The Dissertation Committee for Ashay Rane
certifies that this is the approved version of the following dissertation:

**Broad-Based Side-Channel Defenses
for Modern Microprocessors**

Committee:

Calvin Lin, Supervisor

Mohit Tiwari, Co-Supervisor

Işil Dillig

Emmett Witchel

David Evans

Broad-Based Side-Channel Defenses for Modern Microprocessors

by

Ashay Rane

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2019

Acknowledgments

Had it not been for the heroic efforts of Professor Dan Stanzione, Professor Jim Browne, and Professor Keshav Pingali in enabling me to come to UT Austin, I would never have been able to enroll as a PhD student and, therefore, work on this dissertation. Professor Browne and my PhD supervisor, Professor Calvin Lin, were both instrumental in helping me become acclimatized to the research environment. By virtue of the enthusiastic guidance of Professor Browne, Professor Lin, and my co-supervisor, Professor Mohit Tiwari, I have learned a great deal about various aspects of research, including skills such as picking the right research problems, being thorough with my work, and writing and presenting well. My committee members, Professor Işil Dillig, Professor Emmett Witchel, and Professor David Evans, have also been crucial in improving this dissertation.

Various colleagues have tirelessly provided feedback on my research, and I want to thank them for their contribution. The Spark lab members (Austin, Aydin, Casen, Mikhail, Pranav, Prateek, Riley, Rohith, Shijia, and Willy) and members from Professor Lin's research group (Akanksha, Anjana, Chirag, Curtis, Hao, Jia, Kai, Matthew, Molly, Pawan, and Zhan) have offered both valuable insights and technical assistance at various times throughout my term as a graduate student. Several fellow PhD students including Arthur, John, Rasheed, Roshan, Shilpi, Tyler, and Vance have provided both intellectual stimulation as well as social support through various times. I also thank Greg, Joshua, Raymond, and Varun for opting to work with me towards their Honors thesis.

Finally, and also most importantly, the contributions of my family and friends are invaluable. Without their support and their frequent reminders to stay sane and healthy, none of this work would have seen the light of the day.

Broad-Based Side-Channel Defenses for Modern Microprocessors

Publication No. ____________________

Ashay Rane, Ph.D.
The University of Texas at Austin, 2019

Supervisors: Calvin Lin
Mohit Tiwari

Private or confidential information is used in a wide variety of applications, not just including implementations of cryptographic algorithms but also including machine-learning libraries, databases, and parsers. However, even after using techniques such as encryption, authentication, and isolation, it is difficult to maintain the privacy or confidentiality of such information due to so-called side channels, with which attackers can infer sensitive information by monitoring program execution. Various side channels exist such as execution time, power consumption, exceptions, or micro-architectural components such as caches and branch predictors, and such side channels have been used to steal intellectual property, financial information, and sensitive document contents. Although numerous solutions exist for closing side channels, they are point solutions, since each solution closes an isolated set of side channels.

In this dissertation, we present three compiler-based solutions—Raccoon, Escort, and Vantage—for closing digital side channels (such as the cache, address trace, and branch predictor side channels) that carry information over discrete bits, and for mitigating the a *non-digital* side channel, specifically, the power side channel. Additionally, our compilers are customizable, since they permit the defense to be tailored to the threat model, to the program, and to the microarchitecture.

More broadly, our solutions augment the compiler with information about the lower layers of the computing stack, so that the compiler is aware of potential side channels and so that the compiler can rewrite programs to avoid leaking information through those side channels. In doing so, our solutions define new abstractions that enable the compiler to reason about the program's impact on timing, power consumption, and other similar side channels. Through such abstractions, our compilers detect and prevent a broad set of digital and non-digital leakage on modern microarchitectures.

Table of Contents

List of Tables

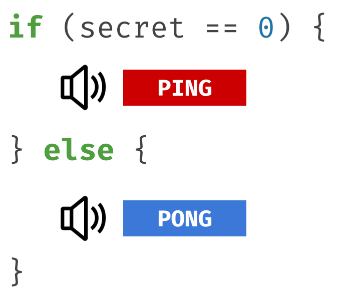
List of Figures

Chapter 1

Introduction

Private or confidential information is used by a wide variety of applications including machine-learning libraries, which analyze personal browsing histories to recommend products [60], social networking tools, which analyze graphs that represent connections between persons [82], and cloud services (such as online maps), which provide directions based on the location of their users [74]. Unfortunately, it is difficult to keep all of this sensitive information private, because although there exist many techniques such as encryption, access control, and isolation to prevent the accidental disclosure or leakage of sensitive information, these techniques are not sufficient. In particular, attackers can steal sensitive information through so-called side channels, which are means by which an attacker can monitor program execution to reveal sensitive information in the program.

To understand side channels, consider the program shown in Figure 1.1, where the code emits a sound *ping* if `secret` is zero and it emits a sound *pong* otherwise. Given such a program, an attacker who hears the sound *ping* knows that `secret` must be equal to zero. Similarly, if the attacker hears the sound *pong*, then she knows that the secret must be not equal to zero. Thus the



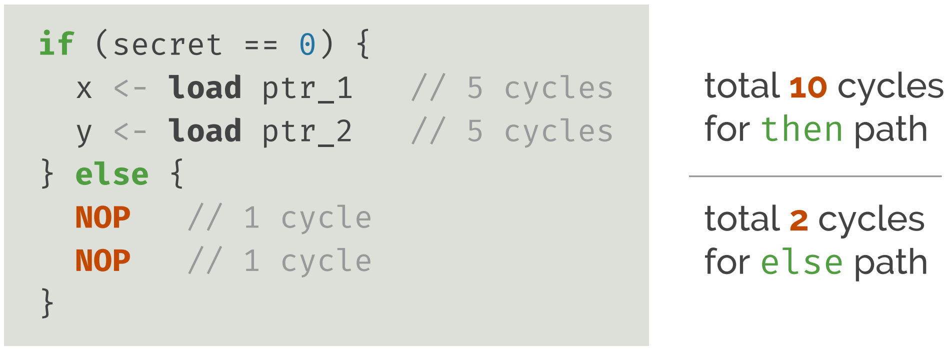
```
if (secret == 0) {
    PING
} else {
    PONG
}
```

Figure 1.1: Code fragment to demonstrate side channels.

presence or absence of the *ping* or *pong* sounds inadvertently leaks the value of the `secret` variable, and hence the audible sounds become side channels. Although programs do not typically emit audible sounds, they usually exhibit many kinds of *variations* that produce the same effect of leaking sensitive information.

More formally, a side channel is any variation in a system's behavior, such that the variation is (1) is visible to an attacker and (2) dependent on sensitive information. Thus, the variations reveal sensitive values, thus becoming side channels of sensitive information.

Numerous side channels exist, including execution time, memory access, instruction count, cache behavior, DRAM address trace, power consumption, and electromagnetic radiation. Side-channel attacks are significant because attackers have used side channels to break AES [80] and RSA [83] encryption schemes, to break the Diffie-Hellman key exchange [50], to fingerprint software libraries [123], and to reverse-engineer commercial processors [53].



```
if (secret == 0) {
  x <- load ptr_1   // 5 cycles
  y <- load ptr_2   // 5 cycles
} else {
  NOP   // 1 cycle
  NOP   // 1 cycle
}
```

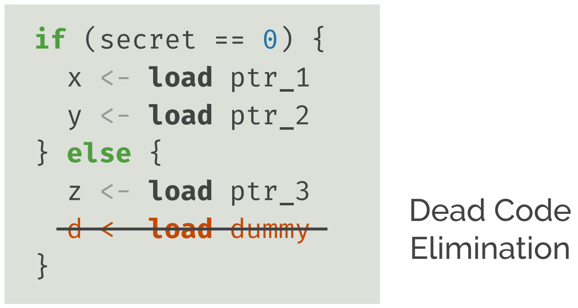
Figure 1.2: In a point solution that adds dummy `NOP` instructions to ensure that the instruction count does not leak information, the execution time can still leak information. A different point solution that adds `NOP` instructions to balance the execution time will need to add many more `NOP` instructions in the `else` path, thus breaking the security guarantee of the former defense.

1.1 Inadequacy of Prior Solutions

Numerous solutions exist for closing side channels [33, 34, 49, 61, 62, 65, 66, 73, 93, 96, 97, 99, 105, 108, 111, 112, 117–119], and we compare our solutions against them in Section 2.4 (Related Work). However, the common characteristic across these prior solutions is that they are *point* solutions, since they each close an isolated set of side channels. Consequently, there exist separate solutions for closing the execution time side channel [66, 73, 94], separate solutions for the branch predictor side channel [25, 56, 73], for the DRAM address trace side channel [4, 8, 61, 62], and so on.

Point solutions suffer from two major drawbacks, which we describe below.

Lack of Composability. Point solutions do not always compose with each



```
if (secret == 0) {
  x <- load ptr_1
  y <- load ptr_2
} else {
  z <- load ptr_3
  ~~d <- load dummy~~
}
```

Figure 1.3: A point solution that adds a dummy load instruction (to hide variations in the number of runtime load instructions) is susceptible to compiler optimizations (like Dead Code Elimination) which will break the solution's security guarantees.

other. More precisely, the use of one point solution can break the security guarantees of another point solution. Moreover, arguing about security can be difficult when multiple point solutions are combined. For instance, consider a cache side-channel defense that randomly evicts cache lines so as to confuse an adversary monitoring the cache usage [63, 111]. Such a defense cannot be easily and securely composed with a DRAM address trace side-channel defense which expects a predictable stream of memory requests from the CPU [61]. As another example, Figure 1.2 shows how a defense for the instruction count side channel breaks the security guarantee of a defense for the execution time side channel.

Disabled Optimizations. Since optimizations can break the security guarantees of point solutions, the use of point solutions can force us to significantly re-engineer compilers and microarchitectures. For instance, consider a solution that adds dummy `load` instructions to hide variations in the load instruction count [61], as shown in Figure 1.3. Unfortunately, common compiler optimizations such as Dead Code Elimination, Loop Invariant Code Motion, and Instruction Combining interfere with the desired security guarantees so such a point solution is forced to disable many compiler optimizations. The same point solution is also vulnerable to microarchitectural optimizations such as caches, since a *dummy* `load` instruction could consume a different execution time than a *real* `load` instruction. Several other microarchitectural optimizations such as prefetchers, branch prediction, and variable-latency instructions also break the security guarantee of this point solution. Consequently, the use of point solutions can require hardware vendors to design new significantly slower hardware to support these side-channel defenses.

Ultimately, point solutions represent a patchwork approach, whose security is difficult to assert in the presence of other point solutions and in the presence of common optimizations. Consequently, point solutions are extremely limited in not just the number of side channels that they can close, but also in the kinds of programs and the kinds of microarchitectures that they can protect. Perhaps unsurprisingly, a number of point solutions have been evaluated using only small cryptographic kernels [56, 63, 100, 114, 121], such as implementations of Advanced Encryption Standard (AES), or on programs using only simple control flow [7, 61, 62, 79], thus ruling out loops and floating-point arithmetic operations.

1.2 Key Contributions of Our Research

In this dissertation, we present three compiler-based solutions—Raccoon [87], Escort [88], and VANTAGE [89]—for closing a broad class of side channels. Although our solutions cannot close speculation-based side channels, our Raccoon and Escort compilers close digital side channels, which are side channels that carry information over discrete bits (*e.g.* cache, address trace, branch predictor, instruction count, etc.), and our VANTAGE compiler augments existing solutions for closing *non-digital* side channels, specifically, the power side channel, to protect a more diverse set of applications compared to prior work. Our solutions are compatible with microarchitectural optimizations such as caching, prefetching, branch prediction, and out-of-order execution, and our solutions leverage the flexibility of software (*i.e.* the compiler) for tailoring the defense according to the program, the microarchitecture, and the threat model.

At a high level, our solutions assume that the attacker cannot bypass the memory and register-level isolation that is typically expected to be enforced by the underlying hardware (either through the Memory Management Unit or through enclaves such as Intel Software Guard Extensions [71]), and our solutions assume that the memory contents are encrypted. Since our solutions are built in the compiler, they assume that the source code is available. Consequently, our solutions cannot transform code fragments that contain system calls or library calls, since their definition is outside the purview of the compiler. We describe the assumptions made by our solutions in greater detail in the following chapters.

1.3 Key Insight Used in Our Research

In our solutions, we use our insight that a broad class of side channels are caused due to variations in the source-level behavior of the program. For instance, the branch predictor and branch target buffer side channels are caused by differences in the outcome of conditional branches in the program. Similarly, the cache, TLB, and DRAM address trace side channels are caused due to differences in pointer dereferences in the program.

Since such source-level variations can be summarized in terms of control flows and data flows, our solutions eliminate a broad class of side channels by making the program's control flows and data flows independent of the program's sensitive information. Conceptually, our solutions execute all paths and access all memory locations in the program. For instance, for the code fragment shown in Figure 1.1, our solutions force the evaluation of both the `then` path as well as the `else` path of the branch, so that the adversary's view in terms of different side channels in always the same regardless of the value of the `secret` variable. Our solutions prevent an explosion in the number of executed paths and accessed memory locations by (1) identifying the instructions that are common to various paths and executing them only once and (2) by limiting our focus to only those parts of the program that potentially leak sensitive information.

Of course, side channels can also exist in the execution of individual assembly instructions, so a program whose control flows and data flows are independent of sensitive information could still leak information. To prevent such leakage, our solutions rewrite the operation of specific assembly instruction using other (safe) assembly instructions. We generate such rewritten operations either manually or by leveraging results from an existing superoptimizer.

1.4 Organization of This Dissertation

The rest of this dissertation is organized as follows. In Chapter 2, we describe an overview of our approach, our threat model, the limitations of our approach, and related work. In Chapter 3, we explain our solution for closing side channels caused due to differences in the source-level behavior of the application. Chapter 4 describes our approach for closing side channels arising from the use of variable-latency floating-point operations, along with an optimized technique for preventing information leakage due to control flows. In Chapter 5, we describe our technique for enabling the compiler to mitigate analog side channels, specifically the power side channel, in diverse applications. Finally, Chapter 6 concludes before highlighting potential future work.

Chapter 2

Overview of This Dissertation

This chapter describes the key ideas used in this dissertation, the threat model, limitations of our approach, and related work.

2.1 Key Ideas

Broadly, our solutions build abstractions from compiler-level instructions to side-channel observations, thus enabling the compiler to reason about potential information leakage in the program through side channels. We build such abstractions by analyzing the underlying layers of the computing stack (*i.e.* the ISA, the microarchitecture, and the physical hardware). Specifically, we apply information flow analysis to determine how sensitive information in registers and memory locations can leak through metrics such as timing, exceptions, power, etc. Using such abstractions from compiler-level instructions to side-channel observations, our solutions both identify potential leakage and transform programs to prevent the adversary from inferring sensitive information despite the existence of side channels.

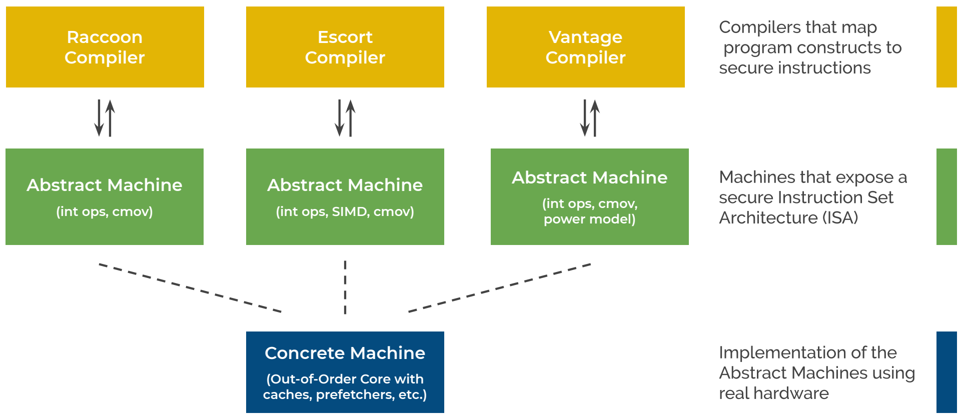

Raccoon Compiler
Escort Compiler
Vantage Compiler
Compilers that map program constructs to secure instructions
Abstract Machine (int ops, cmov)
Abstract Machine (int ops, SIMD, cmov)
Abstract Machine (int ops, cmov, power model)
Machines that expose a secure Instruction Set Architecture (ISA)
Concrete Machine (Out-of-Order Core with caches, prefetchers, etc.)
Implementation of the Abstract Machines using real hardware

Figure 2.1: Our compilers map various programming constructs to instructions of an abstract machine. We derive the abstract machine by analyzing various ISAs and microarchitectures.

Abstract Models for Computational Non-Interferences. In this dissertation, we construct and leverage three main abstract models for implementing computational non-interference. First, our Raccoon compiler uses the conditional move instruction in x86 ISAs to protect integer applications from digital side channels. Effectively, the Raccoon compiler maps various program constructs onto an abstract machine that supports a secure register-level conditional move instruction. Consequently, Raccoon assumes that the concrete machine's microarchitectural implementation of the register-level conditional move instruction does not leak information.

Our Escort compiler, which closes digital side channels arising in floating-point applications, uses an extended abstract machine. Specifically, in addition to assuming a secure implementation of the conditional move instruction, it assumes that SIMD operations that use subnormal values execute in parallel, so that dummy subnormal operations in spare lanes force floating-point operations to consume a fixed, worst-case latency. Although our concrete machine (an Intel Sandy Bridge processor) exhibits a $\sim$1-cycle standard deviation ([88], Table 5) in execution latency despite the use of SIMD operations, this variation is substantially lower than the $\sim$58-cycle standard deviation among non-secure operations. On the other hand, if the concrete machine is built to match the abstract machine's specifications (*i.e.* the SIMD operations that use subnormal values indeed execute in parallel), then the Escort compiler's guarantees are complete.

Finally, our Vantage compiler leverages an abstract machine in which fine-grain power variations are eliminated using existing hardware techniques, thus enabling the compiler to eliminate coarse-grain power variations.

Formal Notation. We illustrate our solutions' generalized approaches for code transformation in the context of a computation function f that we wish to protect from side-channel leakage. Specifically, $f : I \rightarrow O$, *i.e.* the function f accepts inputs I and produces outputs O. By applying the previously-derived abstraction, α (from the compiler-level instructions to the side-channel observations), to each instruction inside the function f, we arrive at a leakage function l which maps the program's inputs to side-channel observations. More precisely, $l : I \rightarrow S$, *i.e.* the function l accepts inputs I and produces side-channel observations S. The adversary's goal is to infer the input based on the side-channel observations, whereas our solutions' goal is to rewrite the

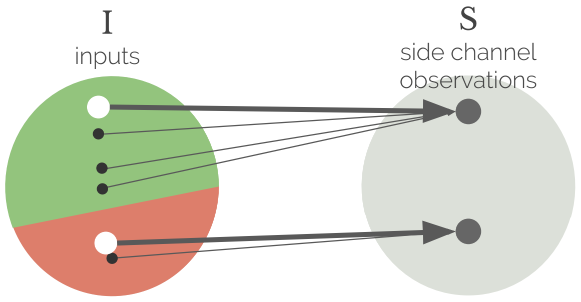


Figure 2.2: If the input set can be partitioned based on the distinct side-channel observations, then we can evaluate the user's computation on only one input per partition instead of evaluating it once for each input, thus reducing the performance impact.

function f so that the adversary's inference is impossible.

We now describe the three broad approaches used by our solutions.

Iterating Over All Inputs. A naive solution to hide sensitive input despite the occurrence of side channels is to evaluate the computation function f on all inputs I, regardless of the desired sensitive input. By doing so, the adversary's view in terms of side channels is always identical regardless of the actual input. Of course, if the set of inputs I is large, then this approach is too expensive.

Iterating Over Partitioned Inputs. Depending on the abstraction α, it may be possible to partition the set of inputs I such that inputs that cause identical side-channel observations are grouped into the same partition. Figure 2.2 illustrates this condition, where inputs are partitioned into two sets based on the distinct side-channel observations. In such

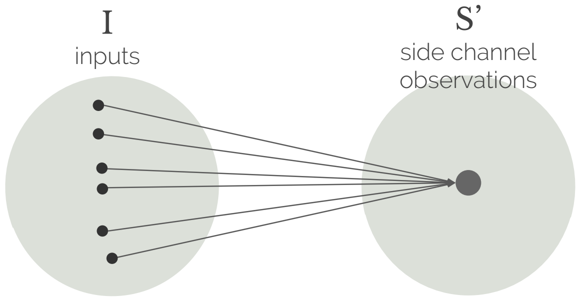


Figure 2.3: Illustration of the leakage function of a computation whose inputs result in an identical side-channel observation.

cases, our solutions evaluate the computation function f only once per partition instead of once for each input, thus improving performance.

Rewriting to Eliminate Leakage. In some cases, it may also be possible to transform the computation function f into another function f', such that f' is functionally equivalent to f, but the side-channel observations of f' are always identical regardless of the input. In other words, $f' : I \to O$ and $l' : I \to S'$, where l' is the leakage function for f' and $|S'| = 1$. Figure 2.3 illustrates this case. If it is possible to derive such a function f' (for example, using superoptimization), then our solutions simply replace the computation function f with the function f', instead of evaluating the computation multiple times.

2.2 Threat Model

This section describes our assumptions about the underlying hardware and software. We assume that the adversary is either an external entity that monitors *observation-based* side channels (*e.g.* time [50], memory address trace [43], or the `/proc` pseudo-filesystem [44]) or a co-resident process/VM that monitors *contention-based* side channels (*e.g.* cache [83] or branch predictor state [2]).

Hardware Assumptions. Our Raccoon and Escort solutions assume that the adversary can monitor and tamper with any digital signals on the processor's I/O pins. We also assume that the processor is a sealed chip [99]. We assume that the CPU encrypts data transferred to and from DRAM. All components other than the processor are untrusted, and we assume that the adversary can observe and tamper with any digital signal. Vantage prevents an attacker from correlating secrets with the processor's power consumption over long sequences of instructions. We assume that the power model analyzed using Vantage is at least as accurate as the power model that will be used by the adversary, but Vantage is flexible enough to permit richer power models as they get developed. The user of Vantage can select the relevant power model depending on the measurement techniques available to the adversary, such as physical oscilloscope probes used in the vicinity of the processor or remotely measured energy-related performance events such as Intel RAPL. Vantage prevents power and energy variations at the level of instructions and microcode operations but not cycle-level power variations; we assume that cycle-level power variations are eliminated using existing techniques such as Computational Blinking [5, 107, 116] or using custom transistors that enforce peak power consumption [28, 84, 86, 103, 104]. Vantage then removes variations in the power consumption at the level of instructions and microcode operations, effectively complementing (instead of replacing) existing power channel defenses. Vantage does not protect the DRAM from power channel attacks. Our solutions assume that there is no speculative execution, since our approach assumes that the processor executes only those instructions that eventually commit.

Software Assumptions. We assume that the adversary can run malicious applications on the same operating system and/or hardware as the victim's application. We allow malicious applications to probe the victim application's run-time statistics exposed by the operating system (*e.g.* the stack pointer in `/proc/pid/stat`). We assume that the input program is free of errors, *i.e.* (1) the program does not contain bugs that will induce application crashes, (2) the program does not exhibit undefined behavior, and (3) if multi-threaded, then the program is data-race free. We also assume that the program does not contain irreducible control flow graphs. Under these assumptions, our solutions do not introduce new termination-channel leaks, and they correctly obfuscate multi-threaded programs. Our solutions statically transform the user code into an obfuscated binary, so we assume that the adversary has access to this transformed binary code and to any symbol table and debug information that may be present. Our solutions do not support all features of the C99 standard. Specifically, our compilers cannot obfuscate I/O statements[1] and non-local `goto` statements. Our compilers cannot analyze libraries since their source code is not available when compiling the end-user's application. Our solutions include static analyses that check if the input program contains these unsupported language constructs. If such constructs are found in the input program, the program is rejected. Unlike Escort and Vantage, Raccoon does not prevent information leakage from loop trip counts, since naïvely obfuscating loop back-edges would create infinite loops. For the same reason, Raccoon does not obfuscate branches that represent terminal cases of recursive function calls. However, to address these issues, it is possible to adapt complementary techniques designed to close timing channels [117], which can limit information leaks from loop trip counts and recursive function calls. Finally, our solutions assume that the programmer annotates variables that store sensitive input values, so that the compiler can identify the parts of the program that need protection.

2.3 Limitations of Our Approach

Of course, our solutions have limitations, which we now describe.

System Calls, I/O Operations, and Library Calls. Compiler-based solutions cannot transform code that is outside the compiler's purview.

[1]Various solutions have been proposed that allow limited use of "transactional" I/O statements through runtime systems [19], operating systems [85], or the underlying hardware [14].

Specifically, I/O operations, system calls, and library calls are outside of the scope of the compiler. I/O operations are perhaps impossible to hide using our approach. Thus, our solutions are largely limited to computational kernels.

Programmer Annotations for Discovering Sensitive Inputs. Our approach relies on the programmer to correctly identify the input variables that store sensitive information. If the programmer fails to properly identify all sensitive inputs, then our solutions' transformations will not adquately prevent side-channel leakage. Although our solutions could conservatively assume that all inputs to the program hold sensitive information, doing so will likely result in poor performance.

Side Channels Caused by Speculation. Speculation-based side channels such as Spectre and Meltdown cannot be closed by our approach, since our approach assumes that the processor executes only those instructions that eventually commit.

Side Channels at the Lowest Levels of the Computing Stack. Although compiler-based defenses can close side channels that are at the lowest levels of the computing stack, such as power, electromagnetic radiation, and heat, the efficiency of such solutions is limited, due to both the undecidability of static analyses and also due to the limited interface exposed by the ISA. For instance, a multiplier unit could leak information about its operands through the power consumption, but the ISA does not permit exerting control over the multiplier's power consumption.

Design of Compiler Transformations. We manually design the code transformations in our solutions' compilers. However, given the variety of threat models and microarchitectures, it is improbable to expect a human to design each code transformation.

2.4 Related Work

Side-channel attacks through the OS, the underlying hardware, or the processor's output pins have been a subject of vigorous research. Formulated as the "confinement problem" by Lampson in 1973 [54], such attacks have become relevant for both cloud infrastructures, where the adversary and victim VMs can be co-resident [91], and for settings where adversaries have physical access to the processor [65, 123]. We first describe the related work in digital side channels, before focusing our attention on the power side channel.

2.4.1 Digital Side Channels

Side-Channels through OS and Microarchitecture. Some application-level information leaks are beyond the application's control, for example, an adversary reading a victim's secrets through the `/proc` filesystem [44], or a victim's floating point registers that are not cleared on a context switch [3]. In addition to such *explicit* information leaks, *implicit* flows rely on *contention* for shared resources, as observed by Wang and Lee [111] for cache channels and extended by Hunger et al. [105] to all microarchitectural channels.

Physical Access Attacks and Secure Processors. Execute-only Memory (XOM) [101] encrypts portions of memory to prevent the adversary from reading secret data or instructions from memory. The AEGIS [99] secure processor provides the notion of tamper-evident execution (recognizing integrity violations using a merkle tree) and tamper-resistant computing (preventing an adversary from learning secret data using memory encryption). Intel's Software Guard Extensions (SGX) [72] create "enclaves" in memory and limit accesses to these enclaves. Both XOM and SGX are only partially successful in prevent the adversary from accessing code because an adversary can still disassemble the program binary that is stored on the disk. In contrast, our solutions permit release of the transformed code to the adversary. Hence our solutions never need to encrypt code memory.

Side-Channel Defenses. Decades of prior research have produced numerous defenses against side channels, the vast majority of which close only a limited number of side channels with a single solution. For instance, numerous solutions exist that close only the cache side channel [27, 52, 111, 112, 119] or only the address-trace side channel [65, 90, 96, 97]. Raccoon [87] is the first solution that closes a broad class of side channels—in particular, the set of digital side channels—with a single solution. Similar to Raccoon, Escort [88] also closes digital side channels with a single solution, but unlike Raccoon, Escort focuses on closing floating-point digital side channels, which can arise from variable latency floating-point instructions and from software implementations of floating-point libraries, in which points-to set sizes are typically small. Given Escort's narrower focus on floating-point computations, Escort is faster than Raccoon by an order of magnitude.

Timing Side-Channel Defenses. Prior defenses against timing side-channel attacks utilize new algorithms [94], compilers [73], runtime systems [66], or secure processors [61]. However, these solutions only address one source of timing variations—either those that stem from the choice of the algorithm [95] or those that stem from the microarchitectural design [37]. By contrast, Raccoon and Escort close timing variations from both sources.

Floating-Point Side-Channel Defenses. Andrysco et al. [7] present `libfixedtimefixedpoint` (FTFP), the first software solution for closing the floating-point timing channel. FTFP has some weaknesses, as we now discuss, but the main contribution of their paper is the demonstration of the significance of this side channel, as they use variable-latency floating-point operations to break a browser's same-origin policy and to break differential privacy guarantees of remote databases. FTFP is a *fixed-point* library that consists of 19 hand-written functions that each operates in fixed time, independent of its inputs. FTFP is slow and imprecise. Cleemput et al. [23] introduce compiler transformations that convert variable-timing code into fixed-timing code. Their technique requires extensive manual intervention, applies only to the division operation, and provides weak security guarantees. Both solutions require manual construction of fixed-time code—a cumbersome process that makes it difficult to support a large number of operations. By contrast, Escort implements a fixed-time floating-point library, while preventing information leaks through timing as well as digital side channels. Escort includes a compiler that we have used to automate the transformation of 112 floating-point functions in the Musl standard C library, a POSIX-compliant C library. Escort also provides precision identical to the standard C library.

2.4.2 Power Side Channel

We now compare our work to prior attempts in closing or mitigating the power channel.

Transistor-Level Solutions. Sub-cycle power variations in any circuit can be eliminated at the level of transistors by either normalizing [28, 86, 103, 104] or randomizing [84] power consumption. Unfortunately, these solutions do not protect from power variations occurring due to microarchitectural optimizations including caching, prefetching, variable-latency instructions, and predication. These solutions also cannot be selectively turned off, so non-secret programs execute with the same overhead as secret programs. By contrast, VANTAGE [89] is a compiler-based solution for eliminating power variations over long sequences of instructions running on modern processors, and VANTAGE selectively transforms portions of programs based on the programs' security requirements. VANTAGE complements existing transistor-level solutions so that a broad-class of programs can be protected from power-channel attacks.

Code Modifications. Several solutions [16, 26, 36, 45, 46, 59] eliminate cycle-level power variations by manually changing the source code of vulnerable programs, but they only support programs that execute a fixed sequence of instructions and which access a fixed set of memory locations in a pre-defined sequence. Virtual Secure Circuits (VSCs) [22] execute the original program concurrently with a so-called shadow program comprising of instructions that are complementary to the original program in such a way that the original and complementary instructions exercise different paths of logic circuits. Unfortunately, VSCs assume tight synchronization between programs running on separate cores, they require caches and branch predictors to be disabled, and VSCs are difficult to use with programs containing branches, function calls, and floating-point arithmetic. Due to these limitations VSCs cannot be run on modern processors and with a broad class a programs.

Microarchitectural Solutions. Yang [115], Ambrose [6], and May [68, 69] suggest closing the power channel by adding noise using dynamic voltage and frequency scaling, out-of-order execution, register renaming, or by randomly inserting random instructions. Unfortunately, these approaches merely fix the symptoms of the problem, and they provide weak guarantees for closing the power channel, since the running time of programs can vary more than the noise introduced by perturbations. In comparison, our approach removes variations in the source code and our evaluation is stronger than the evaluation used in the above approaches.

Masking or Blinding Secrets. Secrets can be hidden or "blinded", either manually [78, 81] or semi-automatically using a compiler [11, 75], by XOR-ing them with a randomly-generated bit mask. The blinding process effectively randomizes the power consumption. However, blinding schemes are inherently limited to a restricted class of application programs, since blinding relies on specific mathematical properties of the computations. VANTAGE protects applications that use conditional branches and floating-point operations using code transformations. In our approach, we rely on the processor's ability to encrypt off-chip communication, thus obviating the need for blinding off-chip communication.

Analysis of Power Models. Several models [17, 48, 58, 92, 102, 120] of the processor's power consumption exist but these models focus on characterizing the execution of programs for efficiency, whereas in this work, we devise a technique to eliminate variations in power consumption. More specifically, we augment the compiler with information about power consumption, so that the compiler can generate code that consumes constant power regardless of application secrets. Unlike the regression model by McCann et al. [70], which models power consumption of *instructions*, VANTAGE's regression model is based on *microarchitectural events*, which are more generic than instructions.

Chapter 3

Closing Side Channels due to Source-Level Behavior

In this chapter, we describe our solution for closing side channels that arise due to differences in the source-level behavior of the program. Specifically, our solution closes *digital side channels*, which we define as side channels that carry information over discrete bits. These side channels are visible to the adversary at the level of both the program state and the instruction set architecture (ISA). Thus, address traces, cache usage, and data size are examples of digital side channels, while power draw, electromagnetic radiation, and heat are not. Portions of this chapter have been published in the 2015 USENIX Security Symposium [87][1].

Over the past five decades, numerous solutions [33, 34, 49, 61, 62, 65, 66, 73, 93, 96, 97, 99, 105, 108, 111, 112, 117–119] have been proposed for defending against side-channel attacks. Unfortunately, these defenses provide point solutions that leave the program open to other side-channel attacks. Thus, our

[1]Full citation: Ashay Rane, Calvin Lin, and Mohit Tiwari, "Raccoon: Closing Digital Side-Channels Through Obfuscated Execution" in USENIX Security Symposium (SEC), pages 431–446, 2015. The author of this dissertation contributed to this paper by performing the research, evaluating the solution, and comparing the solution with prior work.

goal is to instead find a single solution that simultaneously closes a broad class of side channels.

Our key insight is that digital side channels emerge from variations in program execution, so while other solutions attempt to hide the symptoms—for example, by normalizing the number of instructions along two paths of a branch—we instead attack the root cause by executing extraneous program paths, which we refer to as *decoy* paths. Intuitively, after obfuscation, the adversary's view through any digital side channel appears the same as if the program were run many times with different inputs. Of course, we must ensure that our system records the output of only the real path and not the decoy paths, so our solution uses a transaction-like system to update memory. Furthermore, on the real paths, each `store` operation first reads the old value of a memory location before writing the new value, while the decoy paths read the old value and write the same old value.

The only distinction between real and decoy paths lies in the values written to memory: Decoy and real paths will write different values, but unless an adversary can break the data encryption, she cannot distinguish decoy from real paths by monitoring digital side channels. Our solution does *not* defend against non-digital side-channel attacks, because analog side channels might reveal the difference between the encrypted values that are stored. For example, a decoy path might "increment" some variable x multiple times, and an adversary who can precisely monitor some non-digital side channel, such as power-draw, might be able to detect that the "increments" to x all write the same value, thereby revealing that the code belongs to a decoy path.

Nevertheless, our new approach offers several advantages. First, it defends against digital side-channel attacks. Second, it does not require that the programs themselves be secret, just the data. Third, it obviates the need for special-purpose hardware. Thus, standard processor features such as caches, branch predictors and prefetchers do not need to be disabled. Finally, in contrast with previous solutions for hiding specific side channels, it places few fundamental restrictions on the set of supported language features.

This work makes the following contributions:

1. We design a set of mechanisms, embodied in a system that we call Raccoon,[2] that closes digital side channels for programs executing on commodity hardware. Raccoon works for both single- and multi-threaded programs.

2. We evaluate the security aspects of these mechanisms in several ways. First, we argue that the obfuscated data- and control-flows are correct and are always kept secret. Second, we use information flows over inference rules to argue that Raccoon's own code does not leak information. Third, as an example of Raccoon's defense, we show that Raccoon protects against a simple but powerful side channel attack through the OS interface.

[2]Raccoons are known for their clever ability to break their scent trails to elude predators. Raccoons introduce spurious paths as they climb and descend trees, jump into water, and create loops.

3. We evaluate the performance overhead of Raccoon and find that its overhead is 8.9$\times$ smaller than that of GhostRider, which is the most similar prior work [61].[3] Unlike GhostRider, Raccoon defends against a broad range of side-channel attacks and places many fewer restrictions on the programming language, on the set of applicable compiler optimizations, and on the underlying hardware.

This chapter is organized as follows. Section 3.1 explains background information while Section 3.2 describes Raccoon's guarantees. We then describe our solution in detail in Section 3.3 before presenting our security evaluation and our performance evaluation in Sections 3.4 and 3.5, respectively. We discuss the implications of Raccoon's design in Section 3.6, and we conclude in Section 3.7.

3.1 Background: Memory Trace Obliviousness

GhostRider [61, 62] is a set of compiler and hardware modifications that transforms programs to satisfy Memory Trace Obliviousness (MTO). MTO hides control flow by transforming programs to ensure that the memory access traces are the same no matter which control flow path is taken by the program. GhostRider's transformation uses a type system to check whether the program is fit for transformation and to identify security-sensitive program values. It

[3]GhostRider [61] was evaluated with non-optimized programs executing on embedded CPUs, which results in an unrealistically low overhead ($\sim$10$\times$). Our measurements instead use a modern CPU with an aggressively optimized binary as the baseline.

also pads execution paths along both sides of a branch so that the length of the execution does not reveal the branch predicate value.

However, unlike our solutions, GhostRider cannot execute on generally-available processors and software environments because GhostRider makes strict assumptions about the underlying hardware and the user's program. Specifically, GhostRider (1) requires the use of new instructions to load and store data blocks, (2) requires substantial on-chip storage, (3) disallows the use of dynamic branch prediction, (4) assumes in-order execution, and (5) does not permit use of the hardware cache (it instead uses a scratchpad memory controlled by the compiler). GhostRider also does not permit the user code to contain pointers or to contain function calls that use or return secret information. By contrast, our solutions run on SGX-enabled Intel processors (SGX is required to encrypt values on the data bus) and permits user programs to contain pointers, permits the use of possibly unsafe arithmetic statements, and allows the use of function calls that use or return secret information.

3.2 System Guarantees

Raccoon protects against digital side-channel attacks. Raccoon guarantees that an adversary monitoring the digital signals of the processor chip cannot differentiate between the real path execution and the decoy path executions. Even after executing multiple decoy program paths, Raccoon guarantees the same final program output as the original program.

Raccoon guarantees that its obfuscation steps will not introduce new program bugs or crashes, so Raccoon does not introduce new information leaks over the termination channel.

Assuming that the original program is race-free, Raccoon's code transformations respect the original program's control and data dependences. Moreover, Raccoon's obfuscation code uses thread-local storage. Thus, Raccoon's obfuscation technique works seamlessly with multi-threaded applications because it does not introduce new data dependences.

3.3 Raccoon Design

This section describes the design and implementation of Raccoon from the bottom-up. We start by describing the two critical properties of Raccoon that distinguish it from other obfuscation techniques. Then, after describing the key building block upon which higher-level oblivious operations are built, we describe each of Raccoon's individual components: (1) a taint analysis that identifies program statements that require obfuscation (Section 3.3.3), (2) a runtime transaction-like memory mechanism for buffering intermediate results along decoy paths (Section 3.3.4), (3) a program transformation that obfuscates control-flow statements (Section 3.3.5), and (4) a code transformation that uses software Path ORAM to hide array accesses that depend on secrets (Section 3.3.6). We then describe Raccoon's program transformations that ensure crash-free execution (Section 3.3.7). Finally, we illustrate with a simple example the synergy among Raccoon's various obfuscation steps (Section 3.3.8).

```
1: $p \leftarrow \&a$;
2: **if** $secret$ = **true then**
3:     ...                                   ▷ **Real** path.
4: **else**
5:     ...                                   ▷ **Decoy** path.
6:     $p \leftarrow \&b$;          ▷ Dummy instructions do not update $\boldsymbol{p}$.
7:     $*p \leftarrow 10$;          ▷ Accesses variable $\boldsymbol{a}$ instead of $\boldsymbol{b}$!
8: **end if**
```

Figure 3.1: Illustrating the importance of Property 2. This code fragment shows how solutions that do not update memory along decoy paths may leak information. If the decoy path is not allowed to update memory, then the dereferenced pointer in line 7 will access $\boldsymbol{a}$ instead of accessing $\boldsymbol{b}$, which reveals that the statement was part of a decoy path.

3.3.1 Key Properties of Our Solution

Two key properties of Raccoon distinguish it from other branch-obfuscating solutions [27, 61, 62, 73]:

- **Property 1:** Both real and decoy paths execute actual program instructions.
- **Property 2:** Both real and decoy paths are allowed to update memory.

Property 1 produces decoy paths that—from the perspective of an adversary monitoring a digital side-channel—are indistinguishable from from real paths. Without this property, previous solutions can close one side-channel while leaving other side-channels open. To understand this point, we refer to Figure 3.2 and consider a solution that normalizes execution time along the two branch paths in the Figure by adding `NOP` instructions to the Not

```
function SQUARE_AND_MULTIPLY($m, s, n$)
    $z \leftarrow 1$
    for bit $b$ in $s$ from left to right do
        if $b = 1$ then
            $z \leftarrow m \cdot z^2 \bmod n$
        else
            $z \leftarrow z^2 \bmod n$
        end if
    end for
return $z$
end function
```

Figure 3.2: Source code to compute $m^s \bmod n$.

Taken path. This solution closes the timing channel but introduces different instruction counts along the two branch paths. On the other hand, the addition of dummy instructions to normalize instruction counts will likely result in different execution time along the two branch paths, since (on commodity hardware) the `NOP` instructions will have a different execution latency than the multiply instruction.

Property 2 is a special case of Property 1, but we include it because the ability to update memory is critical to Raccoon's ability to obfuscate execution. For example, Figure 3.1 shows that if the decoy path does not update the pointer ***p***, then the subsequent decoy statement will update ***a*** instead of ***b***, revealing that the assignment to ****p*** was part of a decoy path.

```
cmov(uint8_t pred, uint32_t t_val, uint32_t f_val) {
    uint32_t result;
    __asm__ volatile (
        "mov    %2, %0;"
        "test   %1, %1;"
        "cmovz  %3, %0;"
        "test   %2, %2;"
        : "=r" (result)
        : "r" (pred), "r" (t_val), "r" (f_val)
        : "cc"
    );
    return result;
}
```

Figure 3.3: `CMOV` wrapper

3.3.2 Oblivious Store Operation

Raccoon's key building block is the oblivious `store` operation, which we implement using the `CMOV` x86 instruction. This instruction accepts a condition code, a source operand, and a destination operand; if the condition is true, it moves the source operand to the destination. When both the source and the destination operands are in registers, the execution of this instruction does not reveal information about the branch predicate (hence the name *oblivious* store operation).[4] As we describe shortly, many components in Raccoon leverage the oblivious `store` operation. Figure 3.3 shows the x86 assembly code for the `CMOV` wrapper function.

[4]Contrary to the pseudocode describing the `CMOV` instruction in the Intel 64 Architecture Software Developer's Manual, our assembly code tests reveal that in 64-bit operating mode when the operand size is 16-bit or 32-bit, the instruction resets the upper 32 bits regardless of whether the predicate is true. Thus the instruction does not leak the value of the predicate via the upper 32 bits, as one might assume based on the manual.

3.3.3 Taint Analysis

Raccoon requires the user to annotate secret variables using the `__attribute__` construct. With these secret variables identified, Raccoon performs inter-procedural taint analysis to identify branches and data access statements that require obfuscation. Raccoon propagates taint across both implicit and explicit flow edges. The result of the taint analysis is a list of memory accesses and branch statements that must be obfuscated to protect privacy.

3.3.4 Transaction Management

To support Properties 1 and 2, Raccoon executes each branch of an obfuscated if-statement in a transaction. In particular, Raccoon buffers `load` and `store` operations along each path of an if-statement, and Raccoon writes values along the real path to DRAM using the oblivious `store` operation. If a decoy path tries to write a value to the DRAM, Raccoon uses the oblivious store operation to read the existing value and write it back. At compile time, Raccoon transforms `load` and `store` operations so that they will be serviced from the transaction buffers. Figure 3.4 shows pseudocode that implements transactional loads and stores. Loads and stores that appear in non-obfuscated code do not use the transaction buffers.

3.3.5 Control-Flow Obfuscation

To obfuscate control flow, Raccoon forces control flow along both paths of an obfuscated branch, which requires three key facilities: (1) a method of

```
// Writes a value to the transaction buffer.
tx_write(address, value) {
    if (threaded program)
        lock();

    // Write to both the transaction buffer
    // and to the non-transactional storage.
    tls->gl_buffer[address] = value;
    *address = cmov(real_idx == instance,
            value, *address);

    if (threaded program)
        unlock();
}

// Fetches a value from the transaction buffer.
tx_read(address) {
    if (threaded program)
        lock();

    value = *address;
    if (address in tls->gl_buffer)
        value = tls->gl_buffer[address];

    value = cmov(real_idx == instance,
            *address, value);

    if (threaded program)
        unlock();

    return value;
}
```

Figure 3.4: Pseudocode for transaction buffer accesses. Equality checks are implemented using `XOR` operation to prevent the compiler from introducing an explicit branch instruction.

perturbing the branch outcome, (2) a method of bringing execution control back from the end of the `if`-statement to the start of the `if`-statement so that execution can follow along the unexplored path, and (3) a method of ensuring that memory updates along decoy path(s) do not alter non-transactional memory. The first facility is implemented by the `obfuscate()` function (which forces sequential execution of both paths arising out of a conditional branch instruction). Although Raccoon executes both branch paths, it evaluates the (secret) branch predicate only once. This ensures that the execution of the first path does not unexpectedly change the value of the branch predicate. The second facility is implemented by the `epilog()` function (which transfers control-flow from the post-dominator of the `if`-statement to the beginning of the `if`-statement). Finally the third facility is implemented using the oblivious `store` operation described earlier. The control-flow obfuscation functions (`obfuscate()` and `epilog()`) use the libc `setjmp()` and `longjmp()` functions to transfer control between program points.

Safety of setjmp() and longjmp() Operations. The use of `setjmp()` and `longjmp()` is safe as long as the runtime system does not destroy the activation record of the caller of `setjmp()` prior to calling `longjmp()`. Thus, the function that invokes `setjmp()` should not return until `longjmp()` is invoked. To work around this limitation, Raccoon copies the stack contents along with the register state (identified by the `jmp_buff` structure) and restores the stack before calling `longjmp()`. To avoid perturbing the stack while manipulating the stack, Raccoon manipulates the stack using C macros and global variables.

As an additional safety requirement, the runtime system must not remove the code segment containing the call to `setjmp()` from instruction memory before the call to `longjmp()`. Because both `obfuscate()`—which calls `setjmp()`—and `epilog()`—which calls `longjmp()`—are present in the same program module, we know that that the code segment will not vanish before calling `longjmp()`.

Obfuscating Nested Branches. Nested branches are obfuscated in Raccoon by maintaining a stack of transaction buffers that mimics the nesting of transactions. Unlike traditional transactions, transactions in Raccoon are easier to nest because Raccoon can determine whether to commit the results or to store them temporarily in the transaction buffer *at the beginning of the transaction* (based on the secret value of the branch predicate).

3.3.6 Software Path ORAM

Raccoon's implementation of the Path ORAM algorithm builds on the oblivious `store` operation. Since processors such as the Intel x86 do not have a trusted memory (other than a handful of registers) for implementing the *stash*, we modify the Path ORAM algorithm from its original form [97]. Raccoon's Path ORAM implementation cannot directly index into arrays that represent the *position map* or the *stash*, so Raccoon's implementation streams over the *position map* and *stash* arrays and uses the oblivious `store` operation to selectively read or update array elements. Raccoon implements both recursive [96] as well as non-recursive versions of Path ORAM. Our software implementation of Path ORAM permits flexible sizes for both the *stash memory* and the *position map*.

Section 3.5.3 compares recursive and non-recursive ORAM implementations with an implementation that streams over the entire data array. Raccoon uses AVX vector intrinsic operations for streaming over data arrays. We find that even with large data sizes, it is faster to stream over the array than perform a single ORAM access.

3.3.7 Limiting Termination Channel Leaks

By executing instructions along decoy paths, Raccoon might operate on incorrect values. For example, consider the statement `if (y != 0) { z = x / y; }`. If `y` = 0 for a particular execution and if Raccoon executes the decoy path with `y` = 0, then the program will crash due to a division-by-zero error, and the occurrence of this crash in an otherwise bug-free program would reveal that the program was executing a decoy path (and, consequently, that `y` = 0).

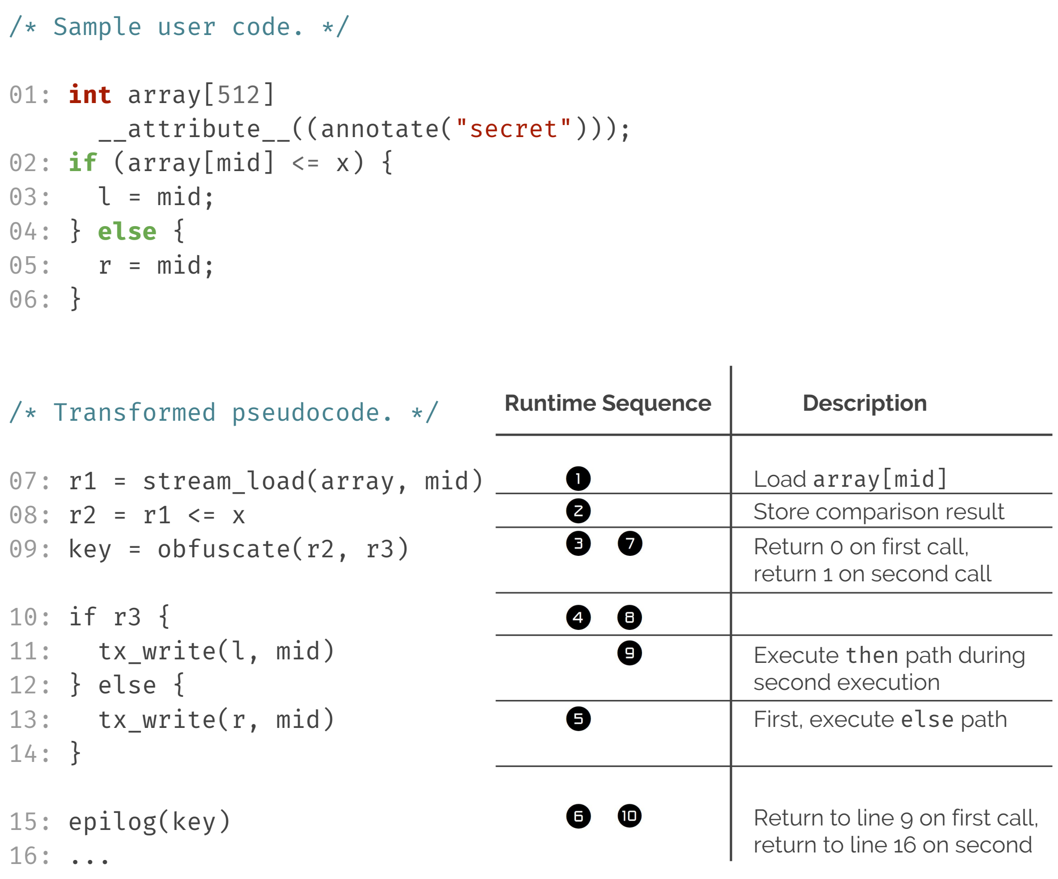
To avoid such situations, Raccoon prevents the program from terminating abnormally due to exceptions. For each integer division that appears in a transaction (along both real and decoy paths), Raccoon instruments the operation so that it obliviously (using `cmov`) replaces the divisor with a non-zero value. To prevent integer division overflow, Raccoon checks whether the dividend is equal to INT_MIN and whether the divisor is equal to -1; if so, Raccoon obliviously substitutes the divisor to prevent a division overflow. Raccoon also disables floating point exceptions using `fedisableexcept()`. Similarly, array `load` and `store` operations appearing on the decoy path are checked (again, obliviously, using `cmov`) for out-of-bounds accesses. Thus, to ensure that the execution of decoy paths does not crash the program, Raccoon patches unsafe operations. Section 3.4.3 demonstrates that this process of patching unsafe operations does not leak secret information to the adversary.

3.3.8 Putting It All Together

We now explain how Raccoon transforms the code shown in Figure 3.5. Here, the `secret` annotation informs Raccoon that the contents of `array` are secret.

Static taint analysis then reveals that the branch predicate (line 2) depends on the secret value, so Raccoon obfuscates this branch. Similarly, implicit flow edges from the branch predicate to the two assignment statements (at lines 3 and 5) indicate that Raccoon should use the oblivious `store` operation for both assignment statements.

Accordingly, Raccoon replaces direct memory stores for `l` and `r` with function calls that write into transaction buffers in lines 11 and 13 of the transformed pseudocode. The access to `array` in line 1 is replaced by an oblivious streaming operation in line 7. Finally, the branch in line 2 is obfuscated by inserting the `obfuscate()` and `epilog()` function calls. The `epilog()` and



```
/* Sample user code. */

01: int array[512]
      __attribute__((annotate("secret")));
02: if (array[mid] <= x) {
03:   l = mid;
04: } else {
05:   r = mid;
06: }
```

```
/* Transformed pseudocode. */

07: r1 = stream_load(array, mid)
08: r2 = r1 <= x
09: key = obfuscate(r2, r3)

10: if r3 {
11:   tx_write(l, mid)
12: } else {
13:   tx_write(r, mid)
14: }

15: epilog(key)
16: ...
```

Runtime Sequence	Description
❶	Load `array[mid]`
❷	Store comparison result
❸ ❼	Return 0 on first call, return 1 on second call
❹ ❽	
❾	Execute `then` path during second execution
❺	First, execute `else` path
❻ ❿	Return to line 9 on first call, return to line 16 on second

Figure 3.5: Sample code and transformed pseudocode.

`obfuscate()` function calls are coordinated over the `key` variable. To prevent the compiler from deleting or optimizing security-sensitive code sections, Raccoon marks security-sensitive functions, variables, and memory access operations as `volatile` (not shown in the transformed IR).[5]

At runtime, the transformed code executes the following steps:

1. Line 7 streams over the array and uses ORAM to load a single element (identified by `mid`) of the array.

2. Line 8 calculates the actual value of the branch predicate.

3. The key to this obfuscation lies in the `epilog()` function on line 15, which forces the transformed code to execute twice. The first time this function is called, it transfers control back to line 9. The second time this function is called, it simply returns, and program execution proceeds to other statements in the user's code.

4. Line 9 obfuscates the branch outcome. The first time the `obfuscate()` function returns, it stores 0 in `r3`, and control is transferred to the statement at line 13, where the `tx_write()` function call updates the transaction buffer. Non-transactional memory is updated only if this path corresponds to the real path.

[5]The C99 standard states that any "any expression referring to [a volatile object] shall be evaluated *strictly* according to the rules of the abstract machine", and the abstract machine is defined in a manner that considers that "issues of optimization are irrelevant".

The second time the `obfuscate()` function returns, it stores 1 in `r3`, and control is transferred to the statement at line 11, again calling the `tx_write()` function to update the transaction buffer. Again, non-transactional memory is updated only if this path corresponds to the real path.

3.4 Security Evaluation

In this section, we first demonstrate that the control-flows and data-flows in obfuscated programs are correct and that they are independent of the secret value. Then, using type-rules that track information flows, we argue that Raccoon's own code does not leak secret information. We then illustrate Raccoon's defenses against termination channels by reasoning about exceptions in x86 processors. Finally, we evaluate Raccoon's ability to prevent side-channel attacks via the `/proc` filesystem.

3.4.1 Security of Obfuscated Code

In this section, we argue that the obfuscated control-flows and data-flows (1) preserve the original program's dependences and (2) do not reveal any secret information. We only describe scalar loads and stores, since all array-loads and array-stores are obfuscated by simply streaming over the array. To simplify the explanation, the following arguments describe a top-level (*i.e.* a non-nested) branch. The same arguments can be extended to nested branches by maintaining a stack of transaction buffers.

Correctness of Obfuscated Data-Flow. To ensure correct data-flow, Raccoon uses a combination of transaction buffers and non-transactional storage (*i.e.* main memory). Raccoon sets up a fresh transaction buffer for each thread that executes a new path. Figure 3.4 shows the implementation of buffered `load` and `store` operations for use with transactions. The `store` operations along real paths write to both transaction buffers and non-transactional storage (since threads cannot share data that is stored in thread-local transaction buffers).

Consider a non-obfuscated program that stores a value to a memory location m in line 10 and loads a value from the same location in line 20. We now consider four possible arrangements of these two `load` and `store` operations in the obfuscated program, where each operation may reside either inside or outside a transaction. Our goal is to ensure that the `load` operation always reads the correct value, whether the correct value resides in a transactional buffer or in non-transactional storage.

- **`store` outside transaction, `load` inside transaction:** This implies that there is no `store` operation on m within the transaction. Thus, the transaction buffer does not contain an entry for m, and the `load` operation reads the value from the non-transactional storage.

- **`store` inside transaction, `load` inside transaction:** Since the transaction has previously written to m, the transaction buffer contains an entry for m, and the `load` operation fetches the value from the transaction buffer.

- **`store` inside transaction, `load` outside transaction:** This implies that the `store` operation must lie along the real path. Real-path execution updates non-transactional storage. Since `load` operations outside of transactions always fetch from non-transactional storage, the `load` operation reads the correct value of m.

- **`store` outside transaction, `load` outside transaction:** Raccoon does not change `load` or `store` operations that are located outside of the transactions. Hence the non-obfuscated reaching definition remains unperturbed.

Raccoon correctly obfuscates multi-threaded code as well. In programs obfuscated by Raccoon, decoy paths only update transactional buffers. Thus, only the `store` operations on real path affect reaching definitions of the obfuscated program. Furthermore, `store` (or `load`) operations along real path immediately update (or fetch) non-transactional storage and do not wait until the transaction execution ends. Thus, memory updates from execution of real paths are immediately visible to all threads, ensuring that inter-thread dependences are not masked by transactional execution. Finally, all transactional `load` and `store` operations use locks to ensure that these accesses are atomic. Put together, `load` and `store` operations on real paths are atomic and globally-visible, whereas `store` operations on decoy paths are only locally-visible and get discarded upon transaction termination. We thus conclude that the obfuscated code maintains correct data-flows for both single- and multi-threaded programs.

Concealing Obfuscated Data-Flow. Raccoon always performs two `store` operations for every transactional write operation, regardless of whether the write operation belongs to a real path or a decoy path. Moreover, by leveraging the oblivious `store` operation, Raccoon hides the specific value written to the transactional buffer or to the non-transactional storage. Although the `tx_read()` function uses an `if`-statement, the predicate of the `if`-statement is not secret, since an adversary can simply inspect the code and differentiate between repeated and first-time memory accesses. Thus, we conclude that the data-flows exposed to the adversary do not leak secret information.

Concealing Obfuscated Control-Flow. Raccoon converts control flow that depends on secret values into static (*i.e.* deterministically repeatable) control-flow that does not depend on secret values. Given a conditional branch instruction and two branch targets in the LLVM Intermediate Representation (IR), Raccoon always forces execution along the first target and then the second target. Thus, the sequence of executed branch targets depends on the (static) encoding of the branch instruction and not on the secret predicate.

Category	**Functions**	**Secret info.**
Control-flow obfuscation.	`obfuscate()`, `epilog()`.	Predicate value
Wrapper functions for unsafe operations.	`stream_load()`, `stream_store()`, `div_wrapper()`.	Array index, division operands.
Registering stack and array information.	`reg_memory()`, `reg_stack_base()`.	-
Initialization and clean-up functions.	`init_handler()`, `exit_handler()`.	-

Table 3.1: Entry-points of Raccoon's library.

3.4.2 Security of Obfuscation Code

Raccoon's own code should never leak secret information, so in this section, we demonstrate the security of the secret information maintained by Raccoon. Because the Raccoon code exposes only a handful of APIs (Table 3.1) to user applications, we can perform a detailed analysis of the code's entry- and exit-points to ensure that these interfaces never spill secret information outside of Raccoon's own code.

Type System for Tracking Information Flows. Figure 3.6 shows a subset of the typing rules used for checking the IR of Raccoon's own code. These rules express small-step semantics that track security labels. We assume the existence of a functions $l_r : \nu \rightarrow \gamma$ and $l_a : \Delta \rightarrow \gamma$ that map LLVM's virtual registers (ν) and addresses (Δ) to security labels (γ), respectively. Security labels can be of two types: `L` represents low-context (or public) information, while `H` represents high-context (or secret) information. Secret information

$$\text{T-LOAD}\quad \frac{l_r(p) = \mathtt{L}, A = pts(p), m = \max_{a \in A} l_a(a)}{\langle x = \mathbf{load} p; c, l_a, l_r\rangle \rightarrow \langle c, l_a, l_r[x \mapsto m]\rangle}$$

$$\text{T-STORE}\quad \frac{l_r(p) = \mathtt{L}, A = pts(p)}{\langle \mathbf{store}(x, p); c, l_a, l_r\rangle \rightarrow \langle c, \bigcup_{a \in A} l_a[a \mapsto max(l_a(a), l_r(x)), l_r]\rangle}$$

$$\text{T-BINOP}\quad \frac{}{\langle v = \mathbf{binary\text{-}op}(x, y); c, l_a, l_r\rangle \rightarrow \langle c, l_a, l_r[v \mapsto \mathbf{max}(l_r(x), l_r(y))]\rangle}$$

$$\text{T-UNOP}\quad \frac{}{\langle v = \mathbf{unary\text{-}op}(x); c, l_a, l_r\rangle \rightarrow \langle c, l_a, l_r[v \mapsto l_r(x)]\rangle}$$

$$\text{T-BRANCH}\quad \frac{l_r(p) = \mathtt{L}, \langle c_t; c, l_a, l_r\rangle \rightarrow \langle c, {l_a}', {l_r}'\rangle, \langle c_f; c, l_a, l_r\rangle \rightarrow \langle c, {l_a}'', {l_r}''\rangle}{\langle \mathbf{branch}(p, c_t, c_f); c, l_a, l_r\rangle \rightarrow \langle c, M({l_a}', {l_a}''), M({l_r}', {l_r}'')\rangle}$$

$$\text{T-CMOV}\quad \frac{}{\langle v = \mathbf{cmov}(p, t, f); c, l_a, l_r\rangle \rightarrow \langle c, l_a, l_r[v \mapsto \mathtt{L}]\rangle}$$

$$\text{T-SKIP}\quad \frac{}{\langle v = \mathbf{skip}; c, l_a, l_r\rangle \rightarrow \langle c, l_a, l_r\rangle}$$

$$\text{T-SEQUENCE}\quad \frac{\langle c_0, l_a, l_r\rangle \rightarrow \langle {c_0}', {l_a}', {l_r}'\rangle}{\langle c_0; c_1, l_a, l_r\rangle \rightarrow \langle {c_0}'; c_1, {l_a}', {l_r}'\rangle}$$

$$M(l', l'') = \forall_{x \,\in\, \{K(l') \,\cup\, K(l'')\}}\ (x,\ max(l'(x), l''(x))) \qquad K(l) = \{x \mid (x, s) \ \in \ l\}$$

Figure 3.6: Typing rules and supporting functions that check security of Raccoon's code.

listed in Table 3.1 is assigned the H security label, while all other information is assigned the L security label. We also assume the existence of a function $pts : r \rightarrow \{\Delta\}$ that returns the points-to set for a given virtual register r.

Our goal is to ensure that Raccoon does not leak secret information either through control-flow (branch instructions) or data-flow (load and store instructions). The typing rules in Figure 3.6 verify that information labeled as secret never appears as an address in a load or store instruction and never appears as a predicate in a branch instruction. Otherwise, the typing rules would result in a stuck transition. To prevent information leaks, Raccoon passes the secret information through the declassifier (cmov) before executing a load, store, or branch operation with a secret value. Due to its oblivious nature, the cmov operation resets the security label of its destination to L.

Security Evaluation of the cmov Operation. The tiny code size of the cmov operation (Figure 3.3) permits us to thoroughly inspect each instruction for possible information leaks. We use the Intel 64 Architecture Software Developer's Manual to understand the side-effects of each instruction.

Since the code operates on the processor registers only and never accesses memory, it operates within the (trusted) boundary of the sealed processor chip. The secret predicate is loaded into the %1 register. The mov instruction in line 4 initializes the destination register with t_val. The test instruction at line 5 checks if pred is zero and updates the Zero flag (ZF), Sign flag (SF), and the Parity flag (PF) to reflect the comparison. The subsequent `cmovz` instruction copies `f_val` into the destination register *only if* `pred` is zero. At this point, `ZF`, `SF`, and `PF` still contain the results of the comparison. The `test` instruction at line 7 overwrites these flags by comparing known non-secret values.

Since none of the instructions ever accesses memory, these instructions can never raise a General Protection Fault, Page Fault, Stack Exception Fault, Segment Not Present exception, or Alignment Check exception. None of these instructions uses the `LOCK` prefix, so they will never generate an Invalid Opcode (#UD) exception. As per the Intel Software Developer's Manual, the above instructions cannot raise any other exception besides the ones listed above. Through a manual analysis of the descriptions of 253 performance events[6] supported by our target platform, we discovered that only two performance events are directly relevant to the code in Figure 3.3: `PARTIAL_RAT_STALLS.FLAGS_MERGE_UOP` and `UOPS_RETIRED.ALL`. The first event (`FLAGS_MERGE_UOP`), which counts the number of performance-sensitive flags-merging micro-operations, produces the same value for our code, no matter whether the predicate is true or false. The second event (`UOPS_RETIRED.ALL`) counts the number of retired micro-operations. Since details of micro-operation counts for x86 instructions are not publicly available, we used an unofficial source of instruction tables[7] to verify that the micro-operation count for a `cmov` instruction is independent of the instruction's predicate. We thus con-

[6]Intel 64 and IA-32 Architectures Software Developers Manual, Section 19.5.

[7]`http://www.agner.org/optimize/instruction_tables.pdf`

Category	**Interrupt list**
Arithmetic errors	Division by zero, invalid operands, overflow, underflow, inexact results.
Memory access interrupts	Stack exception fault, general protection fault, page fault.
Debugging interrupts	Single-step, breakpoint.
Privileged operations	Invalid TSS, segment not present.
Coprocessor (legacy) interrupts	No coprocessor, coprocessor overrun, coprocessor error.
Other	Non-maskable interrupt, invalid opcode, double-fault abort.

Table 3.2: Categorized list of x86 hardware exceptions.

clude that the code in Figure 3.3 does not leak the secret predicate value.

3.4.3 Termination Leaks

In Section 3.3.7, we explained how Raccoon patches division operations and memory access instructions to prevent the program from crashing along decoy paths. We now explain why these patches are sufficient in preventing the introduction of new termination leaks. Table 3.2 shows a categorized list of exception conditions arising in Intel x86 processors[8] that may terminate programs. Among these interrupts, Raccoon transparently handles arithmetic and memory access interrupts.

Debugging interrupts are irrelevant for the program safety discussion because they do not cause the program to terminate. Our threat model does not apply obfuscation to OS or kernel code. Since we do not expect user programs to contain privileged instructions, Raccoon does not need to mask

[8]`http://www.x86-64.org/documentation/abi.pdf`

interrupts from privileged operations. Coprocessor interrupts are relevant to Numeric Processor eXtensions (NPX), which are no longer used today. Non-maskable interrupts are not caused by software events and thus need not be hidden by Raccoon. Branches in Raccoon always jump to the start of valid basic blocks, so invalid opcodes can never occur in an obfuscated version of a correct program. A double-fault exception occurs when the processor encounters an exception while invoking the handler for a previous exception. Aborts due to double-fault need not be hidden by Raccoon because none of the primary exceptions in an obfuscated program will leak secret information. In conclusion, Raccoon prevents abnormal program termination, thus guaranteeing that Raccoon's execution of decoy paths will never cause information leaks over the termination channel.

3.4.4 Defense Against Side-Channel Attacks

We have argued in Sections 3.4.1 and 3.4.2 that Raccoon closes digital side-channels. We now show a concrete example of a simple but powerful side-channel attack, and we use basic machine-learning techniques to visually illustrate Raccoon's defense against this attack. We model the adversary as a process that observes the instruction pointer (IP) values of the victim process. Both the victim process and the adversary process run on the same machine. The `driver` process starts the `victim` process and immediately pauses the `victim` process by sending a `SIGSTOP` signal. The `driver` process then starts the `adversary` process and sends it the process ID of the paused `victim`

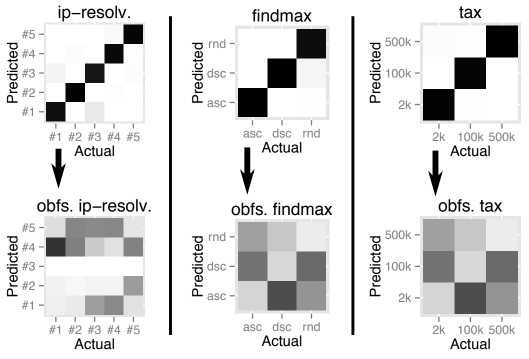


Figure 3.7: Confusion matrices for `ip-resolv`, `find-max` and `tax`. The top matrices describe original execution. The bottom matrices describe obfuscated execution.

process. This `adversary` process polls for the instruction pointer of the `victim` process every 5ms via the `kstkeip` field in `/proc/pid/stat`. When the `victim` process finishes execution, the `driver` process sends a `SIGINT` signal to the `adversary` process, signalling it to save its collection of instruction pointers to a file. We run the `victim` programs with various secret inputs and each run produces a (sampled) trace of instruction pointers. Each such trace is labelled with the name of the program and an identifier for the secret input. We collect 300 traces for each label. For the sake of brevity, we show results for only three programs from our benchmark suite.

The labelled traces are then passed through a Support Vector Machine for k-fold cross-validation (we choose $k = 10$) using LIBSVM v3.18. Using the prediction data, we construct a confusion matrix for each program, which conveys the accuracy of a classification system by counting the number of correctly-predicted and mis-predicted values (see Figure 3.7). The confusion

Name	Lines	Data size
Classifier	86	5 features, 5 records
IP resolver	247	3,500 records
Medical risk analysis	92	3,200 records
CRC32	76	10 KB
Genetic algorithm	446	pop. size = 1 KB
Tax calculator	350	-
Radix sort	675	256K elements
Binary search	35	10K elements
Dijkstra	50	1K edges
Find max	27	1K elements
Heap add	24	1K elements
Heap pop	42	10K elements
Histogram	40	1K elements
Map	29	1K elements
Matrix multiplication	28	500 x 500 values

Table 3.3: Benchmark programs used for performance evaluation of Raccoon. The bottom eight programs are also used to evaluate GhostRider. The remaining seven programs cannot be transformed by GhostRider because these programs use pointers and invoke functions in the secret context.

matrices show that for the non-secure executions, the classifier is able to label instruction pointer traces with high accuracy. By contrast, when using traces from obfuscated execution, the classifier's accuracy is significantly lower.

3.5 Performance Evaluation

Methodology. Raccoon is implemented in the LLVM compiler framework v3.6. In our test setup, the host operating system is CentOS 6.3. To evaluate performance, we use 15 programs (eight small kernels and seven small applications). Table 3.3 summarizes their characteristics and the associated input data sizes. The bottom eight programs in the table are the same programs used to evaluate GhostRider [61, 62], and we use these to compare Raccoon's overhead against that of GhostRider. To simplify the comparison between Raccoon and GhostRider, we use data sizes that are similar to those used to evaluate GhostRider [61]. Raccoon uses the `__attribute__` construct to mark secret variables—which mandates that the input programs are written in C/C++. However the rest of Raccoon operates entirely on the LLVM IR and does not use any source-language features. Thus, Raccoon can easily be ported to work with any language that can be compiled to the LLVM IR. All tests use the LLVM/Clang compiler toolchain.

We run all experiments on a machine with two Intel Xeon (Sandy Bridge) processors and with 32 GB (8×4 GB) DDR3 memory. Each processor has eight cores with 256 KB private L2 caches. The eight cores on a processor chip share a 20 MB L3 cache. Streaming encryption/decryption hardware makes the cost of accessing memory from encrypted RAM banks almost the same as the cost of accessing a DRAM bank. The underlying hardware does not support encrypted RAM banks, but we do not separately add any encryption-related overhead to our measurements because the streaming access cost is almost the same with or without encryption.

Performance measurements of our simulated ORAM use the native hardware performance event—`UNHALTED_CORE_CYCLES`. We measure overhead using `clock_gettime()`. Our software Path ORAM implementation is configured with a block size of 64 bytes. Each node in the Path ORAM tree stores 10 blocks. The stash size is selected at ORAM initialization time and is set to $\frac{\textit{ORAM block count}}{100}$ or 64 entries, whichever is higher.

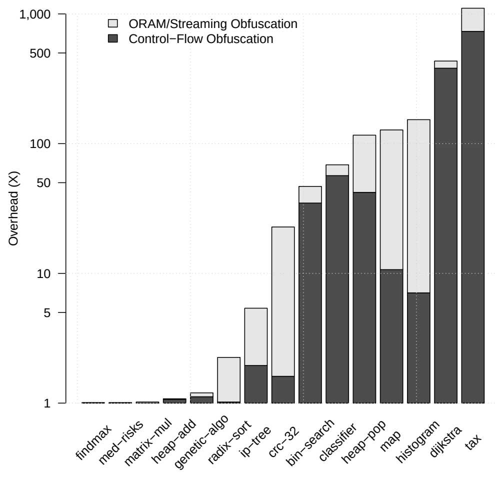


Figure 3.8: Sources of obfuscation overhead.

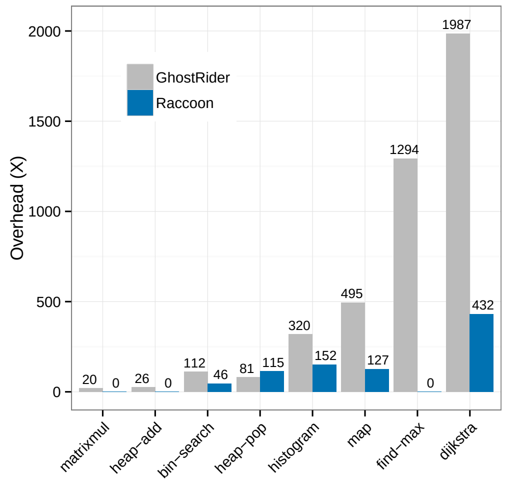


Figure 3.9: Overhead comparison on GhostRider's benchmarks. Even when we generously underestimate GhostRider's overhead, GhostRider sees an average overhead of 195×, while Raccoon's overhead is 21.8×.

3.5.1 Obfuscation Overhead

There are two main sources of Raccoon overhead: (1) the cost of the ORAM operations (or streaming) and (2) the cost of control-flow obfuscation (including the cost of buffering transactional memory accesses, the cost of copying program stack and CPU registers, and the cost of obliviously patching arithmetic and memory access instructions). We account for ORAM/streaming overhead over both real and decoy paths. Of course, the overhead varies with program characteristics, such as size of the input data, number of obfuscated statements, and number of memory access statements. Figure 3.8 shows the obfuscation overhead for the benchmark programs when compared with an aggressively optimized (compiled with `-O3`) non-obfuscated binary executable. The geometric mean of the overhead is $\sim 16.1\times$. Applications closer to the left end of the spectrum had low overheads due to Raccoon's ability to leverage existing compiler optimizations (if-conversion, automatic loop unrolling, and memory to register promotion). In most applications with high obfuscation overhead, a majority of the overhead arises from transactional execution in control-flow obfuscation.

3.5.2 Comparison with GhostRider

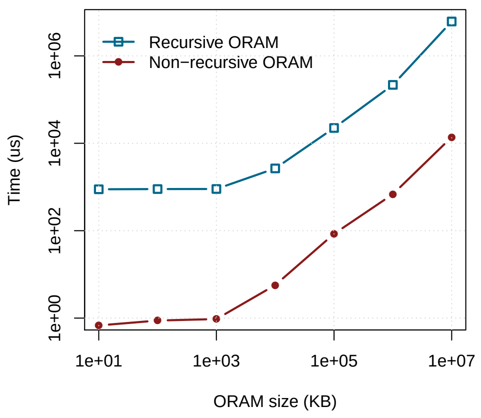
To place our work in the context of similar solutions to side-channel defenses, we compare Raccoon with the GhostRider hardware/software framework [61, 62] that implements Memory Trace Obliviousness. This section focuses on the performance aspects of the two systems, but Raccoon provides significant benefits over GhostRider beyond performance. First, Raccoon provides a broad coverage against many different side-channel attacks. Second, the dynamic obfuscation scheme used in Raccoon strengthens the threat model, since it allows the transformed code to be released to the adversary. Third, Raccoon does not require special-purpose hardware. Finally, since GhostRider adds instructions to mimic address traces in both branch paths, it requires that address traces from obfuscated code be known at compile-time, which significantly limits the programs that GhostRider can obfuscate. Raccoon relaxes this requirement by executing actual code, so Raccoon can transform more complex programs than GhostRider.

Methodology. We now describe our methodology for simulating the GhostRider solution. As with our Raccoon setup, we compare GhostRider's obfuscated program with an aggressively optimized (compiled with `-O3`) non-obfuscated version of the same program. Various compiler optimizations (dead code elimination, vectorization, constant merging, constant propagation, global value optimizations, instruction combining, loop-invariant code motion, and promotion of memory to registers) interfere with GhostRider's security guarantees, so we disable optimizations for the obfuscated program. We manually apply the transformations implemented in the GhostRider compiler. We simulate a processor that is modelled after the GhostRider processor, so we use a single-issue in-order processor that does not allow prefetching into the cache.

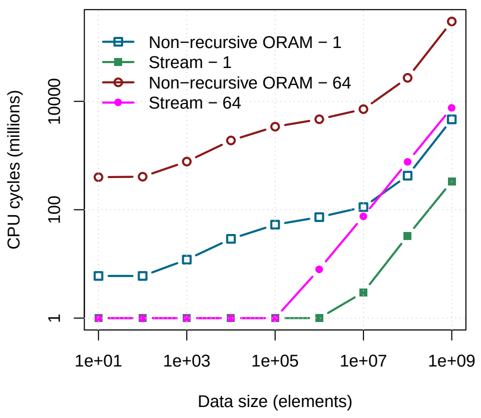
There are four reasons why our methodology significantly underestimates GhostRider's overhead. The first three reasons stem from our inability to faithfully simulate all features of the GhostRider processor: (1) We simulate variable-latency instructions, (2) we simulate the use of a dynamic branch predictor, and (3) we simulate a perfect cache for non-ORAM memory accesses. All three of these discrepancies give GhostRider an unrealistically fast hardware platform. The fourth reason arises because our simulator does not support AVX vector instructions, so we are unable to compare GhostRider against a machine that can execute AVX vector instructions.

The non-obfuscated execution uses a 4-issue, out-of-order core with support for Access Map Pattern Matching prefetching scheme [42] for the L1, L2 and L3 data caches. In all other respects, the two processor configurations are identical. Both processors are clocked at 1 GHz. The processor configuration closely matches the configuration described by Fletcher et al. [33], and based on their measurements, we assume that the latency to all ORAM banks is 1,488 cycles per cache line. We run GhostRider's benchmarks on this modified Marss86 simulator and manually add the cost of each ORAM access to the total program execution latency.

Performance Comparison. Figure 3.9 compares the overhead of GhostRider on the simulated processor and the overhead of Raccoon. Only those benchmark programs that meet GhostRider's assumptions are used in this comparison. The remaining seven applications cannot be transformed by the GhostRider solution because they use pointers or because they invoke func-



(a) Initialization cost of recursive and non-recursive ORAM implementation (median of 10 measurements for each sample).



(b) Performance comparison of software Path ORAM and streaming over the entire array.

Figure 3.10: Software ORAM performance.

tions in the secret context. We see that Raccoon's overhead (geometric mean of 16.1× over all 15 benchmarks, geometric mean of 21.8× over GhostRider-only benchmarks) is significantly lower than GhostRider's overhead (geometric mean of 195×), even when giving GhostRider's processor substantial benefits (perfect caching, lack of AVX-vector support in the baseline processor, and dynamic branch prediction).

3.5.3 Software Path ORAM

This section considers choices for Raccoon's ORAM implementation. In particular, to run on typical general-purpose processors, we need to modify the Path ORAM algorithm to assume just a tiny amount of trusted memory, which forces us to stream the *position map* and *stash* multiple times to obliviously copy or update elements.

We thus consider three possible implementations. The first, recursive ORAM [96], places the *position map* in a smaller ORAM until the *position map* of the smallest ORAM fits in the CPU registers. The second is a non-recursive solution that streams over a single large *position map*. The third uses AVX intrinsic operations and streams over the entire array to access a single element.

Figure 3.10(a) compares the cost of ORAM initialization for different ORAM sizes in our recursive and non-recursive ORAM implementations. On this log-log scale, we see that the non-recursive ORAM is significantly faster than the recursive ORAM for all sizes. Figure 3.10(b) compares our non-recursive ORAM implementation against the streaming approach. In particular, it measures the cost of accessing a single element and the cost of 64 single-element random accesses using ORAM and streaming. We see that the streaming implementation is orders of magnitude faster than our non-recursive ORAM.

In summary, our software implementation of Path ORAM requires non-trivial changes to the original Path ORAM algorithm. Unfortunately, these changes impose a prohibitively large memory bandwidth requirement, making the modified software Path ORAM far costlier than streaming over arrays. Raccoon's obfuscation technique is compatible with the use of dedicated ORAM memory controllers, and Raccoon's overhead can be further reduced by using such special purpose hardware [65].

3.6 Discussion

Closing Other Side-Channels. The existing Raccoon implementation does not defend against kernel-space side-channel attacks. However, many of Raccoon's obfuscation principles can be applied to OS kernels as well. Memory updates in systems such as TxOS [85] can be made oblivious using Raccoon's `cmov` operation. By contrast, non-digital side-channels appear to be fundamentally beyond Raccoon's scope since physical characteristics (power, temperature, EM radiation) of hardware devices make it possible to differentiate between real values and decoy values.

Multi-threaded Programs. Raccoon's data structures are stored in thread-local storage (TLS), so Raccoon can access internal data structures without using locks. Raccoon initializes these data-structures at thread entry-points (identified by `pthread_create()`) and frees them at thread destruction-points (identified by `pthread_exit()`). Raccoon prevents race conditions on the user program's memory by using locks where necessary. Most importantly, as long as the user program is race-free, Raccoon maintains the correct data-flow dependences in both single-threaded and multi-threaded programs, as described in Section 3.4.1.

Taint Analysis. Raccoon's taint analysis is sound but not complete, so it over-approximates the amount of code that must be obfuscated. For large programs, this over-approximation is a significant source of overhead. Raccoon's taint analysis is flow-insensitive, path-insensitive, and context-insensitive, and Raccoon uses a rudimentary alias analysis technique that assumes two pointers alias if they have the same type. We believe that more precise static analysis techniques can be used to greatly shrink Raccoon's taint graph, thus reducing the obfuscation overhead.

Limitations Imposed by Hardware. Various x86 instructions (`DIV`, `SQRT`, etc.) consume different cycles depending on their operands. Such operand-dependent instruction execution latency introduces the biggest hurdle in ensuring the safety of Raccoon-obfuscated programs. We also believe that the performance overhead of obfuscated programs would be substantially smaller than the current overhead if processors came equipped with (small) scratch-pad memory. Based on these conjectures, we plan to explore the impact of modified hardware designs in the near future.

3.7 Conclusions

In this chapter, we have introduced the notion of digital side-channel attacks, and we have presented a system named Raccoon to defend against such attacks. We have evaluated Raccoon's performance against 15 programs to show that its overhead is significantly less than that of the best prior work and that it has several additional benefits: It expands the threat model, it removes special-purpose hardware, it permits the release of the transformed code to the adversary, and it also expands the set of supported language features. In comparing Raccoon against GhostRider, we find that Raccoon's overhead is more than 8.9× lower.

Raccoon's obfuscation technique can be enhanced in several ways. First, while the performance overhead of Raccoon-obfuscated programs is high enough to preclude immediate practical deployment, we believe that this overhead can be substantially reduced by employing deterministic or special-purpose hardware. Second, Raccoon's technique of transactional execution and oblivious memory update can be applied to the operating system (OS) kernel, thus paving the way for protection against OS-based digital side-channel attacks. Finally, in addition to defending against side-channel attacks, we believe that Raccoon can be strengthened to defend against covert-channel communication.

Chapter 4

Closing Side Channels due to Floating-Point Instructions

In this chapter, we present our research on closing side channels arising due to variable-latency floating point instructions. We also present an optimized approach for closing control flow side channels. Portions of this work have been published in the 2016 USENIX Security Symposium [88][1].

Numerous side channels exist, including instruction and data caches [80, 83], branch predictors [2], memory usage [44, 110], execution time [18, 95], heat [67], power [51], and electromagnetic radiation [35], but one particularly insidious side channel arises from the execution of variable-latency floating-point instructions [7, 37], in which an instruction's latency varies widely depending on its operands, as shown in Table 4.1.

Both x86[2] and ARM[3] provide variable-latency floating-point instruc-

[1]Full citation: Ashay Rane, Calvin Lin, and Mohit Tiwari, "Secure, Precise, and Fast Floating-Point Operations on x86 Processors" in USENIX Security Symposium (SEC), pages 71–86, 2016. The author of this dissertation contributed to this paper by designing the research, performing the research, evaluating the solution, and comparing the solution with prior work.

[2]`http://www.agner.org/optimize/instruction_tables.pdf`

[3]`http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344k/ch16s07s01.html`

Zero	**Normal**	**Subnormal**	**Infinity**	**NaN**
7	11	153	7	7

Table 4.1: Latency (in cycles) of the `SQRTSS` instruction for various operands.

tions. This variable latency stems from the desire to have graceful floating-point arithmetic behavior, which, as we explain in Section 4.2, requires the use of so-called *subnormal* values [31], which are processed using special algorithms. Since subnormal values are rare, hardware vendors typically support such values in microcode, so as not to slow down the common case. The resulting difference in instruction latency creates a timing side channel, which has been used to infer cross-origin data in browsers and to break differential privacy guarantees of a remote database [7].

However, variable latency floating-point instructions represent only a part of the problem, since higher level floating-point operations, such as sine and cosine, are typically implemented in software. Thus, the implementation of these floating-point operations can leak secret information through other side channels as well. Depending on the secret values, programs can throw exceptions, thereby leaking the presence of abnormal inputs through termination. Programs can also contain conditional branches, which can leak secrets through the instruction pointer, branch predictor, or memory access count. Finally, programs that index into lookup tables can leak secrets through the memory address trace.

To prevent information leaks in both floating-point instructions and floating-point software, a strong solution should ensure at least four key properties, which correspond to the side channels that we discussed above: (1) fixed-time operations that are independent of secret values, (2) disabled exceptions, (3) sequential control flow, and (4) uniform data accesses that are independent of the value of secret variables. Previous solutions [7, 23] are inadequate because they do not ensure all four properties, are slow, are orders of magnitude less precise, or are difficult to implement.

This work presents a novel solution that closes side channels arising from both hardware and software implementations of floating point operations, providing all four properties mentioned above. Our compiler-based solution has two components.

The first component creates building blocks of elementary floating-point operations for instructions that are natively supported by the hardware (addition, subtraction, multiplication, division, square root, and type conversion). Our solution leverages unused SIMD lanes so that fast operations on normal operands are accompanied by slower dummy computations on subnormal operands, yielding a consistent yet low instruction latency for all types of operands.

The second component is a software library of higher-level floating-point operations like sine and cosine. The key to creating this second component is a new code transformation that produces fixed-latency functions through normalized control flows and data access patterns. Code generated by our compiler closes *digital side-channels*, which have been defined to be those side channels that carry information over discrete bits [87]. Whereas previous work in closing digital side channels employs a runtime system [87], our solution shifts much of the work to compile time, yielding a significantly smaller runtime overhead.

This work makes the following contributions:

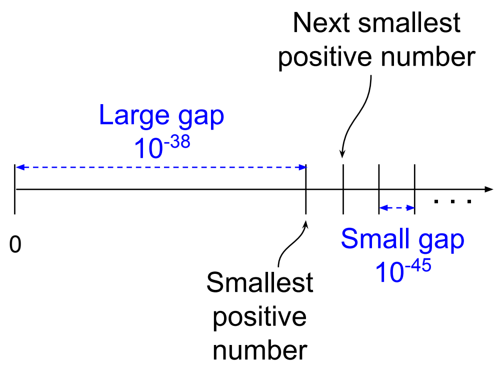
1. We present a novel compiler-based system, called *Escort*, for closing digital side channels that arise from the processing of floating-point instructions.
2. **Secure:** We demonstrate that our solution is secure not just against timing but also against digital side channels. We demonstrate Escort's capabilities by defeating a machine-learning side-channel attack, by defending against a timing attack on the Firefox web browser, by conducting extensive performance measurements on an x86 processor, and by verifying our solution's code using typing rules.
3. **Precise:** We show that Escort provides precision that is *identical* to that of the standard C math library. By contrast, the previous solution's precision is off by several million floating-point values.
4. **Fast:** We show that our solution is fast. On a set of micro-benchmarks that exercise elementary floating-point operations, Escort is 16$\times$ faster than the previous solution [7].
5. As an ancillary contribution, we introduce a methodology for evaluating the precision and security of floating-point operations, which is fraught

with subtleties.

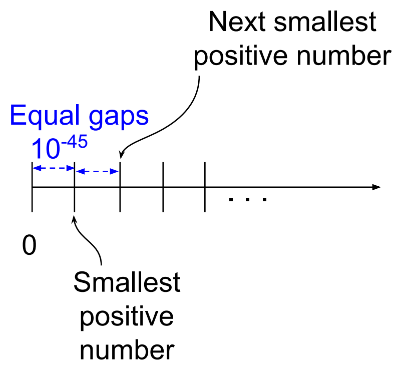
The rest of this work is organized as follows. Section 4.1 describes Escort's guarantees. We provide background in Section 4.2 before presenting our solution in Section 4.3. We evaluate our solution in Sections 4.4–4.6. Finally, we conclude in Section 4.7.

4.1 Escort's Guarantees

Escort rejects programs that contain unsupported features—I/O operations and recursive function calls. Unlike prior work [61, 87], Escort *does* transform loops that leak information through trip counts. Escort is unable to handle programs containing irreducible control flow graphs (CFGs), but standard compiler transformations [76] can transform irreducible CFGs into reducible CFGs. Escort assumes that the input program does not use vector instructions, does not exhibit undefined behavior, does not terminate abnormally through exceptions, and is free of race conditions. Given a program that abides by these limitations, Escort guarantees that the transformed code produces identical results as the original program, does not leak secrets through timing or digital side channels, and that the transformed code does not terminate abnormally.



(a) Without subnormal values.



(b) With subnormal values.

Figure 4.1: Impact of allowing subnormal numbers. Without subnormal values, there exists a much larger gap between zero and the smallest positive number than between the first two smallest positive numbers. With subnormal numbers, the values are more equally spaced. (The figure is not drawn to scale.)

4.2 Background

The variable latency of floating-point instructions creates security vulnerabilities. In this section, we explain subnormal numbers, which are the cause of the variable latency, and we explain the difficulty of fixing the resulting vulnerability. We also explain how the Unit of Least Precision (ULP) can be used to quantify the precision of our and competing solutions.

4.2.1 Subnormal Numbers

Subnormal numbers have tiny exponents, which result in floating-point values that are extremely close to zero: $10^{-45} < |x| < 10^{-38}$ for single-precision numbers and $10^{-324} < |x| < 10^{-308}$ for double-precision numbers. Subnormal values extend the range of floating-point numbers that can be represented, but more importantly, they enable *gradual underflow*—the property that as floating-point numbers approach zero along the number scale, the difference between successive floating-point numbers does not increase[4]. Figures 4.1a and 4.1b show the differences between zero and the two smallest positive floating-point numbers. With subnormal numbers, the gap between any two consecutive floating-point values is never larger than the values themselves, thus exhibiting Gradual Underflow. Subnormal numbers are indispensable because gradual underflow is required for reliable equation solving and convergence acceleration [31, 47].

To avoid the added hardware complexity of supporting subnormal numbers, which occur infrequently, vendors typically process subnormal values in microcode, which is orders of magnitude slower than hardwired logic.

The resulting difference in latencies creates a security vulnerability. An adversary that can measure the latency of a floating-point instruction can make reasonable estimates about the operand type, potentially inferring secret values using the timing channel. While subnormal values occur infrequently in typical program execution, an adversary can deliberately induce subnormal values in the application's inputs to enable subnormal operand timing attacks.

[4] `https://www.cs.berkeley.edu/~wkahan/ARITH_17U.pdf`

4.2.2 Floating-Point Error Measurement

Unlike real (infinite precision) numbers, floating-point numbers use a limited number of bits to store values, thus making them prone to rounding errors. Rounding errors in floating-point numbers are typically measured in terms of the Unit of Least Precision (ULP) [77]. The ULP distance between two floating-point numbers is the number of distinct representable floating-point numbers between them, which is simply the result of subtracting their integer representations. If the result of the subtraction is zero, the floating-point numbers must be exactly the same.

4.3 Our Solution: Escort

Escort offers secure counterparts of ordinary non-secure floating-point operations, including both elementary operations and higher-level math operations. The elementary operations include the six basic floating-point operations that are natively supported by the ISA—type conversion, addition, subtraction, multiplication, division, and square root—and a conditional data copy operation. The 112 higher-level math operations are those that are implemented using a combination of native instructions. Examples of higher-level functions include sine, cosine, tangent, power, logarithm, exponentiation, absolute value, floor, and ceiling.

The next subsections describe Escort's design in three parts. First, we describe the design of Escort's secure elementary operations. These operations collectively form the foundation of Escort's security guarantees. Second,

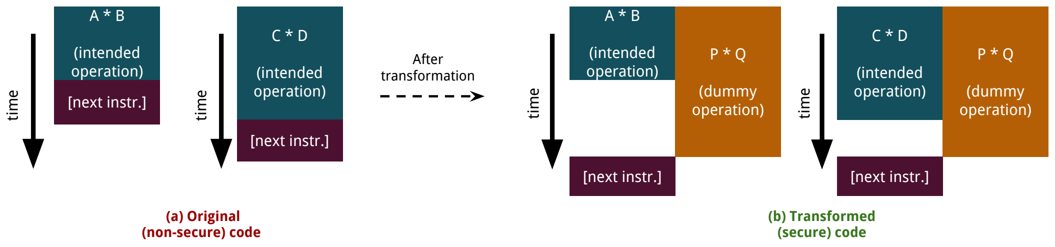


Figure 4.2: The key idea behind Escort's secure elementary operations. The operation is forced to exhibit a fixed latency by executing a fixed-latency long-running operation in a spare SIMD lane.

we describe Escort's compiler, which accepts non-secure code for higher-level operations and converts it into secure code. This compiler combines a code transformation technique with Escort's secure elementary operations. Third, we present an example that shows the synergy among Escort's components.

4.3.1 Elementary Operations

The key insight behind Escort's secure elementary operations is that the latencies of SIMD instructions are determined by the slowest operation among the SIMD lanes (see Figure 4.2), so the Escort compiler ensures that each elementary instruction runs along side a dummy instruction whose operand will produce the longest possible latency. Our analysis of 94 x86 SSE and SSE2 instructions (which includes single- and double-precision arithmetic, comparison, logical, and conversion instructions) reveals: (1) that only the multiplication, division, square root, and single-precision to double-precision conversion (upcast) instructions exhibit latencies that depend on their operands and (2) that subnormal operands induce the longest latency.

```
double escort_mul_dp(double x, double y) {
    const double k_normal_dp = 1.4;
    const double k_subnormal_dp = 2.225e-322;

    double result;
    __asm__ volatile(
            "movdqa     %1, %%xmm14;"
            "movdqa     %2, %%xmm15;"
            "pslldq     $8, %1;"
            "pslldq     $8, %2;"
            "por        %3, %1;"
            "por        %4, %2;"
            "movdqa     %2, %0;"
            "mulpd      %1, %0;"
            "psrldq     $8, %0;"
            "movdqa     %%xmm14, %1;"
            "movdqa     %%xmm15, %2;"
            : "=x" (result), "+x" (x), "+x" (y)
            : "x" (k_subnormal_dp), "x" (k_normal_dp)
            : "xmm15", "xmm14");
    return result;
}
```

Figure 4.3: Escort's implementation of double-precision multiplication, using the AT&T syntax.

```
copy(uint8_t pred, uint32_t t_val, uint32_t f_val) {
    uint32_t result;
    __asm__ volatile (
        "mov    %2, %0;"
        "test   %1, %1;"
        "cmovz  %3, %0;"
        "test   %2, %2;"
        : "=r" (result)
        : "r" (pred), "r" (t_val), "r" (f_val)
        : "cc"
    );
    return result;
}
```

Figure 4.4: Code for conditional data copy operation that does not leak information over digital side channels. This function returns `t_val` if `pred` is `true`; otherwise it returns `f_val`. The assembly code uses AT&T syntax.

In particular, Escort's fixed-time floating-point operations utilize SIMD lanes in x86 SSE and SSE2 instructions. Our solution (1) loads genuine and dummy (subnormal) inputs in spare SIMD lanes of the same input register, (2) invokes the desired SIMD instruction, and (3) retains only the result of the operation on the genuine inputs. Our tests confirm that the resulting SIMD instruction exhibits the worst-case latency, with negligible variation in running time (standard deviation is at most 1.5% of the mean). Figure 4.3 shows Escort's implementation of one such operation.

Escort includes Raccoon's conditional data copy operation (see Figure 4.4) which does not leak information through digital side channels. This operation copies the contents of one register to another register if the given condition is true. However, regardless of the condition, this operation consumes a fixed amount of time, executes the same set of instructions, and does

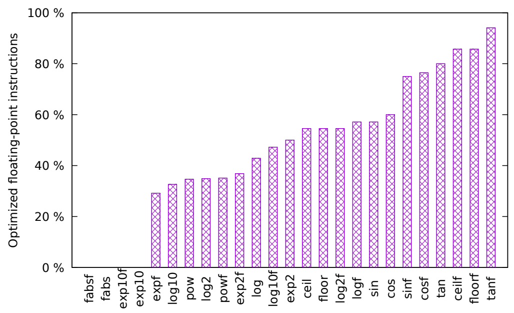


Figure 4.5: Percentage of instructions that are left uninstrumented (without sacrificing security) after consulting the SMT solver.

not access application memory.

4.3.2 Compiling Higher-Level Operations

Escort's compiler converts existing non-secure code into secure code that prevents information leakage through digital side channels. First, our compiler replaces all elementary floating-point operations with their secure counterparts. Next, our compiler produces straight-line code that preserves control dependences among basic blocks while preventing instruction side effects from leaking secrets. Our compiler then transforms array access statements so that they do not leak information through memory address traces. Finally, our compiler transforms loops whose trip count reveals secrets over digital side channels. We now describe each step in turn.

4.3.2.1 Step 1: Using Secure Elementary Operations

The Escort compiler replaces x86 floating-point type-conversion, multiplication, division, and square root assembly instructions with their Escort counterparts. However, Escort's secure elementary operations can be up to two orders of magnitude slower than their non-secure counterparts. Hence, our compiler minimizes their usage by using taint tracking and by employing the quantifier-free bit-vector logic in the Z3 SMT solver [30], which is equipped with floating-point number theory. If the solver can prove that the operands can never be subnormal values, then Escort refrains from replacing that instruction.

In effect, the Escort compiler constructs path-sensitive Z3 expressions for each arithmetic statement in the LLVM IR. For every Φ-node that produces an operand for an arithmetic expression, Escort creates one copy of the expression for each input to the Φ-node. If the solver reports that no operand can have a subnormal value, then Escort skips instrumentation of that floating-point operation.

We set a timeout of 40 seconds for each invocation of the SMT solver. If the solver can prove that the instruction never uses subnormal operands, then Escort skips replacing that floating-point instruction with its secure counterpart. Figure 4.5 shows the percentage of floating-point instructions in commonly used math functions that are left untransformed by Escort.

This optimization is conservative because it assumes that all floating-point instructions in the program have subnormal operands unless proven otherwise. The correctness of the optimization is independent of the code's use of pointers, library calls, system calls, or dynamic values. The static analysis used in this optimization is flow-sensitive, path-sensitive, and intra-procedural.

```
for each basic block $bb$ in function do
    if $entry\_block(bb)$ then
        $pred[bb] \leftarrow$ true
    else
        $pred[bb] \leftarrow$ false
    end if
end for

for each basic block $bb$ in function do
    $br \leftarrow branch(bb)$
    if $unconditional\_branch(br)$ then
        $\{s\} \leftarrow successors(bb)$
        $pred[s] \leftarrow pred[s] \vee pred[bb]$
        $pred[s] \leftarrow simplify(pred[s])$
    else                                                  ▷ Conditional Branch.
        $\{s_1, s_2\} \leftarrow successors(bb)$
        if $loop\_condition\_branch(br)$ then
                                              ▷ Skip branches that represent loops.
            $pred[s_1] \leftarrow pred[s_1] \vee pred[bb]$
            $pred[s_2] \leftarrow pred[s_2] \vee pred[bb]$
        else
            $p \leftarrow condition(br)$
            $pred[s_1] \leftarrow pred[s_1] \vee (pred[bb] \wedge p)$
            $pred[s_2] \leftarrow pred[s_2] \vee (pred[bb] \wedge \neg p)$
        end if
        $pred[s_1] \leftarrow simplify(pred[s_1])$
        $pred[s_2] \leftarrow simplify(pred[s_2])$
    end if
end for
```

Figure 4.6: Algorithm for predicating basic blocks.

4.3.2.2 Step 2: Predicating Basic Blocks

Basic block predicates represent the conditions that dictate whether an instruction should execute. These predicates are derived by analyzing conditional branch instructions. For each conditional branch instruction that evaluates a predicate p, the Escort compiler associates the predicate p with all basic blocks that execute if the predicate is true, and it associates the predicate $\neg p$ with all basic blocks that execute if the predicate is false. For unconditional branches, the compiler copies the predicate of the previous block into the next block. Finally, if the Escort compiler comes across a block that already has a predicate, then the compiler sets the block's new predicate to the logical `OR` of the input predicates. At each step, the Escort compiler uses Z3 as a SAT solver to simplify predicates by eliminating unnecessary variables in predicate formulas. Figure 4.6 shows the algorithm for basic block predication.

4.3.2.3 Step 3: Linearizing Basic Blocks

The Escort compiler converts the given code into straight-line code so that every invocation of the code executes the same instructions. To preserve control dependences, the basic blocks are topologically sorted, and then the code is assembled into a single basic block with branch instructions removed.

4.3.2.4 Step 4: Controlling Side Effects

We now explain how Escort prevents side effects from leaking secrets. Here, *side effects* are modifications to the program state or any observable in-

```
1: $p \leftarrow \&a$
2: $secret \leftarrow input()$                          ▷ Assume input() returns **true**.
3: **if** $secret =$ **true then**
4:     ...
5: **else**
6:     ...
7:     $p \leftarrow \&b$    ▷ Instruction does not update pointer $\boldsymbol{p}$, since basic block's
       execution-time predicate is **false**.
8:     $*p \leftarrow 10$                                ▷ Accesses $\boldsymbol{a}$ instead of $\boldsymbol{b}$!
9: **end if**
```

Figure 4.7: The use of pointers can leak information. If `store` instructions are not allowed to access memory when the basic block's predicate is `false`, then pointer $\boldsymbol{p}$ will dereference the address for $\boldsymbol{a}$ instead of $\boldsymbol{b}$, thus revealing that *secret* is `true`.

teraction, including memory accesses, exceptions, function calls, or I/O. Escort controls all side effects except for I/O statements.

Memory Access Side Effects. To ensure proper memory access side effects, the Escort compiler replaces store instructions with conditional data-copy operations that are guarded by the basic block's predicate, so memory is only updated by instructions whose predicate is true.

Unfortunately, this naïve approach can leak secret information when the program uses pointers. Figure 4.7 illustrates the problem: If store instructions are not allowed to update a pointer variable when the basic block predicate is false, then the address trace from subsequent load instructions on the pointer variable will expose the fact that the pointer variable was not updated.

The Escort compiler prevents such information leaks by statically replacing pointer dereferences with loads or stores to each element of the points-to set[5]. Thus Escort replaces the statement in line 8 (Figure 4.7) with a store operation on *b*. When the points-to set is larger than a singleton set, Escort uses the conditional data copy operation on all potential *pointees i.e.* the elements of the points-to set. The predicate of the conditional copy operation checks whether the pointer points to the candidate pointee. If the predicate is false, the pointee's existing value is overwritten, whereas if the predicate is true, the new value is written to the pointee.

Function Call Side Effects. Adversaries can observe the invocation of functions (or lack thereof) using side channels like the Instruction Pointer. Thus, a solution incapable of handling function calls will leak information to the adversary. While inlining functions is a potential solution, inlining is impractical for large applications.

Escort handles side effects from function calls by propagating the predicate from the calling function to the callee. Thus, each user-defined function is given an additional argument that represents the predicate of the call site's basic block. The callee ensures correct handling of side effects by `AND`ing its own predicates with the caller's predicate.

[5]Escort uses a flow-sensitive, context-insensitive pointer analysis: `https://github.com/grievejia/tpa`. Replacing a pointer dereference with a `store` operation on all elements of the points-to set is feasible for Escort because points-to set sizes in the Musl C library are very small.

Side Effects from Exceptions. Program termination caused by exceptions will leak the presence or absence of abnormal operands. To prevent such information leakage, Escort requires that exceptions not occur during program execution[6].

Escort manages floating-point and integer exceptions differently. Escort requires that the programmer disable floating-point exceptions (*e.g.* using `feclearexcept()`). For integer exceptions, Escort borrows ideas from Raccoon by replacing abnormal operands with benign operands (*e.g.* Escort prevents integer division-by-zero by replacing a zero divisor with a non-zero divisor).

4.3.2.5 Step 5: Transforming Array Accesses

Array index values reveal secrets as well. For instance, if the adversary observes that accesses to `array[0]` and `array[secret_index]` result in accesses to locations 10 and 50, then the adversary knows that `secret_index` = 40. To eliminate such information leaks, the Escort compiler transforms each array access into a linear sweep over the entire array, which hides from the adversary the address of the program's actual array index.

Of course, the transformed code is expensive, but this approach is feasible because (1) math library functions typically use only a few small lookup tables, thus requiring relatively few memory accesses and (2) the processor's

[6]Escort assumes that the input program does not throw exceptions, so masking exceptions does not change the semantics of the program.

caches and prefetchers dramatically reduce the cost of sweeping over the arrays.

4.3.2.6 Step 6: Transforming Loops

Some loops introduce timing channels because their trip counts depend on secret values. The Escort compiler transforms such loops using predictive mitigation [117]. The loop body executes as many times as the smallest power of 2 that is greater than or equal to the loop trip count. For instance, if the actual loop trip count is 10, then the loop body is executed 16 times. The basic block predicate ensures that dummy iterations do not cause side effects. With this transformed code, an adversary that observes a loop trip count of l can infer that the actual loop trip count l' is between l and $0.5 \times l$. However, the exact value of l' is not revealed to the adversary.

Unfortunately, this naive approach can still leak information. For instance, if two distinct inputs cause the loop to iterate 10 and 1000 times respectively, the transformed codes will iterate 16 and 1024 times respectively—a large difference that may create timing variations. To mitigate this problem, Escort allows the programmer to manually specify the minimum and maximum loop trip counts using programmer annotations. These annotations override the default settings used by the Escort compiler.

4.3.3 Example Transformation: `exp10f`

We now explain how Escort transforms an example non-secure function

```
float e10(float x) {
  float n, y = mf(x, &n);
  if (int(n) >> 23 & 0xff < 0x82) {
    float p = p10[(int) n + 7];
    if (y == 0.0f) {
      return p;
    }
    return exp2f(3.322f * y) * p;
  }
  return exp2(3.322 * x);
}
```

(a) Original code for `exp10f()`.

```
float e10(float x) {
  float n, y = mf(x, &n);
  if (int(n) >> 23 & 0xff < 0x82) {
    float p = p10[(int) n + 7];
    if (y == 0.0f)
      result = p;
    else
      result =
            exp2f(3.322f * y) * p;
  } else
    result = exp2(3.322 * x);
  return result;
}
```

(b) Result after applying LLVM's `mergereturn` pass. This code becomes the input for the Escort compiler.

```
float e10(float x) {
 return e10_cloned(x, true);
}

float e10_cloned(float x, uint pred) {
  float n, y = mf_cloned(x, &n, pred);
  float p = write(int(n) >> 23 & 0xff
    < 0x82, stream_load(p10, (int) n + 7]));
  bool p2 = y == 0.0f;
  write(pred & p1 & p2, p, &result);
  write(pred & p1 & !p2,
    escort_mul(
     escort_mul(
       exp2f_cloned(3.322f,
         pred & p1 & !p2),
       y),
     p),
   &result);

  write(!p1,
   escort_mul(
     exp2_cloned(3.322, pred & !p1),
     escort_upcast(x))),
   result);

  return result;
}
```

(c) Result of the Escort compiler's transformation.

Figure 4.8: Escort's transformation of `exp10f()`.

Line #	**Predicate**
2, 3, 11	`TRUE`
4, 5	`(n >> 23 & 0xff) < 0x82`
6	`(n >> 23 & 0xff) < 0x82` $\wedge$ `y = 0`
8	`(n >> 23 & 0xff) < 0x82` $\wedge$ `y` $\neq$ `0`
10	$\neg$`((n >> 23 & 0xff) < 0x82)`

Table 4.2: Predicates per line for function in Figure 4.8b.

(Figure 4.8a) into a secure function (Figure 4.8c). To simplify subsequent analyses and transformations, the Escort compiler applies LLVM's `mergereturn` transformation pass, which unifies all exit nodes in the input function (see Figure 4.8b).

First, the Escort compiler replaces elementary floating-point operations in lines 8 and 10 with their secure counterpart function shown in lines 21 and 22 of the transformed code. Second, using the algorithm shown in Figure 4.6, the Escort compiler associates predicates with each basic block, which we list in Table 4.2. Third, the Escort compiler linearizes basic blocks by applying a topological sort on the control flow graph (see Figure 4.9) and fuses the basic blocks together. Finally, the Escort compiler replaces the array access statement in line 4 with a function that sweeps over the entire array. The resulting code, shown in Figure 4.8c, eliminates control flows and data flows that depend on secret values. In addition to closing digital side channels, the code also uses secure floating-point operations.

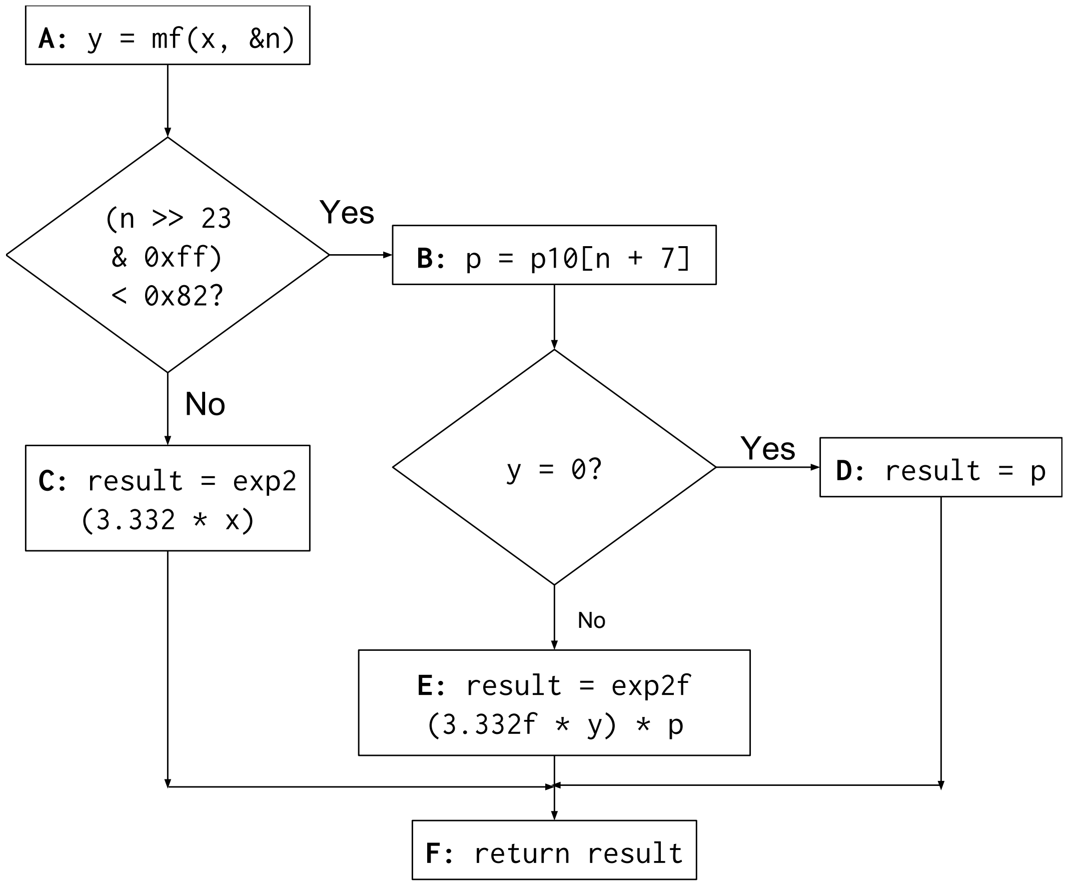


Figure 4.9: Control flow graph with labeled statements for the code in Figure 4.8b. `A`, `B`, `D`, `E`, `C`, `F` is one possible sequence of basic blocks when linearized by the Escort compiler.

4.4 Security Evaluation

This section demonstrates that Escort's floating-point operations run in fixed time and do not leak information through digital side channels. Since precise timing measurement on x86 processors is tricky due to complex processor and OS design, we take special measures to ensure that our measurements are accurate. In addition to Escort's timing and digital side channel defense, we also demonstrate Escort's defense against a floating-point timing channel attack on the Firefox web browser.

4.4.1 Experimental Setup

We run all experiments on a 4-core Intel Core i7-2600 (Sandy Bridge) processor. The processor is clocked at 3.4 GHz. Each core on this processor has a 32 KB private L1 instruction cache, a 32 KB private L1 data cache, and a 256 KB private L2 cache. A single 8 MB L3 cache is shared among all four cores. The host operating system is Ubuntu 14.04 running kernel version 3.13. We implement compiler transformations using the LLVM compiler framework [55] version 3.8.

We measure instruction latencies using the `RDTSC` instruction that returns the number of elapsed cycles since resetting the processor. Since the latency of executing the `RDTSC` instruction is usually higher than the latency of executing operations, our setup measures the latency of executing 1024 consecutive operations and divides the measured latency by 1024. Our setup uses the `CPUID` instruction and `volatile` variables for preventing the processor and the compiler from reordering critical instructions. Finally, our setup measures overhead by executing an empty loop body—a loop body that contains no instructions other than those in the test harness. By placing an empty `volatile __asm__` block in the empty loop body, our setup prevents the compiler from deleting the empty loop body.

4.4.1.1 Outlier Elimination

Many factors outside of the experiment's control, like interrupts, scheduling policies, etc., may result in outliers in performance measurements. We now explain our procedure for eliminating outliers, before demonstrating that the elimination of these outliers does not bias the conclusions.

We use Tukey's method [106] for identifying outliers, but we adapt it to conservatively classify fewer values as outliers (thus including more values as valid data points). The original Tukey's method first finds the minimum (M_n), median (M_d), and maximum (M_x) of a set of values. The first quartile, Q_1, is the median of values between M_n and M_d. The third quartile, Q_3, is the median of values between M_x and M_d. The difference between the first and the third quartiles ($Q_3 - Q_1$) is called the Inter-Quartile Range, R_{IQ}. Tukey's method states that any value v, such that $v > Q_3 + 3 \times R_{IQ}$ or $v < Q_1 - 3 \times R_{IQ}$ is a probable outlier. In our evaluation, we weaken our outlier elimination process (*i.e.* we count fewer values as outliers), by (1) setting the R_{IQ} to be at least equal to 1.0, and (2) classifying v as an outlier when $v > Q_3 + 20 \times R_{IQ}$ or $v < Q_1 - 20 \times R_{IQ}$. Results presented in the following sections use the relaxed

	Mean	**Median**	**Std. Dev.**
Different Operands	847,323 (0.81%)	1,066,270 (1.02%)	381,467
Same Operands	929,703 (0.89%)	1,139,961 (1.09%)	364,192

Table 4.3: Number of discarded outliers from 100 million double-precision square-root operations. The results indicate that our outlier elimination process is statistically independent of the input operand values.

Tukey method described above.

To demonstrate that our outlier elimination process does not bias conclusions, we compare the distribution of outliers between (a) 100 million operations using randomly-generated operands, and (b) 100 million operations using one fixed operand. The two experiments do not differ in any way other than the difference in their input operands. Table 4.3 shows the mean, median, and standard deviation of outliers for the double-precision square-root operation. Results for other floating-point operations are similar and are elided for space reasons. Since the difference in mean values as well as the difference in median values is within a quarter of the standard deviation from the mean, we conclude that the discarded outlier count is statistically independent of the input operand values.

4.4.2 Timing Assurance of Elementary Operations

Since exhaustively testing all possible inputs for each operation is infeasible, we instead take the following three-step approach for demonstrating the timing channel defense for Escort's elementary operations: (1) We charac-

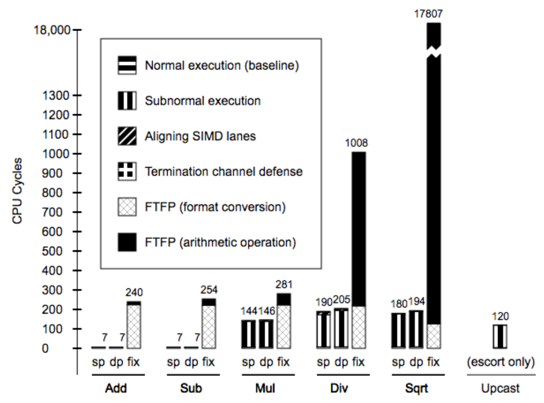


Figure 4.10: Comparison of running times of elementary operations. **sp** identifies Escort's single-precision operations, **dp** identifies Escort's double-precision operations, and **fix** identifies FTFP's fixed-point operations. Numbers at the top of the bars show the total cycle count. We see that Escort's execution times are dominated by the cost of subnormal operations, and we see that FTFP's overheads are significantly greater than Escort's.

terize the performance of Escort's elementary operations using a specific, fixed floating-point value (*e.g.* 1.0), (2) using one value from each of the six different types of values (zero, normal, subnormal, $+\infty$, $-\infty$, and not-a-number), we show that our solution exhibits negligible variance in running time, and (3) to demonstrate that each of the six values in the previous experiment is representative of the class to which it belongs, we generate 10 million normal, subnormal, and not-a-number (NaN) values, and show that the variance in running time among each set of 10 million values is negligible. Our key findings are that Escort's operations run in fixed time, are fast, and that their performance is closely tied to the performance of the hardware's subnormal operations.

Figure 4.10 compares the running times of elementary operations of Es-

Function	**Escort**	**Native (SSE)**
add-sp	0	0
add-dp	0	0
sub-sp	0	0
sub-dp	0	0
mul-sp	0	49.2 (175%)
mul-dp	0	49.2 (175%)
div-sp	0.66 (0.4%)	65.67 (163%)
div-dp	1.66 (0.8%)	69.08 (164%)
sqrt-sp	1.49 (0.8%)	62.7 (170%)
sqrt-dp	2.98 (1.5%)	66.87 (169%)
upcast	0	40.99 (178%)

Table 4.4: Comparison of standard deviation of running times of elementary operations across six types of values (zero, normal, subnormal, $+\infty$, $-\infty$, and not-a-number). Numbers in parenthesis show the standard deviation as a percentage of the mean. The **-sp** suffix identifies single-precision operations while the **-dp** suffix identifies double-precision operations. Compared to SSE operations, Escort exhibits negligible variation in running times.

cort and of previous solutions (FTFP). First, we observe that the running times of Escort's single- and double-precision operations are an order-of-magnitude lower than those of FTFP's fixed-precision operations. Second, Escort's running time is almost entirely dominated by the processor's operation on subnormal numbers. Third, conversion between fixed-point and floating-point takes a non-trivial amount of time, further increasing the overhead of FTFP's operations. Overall, Escort elementary operations are about 16× faster than FTFP's.

Table 4.4 shows the variation in running time of elementary operations across six different types of inputs (zero, normal value, subnormal value, $+\infty$,

Fn.	**NaN**	**Normal**	**Subnormal**
add-sp	0.21 (3.1%)	0.21 (2.9%)	0.19 (2.7%)
add-dp	0.21 (3.0%)	0.20 (2.9%)	0.21 (3.0%)
sub-sp	0.18 (2.6%)	0.19 (2.7%)	0.20 (2.9%)
sub-dp	0.19 (2.7%)	0.19 (2.7%)	0.19 (2.7%)
mul-sp	0.98 (0.7%)	0.94 (0.7%)	1.05 (0.7%)
mul-dp	0.90 (0.6%)	1.04 (0.7%)	1.02 (0.7%)
div-sp	1.22 (0.6%)	1.27 (0.7%)	1.23 (0.6%)
div-dp	1.39 (0.7%)	1.37 (0.6%)	1.17 (0.6%)
sqrt-sp	1.15 (0.6%)	1.13 (0.6%)	1.14 (0.6%)
sqrt-dp	1.29 (0.7%)	1.41 (0.7%)	1.33 (0.7%)
upcast	1.03 (0.9%)	0.89 (0.8%)	0.95 (0.8%)

Table 4.5: Standard deviation of 10 million measurements for each type of value (normal, subnormal, and not-a-number). All standard deviation values are within 3.1% of the mean. Furthermore, the mean of these 10,000,000 measurements is always within 2.7% of the representative measurement.

$-\infty$, and not-a-number value) and compares it with the variation of SSE (native) operations. While SSE operations exhibit high variation (the maximum observed standard deviation is 176% of the mean), Escort's operations show negligible variation across different input types.

Finally, we measure Escort's running time for 10 million random normal, subnormal, and not-a-number values. We observe that the standard deviation of these measurements, shown in Table 4.5, is extremely low (at most 3.1% of the mean). We thus conclude that our chosen values for each of the six classes faithfully represent their class.

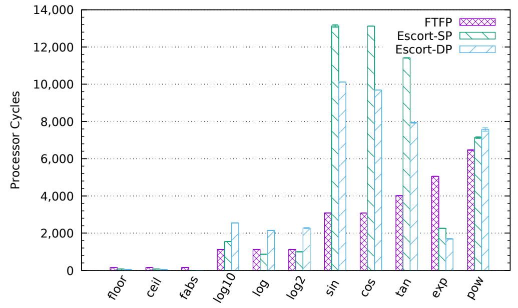


Figure 4.11: Comparison of running times of commonly used higher-level functions. Error bars (visible for only a few functions) show the maximum variation in running time for different kinds of input values.

4.4.3 Timing Assurance of Higher-Level Operations

Using different types of floating-point values (zero, normal, subnormal, $+\infty$, $-\infty$, and not-a-number), Figure 4.11 compares the performance of most of the commonly used single- and double-precision higher-level operations[7]. Overall Escort's higher-level operations are about 2× slower than their corresponding FTFP operation, which is the price for closing side channels that FTFP does not close.

Figure 4.12 shows the breakdown of the performance of commonly used higher-level functions. We observe that the performance of most higher-level functions is dominated by the latency of operations on subnormal operands, which is closely tied to the performance of the underlying hardware. A handful of routines (`exp10()`, `exp10f()`, `exp2()`, and `exp2f()`) use lookup tables that

[7]We exclude the `exp2()` (6,617 cycles), `exp10()` (14,910 cycles), `exp2f()` (1,693 cycles), and `exp10f()` (9,134 cycles) from Figure 4.11 because FTFP does not implement these operations.

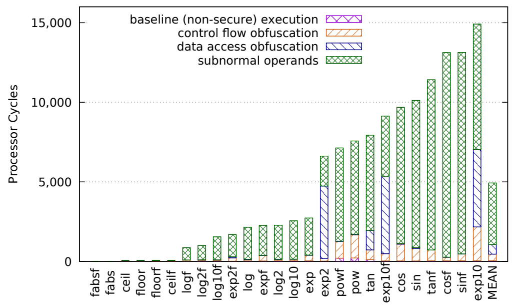
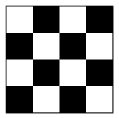
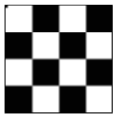


Figure 4.12: Performance breakdown of Escort's commonly used higher-level functions. The baseline (non-secure) execution and exception handling together cost less than 250 cycles for each function, making them too small to be clearly visible in the above plot.


(a) Original image.


(b) Reconstructed image using timing attack.

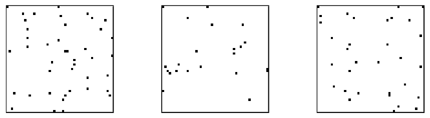

(c) Reconstructed images in 3 independent, consecutive experiments after patching Firefox with Escort.

Figure 4.13: Results of attack and defense on a vulnerable Firefox browser using timing-channel information leaks arising from the use of subnormal floating-point numbers.

are susceptible to address-trace-based side-channel information leaks, so the code transformed by Escort sweeps over these lookup tables for each access to the table. Finally, we see that the cost of control flow obfuscation (*i.e.* the cost of executing all instructions in the program) contributes the least to the total overhead.

4.4.4 Side-Channel Defense in Firefox

We now evaluate Escort's defense against the timing channel attack by Andrysco et al. [7] on the Firefox web browser. The attack reconstructs a two-color image inside a victim web page using only the timing side channel in floating-point operations. The attack convolves the given secret image with a matrix of subnormal values. The convolution step for each pixel is timed using high resolution Javascript timers. By comparing the measured time to a threshold, each pixel is classified as either black or white, effectively reconstructing the secret image.

We integrate Escort into Firefox's convolution code[8] and re-run the timing attack. The results (see Figure 4.13c) show that Escort successfully disables the timing attack.

4.4.5 Control- and Data-Flow Assurance

We now show that Escort's operations do not leak information through control flow or data flow. We first use inference rules over the LLVM IR to demonstrate non-interference between secret inputs and digital side channels. We run a machine-learning attack on Escort and demonstrate that Escort successfully disables the attack.

[8]Specifically, we replace three single-precision multiplication operations with invocations to the equivalent Escort function. All source code changes are limited to the code in the `ConvolvePixel()` function in `SVGFEConvolveMatrixElement.cpp`.

T-PUBLIC-LOAD	$\dfrac{\Gamma(\texttt{ptr}) = \texttt{L} \quad P = \mathit{ptset}(\texttt{ptr}) \quad \texttt{m} = \max_{\texttt{addr} \in P} \Gamma(\texttt{addr}) \quad \Gamma' = \Gamma[\texttt{val} \mapsto \texttt{m}]}{\Gamma \vdash \texttt{val := public-load ptr} : \Gamma'}$
T-PUBLIC-STORE	$\dfrac{\Gamma(\texttt{ptr}) = \texttt{L} \quad \forall\, \texttt{addr} \in \mathit{ptset}(\texttt{p}) \quad \texttt{m} = \mathit{max}(\Gamma(\texttt{val}), \Gamma(\texttt{addr})) \quad \Gamma' = \Gamma[\texttt{addr} \mapsto \texttt{m}]}{\Gamma \vdash \texttt{public-store ptr, val} : \Gamma'}$
T-SECRET-LOAD	$\dfrac{\Gamma' = \Gamma[\texttt{val} \mapsto \texttt{H}]}{\Gamma \vdash \texttt{val := secret-load ptr} : \Gamma'}$
T-SECRET-STORE	$\dfrac{\forall\, \texttt{addr} \in \mathit{ptset}(\texttt{p}) \quad \Gamma' = \Gamma[\texttt{addr} \mapsto \texttt{H}]}{\Gamma \vdash \texttt{secret-store ptr, val} : \Gamma'}$
T-BRANCH	$\dfrac{\Gamma(\texttt{cond}) = \texttt{L}}{\Gamma \vdash \texttt{br cond,block1,block2} : \Gamma}$
T-OTHER	$\dfrac{\Gamma' = \Gamma[\texttt{x} \mapsto \Gamma(\texttt{y})]}{\Gamma \vdash \texttt{x:=y} : \Gamma'}$
T-COMPOSITION	$\dfrac{\Gamma \vdash S_1 : \Gamma', \quad \Gamma' \vdash S_2 : \Gamma''}{\Gamma \vdash S_1; S_2 : \Gamma''}$
T-SANITIZER	$\dfrac{\Gamma' = \Gamma[\texttt{x} \mapsto \texttt{L}]}{\Gamma \vdash S(\texttt{x}) : \Gamma'}$

Table 4.6: Inference rules for verifying the security of Escort's higher-level operations.

4.4.5.1 Non-Interference Using Inference Rules

Since Escort's elementary operations are small and simple—they are implemented using fewer than 15 lines of assembly code, they do not access memory, and they do not contain branch instructions—they are easily verified for non-interference between secret inputs and digital side channels. Using an LLVM pass that applies the inference rules from Table 4.6, tracking labels that can be either `L` (for low-context *i.e.* public information) or `H` (for high-context *i.e.* private information), we verify that Escort's higher-level operations close digital side channels. This compiler pass initializes all function arguments with the label `H`, since arguments represent secret inputs.

Inference rules for various instructions dictate updates to the labels. The environment Γ tracks the label of each pointer and each address. The Escort compiler tags `load` and `store` instructions as secret if the pointer is tainted, or public otherwise. Unlike a public `load` or `store` instruction, a secret `load` or `store` instruction is allowed to use a tainted pointer since Escort generates corresponding loads and stores to *all* statically-determined candidate values in the points-to set. The sanitization rule resets the value's label to `L` and is required to suppress false alarms from Escort's loop condition transformation. Escort's transformed code includes instructions with special LLVM metadata that trigger the sanitization rule.

During verification, the compiler pass iterates over each instruction and checks whether a rule is applicable using the rule's antecedents (the statement above the horizontal line); if so, it updates its local state as per the rule's consequent (the statement below the horizontal line). If no applicable rule is found, then the compiler pass throws an error. The compiler pass processes the code for Escort's 112 higher-level operations without throwing errors.

4.4.5.2 Defense Against Machine-Learning Attack

We use the TensorFlow [1] library to design a machine-learning classifier, which we use to launch a side-channel attack on the execution of the `expf()` function, where the input to the `expf()` function is assumed to be secret. Using three distinct inputs, we run this attack on the implementations in the (non-secure) Musl C library and in the (secure) Escort library. We first use the Pin dynamic binary instrumentation tool [64] to gather the full instruction address traces of both `expf()` implementations[9]. We train the TensorFlow machine-learning classifier by feeding the instruction address traces to the classifier, associating each trace with the secret input to `expf()`. We use cross entropy as the cost function for TensorFlow's training phase. In the subsequent testing phase, we randomly select one of the collected address traces and ask the classifier to predict the secret input value.

We find that for the Musl implementation, the classifier is accurately able to predict the correct secret value from the address trace. On the other hand, for the Escort implementation, the classifier's accuracy drops to 33%, which is no better than randomly guessing one of the three secret input values.

[9]Using the `md5sum` program, we observe that Escort's address traces for all three inputs are identical.

Function	**Min.**	**Median**	**Max.**
add	16	1,743,272	210,125,824
sub	1,312	6,026,976	84,089,503,744
mul	317	8,587,410	112,134,679,849
div	829	5,834,095	30,899,033,427
sqrt	562	2,815,331	21,257,836,468
floor	0	0	0
ceil	0	0	0
log	1,698	5,908,547	2,705,277,8104
log2	262	5,812,840	13,890,632,367
log10	981	10,105,199	40,631,590,323
exp	132	1,409,624	6,066,894
sin	1,316	4,173,786	40,138,955,131
cos	2,166	2,241,360	10,127,702
tan	717	5,576,540	40,126,401,802
pow	522	3,425,870	26,876,068,127
fabs	352	3,129,984	40,134,770,688

Table 4.7: Floating-point difference for 10,000 operations on random inputs in terms of Unit of Least Precision (ULP) in **FTFP versus Musl C library**. Since we observe zero ULP distance between Escort's results and Musl's results, this table omits Escort's results.

4.5 Precision Evaluation

We examine the precision of Escort and FTFP by comparing Escort's and FTFP's results with those produced by a standard C library.

4.5.1 Comparison Using Unit of Least Precision

Methodology. We adopt an empirical approach to estimate precision in terms of Unit of Least Precision (ULP), since formal derivation of maximum ULP difference requires an intricate understanding of theorem provers and floating-point algorithms. We run various floating-point operations on 10,000 randomly generated pairs (using `drand48()`) of floating-point numbers between zero and one. For elementary operations, we compare the outputs of Escort and FTFP with the outputs of native x86 instructions. For all other operations, we compare the outputs of Escort and FTFP with the outputs produced by corresponding function from the Musl C library.

Results. We observe that Escort's results are identical to the results produced by the reference implementations, *i.e.* the native (x86) instructions and the Musl C library. More precisely, the ULP difference between Escort's results and reference implementation's results is zero. On the other hand, FTFP, which computes arithmetic in fixed-point precision, produces output that differs substantially from the output of Musl's double-precision functions (see Table 4.7). The IEEE 754 standard requires that addition, subtraction, multiplication, division, and square root operations are computed with ULP difference of at most 0.5. Well-known libraries compute results for most higher-level operations within 1 ULP.

4.5.2 Comparison of Program Output

Methodology. Since differences in program outputs provide an intuitive understanding of the error introduced by approximate arithmetic operations, we compare the output of the test suite of Minpack[10], a library for solving non-

[10]`https://github.com/devernay/cminpack`

$< 10^{-5}$	10^{-5} to 10^{-3}	10^{-3} to 10^{0}	10^{0} to 10^{3}	$> 10^{3}$
49%	9%	21%	10%	11%

Table 4.8: Distribution of differences in answers produced by Minpack-FTFP and Minpack-C. In all, 321 values differ between the outputs of the two programs.

linear equations and non-linear least squares problems. We generate three variants of Minpack: Minpack-C uses the standard GNU C library, Minpack-Escort uses the Escort library, and Minpack-FTFP uses the FTFP library. We run the 29 programs in Minpack's test suite and compare the outputs produced by the three program variants.

Results. We observe that Minpack-Escort produces output that is identical to Minpack-C's output. We also observe that all outputs of Minpack-FTFP differ from Minpack-C. Specifically, 321 values differ between the outputs of Minpack-FTFP and Minpack-C. We analyze all 321 differences between Minpack-FTFP and Minpack-C by classifying them into the following five categories: (1) smaller than 10^{-5}, (2) between 10^{-5} and 10^{-3}, (3) between 10^{-3} and 10^{0}, (4) between 10^{0} and 10^{3}, and (5) larger than 10^{3}. As seen in Table 4.8, almost half of the differences (49%) are extremely small (less than 10^{-5}), possibly arising from relatively small differences between fixed-point and floating-point calculations. However, we hypothesize that differences amplify from propagation, since nearly 42% of the differences are larger than 10^{-3}.

Application	**Escort Overhead**	**Static (LLVM) Floating-Point Instruction Count**
433.milc	29.33×	2,791
444.namd	57.32×	9,647
447.dealII	20.31×	21,963
450.soplex	4.74×	4,177
453.povray	82.53×	25,671
470.lbm	56.19×	711
480.sphinx3	52.46×	629
MEAN	**32.63×** **(geo. mean)**	**9,370** **(arith. mean)**

Table 4.9: Overhead of SPEC-Escort (SPECfp2006 using Escort operations) relative to SPEC-Libc (SPECfp2006 using libc).

4.6 Performance Evaluation

We now evaluate the end-to-end application performance impact of Escort's floating-point library and Escort's control flow obfuscation.

4.6.1 Impact of Floating-Point Library

This section evaluates the performance impact of Escort on the SPEC floating point benchmarks, as well as on a security-sensitive program SVM^{light}, a machine-learning classifier.

Evaluation Using SPEC Benchmarks. We use the C and C++ floating-point applications in the SPEC CPU 2006 benchmark suite with reference inputs. We generate two versions of each program—the first version (SPEC-

Test Case	**Overhead for Training**	**Overhead for Classification**
#1	8.66×	1.34×
#2	30.24×	0.96×
#3	1.41×	1.11×
#4	12.75×	0.92×
GEO MEAN	**8.28**×	**1.07**×

Table 4.10: Overhead of Escort on SVM^{light} program.

Libc) uses the standard C library functions, and the second version (SPEC-Escort) uses functions from the Escort library[11]. We compile the SPEC-Libc program using the Clang/LLVM 3.8 compiler with the `-O3` flag, and we disable auto-vectorization while compiling the SPEC-Escort program. The following results demonstrate the *worst* case performance overhead of Escort for these programs, since we transform *all* floating-point operations in SPEC-Escort to use the Escort library. More precisely, we do not reduce the number of transformations either using taint tracking or using SMT solvers.

Table 4.9 shows that Escort's overhead is substantial, with a geometric mean of 32.6×. We expect a lower average overhead for applications that use secret data, since taint tracking would reduce the number of floating-point operations that would need to be transformed.

[11]We also ran the same programs using the FTFP library, but the programs either crashed due to errors or ran for longer than two hours, after which they were manually terminated.

Evaluation Using SVM^{light}. To evaluate Escort's overhead on a security-sensitive benchmark, we measure Escort's performance on SVM^{light}, an implementation of Support Vector Machines in C, using the four example test cases documented on the SVM^{light} website[12]. We mark the training data and the classification data as secret. Before replacing floating-point computations, Escort's taint analysis discovers all floating-point computations that depend on the secret data, thus reducing the list of replacements. We also instruct Escort to query the Z3 SMT solver to determine whether candidate floating-point computations could use subnormal operands. Escort then replaces these computations with secure operations from its library. We compile the baseline (non-secure) program using the Clang/LLVM 3.8 compiler with the `-O3` flag, and we disable auto-vectorization while compiling SVM^{light} with Escort. We measure the total execution time using the `RDTSC` instruction. Table 4.10 shows that Escort's overhead on SVM^{light}. We observe that Escort's overhead on SVM^{light} is substantially lower than that on SPEC benchmarks. Using the `md5sum` program, we verify that the output files before and after transformation of SVM^{light} are identical.

4.6.2 Impact of Control Flow Obfuscation

To compare the performance impact of Escort's control flow obfuscation technique with that of Raccoon, we use the same benchmarks that were used to evaluate Raccoon [87], while compiling the baseline (non-transformed)

[12]`http://svmlight.joachims.org/`

Benchmark	**Raccoon Overhead**	**Escort Overhead**
ip-tree	1.01×	2.40×
matrix-mul	1.01×	1.01×
radix-sort	1.01×	1.06×
findmax	1.01×	1.27×
crc32	1.02×	1.00×
genetic-algo	1.03×	1.03×
heap-add	1.03×	1.27×
med-risks	1.76×	1.99×
histogram	1.76×	2.26×
map	2.04×	1.01×
bin-search	11.85×	1.01×
heap-pop	45.40×	1.44×
classifier	53.29×	1.24×
tax	444.36×	1.67×
dijkstra	859.65×	1.10×
GEO MEAN	**5.32**×	**1.32**×

Table 4.11: Performance comparison of benchmarks compiled using Raccoon and Escort. We only compare the control flow obfuscation overhead, since both Raccoon and Escort use the same technique for data access obfuscation.

application with the `-O3` optimization flag. Although both Escort and Raccoon obfuscate control flow *and* data accesses, we compare the cost of control flow obfuscation only, since both Escort and Raccoon obfuscate data accesses using the identical technique. Table 4.11 shows the results.

We find that programs compiled with Escort have a significantly lower overhead than those compiled with Raccoon. Escort's geometric mean overhead is 32%, while that of Raccoon is 5.32×. The worst-case overhead for Escort is 2.4× (for `ip-tree`).

The main reason for the vast difference in overhead is that Raccoon obfuscates branch instructions at *execution* time, which requires the copying and restoring of the stack for each branch instruction. Since the stack can be arbitrarily large, such copying and restoring adds substantial overhead to the running time of the program. On the other hand, Escort's code rewriting technique obfuscates code at *compile* time using basic block predicates, which enables significant performance boosts on the above benchmarks.

4.7 Conclusions

In this work, we have presented Escort, a compiler-based tool that closes side channels that stem from floating-point operations. Escort prevents an attacker from inferring secret floating-point operands through the timing channel, though micro-architectural state, and also through off-chip digital side channels, such as memory address trace.

Escort uses native SSE instructions to provide speed and precision. Escort's compiler-based approach enables it to support a significantly larger number of floating-point operations (112) than FTFP (19).

Escort's design motivates further research into hardware support for side-channel resistant systems. For example, by allowing software to control the timing of integer instruction latencies and their pipelined execution, Escort's guarantees could be extended to instructions beyond floating-point instructions.

Chapter 5

Mitigating Non-Digital Side Channels

In this chapter, we present our research on extending existing power side channel defenses, so that we can protect a broad class of applications running on modern microprocessors.

Mechanisms for closing *digital* side-channels—those that leak discrete bits of information through entities such as caches, the address trace, and the branch predictor—have been well studied for a variety of programs and architectures [7, 15, 27, 52, 61, 65, 73, 87, 88, 97, 111, 119]. However, as Figure 5.1 shows, existing defenses for *analog* side channels—such as power, electromagnetic radiation, and temperature—cannot protect a broad class of programs running on modern microprocessors. In particular, defenses based on blinding or masking [11, 16, 22, 26, 36, 45, 46, 59, 75, 78, 81] rely on mathematical properties of cryptographic computations, so they are not applicable to many important programs, such as databases, social media platforms, and machine-learning programs. Approaches that use custom transistors whose power consumption is independent of the data [28, 84, 86, 103, 104] are prohibitively expensive to scale to an entire modern processor, and they do not protect against power variations that arise due to microarchitectural optimizations, such as

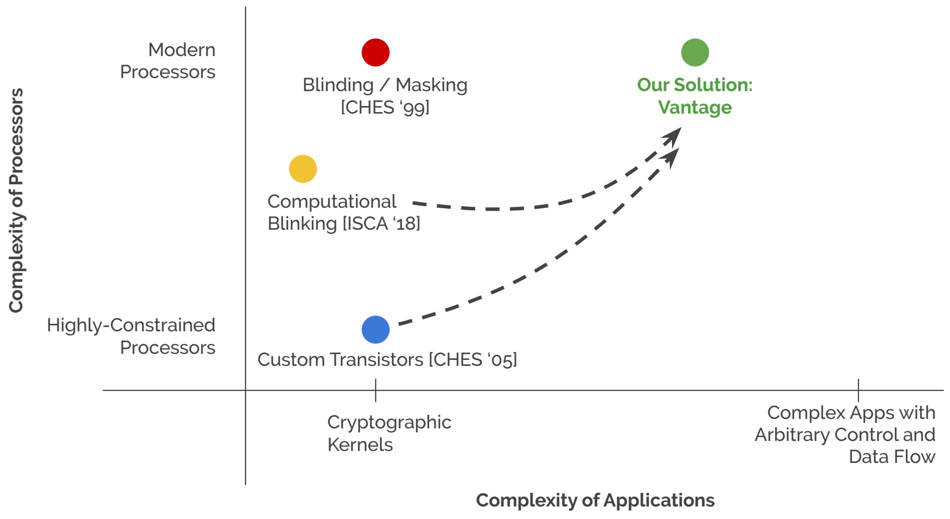

Modern Processors
Blinding / Masking [CHES '99]
Our Solution: Vantage
Computational Blinking [ISCA '18]
Highly-Constrained Processors
Custom Transistors [CHES '05]
Cryptographic Kernels
Complex Apps with Arbitrary Control and Data Flow
Complexity of Processors
Complexity of Applications

Figure 5.1: Relative comparison of power side-channel defenses. Our solution, VANTAGE, enables protection for a broad class of applications running on modern microprocessors, by building on existing techniques like Computational Blinking or Custom Transistors.

out-of-order execution, caches, or branch predictors. Hardware defenses also include computational blinking [5], which hides power consumption for short bursts of time [107, 116], but program behavior can vary so widely that these solutions are incompatible with complex control and data flow.

For several reasons, compilers would seem to be an ideal tool for defending against side-channel attacks. First, compilers can reason about and close whole-program information flow, so compilers can eliminate analog side-channel variations over long sequences of instructions, thereby complementing existing defenses. Second, compilers offer lower performance overheads than a possible hardware solution, because they can selectively apply defenses to just those parts of the program that manipulate sensitive data. Finally, compilers can adapt their transformations based on the target processor and all of their microarchitectural optimizations. Unfortunately, compilers also appear to be fundamentally ill-suited to closing analog side channels, because compilers target the instruction set architecture (ISA), a functional interface that hides from the compiler the very implementation details—the analog channels—that we wish to regulate.

This work explains how compilers can in fact be used to close analog side channels—specifically the power side channel—for a broad class of applications running on x64, ARM 32, and ARM 64 processors. Our key idea is to use a power model[1] to intentionally change a microprocessor's power consumption through program execution. In essence, the power model helps us selectively break the ISA barrier, allowing our compiler—the VANTAGE compiler—to deliberately change the program's power consumption for the sake of closing the power channel over long sequences of instructions.

VANTAGE can be viewed as a system for analyzing and translating programs to a new abstract domain—the power domain—which allows compilers to reason about the power consumption of the program. More broadly, the key ideas behind VANTAGE can be used to build compilers for similar abstract domains for the purpose of closing other side channels.

Our compiler is not tied to any particular power model, so it can be

[1]Power models characterize a microprocessor's power consumption based on microarchitectural events such as instruction count, cache misses, etc.

customized to use more precise and accurate power models as they become available. To illustrate the flexibility of our approach, we evaluate versions of the VANTAGE compiler that use two very different power models—the open-source McPAT [58] model and the closed-source Intel Running Average Power Limit (RAPL) [29] model. We make no claims that McPAT and RAPL are the best possible power models. Indeed, there has been a long history in the development of increasingly-accurate power models [17, 48, 58, 92, 102, 120], and our key contribution is the *approach* for augmenting compilers with power models to close analog side channels.

This work makes the following contributions:

1. **Importance of Closing Power Channels on Modern Processors for a Variety of Programs.** We argue that current techniques are limited because they do not protect complex programs running on existing commercial hardware.

2. **Use of Power Models to Transcend the ISA Barrier.** We observe that because power models link digital events to analog behavior, a compiler can selectively use a power model to peek through the ISA to modulate power consumption, thereby closing power channels over long sequences of instructions.

3. **Compilers for Two Distinct Power Models.** We introduce the VANTAGE compiler that complements existing techniques to close the power channel in a broad variety of applications running on modern

	Label #1 (Malicious Packet)	**Label #2 (Benign Packet)**
Mean	2003.0	1893.1
Stdev	29.8	30.8

Table 5.1: Energy consumption (measured using Intel RAPL) while running the LibSVM classifier [20] that labels data from the KDD Cup dataset [10]. We observe that energy consumption is a reasonable indicator of the label of the input data.

processors. We evaluate Vantage using two power models, one an open-source model (McPAT) that explicitly exposes power leakage and the second, a closed-source model (RAPL) from which we stastically identify power leaks.

4. **Security and Performance Evaluation.** We show that Vantage protects otherwise vulnerable programs. Depending on the underlying power model and the target microprocessor, Vantage imposes a *mean* slowdown from a few percent to about 7×, which is 33× to 137× more efficient than a baseline hardware-only solution.

5.1 Motivation

In this section, we explain the importance of closing the power channel for a broad class of applications running on modern processors.

Power side channels leak secret information through not just arithmetic and logic operations, but also through branches, memory accesses, and through microarchitectural optimizations. Table 5.1 shows, for two secret inputs passed

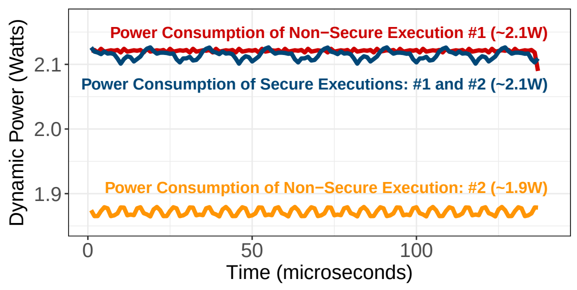


Figure 5.2: Power consumption of the LibSVM classifier before and after using our solution (measured using McPAT). The non-secure executions (shown in red and orange), produce a visually distinct profile of power consumption, whereas after using our solution, for all secrets, the power profiles are identical (shown as a single line in blue), effectively mitigating the power channel attack.

to the LibSVM classifier [20], the total energy consumption measured using Intel RAPL [29]. We see that when running the classifier on a reference network intrusion dataset [10], there exists a correlation between the processor's energy consumption and the classification of network packets as either benign or malicious. Thus, we see that an adversary can infer the packet label by simply observing differences in the processor's energy consumption.

In addition to the total energy consumption, the *profile* of the energy consumption depends on the inputs as well, enabling the adversary to infer the secret through just a partial observation of the program execution. Figure 5.2 shows that the power consumption (measured using the McPAT power modeling framework [58]) during a 140μs time window differs between the two non-secure executions (shown in red and orange), where one execution produces more peaks than the other. Thus, we see that the power channel is dangerous and easy to exploit, in applications that may operate on private or confidential information.

5.2 Open-Source Power Model: McPAT

To be able to close the power channel, we first need to understand how source code can affect the power consumption. In the following sections, we describe our analysis of the McPAT power model, our steps to validate the analysis results, and finally, the code transformations for eliminating power variations.

5.2.1 Analysis of McPAT

To understand the impact of source code on power consumption, we analyze the power model, the microarchitectural simulator whose events drive the power model, and the compiler backend whose instructions trigger events in the microarchitecture. From our analysis of the McPAT power model, we find that it estimates the processor's power consumption based on the processor's physical characteristics (*e.g.* supply voltage), its features (*e.g.* cache sizes), and its microarchitectural events (*e.g.* cache hits and misses). Since the processor's physical characteristics and its features are constant for the program lifetime, we focus our effort on the impact of microarchitectural events on power consumption. We then find the assembly instructions whose operands may cause variations in the microarchitectural events produced by the Gem5 simulator, by analyzing the microcode and exceptions for the x64, ARM 32, and ARM 64 microprocessors and also the common portions of the microarchitectures such as out-of-order execution, caches, and TLBs. Finally, we analyze the translation of LLVM IR instructions into assembly programs to understand the impact of IR instruction operands on power consumption. Although we perform this analysis manually, existing static analysis techniques such as Information Flow Analysis [32] can be used to derive the same results.

5.2.2 Our Findings

Across all three targets (x64, ARM 32 and ARM 64), we find that branch instructions and memory accesses induce variations in power consumption due to their effect on the fetched instructions and the cache usage. In the following paragraphs, we describe computational instructions whose execution induces power variations.

Vulnerable Instructions for x64 Target. We find that 22 instructions (see Table 5.2) on x64 processors execute varying number of microcode operations depending on their operands, thus affecting the power consumption. Among these, 19 instructions are either used only in kernel mode (thus being outside of VANTAGE's threat model) or are rarely used in code generated by modern compilers (*e.g.* string instructions). Hence, we focus our attention on the remaining three instructions (`IDIV`, `BSF`, and `BSR`).

Vulnerable Instructions for ARM 32 Target. We find that predicated instructions on ARM 32 (generated by LLVM for `compare` and `select` IR

Instruction	**Reason for executing variable number of micro operations**
`ENTER`	Value of the nesting depth
`MOV`	If the register operand is a segment selector register
`LODS`, `STOS`, `SCAS`, `CMPS`, `MOVS`, `INS`, `OUTS`	If instructions are prefixed with REP
`IDIV`	Value of dividend (`RAX`)
`JMP`, `RET`	If processor executes far jump or far return
`IRET`	If processor returns to virtual 8086 mode
`INT`, `INT3`	If processor is in long mode
`BSR`, `BSF`	Early termination if input is zero
`CMPXCHG8B`, `CMPXCHG16B`	Early termination if `EAX` or `RAX` does not match with m64 or m128 respectively
`LLDT`	Early termination if invalid operand
`PSRLDQ`, `PSLLDQ`	Value of immediate (constant) operand

Table 5.2: x64 instructions whose operand values trigger variable number of microcode operations.

instructions) can cause variations in power consumption depending on their operands, and that various software runtime library functions (*e.g.* integer division) cause variations in power consumption due to their use of conditional branches and predicated instructions. For processors with *software* floating-point ABI, we find that the elementary floating-point operations (*i.e.* `add`, `sub`, `mul`, `div`, and `sqrt`) and conversion operations between integer and floating-point numbers use conditional branches and predicated instructions, causing the inputs to create variations in power consumption.

5.2.3 Validation of the Analysis Results

We validate the results of the above analysis using randomized testing, where our goal is to check whether there exist other sources of variations in the power consumption, beyond the ones that we discovered. We thus use several different inputs to execute randomly-generated instruction sequences that our previous analysis concluded as safe from power variations, and we observe the power profile of the instruction sequences across the different inputs. If our manual analysis is incorrect (*i.e.* if additional instructions exist whose power consumption varies with the input), we expect to see differences in the power profile when the input values differ.

Validation for x64 Target. We implement this experiment for the x64 platform using the Intel XED (X86 Encoder Decoder) library[2]. From the

[2]`https://intelxed.github.io/`

roughly 400 instructions supported by the Gem5 simulator, our test generator emits 277 instructions (187 SSE instruction and 90 integer instructions). The instructions not emitted by our randomized tests include 71 x87 (coprocessor) instructions, 38 ring 0 (or system management) instructions, 17 branch instructions, and 7 string instructions; we do not include these instructions in our randomized test since they are either removed by our compiler during the code transformation or because these instructions are not used or rarely used in modern user-level code. Our randomized test generates instructions using five addressing modes, whenever supported by the instructions: immediate mode, register mode, indirect mode, base-relative mode, and offset-scaled-base-relative mode. Each instruction sequence is 1,000 instructions long, and in each execution we iterate 10,000 times over the generated instruction sequence. At every 1μs, we measure the power consumption, and we find that regardless of the operand values used in the instruction sequence, all executions produce an *identical* power profile.

Since the Intel XED library does not support encoding instructions for ARM 32 and ARM 64 microprocessors, we are currently in the process of manually encoding instruction mnemonics to port this experiment to the ARM platforms.

5.2.4 Design of the Vantage Compiler

Armed with the list of instruction that may induce power variations, we then devise code transformations of VANTAGE. Fortunately, many trans-

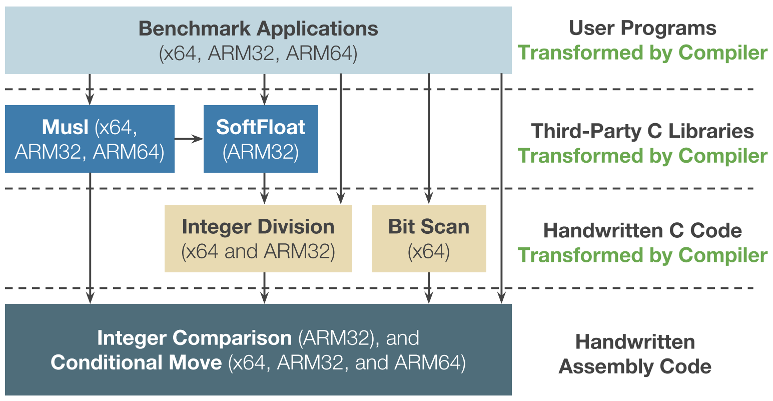


Figure 5.3: Dependences among runtime components for compiling programs using VANTAGE.

formed operations can be generated incrementally using the VANTAGE compiler themselves, thus eliminating the need to manually write all transformed operations in assembly code. As Figure 5.3 shows, we only need a handful of operations in assembly code, while the VANTAGE compiler generates the remaining operations from C code and the handwritten assembly code. Using the VANTAGE compiler, we also transform the Berkeley SoftFloat library[3] for elementary floating-point operations (32- and 64-bit floating-point addition, subtraction, multiplication, division, and square root) and Musl C library[4] for higher-level floating-point operations (sine, cosine, logarithm, and other similar operations). Finally, these transformed components are linked with the transformed version of the benchmarks to produce the executable file.

The VANTAGE compiler first applies inter-procedural, flow-sensitive,

[3]`http://www.jhauser.us/arithmetic/SoftFloat.html`
[4]`https://www.musl-libc.org`

and context-insensitive taint propagation to identify instructions that use sensitive values. VANTAGE relies on the programmer to identify those variables that store sensitive inputs. The VANTAGE compiler then removes or replaces instructions that may operate on sensitive information and which may leak the sensitive information through power consumption. VANTAGE borrows a transformation from the Escort [88] compiler, which closes digital side channels, and we briefly illustrate this transformation below. VANTAGE also includes new transformations that are not present in the other compilers.

Transformation of Tainted `store` Instructions. VANTAGE replaces `store` instructions with predicated write operations, whose predicate controls whether the operation writes new data or pre-existing data. Most importantly, the predicated write operation uses bitwise operations in place of the conditional branch, so that the value of the predicate does not cause variations in the power consumption. Figure 5.4 shows the assembly code for the X64, ARM 32, and ARM 64 targets.

Transformation of Tainted `branch` Instructions. Similar to the Escort compiler, VANTAGE performs standard if-conversion by replacing conditional branches with unconditional branches, while also predicating the instructions along the conditionally-execute path using the predicated write operation described above. VANTAGE also transforms loops by unrolling them a fixed number of times, as specified by the programmer's annotation. VANTAGE

```
pred_write(uint8_t cond, uint32_t __t, uint32_t __f) {
    uint32_t ret, tmp;
    __asm__ volatile (
#if defined(__aarch64__)
        "tst    %w[con], #0xff;"
        "csel   %w[ret], %w[f], %w[t], eq;"
#else
#if defined(__arm__)
        "neg    %[tmp], %[con];"
        "and    %[ret], %[tmp], %[t];"
        "mvn    %[tmp], %[tmp];"
        "and    %[tmp], %[tmp], %[f];"
        "orr    %[ret], %[ret], %[tmp];"
#else
#if defined(__x86_64__)
        "mov    %[t],   %[ret];"
        "test   %[con], %[con];"
        "cmove  %[f],   %[ret];"
#endif
#endif
#endif
        : [ret] "=&r" (ret), [tmp] "=&r" (tmp)
        : [con] "r" (cond), [t] "r" (__t),
          [f] "r" (__f)
        : "cc" );
    return ret;
}
```

Figure 5.4: Conditional move operation for x64, ARM 32, and ARM 64. The code does not leak the secret condition through power consumption.

computes predicates for basic blocks whose execution depends on the tainted branch, by propagating branch predicates along the edges of the CFG, before rewriting only those instructions in the conditional blocks that may cause side effects.

Transformation of Tainted Pointer Dereferences. Since memory accesses affect the usage of the processor cache, which affects the power consumption, VANTAGE replaces sensitive pointer dereferences with accesses that forcibly bypass the cache. We implement such accesses by modifying the Gem5 simulator, and our implementation adds the correct latency from such memory accesses to the program's execution time. Such accesses can also be implemented in software using uncacheable memory.

```c
uint32_t bit_scan_forward(uint32_t input) {
uint8_t n = 1;
if ((input & 0xffff) == 0) {
    n += 16; input >>= 16; }
if ((input & 0x00ff) == 0) {
    n += 8;  input >>= 8;  }
if ((input & 0x000f) == 0) {
    n += 4;  input >>= 4;  }
if ((input & 0x0003) == 0) {
    n += 2;  input >>= 2;  }
if (input == 0) {
    return 0;   }
return n + ((input + 1) & 0x01);
```

Figure 5.5: C Code for bit scan forward operation that is later transformed using the VANTAGE compiler.

Transformation of Other Instructions. We implement software versions of `BSF` and `BSR` (see Figure 5.5) and a binary version of long division `IDIV` (see Figure 5.6), all of which execute using a fixed number of microcode operations, after transformation using VANTAGE. For transforming comparison operations

```
udivrem_32(uint32_t numerator, uint32_t denominator,
    uint32_t* quotient, uint32_t* remainder) {
uint32_t __quo = 0, __rem = 0;
int32_t i;
for (i = sizeof(uint32_t) * 8 - 1; i >= 0; i--) {
    __rem <<= 1;
    uint8_t q_bit = 1,num_i = (numerator >> i) & 1;
    __rem |= num_i;
    if (__rem >= denominator) {
        __rem -= denominator;
    } else {
        q_bit = (__quo >> i) & 1;
    }
    __quo |= (q_bit << i);
}
if (quotient != NULL) {
    *quotient = __quo; }
if (remainder != NULL) {
    *remainder = __rem; }
}
```

Figure 5.6: C Code for unsigned integer division that is later transformed using the VANTAGE compiler. We mark the numerator and the denominator inputs as secret.

on the ARM 32 target, we use the GNU Superoptimizer[5] for discovering alternative instruction sequences that are functionally equivalent to the integer comparison operations, and we port the results of the GNU Superoptimizer from PowerPC to ARM 32 assembly code (see Figures 5.7 and 5.8 for examples). Using the X64 and ARM 32 ISA specifications as reference, we use the CVC4 SMT solver [9] to prove that these transformed operations produce the correct output. For floating-point comparisons, we use transformed code from the Berkeley SoftFloat library. Finally, VANTAGE replaces the `select` IR in-

[5]`https://github.com/embecosm/gnu-superopt`

```
uint8_t cmp_ne(uint32_t x, uint32_t y) {
#if defined(__arm__)
    register uint32_t ret, tmp;
    __asm__ volatile (
        "sub    %[t],  %[x],  %[y];"
        "sub    %[r],  %[y],  %[x];"
        "orr    %[r],  %[r],  %[t];"
        "lsr    %[r],  %[r],  #31;"
        : [r] "=r" (ret), [t] "=&r" (tmp)
        : [x] "r" (x), [y] "r" (y)
        : "cc" );
    return ret & 1;
#else
    return x != y;
#endif
}
}
```

Figure 5.7: Not-equals comparison without causing power variations.

struction with a transformed comparison operation followed by the predicated write operation. For the ARM 32 microprocessor with a *software* floating-point ABI, we use transformed versions of 32- and 64-bit floating-point operations from the Berkeley SoftFloat library. We ensure that the transformed operations do not throw exceptions, since exceptions may reveal the instruction predicate through abnormal termination.

5.3 Closed-Source Power Model: RAPL

We now illustrate the key steps for extending our prior design of the Vantage compiler based on Intel RAPL, which is a closed-source power model. Our high-level approach is to create a regression model between mi-

```
uint8_t cmp_ugt(uint32_t x, uint32_t y) {
#if defined(__arm__)
    register uint32_t ret;
    __asm__ volatile (
        "subs   %[r],  %[y],  %[x];"
        "sbc    %[r],  %[r],  %[r];"
        "neg    %[r],  %[r];"
        : [r] "=r" (ret)
        : [x] "r" (x), [y] "r" (y)
        : "cc" );
    return ret;
#else
    return x > y;
#endif
}
}
```

Figure 5.8: Unsigned greater-than comparison without causing power variations.

croarchitectural events and power consumption[6], and we use the model to determine instructions that may leak information through power consumption.

5.3.1 A Regression Model that Approximates RAPL

We construct a flexible regression model whose precision can be controlled using a tuning parameter. Unlike previous regression models for power consumption [13, 24, 40, 41, 98, 113], the coefficients in our regression model (based on the Elastic Net regression technique [124]) depend on a parameter

[6]Indeed, correlation identified by the regression model does not imply causation. However, constructing our compiler-based defense using correlation (instead of causation) does not alter our solution's security guarantees, although it worsens the performance impact since the compiler is forced to control a larger set of microarchitectural events.

λ, which enables a tradeoff between the precision and the simplicity of the regression model. More precisely, the value of λ affects the number of zero coefficients (*i.e.* coefficients whose value is zero), which indicates that the corresponding microarchitectural events have no impact on the estimated power consumption. So a model with more zero coefficients reduces the number of microarchitectural events that need to be controlled using our compiler, effectively reducing the performance overhead of our defense. At the same time, however, a model with more zero coefficients can also be imprecise, since fewer microarchitectural events are used to estimate power. This tradeoff between precision and the number of zero coefficients is crucial since it allows us to construct a defense that is tailored to the desired threat model and to the performance envelope.

Our regression model for our test platform (an Intel x86 Haswell processor) uses 21 raw performance events (see Table 5.3) that form a superset of the microarchitectural events that can be controlled using the x86 ISA. We use programs from the SPEC CPU 2006 benchmark suite with training inputs, and we run them to completion while measuring performance and power every 100 ms, resulting in $\approx$420,000 measurements. Since our x86 processor supports the measurement of only four simultaneous performance events, we run each SPEC program 21 times, measuring one performance event in each instance of the execution.

Type	**Performance Event**	**Coeff. for λ_{min}**	**Coeff. for λ_{thr}**
Instr-uction	Branch Inst.	1.3×10^{-09}	1.3×10^{-09}
	Mispred. Br.	8.9×10^{-09}	3.0×10^{-09}
	Executed u-ops	4.8×10^{-11}	5.3×10^{-11}
	Issued u-ops	7.5×10^{-11}	1.0×10^{-10}
	CPU Clock	8.3×10^{-11}	2.5×10^{-11}
	Retired Inst.	1.2×10^{-10}	7.0×10^{-11}
Mem. Access	DTLB LD Miss	4.3×10^{-08}	0
	DTLB ST Miss	0	0
	ITLB Miss	1.5×10^{-06}	8.6×10^{-07}
	ICache Read	2.4×10^{-10}	2.0×10^{-10}
	ICache Miss	9.2×10^{-09}	1.1×10^{-08}
	L1 Cache Hit	8.4×10^{-11}	2.8×10^{-11}
	L2 Cache Hit	1.2×10^{-08}	8.0×10^{-09}
	L3 Cache Hit	0	0
	L1 Cache Miss	0	0
	L2 Cache Miss	0	0
	L3 Cache Miss	0	0
Math Op.	Arith. u-ops	3.0×10^{-10}	0
	AVX Inst	0	0
	AVX to SSE Transitions	0	0
	SSE to AVX Transitions	0	0
—	Intercept (Avg Energy)	1.1	1.2

Table 5.3: The 21 chosen performance events for computing the regression between microarchitectural events and energy consumption and their corresponding coefficients (after rounding to one decimal).

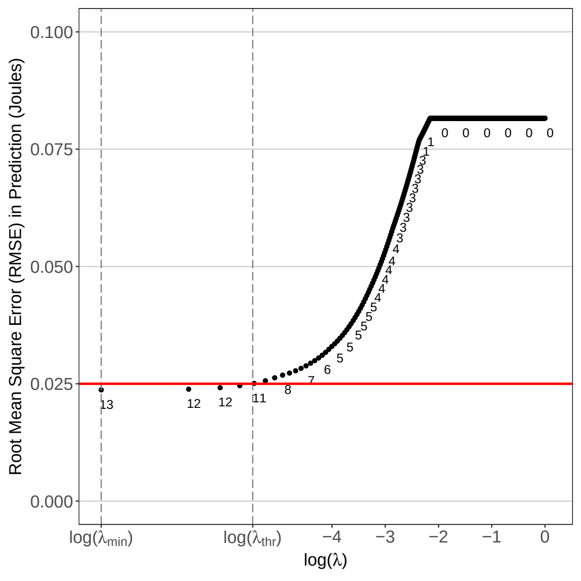


Figure 5.9: The prediction error increases as λ increases beyond λ_{min}, which produces the smallest prediction error. We are interested in the value of λ for which the prediction error is close to the smallest prediction error. Digits on the curve indicate the number of non-zero coefficients for the corresponding values of λ.

Tradeoff Between Simplicity and Precision. We select the value of λ, whose value affects the number of non-zero coefficients, after measuring its impact on the prediction error, which is strictly convex, thus permitting an iterative approach to discover the value of λ that results in the smallest prediction error. Figure 5.9 shows the prediction error after using cross-validation on the SPEC CPU 2006 measurements. The digits on the curve indicate the number of non-zero coefficients. We observe that when $\lambda = \lambda_{min} = 0.00013$, the Root Mean Square (RMS) prediction error is close to 25 mJ, and that the cross validation error continues to be under the 25 mJ threshold until $\lambda = \lambda_{thr} = 0.00413$.

5.3.2 Validation of the Analysis Results

We validate the accuracy of the regression coefficients for both $\lambda = \lambda_{min}$ and $\lambda = \lambda_{thr}$ regression models by predicting the energy consumption of the processor while it executes the PARSEC benchmark applications [12]. Table 5.4 shows the prediction accuracy, computed using the RMS error for every 100 ms of program execution. The prediction accuracy of our regression models is higher than 96%, so we believe that our models are sufficiently accurate.

5.3.3 Design of the Vantage Compiler for Intel RAPL

By leveraging the flexibility of our regression model, we create a power channel defense for $\lambda = \lambda_{thr}$, and we call the corresponding solution as VAN-

Benchmark	**Accuracy of Model when** $\lambda = \lambda_{min}$	**Accuracy of Model when** $\lambda = \lambda_{thr}$
blackscholes	98.96 %	97.19 %
bodytrack	94.92 %	95.28 %
canneal	96.72 %	96.15 %
facesim	97.72 %	98.21 %
ferret	98.06 %	95.52 %
fluidanimate	97.03 %	97.30 %
freqmine	94.61 %	95.85 %
raytrace	96.23 %	97.14 %
streamcluster	96.46 %	95.74 %
vips	98.84 %	96.72 %
Geo. Mean	**96.75 %**	**96.51 %**

Table 5.4: Accuracy of the new regression models based on 100 ms measurements of a subset of the PARSEC benchmarks using Intel RAPL. The remaining benchmarks failed to either compile or run on our platform.

tage-RAPL. The Vantage-RAPL compiler extends the Vantage compiler, by ignoring the transformation of TLB events, since the data TLB coefficients are equal to zero. Specifically, Vantage-RAPL forces cache misses on every access to a secret address using the `clflush` instruction. Such accesses can also be implemented using uncacheable memory.

5.4 Evaluation

We now evaluate the performance of our code transformations using microbenchmarks (which test integer division, bit scan operations, and integer and floating-point comparisons), and we also evaluate the security and performance of our full benchmarks.

5.4.1 Experimental Setup

Vantage uses the LLVM compiler [55] version 7.0 for transforming programs. For Intel RAPL measurements, we gather performance and energy measurements every 100 ms on an 8-core Intel Haswell processor clocked at 3.4 GHz. The processor contains 32 KB private L1 instruction and data caches, 256 KB private L2 unified caches, and a shared 8 MB unified L3 cache. The processor runs Ubuntu 16.04 with Linux kernel version 4.4.0. For measurements based on McPAT, we gather performance and power measurements every $1\mu s$ using the Gem5 microarchitectural simulator that models 1 GHz out-of-order x64, ARM 32, and ARM 64 processors with 32 KB L1 instruction and data caches, a 256 KB L2 cache, and an 8 MB L3 cache. As a reference for performance comparison of Vantage, we use a hypothetical hardware-only defense that does not use contextual information from a compiler like Vantage, forcing the processor to consume worst-case power and execution time for every operation. Our hardware-only defense consumes 1 cycle for arithmetic operations, 2 cycles for branches, 3 cycles for `CALL` instructions, and 400 cycles for memory references[7].

5.4.2 Performance of Microbenchmarks

Figures 5.10 and 5.11 evaluate the performance impact of Vantage's microprocessor-specific code transformations. We observe that the slowdowns are substantial, especially for the bitscan operations, but these instructions do

[7] `https://www.agner.org/optimize/instruction_tables.pdf`

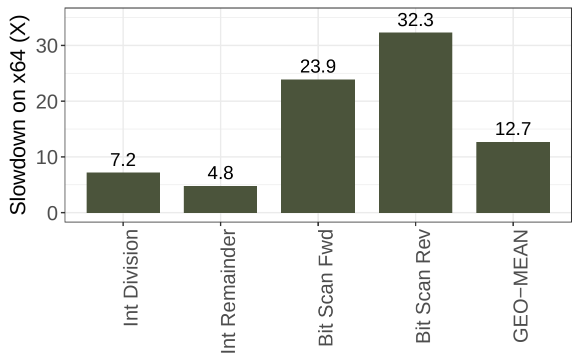


Figure 5.10: Performance overhead of transformed 32-bit division, remainder, and bit scan code on the x64 target.

not dominate the dynamic instruction count in the benchmark applications, so their impact on the performance of the full applications is relatively low.

5.4.3 Benchmark Applications

We now evaluate the security and performance of VANTAGE on x64, ARM 32, and ARM 64 targets with McPAT-based and RAPL-based measurements using 12 benchmarks. Since there are no standardized benchmarks for evaluating side channel defenses, we use commonly used programs whose inputs represent private or confidential information. These benchmarks represent applications from four diverse categories: (1) general user applications (comprising of a Font Renderer[8], a Hash Table implementation[9], and a Bloom Filter implementation[10]), (2) machine-learning kernels (comprising of Dis-

[8]`https://github.com/nothings/stb/blob/master/stb_easy_font.h`
[9]`https://github.com/watmough/jwHash`
[10]`https://github.com/bitly/dablooms`

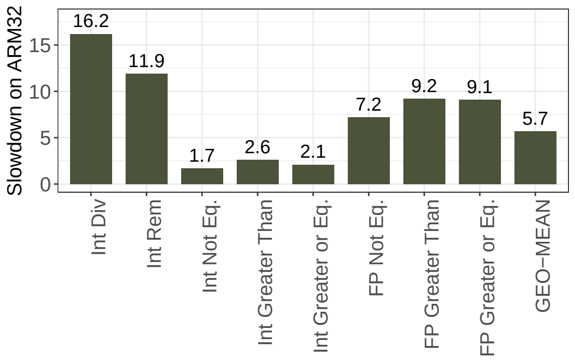


Figure 5.11: Performance overhead of transformed 32-bit division, remainder, and comparison on ARM 32 target.

parity Map computer vision benchmark [109], the LibSVM Support Vector Machine Classifier[11], and an implementation of the K-Means clustering algorithm[12]), (3) graph kernels (which includes an implementation of Top-k Search, the Bellman-Ford shortest path algorithm, and the Pagerank algorithm), and (4) cryptographic kernels (the Microsoft Lattice Cryptography Library[13], a Curve25519 elliptic curve implementation[14], and a Poly1305 message authentication code implementation[15]). Nine of the total 12 benchmarks are written by third party developers. For each application, we mark its inputs as secret, and we use at least three distinct inputs for each application. We use smaller data sizes for McPAT-based measurements since McPAT-based sim-

[11] `https://github.com/cjlin1/libsvm`
[12] `https://wikicoding.org/wiki/c/k-means_clustering_algorithm/`
[13] `https://www.microsoft.com/en-us/research/project/lattice-cryptography-library`
[14] `https://github.com/agl/curve25519-donna`
[15] `https://github.com/floodyberry/poly1305-donna`

ulations run many times slower than RAPL-based executions. Our compiler detects vulnerabilities in, and accordingly transforms, all applications except the cryptographic kernels, since, as per our power models, the cryptographic kernels do not include instructions that leak information.

We emphasize that the evaluation results are closely tied to each power model, so results from one power model are not directly comparable with results from the other power model.

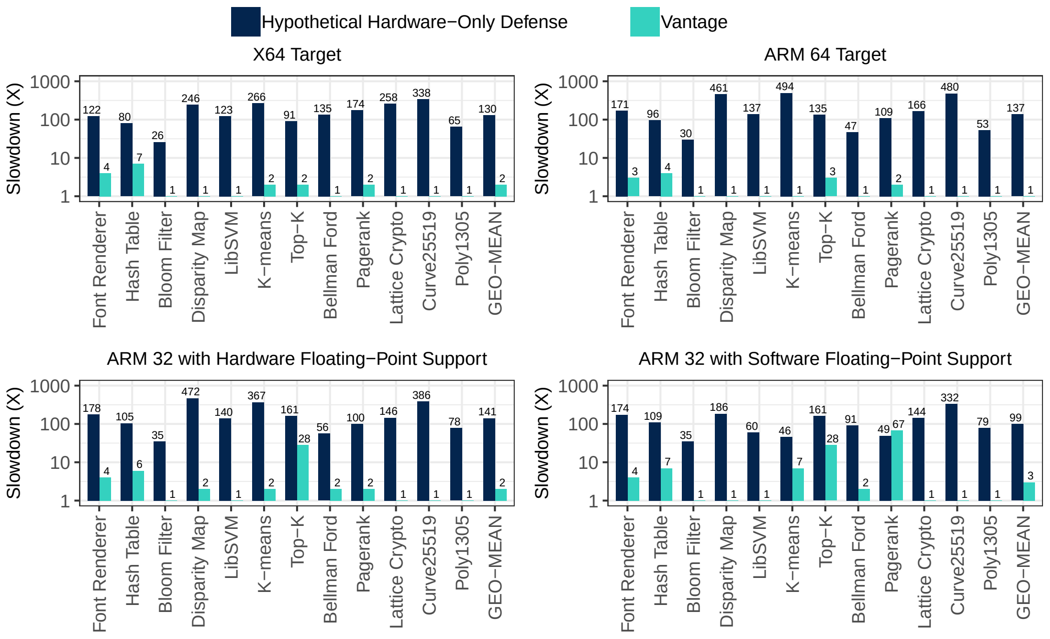


Figure 5.12: Performance overhead of Vantage on x64, ARM 32, and ARM 64 targets.

5.4.3.1 Security Evaluation

Methodology for McPAT-Based Measurements. Our McPAT-based measurements are obtained using the Gem5 simulator, enabling precisely reproducible results on every execution. To determine whether the power trace of the transformed programs is independent of the secrets, we compute the SHA1 checksum for the power trace obtained using Gem5 and McPAT, and we check whether the checksums are exactly identical even when the secret inputs differ. If identical, we conclude that the power trace is independent of the secrets.

Results for McPAT-Based Measurements. For all but the cryptographic benchmarks, we observe that the non-secure execution produces a different SHA1 checksum (elided for space) for each secret input, whereas programs transformed using VANTAGE produce *exactly identical* SHA1 checksums of the power traces regardless of the programs' secret inputs. Indeed, as per our power model analysis, the cryptographic kernels do not use vulnerable instructions that leak secrets.

Methodology for RAPL Measurements. We collect 50 power profiles for every combination of the application and the secret input, and we feed these profiles to an Extreme Gradient Boosting classifier, which we implement using the `xgboost` [21] R package. We first randomly shuffle the power profiles, before using one-third of the profiles for training the classifier. We measure the

Benchmark Application	**Non-Secure Execution AUC**	**Vantage Execution AUC**
Hash Table	**0.91 +/- 0.02**	0.50 +/- 0.03
Disparity Map	**0.85 +/- 0.03**	0.49 +/- 0.04
LibSVM Classifier	**0.93 +/- 0.02**	0.48 +/- 0.04
Top-k Search	**0.94 +/- 0.02**	0.45 +/- 0.05
Page Rank	**0.92 +/- 0.02**	0.50 +/- 0.05
Bellman Ford	**0.99 +/- 0.01**	0.51 +/- 0.05
Font Renderer	0.53 +/- 0.05	0.46 +/- 0.05
Bloom Filter	0.46 +/- 0.04	0.49 +/- 0.04
K-Means Clustering	0.55 +/- 0.05	0.49 +/- 0.05
Lattice Cryto Key Exch.	0.45 +/- 0.05	0.51 +/- 0.05
Curve25519 ECC	0.51 +/- 0.05	0.42 +/- 0.05
Poly1305 MAC	0.47 +/- 0.05	0.44 +/- 0.05

Table 5.5: Mean Area Under the Curve (AUC) and standard deviation for ROC curves corresponding to non-secure and secure (Vantage) execution over 500 summaries. We observe that six benchmark applications are vulnerable to power channel attacks, and Vantage thwarts the attack in the transformed (Vantage) execution.

accuracy of the classification on the remaining two-thirds of the profiles using the Area Under the Curve (AUC) metric of the Receiver Operating Characteristic (ROC) Curve. The random shuffling step perturbs the classification accuracy on each execution, so we perform the classification 500 times and if the mean AUC is close to 0.5, then we conclude that the adversary is unsuccessful at launching a power channel attack. Consequently, if the mean AUC for programs transformed using Vantage drops close to 0.5, then we conclude that Vantage successfully defeats the power channel attack.

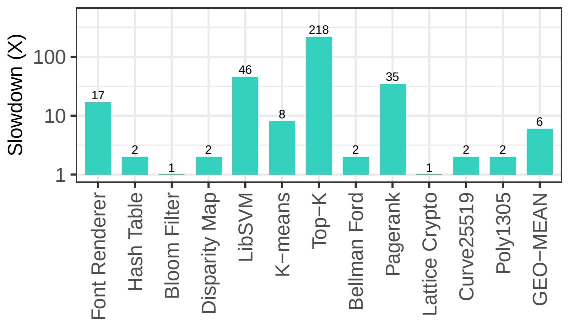


Figure 5.13: Performance overhead of Vantage-RAPL.

Results for RAPL Measurements. Table 5.5 shows the Area Under the Curve (AUC) metric for ROC curves for the original and transformed programs using Intel RAPL measurements. We find that 6 of the 12 benchmarks (shown in the top half of the table) are vulnerable to power channel attacks. In particular, we observe that benchmarks whose dynamic instruction count depends on the secrets are more susceptible to power channel attacks using RAPL, while benchmarks whose memory address trace depends on the secrets are harder to attack. We observe that programs transformed using Vantage-RAPL thwart the power channel attack.

5.4.3.2 Performance Evaluation

Results for McPAT Measurements. Figure 5.12 compares the performance overhead of programs running on the hardware-only defense versus that of programs transformed by Vantage. Since the hardware-only defense lacks contextual information about the program, it needs to treat every operation as secret, thus forcing the worst-case execution time for every operation. In contrast, programs transformed using the VANTAGE compiler can transform only those sections of the code that may leak secrets, thus executing programs two to three orders of magnitude faster than the hardware-only defense. Among programs transformed by VANTAGE, we observe that benchmarks that access memory using secret pointers (Font Renderer and Top-k) incur high overhead. We also find that on the ARM 32 target with software floating-point ABI, benchmarks that use floating-point arithmetic (LibSVM, K-Means, and Pagerank) experience substantial overheads due to their use of software floating-point arithmetic. The Top-k application uses many comparisons (again, implemented in VANTAGE in software), which results in higher overhead on the ARM 32 targets compared to the ARM 64 target. Across all analyzed targets, we find that the *mean* overhead from using VANTAGE is at most $3\times$, while that from using a hardware-only defense ranges between $99\times$ and $141\times$.

Results for RAPL Measurements. Figure 5.13 shows the performance overhead of programs transformed by VANTAGE-RAPL, where we observe that benchmarks which access memory using secret pointers (Font Renderer and Top-k) or which perform floating-point arithmetic on secret values (LibSVM, K-Means, and Pagerank) experience the most slowdowns. These results validate our understanding that VANTAGE's transformation of control flow is substantially cheaper than its transformation of memory references and floating-point computation, since it is more expensive to force cache misses or to perform dummy subnormal floating-point computation compared to executing dummy instructions. On average, we see a 6× slowdown from the use of Vantage-RAPL.

5.5 Discussion

Performance Overhead of Vantage. The performance overhead of Vantage stems from its strong security property of making the running time of the application independent of the secrets, so as to not leak information through the total energy consumption. Like Vantage, any solution that enforces a fixed energy consumption will need to enforce a worst-case execution time. However, we believe that Vantage's performance overhead can be reduced using aggressive compiler optimizations combined with modest microarchitectural changes.

Accuracy of Power Models. Vantage's defense relies crucially on the accuracy of the power model. However, many power models exist whose predictions are close to the actual power consumption [17, 48, 58, 92, 102, 120]. Vantage is not tied to any specific power models; instead Vantage can be adapted based on the available power model.

Other Physical Side Channels. Beyond power, there exist systems that model other aspects of the program execution such as heat [38, 39] and electromagnetic radiation [57, 122], so our approach could be useful for mitigating other analog side channel attacks besides power channel attacks.

5.6 Conclusion and Future Work

Until now, power channel defenses have protected a small minority of programs. This work shows how compiler-based techniques can be used to close power side channels in a more diverse class of applications running on modern processors. The key observation is that to bridge the gap between the program execution and the power consumption, we need a mapping from software events to power consumption, which can be provided by existing power models.

By mapping software events to power consumption, VANTAGE can eliminate variations in all software events—and only those software events—that affect power consumption. At the same time, by reasoning about power consumption at the program level, VANTAGE can selectively apply mitigation techniques only where needed.

Looking to the future, we can combine our compiler-based code transformations with existing cycle-level power side-channel defenses to provide a comprehensive power channel defense. We also plan to improve the performance impact of our solutions through microarchitectural enhancements.

Chapter 6

Conclusion and Future Work

In this dissertation, we presented compiler-based solutions for closing or mitigating digital as well as non-digital side channels. Our solutions require programmer annotations that identify the input variables that hold sensitive information, and the compiler tracks the flow of sensitive information through various instructions, so that it transforms only those parts of the program that operate on sensitive values. Our solutions also adapt the code transformations based on the threat model as well as the microarchitecture.

Our solutions use the key insight that a broad class of side channels arise due to variations in the application's source-level behavior, which can be summarized in terms of control flows and data flows. Consequently, by making the application's control flows and data flows independent of the program's sensitive information, our solutions close a broad class of side channels. But side channels can also exist in the implementation of individual assembly instructions, such as the integer division instruction, which can leak information about the operands through the running time, architectural exceptions, and through power consumption. For closing such side channels, our solutions rewrite the operation without using the vulnerable assembly instruction.

For instance, our Vantage solution rewrites the division instruction using only bitwise arithmetic operations. We derive the rewritten operations either manually or by using results from an existing superoptimizer.

6.1 Future Work

We now list potential topics for extending this work.

Synthesizing Side-Channel Defenses. Based on our generalized approach for closing side channels through abstractions from program instructions to side-channel leakage, it could be possible to synthesize compiler transformations automatically based on microarchitectural models of information leakage. Synthesis techniques can also be useful for proving stronger guarantees through a rich domain-specific language that is amenable to aggressive static analysis, such as determining whether the generated code transformations are composable with other compiler transformations or whether the generated code transformations preseve program correctness.

Extending the ISA to Include Behavioral Constraints. Currently, the only way for a compiler to affect the microarchitecture's behavior is through the ISA, which is a *functional* interface and which does not include a specification of *behavioral* properties such as power consumption or timing. Consequently, software solutions are often forced to craft clever but fragile techniques for influencing microarchitectural behavior, often resulting in large performance overheads. By extending the ISA to include not just functional but also a behavioral specification, software can directly control the microarchitecture's behavioral properties, thus likely improving performance.

Creating a Standardized Benchmark for Evaluating Defenses. Finally, various side channel defenses exist, but it is difficult to compare these defenses against each other because of the lack of a set of programs that can be used as a standardized benchmark. By designing a reference benchmark suite, we can enable a direct comparison of not just the performance impact of various solutions but also their purported security on different programs running on various platforms.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–283, 2016.

[2] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the Power of Simple Branch Prediction Analysis. In *Information, Computer and Communications Security (ICCS)*, pages 312–320, 2007.

[3] Onur Aciiçmez and Jean-Pierre Seifert. Cheap Hardware Parallelism Implies Cheap Security. In *Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 80–91, 2007.

[4] Shaizeen Aga and Satish Narayanasamy. InvisiMem: Smart Memory Defenses for Memory Bus Side Channel. In *International Symposium on Computer Architecture (ISCA)*, pages 94–106, 2017.

[5] Alric Althoff, Joseph McMahan, Luis Vega, Scott Davidson, Timothy

Sherwood, Michael Taylor, and Ryan Kastner. Hiding Intermittent Information Leakage with Architectural Support for Blinking. In *International Symposium on Computer Architecture (ISCA)*, pages 638–649, 2018.

[6] Jude Angelo Ambrose, Roshan G. Ragel, and Sri Parameswaran. RIJID: Random Code Injection to Mask Power Analysis Based Side Channel Attacks. In *Design Automation Conference (DAC)*, pages 489–492, 2007.

[7] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On Subnormal Floating Point and Abnormal Timing. In *Symposium on Security and Privacy (Oakland)*, pages 623–639, 2015.

[8] Amro Awad, Yipeng Wang, Deborah Shands, and Yan Solihin. ObfusMem: A Low-Overhead Access Obfuscation for Trusted Memories. In *International Symposium on Computer Architecture (ISCA)*, pages 107–119, 2017.

[9] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer Aided Verification (CAV)*, pages 171–177, July 2011.

[10] Stephen D. Bay, Dennis Kibler, Michael J. Pazzani, and Padhraic Smyth. The UCI KDD Archive of Large Data Sets for Data Mining Research

and Experimentation. *SIGKDD Explorations Newsletter*, 2(2):81–85, December 2000.

[11] A. G. Bayrak, F. Regazzoni, D. Novo, P. Brisk, F. X. Standaert, and P. Ienne. Automatic Application of Power Analysis Countermeasures. *IEEE Transactions on Computers*, 64(2):329–341, 2015.

[12] Christian Bienia. *Benchmarking Modern Multiprocessors.* PhD thesis, Princeton University, January 2011.

[13] William Lloyd Bircher and Lizy K. John. Complete System Power Estimation Using Processor Performance Events. *IEEE Transactions on Computers*, 61(4):563–577, April 2012.

[14] Colin Blundell, E Christopher Lewis, and Milo Martin. Unrestricted transactional memory: Supporting I/O and system calls within transactions. Technical report, University of Pennsylvania, 2006.

[15] 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 *USENIX Security Symposium (SEC)*, pages 917–934, 2017.

[16] Eric Brier and Marc Joye. Weierstrass Elliptic Curves and Side-Channel Attacks. In *Public Key Cryptography (PKC)*, pages 335–345, 2002.

[17] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations.

In *International Symposium on Computer Architecture (ISCA)*, pages 83–94, 2000.

[18] David Brumley and Dan Boneh. Remote timing attacks are practical. In *USENIX Security Symposium*, 2005.

[19] Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The Atomos Transactional Programming Language. In *Programming Language Design and Implementation (PLDI)*, pages 1–13, 2006.

[20] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A Library for Support Vector Machines. *Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011.

[21] Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In *Knowledge Discovery and Data Mining (KDD)*, pages 785–794, 2016.

[22] Zhimin Chen and Patrick Schaumont. Virtual Secure Circuit: Porting Dual-Rail Pre-Charge Technique into Software on Multicore. *IACR Cryptology ePrint Archive*, page 272, 2010.

[23] Jeroen V. Cleemput, Bart Coppens, and Bjorn De Sutter. Compiler Mitigations for Time Attacks on Modern x86 Processors. *Transactions on Architecture and Code Optimization*, 8(4):23:1–23:20, January 2012.

[24] Gilberto Contreras and Margaret Martonosi. Power Prediction for Intel XScale Processors Using Performance Monitoring Unit Events. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 221–226, 2005.

[25] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *IEEE Symposium on Security and Privacy (Oakland)*, pages 45–60, 2009.

[26] Jean-Sébastien Coron. Resistance Against Differential Power Analysis for Elliptic Curve Cryptosystems. In *Cryptographic Hardware and Embedded Systems (CHES)*, pages 292–302, 1999.

[27] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In *Network and Distributed System Security Symposium (NDSS)*, 2015.

[28] Jean-Luc Danger, Sylvain Guilley, Shivam Bhasin, and Maxime Nassar. Overview of Dual Rail with Precharge Logic Styles to Thwart Implementation-Level Attacks on Hardware Cryptoprocessors. In *Signals, Circuits and Systems (SCS)*, pages 1–8, 2009.

[29] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. RAPL: Memory Power Estimation and Capping. In *Inter-*

national Symposium on Low Power Electronics and Design (ISLPED), pages 189–194, 2010.

[30] 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, 2008.

[31] James W. Demmel. Effects of Underflow on Solving Linear Systems. Technical Report UCB/CSD-83-128, EECS Department, University of California, Berkeley, Aug 1983.

[32] Dorothy E Denning and Peter J Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, 1977.

[33] C. W. Fletcher, Ren Ling, Yu Xiangyao, M. van Dijk, O. Khan, and S. Devadas. Suppressing the Oblivious RAM Timing Channel While Making Information Leakage and Program Efficiency Trade-Offs. In *High Performance Computer Architecture (HPCA)*, pages 213–224, 2014.

[34] Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. A Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *ACM Workshop on Scalable Trusted Computing*, pages 3–8, 2012.

[35] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic Analysis: Concrete Results. In *Cryptographic Hardware and*

Embedded Systems (CHES), pages 251–261, 2001.

[36] Louis Goubin and Jacques Patarin. DES and Differential Power Analysis (The "Duplication" Method). In *Cryptographic Hardware and Embedded Systems (CHES)*, pages 158–172, 1999.

[37] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. Side-Channel Analysis of Cryptographic Software via Early-Terminating Multiplications. In *Information Security and Cryptology (ISC)*, pages 176–192, 2010.

[38] Taliver Heath, Ana Paula Centeno, Pradeep George, Luiz Ramos, Yogesh Jaluria, and Ricardo Bianchini. Mercury and Freon: Temperature Emulation and Management for Server Systems. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 106–116, 2006.

[39] Wei Huang, Shougata Ghosh, Siva Velusamy, Karthik Sankaranarayanan, Kevin Skadron, and Mircea Stan. HotSpot: A Compact Thermal Modeling Methodology for Early-Stage VLSI Design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(5):501–513, 2006.

[40] Canturk Isci and Margaret Martonosi. Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data. In *International Symposium on Microarchitecture (MICRO)*, pages 93–104, 2003.

[41] Canturk Isci and Margaret Martonosi. Phase Characterization for Power: Evaluating Control-Flow-Based and Event-Counter-Based Techniques. In *High-Performance Computer Architecture (HPCA)*, pages 121–132, Feb 2006.

[42] Yasuo Ishii, Mary Inaba, and Kei Hiraki. Access Map Pattern Matching for High Performance Data Cache Prefetch. In *International Conference on Supercomputing (ICS)*, pages 499–500, 2009.

[43] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *Network and Distributed System Security Symposium, NDSS*, 2012.

[44] Suman Jana and Vitaly Shmatikov. Memento: Learning Secrets from Process Footprints. In *IEEE Symposium on Security and Privacy (Oakland)*, pages 142–157, 2012.

[45] Marc Joye and Jean-Jacques Quisquater. Hessian Elliptic Curves and Side-Channel Attacks. In *Cryptographic Hardware and Embedded Systems (CHES)*, pages 402–410, 2001.

[46] Marc Joye and Christophe Tymen. Protections Against Differential Analysis for Elliptic Curve Cryptography. In *Cryptographic Hardware and Embedded Systems (CHES)*, pages 377–390, 2001.

[47] W Kahan. Interval Arithmetic Options in the Proposed IEEE Floating-Point Arithmetic Standard. *Interval Mathematics*, pages 99–128, 1980.

[48] Andrew B. Kahng, Bin Li, Li-Shiuan Peh, and Kambiz Samadi. ORION 2.0: A Fast and Accurate NoC Power and Area Model for Early-Stage Design Space Exploration. In *Design, Automation and Test in Europe (DATE)*, pages 423–428, 2009.

[49] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTH-MEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud. In *USENIX Security Symposium (SEC)*, pages 189–204, 2012.

[50] Paul C Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology (CRYPTO)*, pages 104–113, 1996.

[51] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *19th Annual International Cryptology Conference on Advances in Cryptology*, pages 388–397, 1999.

[52] Jingfei Kong, Onur Aciiçmez, Jean-Pierre Seifert, and Huiyang Zhou. Hardware-Software Integrated Approaches to Defend Against Software Cache-Based Side Channel Attacks. In *High Performance Computer Architecture (HPCA)*, pages 393–404, 2009.

[53] Markus Kuhn. Cipher Instruction Search Attack on the Bus-Encryption Security Microcontroller DS5002FP. *IEEE Transactions on Computers*, 47(10):1153–1157, 1998.

[54] Butler Lampson. A Note on the Confinement Problem. *Communications of the ACM*, pages 613–615, 1973.

[55] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Code Generation and Optimization (CGO)*, pages 75–86, 2004.

[56] 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 *USENIX Security Symposium (SEC)*, pages 557–574, 2017.

[57] Bing Li, Mingzhu Lei, Meiyuan Chen, and Lanyong Zhang. Electro-Magnetic Analysis of High-Frequency Digital Signal Processors. *Springer-Plus*, 5(1):1313, 2016.

[58] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *International Symposium on Microarchitecture (MICRO)*, pages 469–480, 2009.

[59] Pierre-Yvan Liardet and Nigel P. Smart. Preventing SPA/DPA in ECC Systems Using the Jacobi Form. In *Cryptographic Hardware and Embedded Systems (CHES)*, pages 391–401, 2001.

[60] Greg Linden, Brent Smith, and Jeremy York. Amazon.Com Recommendations: Item-to-Item Collaborative Filtering. *IEEE Internet Computing*, 7(1):76–80, January 2003.

[61] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 87–101, 2015.

[62] Chang Liu, M. Hicks, and E. Shi. Memory Trace Oblivious Program Execution. In *Computer Security Foundations Symposium*, pages 51–65, 2013.

[63] Fangfei Liu and Ruby B. Lee. Random Fill Cache Architecture. In *International Symposium on Microarchitecture (MICRO)*, pages 203–215, 2014.

[64] 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. In *Programming Language Design and Implementation (PLDI)*, pages 190–200, 2005.

[65] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. PHANTOM: Practical Oblivious Computation in a Secure Processor. In *Computer and Communications Security (CCS)*, pages 311–324, 2013.

[66] Robert Martin, John Demme, and Simha Sethumadhavan. TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-Channel Attacks. In *International Symposium on Computer Architecture (ISCA)*, pages 118–129, 2012.

[67] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. Thermal Covert Channels on Multi-Core Platforms. In *USENIX Security Symposium (SEC)*, pages 865–880, 2015.

[68] David May, Henk L. Muller, and Nigel P. Smart. Non-Deterministic Processors. In *Australasian Conference on Information Security and Privacy (ACISP)*, pages 115–129, 2001.

[69] David May, Henk L. Muller, and Nigel P. Smart. Random Register Renaming to Foil DPA. In *Cryptographic Hardware and Embedded Systems (CHES)*, pages 28–38, 2001.

[70] David McCann, Elisabeth Oswald, and Carolyn Whitnall. Towards Practical Tools for Side Channel Aware Software Engineering: 'Grey Box' Modelling for Instruction Leakages. In *USENIX Security Symposium (SEC)*, pages 199–216, 2017.

[71] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, pages 10:1–10:1, 2013.

[72] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative Instructions and Software Models for Isolated Execution. In *International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[73] 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 (ICISC)*, pages 156–168, 2005.

[74] Mark Monmonier. The Internet, Cartographic Surveillance, and Locational Privacy. In *Maps and the Internet*, pages 97–113. Elsevier, 2003.

[75] Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. Compiler Assisted Masking. In *Cryptographic Hardware and Embedded Systems (CHES)*, pages 58–75, 2012.

[76] Steven Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc., 1997.

[77] Jean-Michel Muller. On the definition of ulp(x). Technical Report 2005-009, ENS Lyon, February 2005.

[78] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold Implementations Against Side-Channel Attacks and Glitches. In *International Conference on Information and Communications Security (ICICS)*, pages 529–545, 2006.

[79] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious Multi-Party Machine Learning on Trusted Processors. In *USENIX Security Symposium (SEC)*, pages 619–636, 2016.

[80] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In *RSA Conference on Topics in Cryptology*, pages 1–20, 2006.

[81] Elisabeth Oswald, Stefan Mangard, Norbert Pramstaller, and Vincent Rijmen. A Side-Channel Analysis Resistant Description of the AES S-Box. In *Fast Software Encryption (FSE)*, pages 413–423, 2005.

[82] Zizi Papacharissi. The Virtual Geographies of Social Networks: A Comparative Analysis of Facebook, LinkedIn and ASmallWorld. *New media & Society*, 11(1-2):199–220, 2009.

[83] Colin Percival. Cache Missing for Fun and Profit. In *Technical BSD Conference*, 2005.

[84] Thomas Popp and Stefan Mangard. Masked Dual-Rail Pre-Charge Logic: DPA-Resistance Without Routing Constraints. In *Cryptographic Hardware and Embedded Systems (CHES)*, pages 172–186, 2005.

[85] Donald E. Porter, Owen Hofmann, Christopher Rossbach, Alexander Benn, and Emmett Witchel. Operating System Transactions. In *Symposium on Operating Systems Principles (SOSP)*, pages 161–176, 2009.

[86] Srividhya Rammohan, Vijay Sundaresan, and Ranga Vemuri. Reduced Complementary Dynamic and Differential Logic: A CMOS Logic Style for DPA-Resistant Secure IC Design. In *VLSI Design (VLSID)*, pages 699–705, 2008.

[87] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing Digital Side-Channels Through Obfuscated Execution. In *USENIX Security Symposium (SEC)*, pages 431–446, 2015.

[88] Ashay Rane, Calvin Lin, and Mohit Tiwari. Secure, Precise, and Fast Floating-Point Operations on x86 Processors. In *USENIX Security Symposium (SEC)*, pages 71–86, 2016.

[89] Ashay Rane, Mohit Tiwari, and Calvin Lin. Digital Methods for Closing Analog Side Channels. In *Submission*, 2018.

[90] Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. Design Space Exploration and Optimization of Path

Oblivious RAM in Secure Processors. In *International Symposium on Computer Architecture (ISCA)*, pages 571–582, 2013.

[91] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In *Computer and Communications Security (CCS)*, pages 199–212, 2009.

[92] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters (CAL)*, 10(1):16–19, January 2011.

[93] Andrei Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. *IEEE JSAC*, pages 5–19, 2003.

[94] Kouichi Sakurai and Tsuyoshi Takagi. A Reject Timing Attack on an IND-CCA2 Public-Key Cryptosystem. In *International Conference on Information Security and Cryptology*, pages 359–374, 2003.

[95] Werner Schindler. A Timing Attack Against RSA with the Chinese Remainder Theorem. In *Cryptographic Hardware and Embedded Systems (CHES)*, pages 109–124, 2000.

[96] Elaine Shi, Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((logN)^3)$ Worst-Case Cost. In *Advances in Cryptology (CRYPTO)*, pages 197–214, 2011.

[97] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *Conference on Computer and Communications Security (CCS)*, pages 299–310, 2013.

[98] Bo Su, Junli Gu, Li Shen, Wei Huang, Joseph L. Greathouse, and Zhiying Wang. PPEP: Online Performance, Power, and Energy Prediction Framework and DVFS Space Exploration. In *International Symposium on Microarchitecture (MICRO)*, pages 445–457, 2014.

[99] G. Edward Suh, Christopher Fletcher, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Author Retrospective AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *International Conference on Supercomputing (ICS)*, pages 68–70, 2014.

[100] Mohammadkazem Taram, Ashish Venkat, and Dean M. Tullsen. Mobilizing the Micro-ops: Exploiting Context Sensitive Decoding for Security and Energy Efficiency. In *International Symposium on Computer Architecture (ISCA)*, pages 624–637, 2018.

[101] Chandramohan Thekkath, David Lie, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 168–177, 2000.

[102] Shyamkumar Thoziyoor, Jung Ho Ahn, Matteo Monchiero, Jay B. Brockman, and Norman P. Jouppi. A Comprehensive Memory Modeling Tool and Its Application to the Design and Analysis of Future Memory Hierarchies. In *International Symposium on Computer Architecture (ISCA)*, pages 51–62, 2008.

[103] K. Tiri, M. Akmal, and I. Verbauwhede. A Dynamic and Differential CMOS Logic with Signal Independent Power Consumption to Withstand Differential Power Analysis on Smart Cards. In *European Solid-State Circuits Conference (ESSCIRC)*, pages 403–406, 2002.

[104] Kris Tiri and Ingrid Verbauwhede. A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation. In *Design, Automation and Test in Europe (DATE)*, pages 246–251, 2004.

[105] Mohit Tiwari, Casen Hunger, and Mikhail Kazdagli. Understanding Microarchitectural Channels and Using Them for Defense. In *International Symposium on High Performance Computer Architecture*, pages 639–650, 2015.

[106] John Tukey. *Exploratory Data Analysis.* Pearson, 1977.

[107] O. A. Uzun and S. Kse. Converter-Gating: A Power Efficient and Secure On-Chip Power Delivery System. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 4(2):169–179, June 2014.

[108] Bhanu C. Vattikonda, Sambit Das, and Hovav Shacham. Eliminating Fine Grained Timers in Xen. In *Cloud Computing Security Workshop*, pages 41–46, 2011.

[109] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. SD-VBS: The San Diego Vision Benchmark Suite. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 55–64, 2009.

[110] Yao Wang, Andrew Ferraiuolo, and G. Edward Suh. Timing Channel Protection for a Shared Memory Controller. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 225–236, 2014.

[111] Zhenghong Wang and Ruby B. Lee. New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks. In *International Symposium on Computer Architecture (ISCA)*, pages 494–505, 2007.

[112] Zhenghong Wang and Ruby B Lee. A Novel Cache Architecture with Enhanced Performance and Security. In *International Symposium on Microarchitecture (MICRO)*, pages 83–93, 2008.

[113] Wei Wu, Lingling Jin, Jun Yang, Pu Liu, and Sheldon X. Tan. A Systematic Method for Functional Unit Power Estimation in Microprocessors. In *Design Automation Conference (DAC)*, pages 554–557, 2006.

[114] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure Hierarchy-Aware Cache Replacement Policy (SHARP): Defending Against Cache-Based Side Channel Atacks. In *International Symposium on Computer Architecture (ISCA)*, pages 347–360, 2017.

[115] Shengqi Yang, Wayne Wolf, N. Vijaykrishnan, D. N. Serpanos, and Yuan Xie. Power Attack Resistant Cryptosystem Design: A Dynamic Voltage and Frequency Switching Approach. In *Design, Automation and Test in Europe (DATE)*, pages 64–69, 2005.

[116] W. Yu and S. Kse. Time-Delayed Converter-Reshuffling: An Efficient and Secure Power Delivery Architecture. *IEEE Embedded Systems Letters*, 7(3):73–76, Sept 2015.

[117] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Predictive Mitigation of Timing Channels in Interactive Systems. In *Computer and Communications Security (CCS)*, pages 563–574, 2011.

[118] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. Home-Alone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *Symposium on Security and Privacy (Oakland)*, pages 313–328, 2011.

[119] Yinqian Zhang and Michael K. Reiter. Düppel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud. In *Computer and Communications Security (CCS)*, pages 827–838, 2013.

[120] Xinnian Zheng, Lizy K. John, and Andreas Gerstlauer. Accurate Phase-Level Cross-Platform Power and Performance Estimation. In *Design Automation Conference (DAC)*, pages 1–6, 2016.

[121] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. A Software Approach to Defeating Side Channels in Last-Level Caches. In *Computer and Communications Security (CCS)*, pages 871–882, 2016.

[122] Boyuan Zhu, Junwei Lu, and Erping Li. Electromagnetic Radiation Study of Intel Dual Die CPU with Heatsink. In *Symposium on Antennas, Propagation and EM Theory*, pages 949–952, 2008.

[123] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 72–84, 2004.

[124] Hui Zou and Trevor Hastie. Regularization and Variable Selection via the Elastic Net. *Journal of the Royal Statistical Society, Series B*, 67:301–320, 2005.