      then {s} // tag hits and cache hit
    else let // cache miss
      r_line = replace_policy redir_set; // choose line to be replaced
      nset = insert mr (redir_set - {r_line}) in // replace r_line
      {s(|sram := update Cache redir_index nset|)}" // update cache set
```

We show correctness of the specification by proving that it follows the behaviour of the cache implementation. Lemma 4.5 shows that the execution of the read operation in the SA cache is deterministic and only one cache line in the mapped cache set is updated (l_{mr} refers to the mapped cache set index), while other cache sets remain the same when a cache miss happens.

Lemma 4.5 (Correctness of SA Cache Read)**.**

$$[\![\forall e \in s.Cache\ !\ l_{mr}.\ e.ca\_tag \neq mr.tagbits]\!] \Longrightarrow$$
$$[\![\exists! s' \in sa\_read\ mr\ s.\ (\forall l \neq l_{mr}.\ s'.Cache\ !\ l = s.Cache\ !\ l) \wedge$$
$$|\ (s'.Cache\ !\ l_{mr}) \cap (s.Cache\ !\ l_{mr})\ | = |\ s.Cache\ !\ l_{mr}\ | - 1]\!]$$

Next, for the RP cache, the workflow of handling memory requests is shown in Figure 4.3.

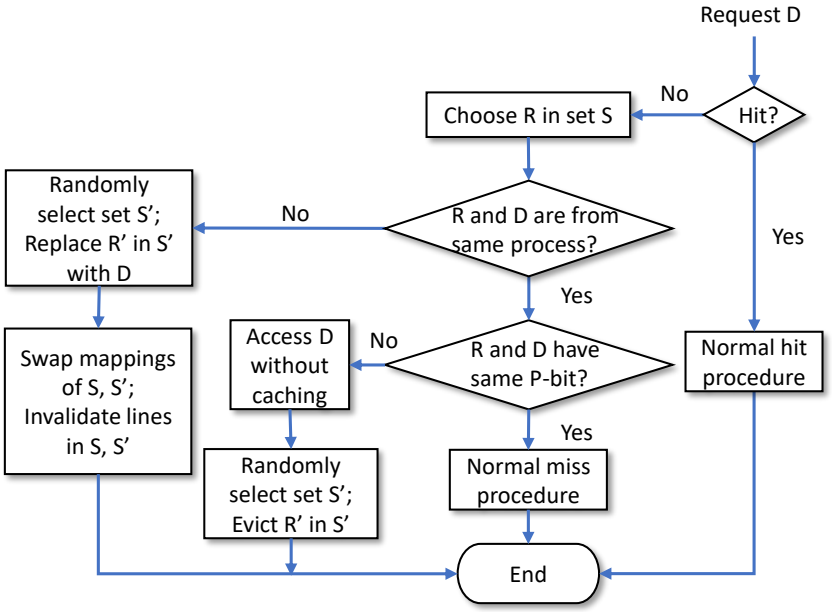


FIGURE 4.3: Workflow of Random Permutation Cache

RP cache utilizes three strategies to randomize the observation of the adversary. (1) When there is an external cache miss (Column 1: the mapped cache line does not belong to the current process), RP cache randomly chooses a cache set and selects one cache line according to the replacement policy in this set to replace the request memory. (2) When there is an internal miss (Column 2: the mapped cache line belongs to the current process but has a different protection flag), RP cache randomly chooses a cache set and selects one cache line according to the replacement policy in this set to evict it without caching. These two strategies will result in non-deterministic cache state transitions. (3) For the external cache miss (Column 1), RP cache also dynamically changes the memory-to-cache mapping, so even if the attacker can deduce the mapping for one read operation, it would fail for the next time. The specification of RP cache is defined below in a way of the event-state transition function.

```
definition rp_read ::
  "state ⇒ memory_request ⇒ state set"
  where "rp_read s mr ≡
    let Mapping = (if mr->thread = H then s->map_H else s->map_L);
        Cache = s->sram;
        redir_index = Mapping -> (mr->index); // find real set index
        redir_set = Cache ! redir_index in // locate cache set
    if probe_line redir_set mr
```

```
    then {s} // tag hits and cache hit
  else let // cache miss
    r_line = replace_policy redir_set; // choose line to be replaced
    policy_line = // randomly select R'
      {t. ∃l ∈ {0..|Cache| - 1}. t = replace_policy (Cache ! l)} in
    if r_line->owned = mr->thread then
      if r_line->lock = mr->protect then let // Column 3
        // normal miss procedure
        nset = insert mr (redir_set - {r_line}) in
        {s(|sram := update Cache redir_index nset|)}
      else // Column 2
        // randomly evict without caching
        {t. ∃r'_line ∈ policy_line. let
            nset = (Cache ! (r'_line->index)) - {r'_line} in
         t = s(|sram := update Cache (r'_line->index) nset|)}
    else // Column 1
      if mr->thread = H then
        {t. ∃r'_line ∈ policy_line.
          if r'_line->index = redir_index then let
          // only invalidate redir_set
            nset = insert mr (empty Cache!(r'_line->index)) in
            t = s(|sram := update Cache (r'_line->index) nset,
              map_H := swap s->map_H redir_index (r'_line->index)|)
          else let
          // invalidate both redir_set and policy_line
            nset = insert mr (empty Cache!(r'_line->index));
            nredir_set = empty redir_set in
            t = s(|sram := update (update Cache (r'_line->index) nset)
             redir_index nredir_set,
               map_H := swap s->map_H redir_index (r'_line->index)|)}
      else // mr -> thread = L
        {t. ∃r'_line ∈ policy_line.
          if r'_line->index = redir_index then let
          // only invalidate redir_set
            nset = insert mr (empty Cache!(r'_line->index)) in
            t = s(|sram := update Cache (r'_line->index) nset,
              map_H := swap s->map_L redir_index (r'_line->index)|)
          else let
```

```
// invalidate both redir_set and policy_line
  nset = insert mr (empty Cache!(r'_line->index));
  nredir_set = empty redir_set in
  t = s(|sram := update (update Cache (r'_line->index) nset)
   redir_index nredir_set,
     map_H := swap s->map_L redir_index (r'_line->index)|)}"
```

We give a lemma to prove the correctness of cache layout when a RP cache read is issued and a miss happens. It indicates that there will be multiple state transitions when a cache miss happens, showing non-deterministic execution. For each transition, there will be one cache line from a random cache set updated, while other cache sets remain the same. Here, l_{mr} refers to the original mapped cache set index.

Lemma 4.6 (Correctness of RP Cache Read)**.**

$$\begin{aligned}&[\![\forall e \in s.Cache\ !\ l_{mr}.\ e.ca\_tag \neq mr.tagbits]\!] \Longrightarrow \\ &[\![\forall s' \in rp\_read\ mr\ s.\ \exists!l.\ (\forall l' \neq l.\ s'.Cache\ !\ l' = s.Cache\ !\ l') \wedge \\ &\qquad\qquad |\ (s'.Cache\ !\ l) \cap (s.Cache\ !\ l)\ | = |\ s.Cache\ !\ l\ | - 1]\!]\end{aligned}$$

Verifying Side-channel Leakage. With the instantiation of the event-state transition function (i.e., the SA and RP cache specifications), we instantiate the remaining two functions to complete the instantiation of the interface layer of the reasoning framework.

Due to direct inheritance, the memory request distribution issued from the victim is regarded as $\mathcal{X}$, of concrete type $\mathbb{P}(mem\_req \times \mathcal{P})$. Next, we instantiate the observation function ϖ. According to the correctness proofs, only one cache set is updated inside those state transitions for each cache miss. Therefore, the attacker can regard the state transition with the same updated cache set as the same observation when re-accessing the cache. For convenience, we use the cache set index to represent the observable state transitions.

Now, we construct the attack-simulated cache layout specification that statically demonstrates how the attacker manipulates the cache layout with its data. It describes the circumstances of the preparation phase shown in the leftmost picture

of Figure 4.1. A cache line e can be differentiated through identifier $e.owned = \mathbf{H}$ and $e.owned = \mathbf{L}$, denoting e is occupied by a confidential or non-confidential process. An attacker use a series of memory accesses without cache collision to fill part of the cache. The specification of manipulated cache below indicates that the attacker's accesses to each memory address in the m_s will result in a cache hit. Thereafter, the victim's access to these manipulated cache areas will change their ownership, resulting in observation to the attacker if re-accessing this memory space.

Definition 4.10 (The Attack-simulated Cache Layout)**.**

$$
\begin{aligned}
&m\_s :: \text{“}mem\_req\ set\text{”}\ and\ it\ satisfies :\\
&\forall m \in m\_s.\ (m.process = L) \wedge (\exists! e \in Cache\ !\ (Mapping \rightharpoonup m.setbits).\\
&\qquad\qquad e.ca\_tag = m.tagbits \wedge e.owned = L)
\end{aligned}
$$

As for the replacement policy in a cache set, we follow the practice in work [11] where they considered the age replacement (e.g., LRU replacement or FIFO replacement) and stipulated that the adversary re-accesses a cache set in a reversed order. In such a way, the victim evicts the oldest cache line at the adversary's waiting state, shown in the middle picture of Figure 4.1. If the adversary probes the cache set in his original order, then the second-oldest (same for the followings) cache line is evicted, leading to a miss on every probe. In contrast, if the attacker probes the cache set in a reversed order, the cache line evicted by the victim can be precisely probed without causing a miss on every probe. We leave the random replacement policy as future work.

With all the preparations above, we turn to the security verification of cache read operation. We prove the read operation in the SA cache breaks two conditions with the following theorem:

Theorem 4.7 (SA Cache Read Produces Fixed Observation)**.**

$$
\begin{aligned}
&\forall x \in \mathcal{X}.\ Cpt(sa\_read, x)/s = \{(s.Mappming_H \rightharpoonup x.setbits, 1)\} \wedge\\
&\qquad (\mid \mathcal{X} \mid > 1 \longrightarrow \sum_{d \in \mathcal{J}_{sub}} d.p \neq 1)\\
&\qquad where\ \mathcal{J}_{sub} = \{j.\ j \in makeJoint.\ j.o = s.Mappming_H \rightharpoonup x.setbits\}
\end{aligned}
$$

Proof. Among the regions controlled by the attacker, any input x can cause an observation with a conditional probability of 1 due to the deterministic execution. Therefore, the SA cache read operation breaks the first unwinding condition. When substituting the conditional probability of 1 with the joint probability distribution, the probability of each element in the joint probability distribution equals the probability of input $x.p$. As we known, a realistic program owns more than one memory address, meaning the number of input distribution $\mathcal{X}$ must be greater than one. Also, other inputs rather than x may be mapped to different cache sets, causing $s.Mappming_H \rightharpoonup x.setbits$ to be different. Then, there exists a marginal output that is not equal to the conditional probability of 1. Therefore, the SA cache read cannot meet the second unwinding condition. □

If we calculate the information it leaks further, the value of each element in the mutual information formula becomes $x.p{*}log_2\frac{1}{x.p}$, where different inputs may result in distinct observations to attackers. Then, we prove the side-channel information leakage in SA cache is $\sum_{x\in\mathcal{X}} x.p * log_2\frac{1}{x.p}$.

Next, we turn to the RP cache read operation. It breaks the first unwinding condition but preserves the second one, which means it produces constant observations to the attacker regardless of the victim's input. We prove the theorem of RP cache read as follows.

Theorem 4.8 (RP Cache Read Produces Constant Observation)**.**

$$\begin{aligned}\forall x \in \mathcal{X}.\ Cpt(rp\_read, x)/s = \{y.\ \exists o \in \{0\ ..\ M-1\}.\ y = (o, \frac{1}{M})\}\ \wedge \\ (\forall y \in Cpt(rp\_read, x)/s.\ y.p = \sum_{d\in\mathcal{J}_{sub}} d.p) \\ where\ \mathcal{J}_{sub} = \{j.\ j \in makeJoint.\ j.o = y.o\}\end{aligned}$$

Proof. With the attack specification, any input that causes an external or internal cache miss requires the RP cache to select one cache line according to replacement policy from each set for replacement or eviction. Therefore, the range of the observation is the whole cache set index under the extreme circumstance, where the attacker can control the whole cache. This is shown in the first line of the above theorem. Meanwhile, the probability of each cache set index equals $\frac{1}{M}$, where M is the length of the whole cache index. When applying this knowledge to the reasoning framework, we can prove that the RP cache read operation satisfies the second

unwinding condition. The observation range narrows under the non-extreme circumstance, while all the observations have a uniform probability. Therefore, there is no information leakage through this process. □

4.5.2 Verifying TLB Designs

Modeling TLB architecture. A memory access to the TLB contains two fields: *va* and *protect*, representing the virtual address issued from CPU and whether the page address is protected, respectively. The TLB structure is similar to the cache specification introduced in Section 4.5.1. Each TLB entry is defined as a **record** $tlb\_line = tlb\_set,\ tlb\_way,\ tlb\_va,\ valid,\ lock$. The field *tlb_va* acts as an identifier of the TLB entry representing the virtual address. The TLB structure also satisfies the same constrains as the cache in Definition 4.9. In a TLB design all the processes use the same index mapping. The state of its state machine at the application layer is defined as **record** $state = TLB,\ Mapping$.

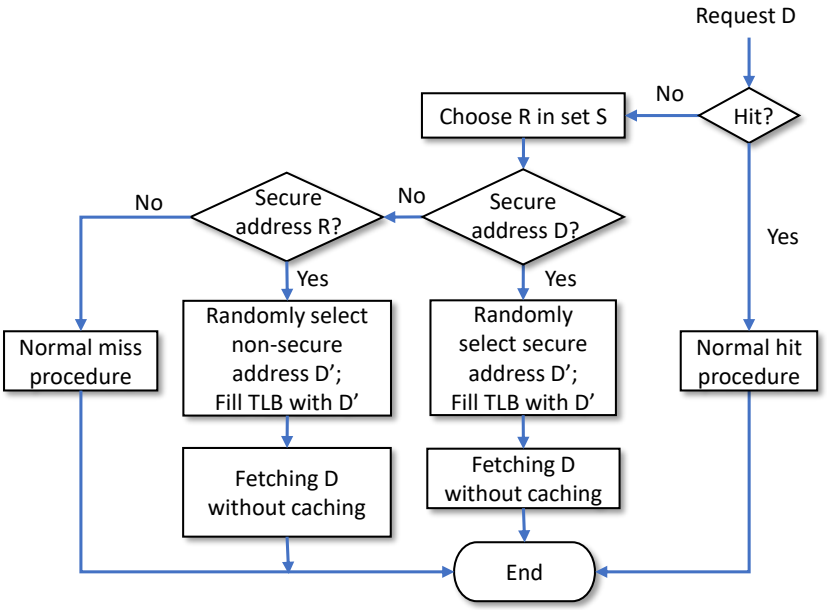


FIGURE 4.4: Workflow of Random Fill TLB

Specifying TLB behaviors. The mechanism of the RF TLB is shown in Figure 4.4. Different from the RP cache, RF TLB randomly selects a page address from secure or non-secure page address regions to fill the TLB when a TLB miss happens. This results in non-deterministic TLB state transitions as well. The specification of RF TLB is defined below.

```
definition rf_search ::
  "state ⇒ memory_request ⇒ memory_request set ⇒ state set"
  where "rf_search s mr mrs ≡
    let Mapping = s->maps; TLB = s->sram;
        redir_index = Mapping -> (mr->index); // find real set index
        redir_tlb = TLB ! redir_index in // locate tlb set
    if probe_line redir_tlb mr
      then {s} // tag hits and tlb hit
    else let // tlb miss
      r_line = replace_policy redir_tlb; // choose line to be replaced
      mm_sec = {t. t ∈ mrs ∧ protect t}; // secure addresses
      mm_nonsec = {t. t ∈ mrs ∧ ¬ protect t} in // non-secure addresses
      if mr->protect then // Column 3
        {t. ∃mm ∈ mm_sec. let // randomly select D'
           mm_index = Mapping -> (mm->index);
           mm_tlb = TLB ! mm_index;
           r'_line = replace_policy mm_tlb; // choose line to be replaced
           nset = insert mm (mm_tlb - {r'_line}) in
         t = s(|sram := update TLB mm_index nset|)}
      else
        if r_line->lock then // Column 2
          {t. ∃mm ∈ mm_nonsec. let
             mm_index = Mapping -> (mm->index);
             mm_tlb = TLB ! mm_index;
             r'_line = replace_policy mm_tlb;
             nset = insert mm (mm_tlb - {r'_line}) in
           t = s(|sram := update TLB mm_index nset|)}
        else let // Column 1
          nset = insert mr (redir_tlb - {r_line}) in
          {s(|sram := update TLB redir_index nset|)}"
```

The correctness lemma is similar with the RP cache read operation.

Lemma 4.9 (Correctness of RF TLB Search)**.**

$$\begin{aligned}&[\![\forall e \in s.TLB\ !\ l_{mr}.\ e.tlb\_va \neq mr.va]\!] \Longrightarrow \\ &[\![\forall s' \in rf\_search\ mr\ s.\ \exists!l.\ (\forall l' \neq l.\ s'.TLB\ !\ l' = s.TLB\ !\ l') \wedge \\ &\qquad\qquad |\ (s'.TLB\ !\ l) \cap (s.TLB\ !\ l)\ | = |\ s.TLB\ !\ l\ | - 1]\!]\end{aligned}$$

Verifying Side-channel Leakage. The search process of the RF TLB preserves the second unwinding conditions, while it breaks the first one, as formalized in the following theorem. Different from cache security proofs above, we care about whether TLB state transitions leak information when any input in secure page address region $\mathcal{X}_{sec}$ resulting in a TLB miss. $\mathcal{L}_{sec}$ represents the TLB set indices that $\mathcal{X}_{sec}$ may map to. $C(l)$ represents the times l appears among all indices.

Theorem 4.10 (RF TLB Search Produces Constant Observation)**.**

$$\forall x \in \mathcal{X}_{sec}.\ Cpt(rf\_search, x)/s = \{y.\ \exists o \in \mathcal{L}_{sec}.\ y = (o, \frac{C(o)}{\mid \mathcal{X}_{sec} \mid})\} \wedge$$
$$(\forall y \in Cpt(rf\_search, x)/s.\ y.p = \sum_{d \in \mathcal{J}_{sub}} d.p)$$
$$where\ \mathcal{J}_{sub} = \{j.\ j \in makeJoint.\ j.o = y.o\}\ and$$
$$\mathcal{L}_{sec} = \{l.\ \exists x \in \mathcal{X}_{sec}.\ l = s.Mapping \rightharpoonup x.va\}$$

Proof. According to the workflow of the RF TLB in Figure 4.4, the attacker has the chance to deduce confidential information only when the memory request is among the secure page addresses and a TLB miss happens. We still use the TLB set index to conveniently represent the observable state transitions. Under this circumstance, the output for each secure page address $x \in \mathcal{X}_{sec}$ is one of the set indices $l \in \mathcal{L}_{sec}$, occupying fixed proportion $\frac{C(l)}{|\mathcal{X}_{sec}|}$ shown in the first line of Theorem 4.10. When applying this conclusion to the reasoning framework, we can prove the RF TLB search operation satisfies the second unwinding condition. Therefore, there is no information leakage. □

4.6 Evaluation

We successfully verify eight state-of-the-art cache designs. We implement all the verification work in Isabelle/HOL. Table 4.1 shows the verification results, as well as the implementation complexity (lines of codes) for each cache. We give the security analysis of eight state-of-the-art cache designs as follows.

Set-Associative (SA) Cache. This conventional cache is well known to be vulnerable to side-channel attacks. Among the cache regions controlled by the

TABLE 4.1: Verification Results of Cache Designs.

Cache Design	NO†	CO♮	Leakage	LOC
Set-Associative Cache	×	×	*yes*	*490+*
Random Fill Cache	×	×	*yes*	*930+*
Partition Locked Cache	√	○	*no*	*380+*
Random Permutation Cache	×	√	*no*	*1490+*
NewCache	×	√	*no*	*1260+*
CEASE Cache	×	×	*yes*	*510+*
CEASER Cache	×	×	*yes*	*580+*
SCATTER Cache	×	×	*yes*	*500+*

† NO denotes No-Observation.
♮ CO denotes Constant-Observation.

attacker, any memory request from the victim can cause an observation with a conditional probability of 1 due to the deterministic execution. Therefore, the cache operation breaks the first condition. Also, a program owns multiple memory address mapped to different cache sets, resulting in different observations. Then, there exists a marginal output that is not equal to the conditional probability of 1, breaking the second condition. Hence, SA cache leaks side-channel information.

Random Fill (RF) Cache [236]. RF cache fills the cache line to be replaced with a random memory line from a neighborhood window of the request memory line. It can result in observations to the attacker, breaking the first condition. Although it randomly picks up a memory line among a stated window, it cannot promise to produce the same range of observations and of equal probability. In extreme cases, RF cache can degrade to a SA cache if the neighborhood window is small, making each memory line mapped to the same cache set. This property cannot imply the second condition. Therefore, there exists information leakage.

Partition Locked (PL) Cache [67]. PL cache with prefetching strategy adds a lock mechanism to the cache line. A replacement policy will work only when the cache line chosen to be replaced is unlocked or belongs to the same process. Therefore, an attacker has no chance to obtain any observations. This cache design is the only one that satisfies the first unwinding condition.

Random Permutation (RP) Cache [67]. Any memory request that causes an external or internal cache miss requires the RP cache to select one cache line from each set for replacement or eviction. Therefore, the range of the observation is

the whole cache set index under the extreme circumstance, where the attacker can control the whole cache. Meanwhile, the probability of each cache set index equals $\frac{1}{M}$, where M is the length of the whole cache index. We can prove that the RP cache operation satisfies the second unwinding condition. The observation range narrows under the non-extreme circumstance, while all the observations have a uniform probability. Therefore, there is no information leakage through this progress.

NewCache [78]. For the protected memory requests, NewCache employs similar strategies as the RP cache. Therefore, it responses to an external or internal cache access by selecting one cache line from each cache set to replace the memory block or to cause an eviction deliberately. Applying similar verification proves the security of NewCache against side-channel attacks.

CEASE Cache [79]. The CEASE cache employs encryption over the physical address and uses the ciphertext to index the cache. However, the memory to cache-set mapping remains constant as long as the encryption key remains the same. Unfortunately, it degenerates to a set-associative cache, which means an adversary can observe a deterministic change for each cache miss. As a result, the CEASE cache may leak confidential information.

CEASER Cache [79]. CEASER cache is an advanced version of CEASE, which adopts dynamic remapping to periodically change the key and remap the lines based on the new key. In the phase when both the current and next keys exist, previous remapping and the victim's access can provide useful observations to the attacker. Therefore, it breaks the first unwinding condition. Although part of the observations is created by remapping of cache lines, the attacker can obtain a deterministic observation during each epoch, and thus there exists such a marginal output that is not equal to its corresponding conditional probability. Therefore, it can not satisfy the second unwinding condition, and information leakage exits.

SCATTER Cache [80]. The SCATTER cache employs the index derivation function that takes the secure domain identifier, the encryption key, and the memory request as inputs to form a nominal cache set (the cache line of each cache way comes from different cache sets). The mapping table is deterministic to a process as long as the encryption key is constant. Another characteristic is that the conditional probability of each output is uniform, while the ranges of these outputs are

inconsistent. Therefore, it can not satisfy the second unwinding condition, leaving a trail of telltale information.

4.7 Conclusion

In this chapter, we propose a novel verification methodology to verify side-channel vulnerabilities resident in the cache designs. We construct an entropy-based non-interference reasoning framework with two unwinding conditions for the evaluation of side-channel threats. We use our methodology to successfully assess and evaluate the security of eight state-of-the-art cache solutions. The verification practice indicates that our verification framework offers strong accuracy and persuasion for the verification of cache side channels.

Although our methodology provides a strong guarantee for the security verification of cache designs from the perspective of formal methods, it still has drawbacks. Our reasoning framework asks a high-level requirement of professionalism of theory proving on users and cannot offer an automated derivation process. Therefore, in future work, we plan to automate the validation process and provide more refined cache models.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In the realm of cache side-channel attacks, a growing body of research is dedicated to their detection, yet significant challenges persist in terms of comprehensiveness and scalability. The intricate interplay between hardware, software, and microarchitectural intricacies makes cache side-channel detection inherently complex, often resulting in an incomplete understanding of mechanisms and vulnerabilities, thereby hindering the effectiveness of detection methods. Additionally, scalability concerns arise as systems become more intricate and expansive, rendering existing detection techniques less comprehensive. These challenges underscore the importance of addressing cache attack detection issues to ensure the efficacy of security measures in the face of evolving threats and intricate computing infrastructures.

Concurrently, the effectiveness of constant-time coding practices, which traditionally mitigated cache side channels, diminishes when confronted with new challenges posed by ciphertext side channels. These attacks exploit deterministic memory encryption, where consistent encryption of the same physical address into identical ciphertext blocks reveals footprints of execution, rendering constant-time coding practices ineffective. Defeating ciphertext side channels necessitates a holistic approach that combines constant-time coding practices with attack mechanisms.

Moreover, the development of new cache designs to counter side channels lacks formal guarantees of effectiveness due to their complexity and potential for emerging

vulnerabilities. The absence of formal assurances underscores the need for continuous evaluation and adaptability to ensure these designs can effectively thwart evolving side-channel threats in real-world contexts.

Motivated by these challenges, this thesis aims to make three key contributions: enhancing cache side-channel detection methods, addressing the emerging threat of ciphertext side channels, and evaluating the security of new cache designs through formal verification. These endeavors aim to bolster the security of modern computing systems in the face of increasingly sophisticated side-channel threats, aligning with the ongoing efforts in this dynamic field.

5.2 Future Work

Quantifying cache side-channel vulnerability leakage represents a promising and crucial research direction for several reasons. As side-channel attacks continue to evolve, the ability to precisely measure and quantify the extent of information leakage through cache channels becomes imperative for assessing the security posture of computing systems. Traditional metrics like timing analysis alone may not adequately capture the nuanced nature of vulnerabilities, and hence, a quantitative approach provides a more nuanced understanding. By developing quantifiable metrics, researchers can assess the severity of cache side-channel vulnerabilities, enabling a more accurate evaluation of the potential risks associated with different attack scenarios. This approach also facilitates the comparison of the effectiveness of various countermeasures and mitigation strategies, allowing for the identification of the most robust solutions. Moreover, a quantitative framework for vulnerability leakage can provide valuable insights into the impact of emerging technologies, such as novel cache designs or encryption methods, on the overall security landscape. Overall, advancing the quantification of cache side-channel vulnerability leakage enhances our ability to systematically analyze, benchmark, and fortify systems against sophisticated side-channel threats.

The detection and analysis of TEE-specific vulnerabilities represent a crucial research direction focusing on identifying and understanding security weaknesses unique to TEE architectures. In this context, researchers aim to develop specialized techniques and methodologies to uncover potential vulnerabilities that adversaries

could exploit within the isolated and secure execution environments of TEEs. This research involves a comprehensive exploration of TEE-specific attack vectors, considering the interplay between hardware, TEE software, and the operating system. By conducting thorough analyses, researchers can uncover nuances in TEE design and implementation that may inadvertently introduce security risks, particularly concerning cache side-channel vulnerabilities. This research direction is essential for enhancing the robustness of TEEs against sophisticated threats, contributing to the ongoing development of secure computing environments where sensitive operations can be conducted with a high degree of confidentiality and integrity.

Cross-layer collaborations for TEE security involve a cohesive approach that unites hardware, TEE software, and the operating system to collectively fortify defenses against cache side-channel attacks. This research direction recognizes the interdependence of these layers in ensuring TEE security and aims to design holistic solutions that address vulnerabilities across the computing stack. Collaborative efforts encompass hardware modifications to create cache architectures resilient to side-channel threats, enhancements to TEE software for adaptive runtime defenses, integration of cache side-channel defenses into the operating system, development of secure communication protocols, implementation of runtime monitoring, and the establishment of standardized benchmarks for evaluation. By fostering interdisciplinary research teams, this approach seeks to comprehensively understand and mitigate cache side-channel vulnerabilities in TEEs, thereby contributing to the overall security and resilience of these critical components in contemporary computing environments.

List of Author's Awards, Patents, and Publications

Publications

- **Ke Jiang**, David Sanán, Yongwang Zhao, Shuanglong Kan, and Yang Liu. 2019. A Formally Verified Buddy Memory Allocation Model. In 24th International Conference on Engineering of Complex Computer Systems (ICECCS' 19), Guangzhou, China, pp. 144-153. https://doi.org/10.1109/ICECCS.2019.00023.

- **Ke Jiang**, Tianwei Zhang, David Sanán, Yongwang Zhao, and Yang Liu. 2022. A Formal Methodology for Verifying Side-channel Vulnerabilities in Cache Architectures. In Proceedings of the 23rd International Conference on Formal Engineering Methods (ICFEM' 22), Madrid, Spain, 190-208. https://doi.org/10.1007/978-3-031-17244-1_12.

- **Ke Jiang**, Yuyan Bao, Shuai Wang, Zhibo Liu, and Tianwei Zhang. 2022. Cache Refinement Type for Side-Channel Detection of Cryptographic Software. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS' 22). Los Angeles, L.A., USA, 1583–1597. https://doi.org/10.1145/3548606.3560672.

- **Ke Jiang**, Sen Deng, Yinshuai Li, Shuai Wang, Tianwei Zhang, and Yinqian Zhang. 2024. CipherGuard: Compiler-aided Mitigation against Ciphertext Side-channel Attacks.

Bibliography

[1] Clémentine Maurice. *Micro-architectural side channels: Studying the attack surface from hardware to browsers*. PhD thesis, Université de Lille, 2023. 2

[2] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996. 2, 111

[3] Daniel J Bernstein. Cache-timing attacks on aes. 2005. 2, 107, 111

[4] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against aes. In *Cryptographic Hardware and Embedded Systems-CHES 2006: 8th International Workshop, Yokohama, Japan, October 10-13, 2006. Proceedings 8*, pages 201–215. Springer, 2006.

[5] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer, 2006. 17, 25, 71, 107, 111, 114

[6] Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. *Cryptology ePrint Archive*, 2002. 111

[7] Colin Percival. Cache missing for fun and profit, 2005. 17, 25, 71, 107, 114

[8] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of des implemented on computers with cache. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 62–76. Springer, 2003. 2, 111

[9] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. Reload+ refresh: Abusing cache replacement policies to perform stealthy cache attacks. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1967–1984, 2020. 2

[10] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross processor cache attacks. In *Proceedings of the 11th ACM on Asia conference on computer and communications security*, pages 353–364, 2016.

[11] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on*

security and privacy, pages 605–622. IEEE, 2015. 17, 23, 25, 71, 107, 114, 127

[12] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 888–904. IEEE, 2019.

[13] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, 2014. 17, 23, 25, 71, 107, 111, 114

[14] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7:99–112, 2017. 2

[15] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016. 2

[16] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Understanding and mitigating covert channels through branch predictors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(1):1–23, 2016.

[17] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. *ACM SIGPLAN Notices*, 53(2):693–707, 2018. 2, 4

[18] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with tlb attacks. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 955–972, 2018. 2, 3, 107, 112

[19] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. Aslr on the line: Practical cache attacks on the mmu. In *NDSS*, volume 17, page 26, 2017. 2

[20] Stephan Van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious management unit: Why stopping cache attacks in software is harder than you think. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 937–954, 2018. 2

[21] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *2015 IEEE Symposium on Security and Privacy*, pages 623–639. IEEE, 2015. 2

[22] 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 *30th USENIX Security Symposium (USENIX Security 21)*, pages 645–662, 2021. 2

[23] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 870–887. IEEE, 2019. 2

[24] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. Absynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures. In *NDSS*, 2020. 2

[25] Dmitry Evtyushkin and Dmitry Ponomarev. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 843–857, 2016. 2

[26] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *2016 IEEE symposium on security and privacy (SP)*, pages 987–1004. IEEE, 2016. 2, 7

[27] Lucian Cojocar, Jeremie Kim, Minesh Patel, Lillian Tsai, Stefan Saroiu, Alec Wolman, and Onur Mutlu. Are we susceptible to rowhammer? an end-to-end methodology for cloud providers. In *2020 IEEE symposium on security and privacy (SP)*, pages 712–728. IEEE, 2020.

[28] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 55–71. IEEE, 2019.

[29] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Trrespass: Exploiting the many sides of target row refresh. *2020 IEEE Symposium on Security and Privacy (SP)*, pages 747–762, 2020.

[30] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another flip in the wall of rowhammer defenses. *2018 IEEE Symposium on Security and Privacy (SP)*, pages 245–261, 2017.

[31] Hasan Hassan, Yahya Can Tugrul, Jeremie S. Kim, Victor van der Veen, Kaveh Razavi, and Onur Mutlu. Uncovering in-dram rowhammer protection mechanisms:a new methodology, custom rowhammer patterns, and implications. *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021.

[32] Jeremie S. Kim, Minesh Patel, Abdullah Giray Yaglikçi, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques. *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 638–651, 2020.

[33] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: an experimental study of dram disturbance errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, page 361–372. IEEE Press, 2014.

[34] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. Rambleed: Reading bits in memory without accessing them. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 695–711, 2020.

[35] Moritz Lipp, Michael Schwarz, Lukas Raab, Lukas Lamster, Misiker Tadesse Aga, Clémentine Maurice, and Daniel Gruss. Nethammer: Inducing rowhammer faults through network requests. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 710–719, 2020.

[36] Onur Mutlu and Jeremie S. Kim. Rowhammer: A retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(8):1555–1571, 2020.

[37] Lois Orosa, Abdullah Giray Yaglikci, Haocong Luo, Ataberk Olgun, Jisung Park, Hasan Hassan, Minesh Patel, Jeremie S. Kim, and Onur Mutlu. A deeper look into rowhammer's sensitivities: Experimental analysis of real dram chips and implications on future attacks and defenses. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, page 1182–1197, New York, NY, USA, 2021. Association for Computing Machinery.

[38] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1–18, Austin, TX, 2016. USENIX Association. ISBN 978-1-931971-32-4.

[39] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. SMASH: Synchronized many-sided rowhammer attacks from JavaScript. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1001–1018. USENIX Association, 2021.

[40] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer attacks over the network and defenses. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 213–226, Boston, MA, 2018. USENIX Association.

[41] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, page 1675–1689, New York, NY, USA, 2016. Association for Computing Machinery. 2

[42] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 249–266, Santa Clara, CA, 2019. USENIX Association. 3

[43] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019. 3

[44] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, Baltimore, MD, 2018. USENIX Association. 3

[45] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. HybCache: Hybrid Side-Channel-Resilient caches for trusted execution environments. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 451–468. USENIX Association, 2020. 3

[46] Haehyun Cho, Penghui Zhang, Donguk Kim, Jinbum Park, Choong-Hoon Lee, Ziming Zhao, Adam Doupé, and Gail-Joon Ahn. Prime+count: Novel cross-world covert channels on arm trustzone. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 441–452, New York, NY, USA, 2018. Association for Computing Machinery. 3

[47] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 557–574, Vancouver, BC, 2017. USENIX Association. ISBN 978-1-931971-40-9. 3

[48] Yangchun Fu, Erick Bauman, Raul Quinonez, and Zhiqiang Lin. Sgx-lapd: Thwarting controlled side channel attacks via enclave verifiable page faults. In *Research in Attacks, Intrusions, and Defenses*, pages 357–380. Springer International Publishing, 2017. 4

[49] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656, 2015. 4

[50] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, page 2421–2434, New York, NY, USA, 2017. Association for Computing Machinery. 4

[51] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient Out-of-Order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, page 991–1008, Baltimore, MD, 2018. USENIX Association. 4

[52] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. ÆPIC leak: Architecturally leaking uninitialized data from the microarchitecture. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3917–3934, Boston, MA, 2022. USENIX Association. ISBN 978-1-939133-31-1. 4

[53] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, page 753–768, New York, NY, USA, 2019. Association for Computing Machinery. 4

[54] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. *IEEE Security & Privacy*, 18(03):28–37, 2020. 4

[55] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, Baltimore, MD, 2018. USENIX Association. 4

[56] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. Cipherleaks: Breaking constant-time cryptography on amd sev via the ciphertext side channel. In *USENIX Security Symposium*, pages 717–732, 2021. 4, 66, 69, 71, 72

[57] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. A systematic look at ciphertext side channels on amd sev-snp. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 337–351. IEEE, 2022. 4, 66, 69, 71, 72, 73

[58] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. *White paper*, page 13, 2016. 4, 65, 69

[59] Dayeol Lee, Dongha Jung, Ian T Fang, Chia-Che Tsai, and Raluca Ada Popa. An off-chip attack on hardware enclaves via the memory bus. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020. 4, 66, 69

[60] Rita Mayer-Sommer. Smartly analyzing the simplicity and the power of simple power analysis on smartcards. In *Workshop on Cryptographic Hardware and Embedded Systems*, 2000. 4

[61] Stefan Mangard. A simple power-analysis (spa) attack on implementations of the aes key expansion. In *Information Security and Cryptology — ICISC 2002*, pages 343–358, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. 4

[62] Louis Goubin and Jacques Patarin. Des and differential power analysis (the "duplication" method). In *Workshop on Cryptographic Hardware and Embedded Systems*, 1999. 4

[63] Christophe Clavier, Jean-Sébastien Coron, and Nora Dabbous. Differential power analysis in the presence of hardware countermeasures. In *Workshop on Cryptographic Hardware and Embedded Systems*, 2000. 4

[64] E. De Mulder, P. Buysschaert, S.B. Ors, P. Delmotte, B. Preneel, G. Vandenbosch, and I. Verbauwhede. Electromagnetic analysis attack on an fpga implementation of an elliptic curve cryptosystem. In *EUROCON 2005 - The International Conference on "Computer as a Tool"*, volume 2, pages 1879–1882, 2005. 5

[65] François-Xavier Standaert and C. Archambeau. Using subspace-based template attacks to compare and combine power and electromagnetic information leakages. In *Workshop on Cryptographic Hardware and Embedded Systems*, 2008. 5

[66] R. Karri, K. Wu, P. Mishra, and Yongkook Kim. Fault-based side-channel cryptanalysis tolerant rijndael symmetric block cipher architecture. In *Proceedings 2001 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 427–435, 2001. 5

[67] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 494–505, 2007. 6, 7, 12, 17, 107, 108, 121, 122, 132

[68] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):1–21, 2012. 6, 17

[69] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. Resource management for isolation enhanced cloud services. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 77–84, 2009. 6

[70] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 194–199. IEEE, 2011. 6

[71] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. Stealthmem: System-level protection against cache-based side channel attacks in the cloud. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 189–204, 2012. 6

[72] CAT Intel. Improving real-time performance by utilizing cache allocation technology. *Intel Corporation, April*, 2015. 6

[73] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE international symposium on high performance computer architecture (HPCA)*, pages 406–418. IEEE, 2016. 6

[74] Ziqiao Zhou, Michael K Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 871–882, 2016. 6, 7

[75] Michael Godfrey and Mohammad Zulkernine. A server-side solution to cache-based side-channel attacks in the cloud. In *2013 IEEE Sixth International Conference on Cloud Computing*, pages 163–170. IEEE, 2013. 6

[76] Read Sprabery, Konstantin Evchenko, Abhilash Raj, Rakesh B Bobba, Sibin Mohan, and Roy Campbell. Scheduling, isolation, and cache allocation: A side-channel defense. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 34–40. IEEE, 2018. 6

[77] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. Scheduler-based defenses against cross-vm side-channels. In *23rd USENIX security symposium (USENIX security 14)*, pages 687–702, 2014. 6

[78] Zhenghong Wang and Ruby B Lee. A novel cache architecture with enhanced performance and security. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 83–93. IEEE, 2008. 7, 12, 17, 108, 133

[79] Moinuddin K Qureshi. Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 775–787. IEEE, 2018. 133

[80] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. Scattercache: Thwarting cache attacks via cache set randomization. In *USENIX Security Symposium*, 2019. 7, 12, 17, 108, 133

[81] Wei Song, Boya Li, Zihan Xue, Zhenzhen Li, Wenhao Wang, and Peng Liu. Randomized last-level caches are still vulnerable to cache side-channel attacks! but we can fix it. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 955–969. IEEE, 2021. 7, 17

[82] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. Systematic analysis of randomization-based protected cache architectures. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 987–1002. IEEE, 2021. 7, 17

[83] Amittai Aviram, Sen Hu, Bryan Ford, and Ramakrishna Gummadi. Determinating timing channels in compute clouds. In *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, pages 103–108, 2010. 7

[84] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in xen. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 41–46, 2011. 7

[85] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 118–129. IEEE, 2012. 7

[86] Peng Li, Debin Gao, and Michael K Reiter. Stopwatch: a cloud architecture for timing channel mitigation. *ACM Transactions on Information and System Security (TISSEC)*, 17(2):1–28, 2014. 7

[87] Yinqian Zhang and Michael K Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 827–838, 2013. 7

[88] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. *ACM SIGPLAN Notices*, 50(4):87–101, 2015. 7

[89] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital {Side-Channels} through obfuscated execution. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 431–446, 2015. 7

[90] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting cache side-channel attacks through dynamic software diversity. In *NDSS*, pages 8–11, 2015. 7

[91] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting sgx enclaves from practical side-channel attacks. In *2018 Usenix Annual Technical Conference (USENIX ATC 18)*, pages 227–240, 2018. 8

[92] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. Racing in hyperspace: Closing hyper-threading side channels on sgx with contrived data races. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 178–194. IEEE, 2018. 8

[93] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with déjá vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 7–18, 2017. 8

[94] Guoxing Chen and Yinqian Zhang. Securing tees with verifiable execution contracts. *IEEE Transactions on Dependable and Secure Computing*, 20(4): 3222–3237, 2022. 8

[95] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. Zerotrace: Oblivious memory primitives from intel sgx. *Cryptology ePrint Archive*, 2017. 8

[96] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. Obfuscuro: A commodity obfuscation engine on intel sgx. In *Network and Distributed System Security Symposium*, 2019. 8, 63

[97] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *NDSS*, 2017. 8

[98] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 217–233, 2017.

[99] Sanchuan Chen, Fangfei Liu, Zeyu Mi, Yinqian Zhang, Ruby B Lee, Haibo Chen, and XiaoFeng Wang. Leveraging hardware transactional memory for cache side-channel defenses. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 601–608, 2018. 8

[100] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. Sgx-shield: Enabling address space layout randomization for sgx programs. In *NDSS*, 2017. 8

[101] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. Speculative taint tracking (stt) a comprehensive protection for speculatively accessed data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 954–968, 2019. 8

[102] Marco Patrignani and Marco Guarnieri. Exorcising spectres with secure compilers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 445–461, 2021. 8

[103] Johan Agat. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 40–53, 2000. 9, 10, 71

[104] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *International Conference on Information Security and Cryptology*, pages 156–168. Springer, 2005. 10

[105] Gilles Barthe, Tamara Rezk, and Martijn Warnier. Preventing timing leaks through transactional branching instructions. *Electronic Notes in Theoretical Computer Science*, 153(2):33–55, 2006. 10

[106] Boris Köpf and Heiko Mantel. Transformational typing and unification for automatically correcting insecure programs. *International Journal of Information Security*, 6(2):107–131, 2007. 10

[107] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1217–1230, 2013.

[108] Heiko Mantel and Artem Starostin. Transforming out timing leaks, more or less. In *European Symposium on Research in Computer Security*, pages 447–467. Springer, 2015. 9, 71

[109] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1267–1279, 2014. 9, 71

[110] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. Verifiable side-channel security of cryptographic implementations: constant-time mee-cbc. In *International Conference on Fast Software Encryption*, pages 163–184. Springer, 2016. 9, 71

[111] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 42–54, 2006. 9, 10

[112] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F Aranha. Sparse representation of implicit flows with applications to side-channel detection. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 110–120, 2016. 9, 71

[113] Mario Dehesa-Azuara, Matthew Fredrikson, Jan Hoffmann, et al. Verifying and synthesizing constant-resource implementations with types. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 710–728. IEEE, 2017. 9, 71

[114] J Bacelar Almeida, Manuel Barbosa, Jorge S Pinto, and Bárbara Vieira. Formal verification of side-channel countermeasures using self-composition. *Science of Computer Programming*, 78(7):796–812, 2013. 9, 71

[115] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 53–70, 2016. 9

[116] Jia Chen, Yu Feng, and Isil Dillig. Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 875–890, 2017. 9

[117] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. Decomposition instead of self-composition for proving the absence of timing channels. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 362–375, 2017. 9

[118] Weikun Yang, Yakir Vizel, Pramod Subramanyan, Aarti Gupta, and Sharad Malik. Lazy self-composition for security verification. In *International Conference on Computer Aided Verification*, pages 136–156. Springer, 2018. 9

[119] Sandrine Blazy, David Pichardie, and Alix Trieu. Verifying constant-time implementations by abstract interpretation. *Journal of Computer Security*, 27(1):137–163, 2019. 9

[120] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1021–1038. IEEE, 2020. 9, 25, 44, 46, 71

[121] Jean Karim Zinzindohoue, Evmorfia-Iro Bartzia, and Karthikeyan Bhargavan. A verified extensible library of elliptic curves. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 296–309. IEEE, 2016. 9, 10, 71

[122] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. Hacl*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1806, 2017. 9, 10, 71

[123] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet,

Pierre-Yves Strub, Markulf Kohlweiss, 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*, pages 256–270, 2016. 10

[124] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K Rustan M Leino, Jacob R Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying high-performance cryptographic assembly code. In *26th USENIX security symposium (USENIX security 17)*, pages 917–934, 2017. 10, 71

[125] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1807–1823, 2017. 10, 71

[126] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. Fact: a dsl for timing-sensitive computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–189, 2019. 10, 71

[127] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *2009 30th IEEE Symposium on Security and Privacy*, pages 45–60. IEEE, 2009. 10

[128] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time". In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 328–343. IEEE, 2018. 10

[129] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. Formal verification of a constant-time preserving c compiler. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–30, 2020. 10

[130] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 15–26, 2018. 10, 11, 71

[131] Luigi Soares and Fernando Magno Quintão Pereira. Memory-safe elimination of side channels. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 200–210. IEEE, 2021. 10, 11, 71

[132] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. 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, pages 715–733, 2021. 11, 71, 80

[133] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. Cached: Identifying cache-based timing channels in production software. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 235–252, 2017. 11, 14, 18, 19, 24, 25, 26, 34, 40, 42, 43, 45, 47, 80

[134] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. Identifying cache-based side channels through secret-augmented abstract interpretation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 657–674, 2019. 11, 14, 18, 19, 24, 25, 26, 34, 40, 43, 45, 47, 104

[135] Jan Wichelmann, Anna Pätschke, Luca Wilke, and Thomas Eisenbarth. Cipherfix: Mitigating ciphertext side-channel attacks in software. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6789–6806, 2023. 12, 14, 66, 72

[136] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. Cache timing side-channel vulnerability checking with computation tree logic. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–8, 2018. 12, 108

[137] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. Analysis of secure caches using a three-step model for timing-based attacks. *Journal of Hardware and Systems Security*, 3(4):397–425, 2019.

[138] Limin Wang, Ziyuan Zhu, Zhanpeng Wang, and Dan Meng. Analyzing the security of the cache side channel defences with attack graphs. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 50–55. IEEE, 2020. 108

[139] Tianwei Zhang and Ruby B Lee. New models of cache architectures characterizing information leakage from cache side channels. In *Proceedings of the 30th annual computer security applications conference*, pages 96–105, 2014. 108

[140] Zecheng He and Ruby B Lee. How secure is your cache against side-channel attacks? In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 341–353, 2017. 12, 108

[141] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 305–316, 2012. 17, 71

[142] Yuval Yarom and Naomi Benger. Recovering openssl ecdsa nonces using the flush+ reload cache side-channel attack. *IACR Cryptol. ePrint Arch.*, 2014: 140, 2014. 17, 51, 107

[143] Keegan Ryan. Return of the hidden number problem. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 146–168, 2019. 51

[144] Diego F Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. Ladderleak: Breaking ecdsa with less than one bit of nonce leakage. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 225–242, 2020. 17, 71, 107

[145] Goran Doychev, Dominik Feld, Boris Kopf, Laurent Mauborgne, and Jan Reineke. Cacheaudit: A tool for the static analysis of cache side channels. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 431–446, 2013. 18, 19, 25

[146] Goran Doychev and Boris Köpf. Rigorous analysis of software countermeasures against cache attacks. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 406–421, 2017. 18, 19, 24

[147] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. Data–differential address trace analysis: Finding address-based side-channels in binaries. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 603–620, 2018. 18, 19, 25, 43, 45, 46, 80

[148] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. Casym: Cache aware symbolic execution for side channel detection and mitigation. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 505–521. IEEE, 2019. 18, 19, 24, 25, 26, 40, 47, 104

[149] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. 18

[150] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, 1977. 18

[151] Qinkun Bao, Zihao Wang, Xiaoting Li, James R Larus, and Dinghao Wu. Abacus: Precise side-channel analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 797–809. IEEE, 2021. 19, 24, 25, 26, 34, 40, 43, 47

[152] Luca Cardelli. Type systems. *ACM Computing Surveys (CSUR)*, 28(1): 263–264, 1996. 21

[153] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002. 21

[154] Danfeng Zhang, Aslan Askarov, and Andrew C Myers. Language-based control and mitigation of timing channels. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 99–110, 2012. 21

[155] Ranjit Jhala, Niki Vazou, et al. Refinement types: A tutorial. *Foundations and Trends in Programming Languages*, 6(3–4):159–317, 2021. 21

[156] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D Gordon. Modular verification of security protocol code by typing. *ACM Sigplan Notices*, 45(1): 445–456, 2010. 22

[157] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D Gordon, and Sergio Maffeis. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(2):1–45, 2011.

[158] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing tls with verified cryptographic security. In *2013 IEEE Symposium on Security and Privacy*, pages 445–459. IEEE, 2013.

[159] Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella-Béguelin. Probabilistic relational verification for cryptographic implementations. *ACM SIGPLAN Notices*, 49(1):193–205, 2014. 22

[160] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. *ACM Computing Surveys (CSUR)*, 54(6):1–37, 2021. 23, 107

[161] Werner Schindler. Exclusive exponent blinding may not suffice to prevent timing attacks on rsa. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 229–247. Springer, 2015. 24

[162] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. Microwalk: A framework for finding side channels in binaries. In *ACSAC*, 2018. 24, 25

[163] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016. 25, 111, 114

[164] Sudipta Chattopadhyay, Moritz Beck, Ahmed Rezine, and Andreas Zeller. Quantifying information leakage in cache attacks via symbolic execution. *TECS*, 2019. 25

[165] Chungha Sung, Brandon Paulsen, and Chao Wang. Canal: a cache timing analysis framework via llvm transformation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 904–907, 2018. 25

[166] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE J. Sel. Areas Commun.*, 21(1):5–19, 2003. doi: 10.1109/JSAC.2002.806121. 27

[167] Hassan Eldib, Chao Wang, Mostafa Taha, and Patrick Schaumont. Qms: Evaluating the side-channel resistance of masked software from source code. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2014. 29

[168] Hassan Eldib, Chao Wang, and Patrick Schaumont. Smt-based verification of software countermeasures against side-channel attacks. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 62–77. Springer, 2014.

[169] Hassan Eldib, Chao Wang, and Patrick Schaumont. Formal verification of software countermeasures against side-channel attacks. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):1–24, 2014. 29

[170] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005. 30, 42, 101

[171] Laurent Simon, David Chisnall, and Ross Anderson. What you get is what you c: Controlling side effects in mainstream c compilers. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 1–15. IEEE, 2018. 31

[172] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap: A binary analysis platform. In *International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011. 42

[173] Samuel Weiser, David Schrammel, Lukas Bodner, and Raphael Spreitzer. Big numbers-big troubles: Systematically analyzing nonce leakage in (ec)dsa implementations. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1767–1784, 2020. 45, 46, 52

[174] OpenSSL-972c87d. Make bn_num_bits_word constant-time., 2018. URL `https://github.com/openssl/openssl/commit/972c87dfc7e765bd28a4964519c362f0d3a58ca4`. 51

[175] Dan Boneh and Ramarathnam Venkatesan. Hardness of computing the most significant bits of secret keys in diffie-hellman and related schemes. In *Annual International Cryptology Conference*, pages 129–142. Springer, 1996. 51

[176] Dan Boneh and Ramarathnam Venkatesan. Rounding in lattices and its cryptographic applications. In *SODA*, volume 1997, pages 675–681. Citeseer, 1997. 51

[177] Naomi Benger, Joop van de Pol, Nigel P Smart, and Yuval Yarom. ?ooh aah... just a little bit?: a small amount of side channel can go a long way. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 75–92. Springer, 2014. 51

[178] Phong Q Nguyen and Igor E Shparlinski. The insecurity of the digital signature algorithm with partially known nonces. *Journal of Cryptology*, 15(3), 2002.

[179] Phong Q Nguyen and Igor E Shparlinski. The insecurity of the elliptic curve digital signature algorithm with partially known nonces. *Designs, codes and cryptography*, 30(2):201–217, 2003. 51

[180] OpenSSL-4b7a4ba. Fix for cve-2014-0076., 2014. URL `https://github.com/openssl/openssl/commit/4b7a4ba29cafa432fc4266fe6e59e60bc1c96332`. 51

[181] OpenSSL-2198be3. Fix for cve-2014-0076., 2014. URL `https://github.com/openssl/openssl/commit/2198be3483259de374f91e57d247d0fc667aef29`. 51

[182] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. Single trace attack against rsa key generation in intel sgx ssl. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 575–586, 2018. 54

[183] Libressl-2cd28f9. Use a blinding value when generating a dsa signature., 2018. URL `https://github.com/libressl-portable/openbsd/commit/2cd28f9?diff=unified`. 56

[184] Libressl-1f6b35b. Remove the blinding later to avoid leaking information on the length., 2019. URL `https://github.com/libressl-portable/openbsd/commit/1f6b35b`. 57

[185] John Toman, Ren Siqi, Kohei Suenaga, Atsushi Igarashi, and Naoki Kobayashi. Consort: Context- and flow-sensitive ownership refinement types for imperative programs. In *ESOP*, volume 12075 of *Lecture Notes in Computer Science*, pages 684–714. Springer, 2020. doi: 10.1007/978-3-030-44914-8_25. 60

[186] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. Refinement types for haskell. In *ICFP*, pages 269–282. ACM, 2014. doi: 10.1145/2628136.2628161.

[187] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Refinement types for typescript. In *PLDI*, pages 310–325. ACM, 2016. doi: 10.1145/2908080.2908110. 60

[188] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 142–157. IEEE, 2019. 63

[189] Wubing Wang, Yinqian Zhang, and Zhiqiang Lin. Time and order: Towards automatically identifying side-channel vulnerabilities in enclave binaries. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 443–457, 2019. 63

[190] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual international cryptology conference*, pages 388–397. Springer, 1999. 63

[191] Amir Moradi, Alessandro Barenghi, Timo Kasper, and Christof Paar. On the vulnerability of fpga bitstream encryption against power analysis attacks: Extracting keys from xilinx virtex-ii fpgas. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 111–124, 2011. 63

[192] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 457–485. Springer, 2015. 63

[193] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 116–129, 2016.

[194] Inès Ben El Ouahma, Quentin L Meunier, Karine Heydemann, and Emmanuelle Encrenaz. Symbolic approach for side-channel resistance analysis of masked assembly codes. In *Security Proofs for Embedded Systems*, 2017. 63

[195] Jun Zhang, Pengfei Gao, Fu Song, and Chao Wang. Sc infer: refinement-based verification of software countermeasures against side-channel attacks. In *International Conference on Computer Aided Verification*, pages 157–177. Springer, 2018. 63

[196] Pengfei Gao, Jun Zhang, Fu Song, and Chao Wang. Verifying and quantifying side-channel resistance of masked software implementations. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(3):1–32, 2019. 63

[197] Pengfei Gao, Hongyi Xie, Jun Zhang, Fu Song, and Taolue Chen. Quantitative verification of masked arithmetic programs against side-channel attacks. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 155–173. Springer, 2019. 63

[198] Gao Pengfei, Xie Hongyi, Pu Sun, Jun Zhang, Fu Song, and Taolue Chen. Formal verification of masking countermeasures for arithmetic programs. *IEEE Transactions on Software Engineering*, 2020. 63

[199] David Kaplan. Protecting vm register state with sev-es. *White paper*, page 13, 2017. 65

[200] David Kaplan, Jeremy Powell, and Tom Woller. Amd sev-snp: Strengthening vm isolationwith integrity protection and more. *White paper, Advanced Micro Devices Inc*, 2020. 65

[201] Intel. Product brief, 3rd gen intel xeon scaleable processor for iot. `https://www.intel.com/content/www/us/en/products/docs/processors/embedded/3rd-gen-xeon-scalable-iot-product-brief.html`, 2021. 65

[202] Simon Johnson, Raghunandan Makaram, Amy Santoni, and Vinnie Scarlata. Supporting intel sgx on multi-socket platforms. *Intel Corp*, 2021. 65

[203] Intel. Intel Trust Domain Extensions (Intel TDX). `https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html`, 2020. 65

[204] ARM. Arm Confidential Compute Architecture software stack. `https://developer.arm.com/documentation/den0127/latest`, 2021. 65

[205] Robert Buhren, Shay Gueron, Jan Nordholz, Jean-Pierre Seifert, and Julian Vetter. Fault attacks on encrypted general purpose compute platforms. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 197–204, 2017. 66

[206] Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. Secure encrypted virtualization is unsecure. *arXiv preprint arXiv:1712.05090*, 2017.

[207] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. Sevurity: No security without integrity: Breaking integrity-free memory encryption with minimal assumptions. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1483–1496. IEEE, 2020. 66

[208] Felicitas Hetzelt and Robert Buhren. Security analysis of encrypted virtual machines. *ACM SIGPLAN Notices*, 52(7):129–142, 2017. 66

[209] Mathias Morbitzer, Manuel Huber, and Julian Horsch. Extracting secrets from encrypted virtual machines. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, pages 221–230, 2019.

[210] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. Severed: Subverting amd's virtual machine encryption. In *Proceedings of the 11th European Workshop on Systems Security*, pages 1–6, 2018. 66

[211] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. Exploiting unprotected i/o operations in amd's secure encrypted virtualization. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1257–1272, 2019. 66

[212] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. Crossline: Breaking" security-by-crash" based memory isolation in amd sev. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2937–2950, 2021. 66

[213] Sen Deng, Mengyuan Li, Yining Tang, Shuai Wang, Shoumeng Yan, and Yinqian Zhang. Cipherh: Automated detection of ciphertext side-channel vulnerabilities in cryptographic implementations. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6843–6860, 2023. 66, 71, 80, 81

[214] LLVM. DataFlowSanitizer. `https://clang.llvm.org/docs/DataFlowSanitizer.html`, 2020. 66

[215] Jan Wichelmann, Anja Rabich, Anna Pätschke, and Thomas Eisenbarth. Obelix: Mitigating side-channels through dynamic obfuscation. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 189–189. IEEE Computer Society, 2024. 66

[216] AMD. Technical guidance for mitigating effects of ciphertext visibility under amd sev. `https://www.amd.com/system/files/documents/221404394-a_security_wp_final.pdf`, 2022. 71, 72

[217] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. Sev-step: A single-stepping framework for amd-sev. *arXiv preprint arXiv:2307.14757*, 2023. 72

[218] Sebastiano Vigna. Further scramblings of marsaglia's xorshift generators. *Journal of Computational and Applied Mathematics*, 315:175–181, 2017. 77

[219] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 380–392, 2016. 101

[220] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, et al. Fallout: Leaking data on meltdown-resistant cpus. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 769–784, 2019. 107, 112

[221] Joseph A Goguen and José Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11. IEEE, 1982. 107, 118

[222] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 305–316, 2012. 107

[223] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, gauss, and reload–a cache attack on the bliss lattice-based signature scheme. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 323–345. Springer, 2016. 107

[224] Daimeng Wang, Ajaya Neupane, Zhiyun Qian, Nael B Abu-Ghazaleh, Srikanth V Krishnamurthy, Edward JM Colbert, and Paul Yu. Unveiling your keystrokes: A cache-based side-channel attack on graphics libraries. In *NDSS*, 2019. 107

[225] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 639–656, 2019. 107

[226] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205. IEEE, 2013. 107

[227] Xun Li, Vineeth Kashyap, Jason K Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T Chong. Sapper: A language for hardware-level security policy enforcement. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 97–112, 2014. 109

[228] Danfeng Zhang, Yao Wang, G Edward Suh, and Andrew C Myers. A hardware design language for timing-sensitive information-flow security. *Acm Sigplan Notices*, 50(4):503–516, 2015.

[229] Shuwen Deng, Doğuhan Gümüşoğlu, Wenjie Xiong, Sercan Sari, Y Serhan Gener, Corine Lu, Onur Demir, and Jakub Szefer. Secchisel framework for security verification of secure processor architectures. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–8, 2019. 109

[230] Thomas M Cover. *Elements of information theory*. John Wiley & Sons, 1999. 109, 112

[231] Konstantinos Chatzikokolakis, Tom Chothia, and Apratim Guha. Statistical measurement of information leakage. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 390–404. Springer, 2010. 109, 112

[232] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer Science & Business Media, 2002. 109, 113

[233] Pepe Vila, Boris Köpf, and José F Morales. Theory and practice of finding eviction sets. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 39–54. IEEE, 2019. 111

[234] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002. 116

[235] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. Secure tlbs. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 346–359. IEEE, 2019. 122

[236] Fangfei Liu and Ruby B Lee. Random fill cache architecture. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 203–215. IEEE, 2014. 132