Software-driven Security Attacks: From Vulnerability Sources to Durable Hardware Defenses

LAUREN BIERNACKI and MARK GALLAGHER, University of Michigan
ZHIXING XU, Princeton University
MISIKER TADESSE AGA, University of Michigan
AUSTIN HARRIS, SHIJIA WEI, and MOHIT TIWARI, University of Texas at Austin
BARIS KASIKCI, University of Michigan
SHARAD MALIK, Princeton University
TODD AUSTIN, University of Michigan

There is an increasing body of work in the area of hardware defenses for software-driven security attacks. A significant challenge in developing these defenses is that the space of security vulnerabilities and exploits is large and not fully understood. This results in specific point defenses that aim to patch particular vulnerabilities. While these defenses are valuable, they are often blindsided by fresh attacks that exploit new vulnerabilities. This article aims to address this issue by suggesting ways to make future defenses more durable based on an organization of security vulnerabilities as they arise throughout the program life cycle. We classify these vulnerability sources through programming, compilation, and hardware realization, and we show how each source introduces unintended states and transitions into the implementation. Further, we show how security exploits gain control by moving the implementation to an unintended state using knowledge of these sources and how defenses work to prevent these transitions. This framework of analyzing vulnerability sources, exploits, and defenses provides insights into developing durable defenses that could defend against broader categories of exploits. We present illustrative case studies of four important attack genealogies—showing how they fit into the presented framework and how the sophistication of the exploits and defenses have evolved over time, providing us insights for the future.

CCS Concepts: • **General and reference** → *Surveys and overviews*; • **Security and privacy** → *Hardware attacks and countermeasures*; *Systems security*; *Formal security models;*

Additional Key Words and Phrases: Taxonomy, security attacks and defenses, vulnerabilities, implementation information, undefined semantics

This work was supported by DARPA under Contract No. HR0011-18-C-0019. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.
Authors' addresses: L. Biernacki, M. Gallagher, M. T. Aga, B. Kasikci, and T. Austin, College of Engineering, University of Michigan, Ann Arbor, 2260 Hayward Street, MI 48109; emails: {lbiernac, markgall, misiker, barisk, austin}@umich.edu; Z. Xu and S. Malik, School of Engineering and Applied Science, Princeton University, 41 Olden St., Princeton, NJ 08544; emails: {zhixingx, sharad}@princeton.edu; A. Harris, S. Wei, and M. Tiwari, Cockrell School of Engineering, University of Texas at Austin, 301 E Dean Keeton St., Austin, TX 78705; emails: {austin.harris, shijiawei, tiwari}@austin.utexas.edu.

1550-4832/2021/07-ART42 $15.00
https://doi.org/10.1145/3456299

ACM Reference format:
Lauren Biernacki, Mark Gallagher, Zhixing Xu, Misiker Tadesse Aga, Austin Harris, Shijia Wei, Mohit Tiwari, Baris Kasikci, Sharad Malik, and Todd Austin. 2021. Software-driven Security Attacks: From Vulnerability Sources to Durable Hardware Defenses. *J. Emerg. Technol. Comput. Syst.* 17, 3, Article 42 (July 2021), 38 pages. https://doi.org/10.1145/3456299

1 INTRODUCTION

There is an increasing body of work in the area of hardware defenses for software-driven security attacks. A major challenge in developing these defenses is that the space of security vulnerabilities and exploits is large and not fully understood. This results in specific point defenses that aim to patch particular vulnerabilities. While these defenses are still valuable, they are often blindsided by fresh attacks that exploit new vulnerabilities. We believe that there is value to be gained by organizing the sources of these vulnerabilities and understanding how they lead to specific attacks and defenses.

At an abstract level, security exploits prevail due to a mismatch between the programmer's intended **finite-state machine (FSM)** for the application/algorithm and its concrete implementation on a general-purpose processor. Through the process of programming, compilation, and hardware realization, there are various sources of security vulnerabilities. Each of these sources introduces states in the concrete implementation that were not intended by the programmer. To synthesize a security exploit, attackers first leverage a vulnerability to move the concrete implementation to a state external to the programmer's intent. Now, outside of the constraints of the original program, an attacker can manipulate the finite-state machine to subvert security guarantees. This process requires accurate knowledge of implementation information and the state space to be successful.

In this work,[1] we classify the sources of vulnerabilities and implementation information leveraged by security attacks. Accordingly, we discuss how defenses directly work to address these sources of exploits-through thwarting access to the implementation information and the implementation state space. To support our formalization, we present case studies of prominent attack genealogies, including control-flow attacks that leverage program-level vulnerabilities to affect execution. Furthermore, we show that side-channel attacks have relied heavily on microarchitectural implementation information and that these two techniques have been combined to break some of the field's most sophisticated defenses. Additionally, we present the historical account of how attacks have evolved in response to various defense mechanisms. Our study reveals that, while attacks evolve quickly, the vulnerability sources and their impact on the final state space of the implementation evolve much more slowly. These observations suggest that defenses that focus on addressing this connection between the vulnerability sources and their impact on the implementation state space may be more effective and durable than traditional security approaches, which focus on finding and fixing vulnerabilities. This article makes the following contributions:

- It provides a framework for organizing vulnerabilities that are introduced through the programming, compilation, and hardware realization phases of the program life cycle.
- It shows how the vulnerabilities result in mismatches between the programmer's intended FSM and the implementation FSM. In particular, it shows how the notion of auxiliary states

[1]This is a systematization-of-knowledge article where we organize a large body of related work into a framework that helps us better understand the relationships between different parts, the evolution of ideas over time, and insights for the future.

in the implementation FSM can capture side-channel information as well as analog effects due to charge leakage exploited in attacks like Rowhammer.
- It shows how exploiting a vulnerability requires knowledge of implementation information throughout the program life cycle and the state space of the resultant implementation FSM.
- It shows how the space of defenses can be classified in terms of how each defense protects knowledge of the implementation and/or the implementation state space.
- It suggests how defenses can be made more durable through an understanding of the above framework and by targeting knowledge of the implementation and/or the implementation state space rather than specific vulnerabilities.
- It presents case studies of the following important attack genealogies by showing how they fit into the presented framework and how the sophistication of the exploits and defenses has evolved over time, providing us insights for the future:
  - ***Control-flow Attacks***—Hijack the victim's control flow to execute malicious code.
  - ***Derandomization Attacks***—Leak implementation information of a randomization defense in an effort to subvert the protection.
  - ***Timing Side-channel Attacks***—Leak a program secret by measuring timing information.
  - ***Transient Execution Attacks***—Leverage mispeculation to induce unintended execution and exfiltrate a program secret.

This article is organized as follows. Section 2 presents the life cycle of a program and outlines the vulnerability sources at each phase in this life cycle. Section 3 discusses how vulnerability sources throughout the program life cycle are leveraged for attacks. Section 4 characterizes how defenses work within our framework to mitigate attacks. Section 5 presents case studies of four important attack genealogies-control-flow attacks, derandomization attacks, timing side-channel attacks, and transient execution attacks-and shows how they fit into our framework. Section 6 concludes this article by highlighting trends and takeaways from this work.

2 MODELING THE PROGRAM LIFE CYCLE

A programmer has some notion of software design even before writing a line of code. We model the desired program functionality that captures the programmer's intent using a FSM, termed the ***intended FSM***. This FSM exists only in the mind of the programmer and captures program behavior as transitions between abstract states. The programmer eventually *programs* hardware to emulate the functionality of the intended FSM. This act of programming requires multiple transformations to the intended FSM, ultimately yielding an ***implementation FSM*** that emulates the intended FSM and runs on the target processor.

Previous work has similarly presented the intended FSM to formalize disparities between the programmer's intent and the final implementation [34]. We take this a step further by understanding the transformations across multiple phases of the program life cycle, shown in Figure 1: high-level programming, compilation, and hardware realization. The first two phases produce intermediate FSMs, termed the language-level and instruction-level FSMs, respectively. Each phase employs an abstract *model* of computation to produce the resultant FSM and may contain other sources that add state and transitions not in the intended FSM (referred to as unintended states and transitions). These unintended states are dangerous, as cleverly crafted inputs that reach them can force the program outside of the original specification. Once no longer bounded by the program's constraints, the implementation may produce unexpected results or violate security guarantees. Thus, these sources of unintended states are effectively ***vulnerability sources***, shown in blue in Figure 1.

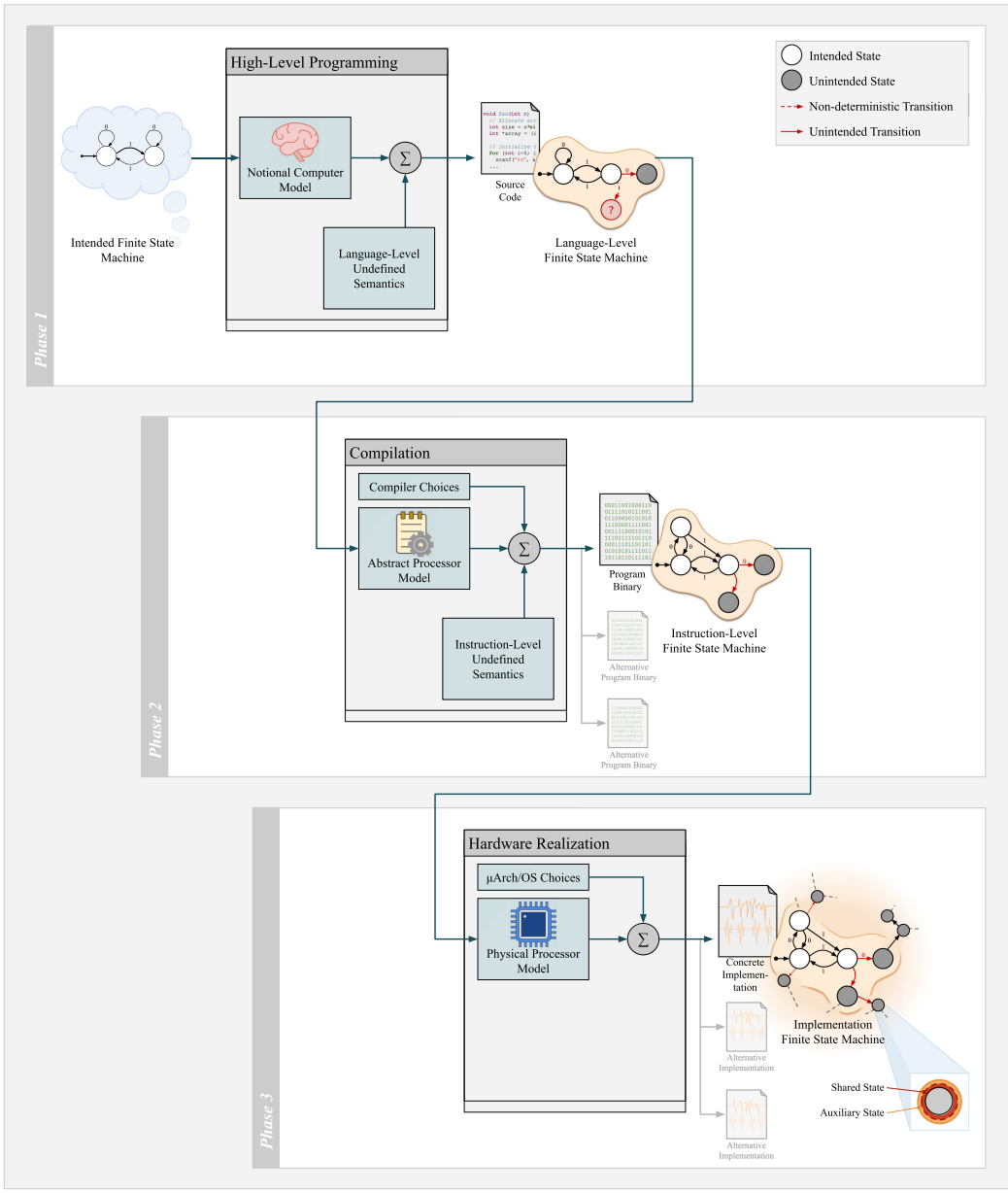


Fig. 1. Program Life Cycle. A programmer's intended finite-state machine (FSM) is transformed into an implementation FSM through multiple phases of the program life cycle: high-level programming, compilation, and hardware realization. The first two phases produce intermediate FSMs, termed the language-level and instruction-level FSMs, respectively, and each phase employs an abstract model of computation to produce the resultant FSM. Additionally, each phase may contain other sources that add unintended state (grey nodes) and transitions (red edges) that are absent from the intended FSM. Undefined semantics in high-level programming and compilation introduce non-deterministic transitions (dashed red edges) that are defined later in the program life cycle. Within this framework, every input into Sigma (Σ) is a vulnerability source and introduces unintended states and transitions that can be leveraged for security exploits.

At each stage in the program life cycle it is useful to think of the state of the program and what causes it to move away from its intended next state to one where it is susceptible to an attack. By representing programs as transition systems, we can reason about security vulnerabilities as unintended transitions within the state space of program evolution (i.e., the program's FSM), where state is characterized by what the program is doing (i.e., data state, like memory contents) and where the program is in execution (i.e., control state, like the program counter). Similarly, we can model defenses as preventing a program from making these unintended transitions or limiting subsequent unintended transitions that can be forced by an attacker to induce malicious functionality. To address the significant challenge of understanding the large space of security exploits, our program life cycle framework is designed as a conceptual model to organize vulnerability sources across the program life cycle and model how defenses address these vulnerability sources.

2.1 High-level Programming

High-level programming is the process of expressing the programmer's intended application into source code in some programming language. High-level programming modifies the intended FSM, concretizing potentially abstract states, expanding the state space, and introducing new transitions. Two distinct mechanisms drive this modification: the programmer's notional computer model and language-level undefined semantics.

2.1.1 Notional Computer Model. A ***notional computer model*** is specific to a programmer and captures all assumptions they hold of the computation engine, including notions of the programming language, compiler, and processor. The specifics of this model depend heavily on the programmer's expertise, with a novice programmer's notional computer model likely capturing only a subset of the operations that the programming language affords. However, an expert programmer's notional computer model is likely to span beyond the language to capture elements of the entire system. For example, a programmer with knowledge of the target processor's memory layout may manually attempt structure packing to reduce their program's memory footprint. While a programmer may have different notional computer models specific to each programming language, this distinction is not relevant to the ideas presented in this work.

The notional computer model permits the introduction of logic bugs and vulnerabilities into the resultant source code. These flaws can arise from discrepancies between the programmer's notional computer model and actual program execution. For example, consider a novice programmer performing array operations in C who is unaware that C uses zero-based indexing. To access the first element of an array of length $n \geq 1$, they load index one, rather than index zero. While the programmer intends to access the first element of the array, they mistakenly load the second. This operation is unintended with reference to the intended FSM, introducing a transition to an unintended state in the resultant ***language-level FSM***. In this example, the result of this transition is well defined within the high-level programming phase. However, other transitions introduced within this phase may be non-deterministic, only to be defined later in the program life cycle.

2.1.2 Language-level Undefined Semantics. Not all programming mistakes result in well-defined transitions in the language-level FSM. Non-deterministic transitions are also introduced during the high-level programming phase due to the effects of ***language-level undefined semantics***. In this work, we define language-level undefined semantics as any source code construct for which the programming language standard does not impose a specific behavior. Continuing the above example, to access the last element of the array, the programmer loads index n, a nonexistent item outside of the array's bounds. Out-of-bounds memory accesses are undefined in the C-language specification, meaning they are permitted but their effect is not guaranteed. This operation is unintended with reference to the intended FSM, and its result is *undefined*, introducing a

non-deterministic transition (unspecified destination state) in the resultant machine. Only after compilation and hardware realization will we know the destination state of this transition.

Language-level undefined semantics comprise the traditional notions of undefined behavior, unspecified behavior, and implementation-defined behavior in the C/C++ standard. For undefined and unspecified behavior, the standard imposes no deterministic outcomes-for undefined behavior, the standard states nothing, whereas, for unspecified behavior, the standard states at least two different outcomes. Implementation-defined behavior is a subset of unspecified behavior that the operating environment implements consistently and documents fully (e.g., whether `char` behaves as `signed char` or `unsigned char`). Programmers with knowledge of the target system can reliably leverage implementation-defined behavior due to the environment's guarantees, whereas the usage of undefined behavior is generally regarded as erroneous. We classify all of these three distinctions as language-level undefined semantics, because the use of these semantics, either unintentionally or intentionally, introduces non-deterministic transitions into the language-level FSM.

2.1.3 Phase Summary. The high-level programming phase produces source code that provides a specification of a language-level FSM. This resultant FSM emulates the intended FSM and includes an expanded state space due to the effects of the notional computer model and language-level undefined semantics. Specifically, the notional computer model introduces unintended states and transitions into the language-level FSM due to misconceptions held by the programmer (e.g., confusion between zero- and one-based indexes). Language-level undefined semantics add non-deterministic transitions that are only defined during the later phases of compilation or hardware realization (e.g., buffer overflow). Attackers leverage knowledge of these effects as they provide a direct means to invoke transitions to unintended states to gain control or leak information.

2.2 Compilation

Compilation transforms source code into a program binary for the target processor. Compilation modifies the language-level FSM, resolving its non-deterministic transitions, expanding the state space, and introducing new transitions, using three mechanisms: the compiler's abstract processor model, compiler choices, and instruction-level undefined semantics.

2.2.1 Abstract Processor Model. The ***abstract processor model*** captures the target processor and context in which the program executes to permit code generation and optimization. It includes notions of the instruction set architecture, such as the number of registers or addressing modes on the target device, needed to transform the source code into a form that can be executed on the target processor. This information is used by the abstract processor model to make decisions regarding register allocation and instruction scheduling. Additionally, the model also captures the functionality of the target device, which is necessary to achieve correctness while performing optimizations. By modeling the semantics of the source and target language, the compiler can assure that any transformations to the FSM maintain input and output consistency (i.e., the program still produces the intended output for possible inputs). An abstract processor model is specific to a compiler instance and can produce alternative program binaries dependent on the decisions made during compilation.

The abstract processor model may be incomplete or even faulty, introducing unintended functionality into the resultant binary. For example, while compilers model program functionality, many do not model microarchitectural state. This disparity permits optimizations that maintain input-output consistency but modify cache contents or memory access patterns. One culprit of this memory alteration is dead store elimination. Consider an application that reads a password from the command line, computes its hash, and sends the hash to the server for authentication. During the calculation of the hash function, the plaintext password is stored in memory. Afterward, the

program erases this variable for security purposes by setting it to zero. From the compiler's perspective, the password is never used after it is set to zero, meaning that this zeroing store is dead. Thus, through dead store elimination, the compiler removes this zeroing store, causing the true plaintext value to persist in memory despite the programmer's intent. With respect to our framework, the compiler introduces unintended states and transitions into the resultant ***instruction-level FSM*** where the program counter advances and the plaintext password persists in memory. The state of the incremented program counter and sensitive password is not intended by the programmer. Ultimately, discrepancies in the abstract processor model introduce unintended functionality to the resultant FSM and have been termed the "correctness-security gap" in compilers [33].

2.2.2 Compiler Choices. ***Compiler choices*** encompass the decisions made by the compiler to implement source code constructs on a hardware platform. Most notably, compiler choices include data and code layout and representation. These decisions reflect the abstract processor model, as the instruction set and target architecture limit viable options. Compiler choices affect the resultant program binary and instruction-level FSM by adding complexity, increasing the size and number of states, and modifying transitions. Compiler choices may also resolve non-determinism in the language-level FSM, for example, by specifying that the type `char` behaves as `unsigned char`.

We find that attackers repeatedly leverage knowledge of compiler choices to perpetrate security exploits. Consider the buffer overflow example discussed in Section 2.1. While this vulnerability is introduced during the high-level programming phase, it can only be exploited for a stack smashing attack with knowledge of compiler choices. Namely, an attacker must know the contents of the stack as well as the target function's location to overwrite the return address and redirect program execution reliably [92]. These two attack assets are determined by compiler choices.

2.2.3 Instruction-level Undefined Semantics. Akin to high-level programming languages, assembly languages may include constructs with no specific defined behavior, termed ***instruction-level undefined semantics***. These undefined semantics within the assembly language similarly add non-deterministic transitions to the resulting instruction-level FSM. For example, the x86-64 standard does not define the value of flags for all instructions. The parity flag, `%pf`, is undefined for the `andn` instruction, enabling processors to either update the flag or keep the previous value. Previous work has shown that processors implement this semantic differently [30]. While instruction-level undefined semantics cause unintended states and transitions in the implementation FSM, we have not found evidence of existing security exploits that leverage these assets.

2.2.4 Phase Summary. The compilation phase produces a program binary file that provides a bit-level specification of an instruction-level FSM. This binary is one of many possible alternative files that can be generated depending on the compiler instance and choices. The instruction-level FSM speaks to these choices, capturing all compiler decisions, including data layout. Hence, the resultant machine includes an expanded state space due to the effects of compiler choices and the abstract processor model. The abstract processor model introduces both unintended states and transitions into the resultant instruction-level FSM (e.g., persistent state security violations). Instruction-level undefined semantics add non-deterministic transitions that are later determinized by the microarchitecture. Attackers leverage knowledge of these effects, especially code and data layout, to perpetrate security exploits on the target hardware platform.

2.3 Hardware Realization

Hardware realization is a process that takes input in the form of a program binary and executes it on a hardware platform. This program execution follows the ***implementation FSM***, which

captures all reachable states in the target processor, including modeling the relevant contents of microarchitectural structures. Hardware realization resolves all remaining non-determinism in the implementation FSM, expands its state space, and introduces new transitions, under the influence of two mechanisms: the physical processor model and microarchitectural choices.

2.3.1 Physical Processor Model. The ***physical processor model*** depicts one of many possible architecture targets for the resultant implementation, modeling the hardware that implements the functionality specified by the instruction set architecture. For example, the physical processor model transforms load/store operations in the program binary into complex hardware-based protocols that implement virtual memory. Because the physical processor model includes notions of all microarchitectural structures, it expands the state space of the instruction-level FSM immensely. The resultant implementation FSM contains states representing all possible combinations of the contents of these structures that are reachable from the initial state (e.g., all possible values of the program counter).

The physical processor model may also perform transformations on the FSM to optimize performance, such as realizing speculative execution. These transformations preserve functional correctness but can modify state in an unintended manner, increasing the complexity of the resultant state space and introducing new transitions among states. The most compelling and currently relevant example of this phenomenon is mispeculation. During branch misprediction, modern processors squash in-flight instructions and roll back execution. These procedures preserve program functionality. However, not all changes to the microarchitecture are reverted. Evidence of the mispeculated instructions remains in structures like the cache, a property that recent attacks such as Spectre [57] and Meltdown [62] have exploited to exfiltrate program secrets. In terms of the implementation FSM, the state before and after rollback is *different* even though functional correctness is preserved.

In isolation, mispeculation is not dangerous. However, it becomes dangerous when combined with other effects of the physical processor model. Specifically, given the current design of microarchitectural structures, the physical processor model introduces state to the resultant FSM that is shared between programs. This ***shared state*** originates from structures that are not flushed during context switching, allowing data to linger after execution completes. In the context of the implementation FSM, this shared state manifests as residues from other programs that can affect the victim program's execution, or alternatively be used to monitor the victim program externally. Shared state can be dangerous as it establishes a channel for third-party observation of a program and may even permit external influence over execution. In the case of mispeculation, shared state enables an attacker to observe the residue of squashed instructions through the cache. Shared state also allows an attacker to influence which control transfers are mispredicted by priming the branch predictor. These mechanisms have been abused for devastating hardware-based exploits [45, 62].

2.3.2 Microarchitectural and OS Choices. ***Microarchitectural choices*** encompass decisions that realize microarchitectural structures on the target device, such as the size and layout of the caches, TLB, store buffer, and so on. There may be multiple microarchitectural choices that implement the functionality required by the physical processor model, but ultimately only one design is employed, affecting the resulting state space in the implementation FSM. We distinguish microarchitectural choices from the physical processor model, because attacks often leverage these specific details in security exploits. For example, shared cache state can be used to observe transient instructions, as mentioned above. However, an attacker must have knowledge about the cache structure, including the size and associativity of the cache, to extract this information with high fidelity. This knowledge of the cache originates from microarchitectural choices. While many attacks are still viable without knowledge of microarchitectural choices (e.g., control-flow attacks), we find that timing side-channels and other architecture-based attacks repeatedly leverage knowledge of these

decisions. ***Operating System (OS) choices*** capture decisions made by the system during runtime. This mechanism primarily represents code and data locations determined at load time due to the use of either position-independent code or **Address Space Layout Randomization (ASLR)** [75]. In this work, we assume code and data location is determined at compile time unless explicitly stated.

2.3.3 Auxiliary State. The implementation FSM also includes ***auxiliary state***, such as wall clock time and occurrence count, that is not reflected at the bit-level in the microarchitecture but retains information and may influence transitions during execution on the target processor. Notably, analog effects can be discretized as auxiliary state for the purpose of modeling them in the FSM. In this work, we distinguish between readable and writeable auxiliary state. Prominent examples of auxiliary states leveraged by security exploits include overall program execution time [58] and occurrence count of memory accesses, which can be used to capture DRAM cell charge leakage [54]. Program execution time is a readable auxiliary state that can be "read" in a variety of ways by an attack. Occurrence count is a writeable auxiliary state that reflects the count of a particular operation (e.g., memory access) and that, on crossing some threshold in some duration, leads to a change of memory state (e.g., bit flip in a DRAM). Thus, occurrence count can be used to effectively model the analog effects of the microarchitecture. As this information is not reflected in the microarchitecture directly, auxiliary state is, in general, hard to model in advance and instead modeled *post-facto* once its possible use is known.

2.3.4 Phase Summary. The hardware realization phase produces an implementation FSM that runs on a specific target architecture (of many) and sufficiently emulates the intended FSM. The implementation captures all additional state resulting from the physical processor model and microarchitectural choices, as well as the prior compilation and high-level programming phases. The physical processor model may also introduce transitions that do not match the programmer's intent, such as leaving execution residues in microarchitectural structures. Finally, the hardware-implemented FSM contains no non-deterministic transitions, as every transition can be fully defined given knowledge of the entire state space. The hardware realization stage produces an implementation that contains numerous unintended states and transitions resulting from the effects of the entire program life cycle. Knowledge of these mechanisms is leveraged across attack classifications to perpetrate security exploits.

2.4 Relation to Previous Works

The framework presented in this article details how the different phases of the program life cycle introduce unintended states and transitions into the final implementation. This framework builds upon the *weird machines* model that captures how security exploits arise from unexpected program input that triggers unintended functionality [20, 34]. In that work, reachable states in the hardware implementation that enable unintended functionality form a new computational device, termed a "weird" machine, that permits security exploits. Work on weird machines specifically looks at the disparities between the programmer's intent and the resultant hardware implementation, *but not the intermediate mechanisms at play*. Dullien has characterized these differences using formal computational models [34]. Other work has studied specific instances of weird machines in the context of page faults [13], ELF executables [86], and proof-carrying code [97].

There is a significant body of work that identifies how undefined semantics in the C and C++ programming languages introduce vulnerabilities during compilation [19, 33, 61, 80, 99]. These works document how choices made during compilation resolve non-determinism from language-level undefined semantics in an unexpected way, such as by removing security checks (Section 3.1.2) [33].

Other work has modeled how weird machines arise during compilation due to behaviors in the target language that cannot be achieved by the semantics of the source language [76].

No work has provided a complete framework of how programmer conceptions, compiler interactions, and microarchitectural details introduce unintended states and influence security exploitation. In this work, we broadly reassess security exploits through the complete program life cycle, expanding the weird machine model to capture effects from the programmer, compiler, and microarchitecture. Our goal is threefold: *(i)* To capture *all* sources of vulnerabilities throughout the program life cycle, *(ii)* To understand how these sources are exploited to force the implementation FSM into an unintended state that compromises security, and *(iii)* To illuminate how knowledge of this framework and the implementation FSM can lead to an understanding of durable defenses that thwart complete categories of unintended functionality rather than patching individual vulnerabilities.

3 SECURITY EXPLOITS

In Section 2, we presented our program life cycle framework, which models how high-level programming, compilation, and hardware realization introduce unintended states and transitions into the implementation FSM. Unintended states permit security exploits by enabling an attacker to circumvent the programmer's intent and cause unwanted program behavior. In this section, we delve further into how security exploits rely on these states to infiltrate victim programs.

Within our proposed framework, we characterize a ***security exploit*** as a path from intended to unintended states that produces unexpected and unwanted behavior in the implementation that is not possible in the intended FSM. Perpetrating an exploit requires two critical components: a transition between an intended and unintended state, termed a ***vulnerability***, to bypass the program intent/specification, and knowledge of the state space, termed ***implementation information***, to reliably manipulate the implementation FSM. We have found that security exploits leverage vulnerabilities and implementation information from all phases of the program life cycle. These ***vulnerability sources*** are the blue colored boxes in Figure 1. We note that numerous exploits for a single vulnerability exist if multiple paths in the implementation FSM can use this vulnerability to achieve the same ends. Alternatively, exploits may also leverage different vulnerabilities to produce the same unwanted behavior. Below, we enumerate the vulnerabilities and implementation information commonly leveraged by attacks and identify their sources.

3.1 Vulnerabilities

Vulnerabilities are transitions in the concrete implementation that move the machine to an unintended state. By leveraging a vulnerability, an attacker moves the implementation FSM to a state that the programmer did not consider, enabling incorrect behavior or the evasion of security protections not represented within the present state. These dangerous transitions originate from all phases of the program life cycle, from programmer misconceptions to the physical hardware's subtleties. We enumerate classes of common vulnerabilities in Table 1, categorizing them by their ***source***, or the mechanism that introduces the offending transition into the FSM at the end of that phase. Below, we detail the vulnerabilities introduced during each phase of our framework.

Vulnerabilities originating from high-level programming are often triggered by program inputs (e.g., arguments passed through the command line), whereas vulnerabilities from subsequent phases are triggered by other influenceable states (e.g., branch predictor state). An attacker can trigger these transitions to shift the implementation FSM beyond the intended specification by carefully interfering with the target processor. However, to do so, an attacker must be able to successfully find and trigger a vulnerability. When this vulnerability originates from high-level programming, an attacker must have intimate knowledge of the program source code to locate this

Table 1. Enumeration of Vulnerabilities

Vulnerability	Phase	Source	Unintended FSM Transition
Memory Access Errors. *Buffer overflow, dangling pointer, use-after-free, etc.*	High-level Programming	Language-level Undefined Semantics	Based on illegal memory access
Uninitialized Values. *Using a value not yet assigned*	High-level Programming	Language-level Undefined Semantics	Based on unpredictable memory value
Signed Integer Overflow. *A numeric value that exceeds the number of storage bits*	High-level Programming	Language-level Undefined Semantics	Based on modulo wrapping properties
Unsigned Integer Overflow. *A numeric value that exceeds the number of storage bits*	High-level Programming	Notional Computer Model	Based on modulo wrapping properties
Sharing-dependent Program Execution. *Program execution that depends on shared state* (e.g., *branch predictor contents)*	Hardware Realization	Physical Processor Model	Based on externally influenced state
Sharing-dependent Program Timing. *Program timing that depends on shared state* (e.g., *external cache conflicts)*	Hardware Realization	Physical Processor Model	Based on externally influenced state
Mispeculation. *Unintended execution of spurious instructions*	Hardware Realization	Physical Processor Model	Based on (incorrect) speculative state
Side Channels. *Confidential information propagated to unintended outlets*	Hardware Realization	Physical Processor Model	Reveals secret state information
Usage-dependent Analog Behavior. *Rowhammer, A2 attack [106], etc.*	Hardware Realization	Physical Processor Model	Based on auxiliary state

Vulnerabilities are transitions to unintended states in the FSM. We classify vulnerabilities by their phase and source: whether they are introduced during high-level programming via language-level undefined semantics or the notional computer model, during compilation via instruction-level undefined semantics, or during hardware realization via the physical processor model. This list is not exhaustive of all vulnerabilities, but rather represents those most prevalent in security exploits.

single fault. While this appears challenging, the commonality of some program-level vulnerabilities, like buffer overflows, eases this task. Additionally, vulnerabilities originating from hardware realization will be shared across multiple programs running on the same target platform. As a result, hardware-based vulnerabilities can be more pervasive and harder to address due to the fixed nature of physical processors.

3.1.1 Vulnerabilities Originating From High-level Programming. During high-level programming, language-level undefined semantics introduce vulnerabilities in the form of nondeterministic edges that are later defined during compilation or hardware realization. Language-level undefined semantics leveraged by the security exploits covered in this work include ***spatial and temporal memory access errors***, the use of ***uninitialized values***, and ***signed integer overflow***. We note that these vulnerabilities may also persist in the source code due to programmer misconceptions about the functionality of these operations. For example, a buffer overflow is a language-level undefined semantic, but it may also exist in the program due to a flawed notional computer model (e.g., a programmer that is unaware that C uses zero-based indexing, as discussed in Section 2.1). Hence, misconceptions and undefined semantics are often intertwined, working concurrently to produce flaws in the source code. Misconceptions can also introduce vulnerabilities when undefined semantics are not in play. For example, ***unsigned integer overflows*** can be just as dangerous and prevalent as signed integer overflows despite being fully defined by the C/C++ specification. Because these vulnerabilities are introduced during high-level programming, their expressiveness depends heavily on the remaining implementation. For example, the layout of the stack and presence of the NX-bit [87] will limit the viability of a buffer overflow

vulnerability. In Section 5.1, we discuss how control-flow attacks heavily use language-level vulnerabilities to subvert program constraints.

3.1.2 Vulnerabilities Originating From Compilation. Compilation can also introduce transitions to unintended states into the resultant instruction-level FSM. These effects include optimizations that modify the timing or memory access patterns of programs, as discussed in Section 2.2.1. Changes made during compilation may also reintroduce program-level vulnerabilities. For example, consider a program that computes the size of an integer array by multiplying the number of array elements by `sizeof(int)` and stores this result in an integer variable called `sz`. If the number of array elements is sufficiently large, then the variable `sz` will overflow, because it is only represented by 32 bits. To prevent this signed integer overflow vulnerability, the program checks if `sz` is smaller than the number of array elements and aborts to indicate this exception. Since a signed integer overflow is undefined in the C language, the compiler may assume that the if-condition is always false (i.e., `sz` cannot be smaller than the number or array elements) and mark it as dead code [33]. Thus, this flaw in the compiler's abstract processor model permits the removal of this security check, resulting in code that contains an integer overflow vulnerability despite the programmer's intent to actively avoid one. With respect to our framework, although this compiler operation removes a security check, it introduces unintended states and transitions into the resultant FSM in which `sz` is smaller than the number of array elements and the program counter continues to advance. Such an integer overflow vulnerability could be used to allocate a zero-sized array or hash table to enable arbitrary code execution, as was done by attacks on SSH1 [1].

3.1.3 Vulnerabilities Originating From Hardware Realization. Vulnerabilities introduced during hardware realization originate from the physical processor model and especially from design choices related to shared and auxiliary state. Specifically, the physical processor model adds transitions dependent on influenceable states (i.e., shared state or writable auxiliary state) or forms paths that propagate confidential information to observable outlets (i.e., shared state or readable auxiliary state). These formulations are unintended with respect to the intended FSM and pervasive across architecture-based attacks. For example, ***mispeculation*** permits the execution of unintended instructions. In isolation, mispeculation is not dangerous, as discussed in Section 2.3. But, when combined with vulnerabilities originating from hardware realization, transient execution becomes a powerful tool to perpetrate security exploits. Attackers can influence transient execution due to ***sharing-dependent program execution*** in the microarchitecture. Specifically, shared state, like the branch predictor contents, affects program transitions, enabling an attacker to control execution without supplying program input directly. Furthermore, transient execution leaves residues in the cache, affecting shared state (cache contents) and readable auxiliary state (timing), due to the presence of a ***side channel*** vulnerability. With these two mechanisms in hand, an attacker can force a program to execute an instruction speculatively and observe the traces of that execution. In addition to sharing-dependent program execution and side channels, we recognize ***usage-dependent analog behavior*** as a vulnerability introduced during hardware realization. This vulnerability captures transitions caused by analog-based auxiliary state (Section 2.3.3), such as how a bit flip can occur in DRAM due to repeated accesses to a neighboring row [54].

3.2 Implementation Information

Knowledge of the state space of the implementation FSM, termed implementation information, is leveraged directly for security exploits. Given this information, an attacker has a better understanding of vulnerability sources and both unintended states and transitions in this FSM. Hence, the attacker has a stronger and more refined model of the semantics of the implementation than the original programmer. Attackers can take advantage of this information to find a way to

Table 2. Enumeration of Implementation Information

Implementation Information	Phase	Source	Acquisition
Code Contents. *Behavior of functions, etc.*	High-level Programming	Notional Computer Model	DOC, ENGR, PROBE
Location of Code/Data. *Location of stack, heap, etc.*	Compilation	Compiler Choices*	ENGR
Order of Code/Data. *Order of variables within a stack frame, etc.*	Compilation	Compiler Choices	DOC, ENGR
Representation of Code. *Binary representation of instructions*	Compilation	Compiler Choices	DOC
Representation of Code/Data Pointers. *Binary representation of addresses*	Compilation	Compiler Choices	DOC
Microarchitectural Sharing. *Cache collisions, shared BTB entries, etc.*	Hardware Realization	Microarch. Choices	ENGR, PROBE
Timing of Operations. *Latency of memory accesses, etc.*	Hardware Realization	Microarch. Choices	PROBE

We classify implementation information by its phase and source, whether it is introduced during high-level programming via the notional computer model, during compilation via compiler choices, or during hardware realization via microarchitectural choices. Furthermore, we specify how this implementation information is acquired by attackers via documentation (DOC), reverse engineering efforts (ENGR), or probing techniques (PROBE). This list is not exhaustive of all implementation information, but rather represents the classes most prevalent in security exploits. *Location of Objects originates from OS Choices when ASLR [75] is in place.

synthesize an exploit and subvert the program's security measures. It should be noted that implementation information does not detail specific states or transitions, but is a property of the state space as a whole. This knowledge is, in effect, more compelling, as an attacker can often successfully synthesize exploits with only a few fundamental pieces of implementation information. Table 2 enumerates implementation information that is typically required by exploits. We categorize implementation information by its *source*, or the mechanism that introduces the state information during the program life cycle. We also categorize implementation information by the method in which knowledge of it is acquired, termed *Implementation Information Acquisition*, through one of the following channels: documentation (DOC), reverse engineering (ENGR), or probing (PROBE). These methods of acquisition broadly capture three possible scenarios for understanding a particular system aspect: *(i)* The system aspect is documented publicly and is known by the attacker or can be obtained easily, *(ii)* The system aspect is not documented publicly but can be directly derived using knowledge of other system aspects, or *(iii)* The system aspect can only be indirectly derived by querying the system and observing a response. Below, we discuss implementation information acquisition in depth, and enumerate information sources in each stage of our framework.

3.2.1 Implementation Information Acquisition. Security exploits require some knowledge of the implementation information to succeed. Control-flow attacks that exploit program-level vulnerabilities typically require knowledge of the memory layout, stack organization, and code locations. Similarly, hardware-based attacks, like timing side-channels, require knowledge of microarchitectural structures and sharing between processes. This knowledge is obtained using acquisition methods (DOC, ENGR, or PROBE) at each level of the program life cycle.

By leveraging documentation (DOC) and reverse engineering techniques (ENGR), an attacker can gain knowledge of system aspects determined during high-level programming and compilation. Documentation and formal specifications of the programming-language and operating system are among the most trivial ways to obtain knowledge of these system aspects. Alternatively, reverse

engineering the program binary can also reveal implementation decisions made during compilation. For example, calling conventions in the executable can expose the organization of the call stack. The program binary also reveals the location of functions or the relative distance between functions, depending on the underlying runtime environment.

Knowledge of the hardware is among the most difficult to obtain, as processor design is proprietary, and often requires sophisticated probing techniques (PROBE) to infer the implementation information. For example, timing side-channel attacks must have knowledge of the cache layout to craft an eviction set. However, constructing an eviction set for the **last-level cache (LLC)** is difficult, because it is physically indexed, and other microarchitectural choices, such as dividing the LLC into per-core slices, complicates the cache mappings. Rather than reverse-engineering the hash function used to distribute the cache to different cores (as in Reference [47]), PRIME+PROBE [63] proceeds by *probing* the LLC to iteratively construct an eviction set. Specifically, PRIME+PROBE selects a potential conflict set from a large buffer, then repeatedly checks if this set evicts a candidate memory line from the LLC by measuring memory access times (i.e., readable auxiliary state). Candidate memory lines are added to the conflict set if they are not evicted (i.e., when the memory access is fast). Once the conflict set is formed, the attack repeats a similar process to partition the conflict set into an eviction set for a specific candidate line. After this second iteration, the attacker has acquired sufficient knowledge of microarchitectural sharing in the LLC to synthesize the side-channel exploit.

If the implementation is totally opaque to the user, then security exploits cannot prevail. However, information leakage is innate to many implementation decisions by design, therefore, opaque implementations are not a universal solution. For example, the timing of a memory access reveals if there is a hit or miss in the cache. This property is inherent to the cache design, yet leaks information that can be used to determine the contents of memory. Rather than making implementation information completely opaque, an alternative solution is to make it unreliable and inconsistent. Such randomization techniques increase the difficulty of implementation information acquisition. For example, ASLR [75] determines memory layout at load-time, forcing attackers to obtain memory information using sophisticated probing techniques when only reverse engineering was required previously [17, 36, 42, 90]. Randomization defenses are further characterized in Section 4.3.

3.2.2 Implementation Information From High-level Programming. Perhaps the most critical implementation information required for exploits is knowledge of the ***contents of the victim source code***. Code contents detail how the intended FSM is implemented as source code, capturing program functionality like function contents, input formatting, and operational behaviors. All security attacks exploit some degree of knowledge of a victim program. Stack smashing attacks must understand how to provide program input that propagates to a buffer-overflow vulnerability [8]. Code reuse attacks must know the functionality of code gadgets [81, 92]. Timing side-channel exploits rely on secret-dependent program execution (e.g., modular exponentiation with key-dependent execution timing) [58]. Spectre attacks require knowledge of a Spectre gadget, or jump instruction in the victim program [57]. To acquire this information, attackers may use documentation, reverse engineering, or probing techniques. Attackers leverage software documentation, open-source code, or knowledge of popular libraries to discover program functionality easily (DOC). If the source code is not widely available but the binary is, then disassembling the binary can provide similar information (ENGR). If neither is available, then attackers may attempt to glean program behavior through probing the API, repeatedly sending inputs to profile the results (PROBE).

3.2.3 Implementation Information From Compilation. The compiler implements many components of the implementation FSM that are critical to program execution, including the ***location***

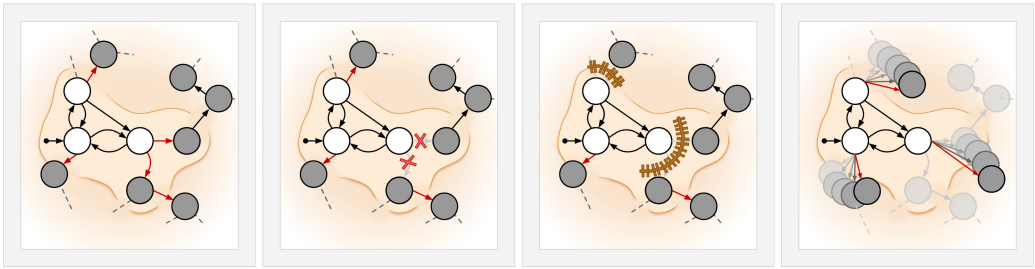
and order of objects. For example, during compilation, the operating system, compiler, and target instruction-set dictate the implementation of function calls in the source code. This is traditionally done through the manipulation of a call stack. Values stored to the stack record the control flow of the program and pass arguments to the requested function. In addition, the call itself decomposes into multiple instructions that calculate the target function pointer and jump to the resulting address. These abstractions are hidden from the programmer in high-level languages and add state to the resulting implementation that is critical for stack smashing exploits [8]. In addition to data and code layout, the compiler also implements the ***representation of code and pointers***. While these mechanisms are heavily constrained by what the target processor supports, they are introduced into the resultant binary and instruction-level FSM during compilation.

3.2.4 Implementation Information From Hardware Realization. Hardware realization massively expands the state space of the implementation FSM based on microarchitectural choices. The resultant machine contains shared and auxiliary state, including properties of ***microarchitectural sharing*** and the ***timing of operations***. Knowledge of these two pieces of implementation information is leveraged directly for transient execution attacks (Section 5.4), like Spectre V2 [57]. Spectre attacks leak information by combining speculative execution with data exfiltration through timing side-channels. Spectre V2 does this by mistraining the processor's branch predictor to mispredict a specific indirect branch of the attacker's choosing, then observing the residues of this transient execution through the cache. While this attack requires some knowledge of program-level information, like the location of the victim branch, it also requires intimate knowledge of the branch predictor. To reliably mistrain the predictor, the attackers used experiments to reverse engineer (ENGR) the branch predictor on the target machine. They found that a predictor on a Haswell i7-4650U only stores the lower 20 bits of the virtual address for twenty-nine previous destination addresses [57]. Using this knowledge, they were able to craft malicious code to mistrain the predictor reliably. The remainder of the attack recovers cache residues by performing FLUSH+RELOAD [107].

Defenses may also seek to instantiate implementation information during hardware realization to prevent attacks. ASLR [75] randomizes the location of the code segment when the program is loaded. Hence, function pointers will vary per instance of the program. ASLR adds diversity to the state space of the implementation as each instance of the program now has a unique memory configuration. With memory locations changing at runtime, obtaining implementation information requires advanced leakage techniques that are much more difficult than if memory locations were determined at compile time.

3.3 Summary of Security Exploits

Security exploits leverage a vulnerability to transition the implementation FSM to an unintended state, bypassing the programmer's intent, then use knowledge of implementation information to manipulate the implementation FSM and produce unexpected and unwanted behavior. With respect to our framework, we enumerate and classify vulnerabilities by their source, identifying prominent vulnerabilities originating from language-level undefined semantics, the notional computer model, and the physical processor model, shown in Table 1. We similarly enumerate and classify implementation information originating from compiler and microarchitectural choices in Table 2. We also characterize how attackers acquire knowledge of implementation information needed for exploits, identifying three primary methods with increasing levels of sophistication: documentation, reverse engineering, and probing. In Section 5, we discuss how prominent attacks fit within our model of security exploits.



(a) **Unprotected.** (b) **Avoidance.** (c) **Enforcement.** (d) **Obfuscation.**

Fig. 2. Defense Techniques. Vulnerabilities are exploited to transition the victim program from an intended state (white) to an unintended state (grey). Once outside of the constraints of the intended IFSM, an attacker can use knowledge of implementation information to subvert security. Avoidance-based protections (b) find and fix vulnerabilities to remove single transitions to unintended states. Enforcement-based protections (c) take a complete approach by addressing *all* transitions of a particular type (e.g., buffer overflows). Finally, obfuscation techniques (d) conceal implementation information and unintended states, preventing attackers from effectively and reliably learning enough information to synthesize an exploit.

4 CLASSIFICATION OF DEFENSES

Exploits subvert the security guarantees of an application by using a vulnerability to transition to an unintended state, then using knowledge of implementation information to produce unexpected behavior. To combat attacks, defenses address the usage of unintended states by either preventing the transition to an unintended state (i.e., via removing a vulnerability), or preventing the subversion of security guarantees (i.e., via obfuscating the state space), effectively making it impossible or useless to acquire unintended states or transitions. Namely, protections either ***avoid*** vulnerabilities by removing their instances in applications, ***enforce*** a property of the machine to mitigate vulnerabilities, or ***obfuscate*** implementation information to make the implementation unpredictable and thus difficult to manipulate. These categories are not mutually exclusive, as some defenses have included both enforcement and obfuscation-based techniques in their designs.

In addition to classifying defenses by type, we specify their level of sophistication, quantified by the ***strength*** or coverage of the defense and the ***phase*** during which it is employed. Strength captures the completeness to which a vulnerability is protected or, for obfuscation-based defenses, the entropy[2] of the system. Phase captures when a defense applies in the program's life cycle. For defenses introduced during hardware realization, we distinguish between *load-time* and *run-time* defenses. Namely, defenses instantiated at load-time consistently change the state space but do not effect execution. For example, ASLR [75] is a load-time address, since the program offsets are constant for each execution. Runtime defenses take on multiple values during program execution. Runtime defenses include ASLR variants that re-randomize the program offset during execution [16, 25, 37, 64, 103]. Due to differing sophistication, two defenses that address the same class of unintended states or transitions in the same manner may not be equivalent. Below, we detail these three defense techniques in more detail.

4.1 Avoidance of Vulnerabilities

Designers use testing or formal verification to avoid unintended states and ensure that these states are not reachable from the program-level. Figure 2(a) illustrates an unprotected system, where

[2]In this work, we define entropy as the number of key bits used to obfuscate unintended state. We denote obfuscation techniques as having low entropy if they use key sizes smaller than 32-bits or employ weak XOR ciphers.

unintended transitions are leveraged to shift the program from an intended state, shown in white, to an unintended state, shown in grey. Avoidance techniques, Figure 2(b), remove single vulnerabilities, eliminating *specific* unintended transitions in the implementation FSM.

For example, a program with a buffer overflow vulnerability will have a transition to a state where memory has been corrupted. Avoidance prevents an exploit by removing the single buffer overflow vulnerability (i.e., the single transition) to the corrupt state, through program testing. However, program testing is not exhaustive. Avoidance does not address the existence of spatial memory access errors as a whole. Other vulnerabilities may still exist, allowing for exploits that abuse vulnerabilities elsewhere in the source code. Rather than relying on program testing, programmers may employ formal verification to eliminate vulnerabilities. While formal verification is comprehensive, it currently does not scale to large programs. Additionally, avoidance defenses are only a durable protection until someone writes more code and introduces a new vulnerability. Last, since testing and formal verification techniques apply to source code or binaries, the avoidance approach cannot be easily extended to address vulnerabilities in the microarchitecture.

4.2 Enforcement of Program Semantics

Enforcement protections seek to remove all vulnerabilities of a particular type from an application by prohibiting the existence of a vulnerability altogether. To eliminate vulnerabilities, enforcement protections detect any uses of the vulnerability, sometimes effectively defining an undefined semantic (i.e., marking it as erroneous). With respect to Figure 2(c), enforcement creates barriers around parts of the implementation FSM. Rather than removing a single transition, as in the case of avoidance, enforcement provides significant obstacles to initiating the security exploits by prohibiting *all* unintended transitions of some type.

4.2.1 Enforcement During High-level Programming. At the programming language level, enforcement works to remove all instances of undefined semantics by fully defining these semantics within the programming language specification. This category includes memory-safe languages, like Java, that eliminate spatial and temporal memory errors. Program-level enforcement techniques are effective but may not be comprehensive. These techniques cannot secure microarchitectural assets; therefore, they are incomplete due to the nature of the underlying hardware. Additionally, these languages are sometimes implemented in an unsafe programming language, leaving possibilities for exploits. While fully defined programming languages are effective, we choose to focus on defenses that secure C or C++ programs in our study, since these programming languages are popular and used in many legacy applications. We assume it is less practical to re-implement these applications in a safe programming language, but we do acknowledge that this possibility exists.

4.2.2 Enforcement During Hardware Realization. Runtime enforcement protections prohibit vulnerabilities and undefined semantics by detecting their use during program execution. For example, processors like CHERI [104] enforce spatial memory access errors at runtime by triggering a security exception whenever these vulnerabilities are detected. Alternatively, **Control-flow Integrity (CFI)** [4] enforces memory access errors that overwrite jump targets by validating all control transfers against the intended FSM. CFI is complete with respect to its threat model, but incurs high performance overheads, motivating coarse-grained variants that proved to be susceptible to attackers [94]. Ultimately, once attacks evolved to no longer rely on the protected asset, enforcement defenses were easily bypassed. Our case-study in Section 5.1 shows evidence of this trend.

4.3 Obfuscation of Implementation Information

Finally, some protections obfuscate implementation information to make knowledge of the implementation FSM unreliable, either by making it difficult to acquire knowledge of the implementation or rendering this knowledge useless. With respect to Figure 2(d), an obfuscation defense conceals or obscures unintended states, increasing the sophistication of information acquisition needed by attackers to synthesize an exploit. Obfuscation defenses mitigate attacks by making the implementation information difficult (albeit not impossible) to acquire by making unintended states unpredictable. Our studies revealed that a majority of attacks require at least some knowledge of implementation information. Thus, obfuscation-based defenses are a promising method to mitigate a range of security attacks.

4.3.1 Obfuscation During Compilation. Compile-time defenses randomly instantiate a parameter during program compilation. Thus, this parameter has the same value every time a program runs. However, the same program on another machine may have a different instantiation. These compiler-based protections are used most commonly for binary obfuscation. For example, Polyverse [3] provides a compiler that randomizes register usage, function locations, import tables, and other targets per compilation. Namely, each binary produced by Polyverse is unique while preserving the semantics of the intended program. This tool effectively mitigates control-flow exploits by obfuscating critical attack assets, like code location. Compiler-level obfuscation techniques apply one-time randomization. Thus, although the implementation information is initially unknown, it can be easily discovered by attackers as this information is constant for each instance of the program. Therefore, these defenses are the most susceptible to information leakage and derandomization attacks (Section 5.2). As the time of obfuscation is moved closer to program execution, discovering a randomized parameter becomes much more difficult.

4.3.2 Obfuscation During Hardware Realization. Defenses during hardware realization randomly instantiate an undefined semantic for each run of a program. Thus, any knowledge the attacker gains about the implementation of a obfuscated semantic cannot be used to attack another instance of that program. The most common example of load-time randomization defenses is ASLR [75]. Many control-flow attacks rely on knowledge of code locations. Finding the address of a target function can be done trivially by disassembling the binary. Yet, with ASLR [75], which randomizes the location of objects at load time, the address of a target function changes per execution. Disassembly no longer reveals the true code location, making the attack more difficult. Unfortunately, clever probing of shared state can still defeat most one-time randomization defenses, including 64-bit ASLR, since the randomized information is constant at runtime [17, 36, 42, 90]. These derandomization attacks are discussed in Section 5.2.

4.3.3 Obfuscation During Runtime Execution. Defenses that obfuscate assets at *runtime* have appeared in response to derandomization attacks. Within these defenses, the protected asset (e.g., data location) takes many forms during a single execution of the program. While these defenses are still instantiated during hardware realization, we distinguish this from the prior section, because the value of the randomized information changes multiple times during the program's lifetime. For example, **Timely Address Space Randomization (TASR)** [16] relocates the code segment during runtime after every system call, effectively obfuscating the location of code objects. This defense aims to mitigate memory leaks that derandomized traditional ASLR by changing code location immediately after a leak could occur (via a system call). Other runtime obfuscation defenses change assets at a constant rate, such as every 50 ms, rather than in response to another event. Generally, these defenses employ higher entropy and more frequent obfuscation schemes to create a tight time limit that probing techniques must overcome to be successful [25, 37, 103].

4.4 Summary of Defenses

Defenses mitigate security exploits in three primary ways: *(i)* Avoidance of vulnerabilities, *(ii)* Enforcement of the implementation FSM to prevent vulnerabilities from being leveraged for attacks, and *(iii)* Obfuscation of implementation information to prevent attackers from reliably exploiting the implementation FSM. These approaches are illustrated in Figure 2. With respect to our framework, we classify defenses by the phase during which it is employed, making a distinction between load-time and runtime defenses within hardware realization, and their coverage of vulnerability sources. In Section 5, we discuss how prominent mitigation techniques fit within this model.

5 CASE STUDIES OF ATTACK GENEALOGIES

In Section 2, we showed how the program life cycle introduces vulnerabilities in its different phases. Following this, Section 3 went deeper into how exploits use these vulnerabilities by obtaining implementation information that can then be used to transition the implementation FSM to unintended states. Section 4 then outlined categories of defenses in terms of how they prevented the attacker from gaining control through transitioning to these unintended states.

In this section, we present a deeper dive into the case studies of four prominent attack genealogies: ***control-flow attacks***, ***derandomization attacks***, ***timing side channels***, and ***transient execution attacks***. The first two genealogies compromise program integrity by hijacking the victim's control flow or, in the case of derandomization attacks, do so after subverting a randomization-based defense. The later two genealogies compromise program confidentiality by using microarchitectural vulnerabilities to leak sensitive information. While we do not consider availability attacks within our case studies, our framework could be adjusted to capture these exploits, as discussed in Section 6.1. For each case study, we provide *(i)* An enumeration of prominent security exploits and the vulnerability sources they leverage, as related to our program life cycle framework through Tables 1 and 2, and *(ii)* An enumeration of defenses classified by their approach (Figure 2) and the vulnerability sources they protect. These case studies serve to illustrate:

- How the attacks and defenses for each case study fit into the framework of this article.
- How the attacks have evolved over time with increasingly sophisticated acquisition of implementation information by the attackers and the mechanisms used to acquire them.
- How the defenses have evolved over time with an increasing shift from avoidance to enforcement and obfuscation-techniques that eliminate access to large parts of the unintended state space and thus are potentially more durable.

5.1 Case Study: Control-flow Attacks

Control-flow attacks are characterized by attempts to redirect program execution from paths intended by the programmer. Control-flow attacks abuse a memory access error to transition the implementation FSM to an unintended state, traditionally this state being one that permits unauthorized access to the computer. Specifically, these exploits abuse a memory access error to overwrite control data (e.g., a return address or code pointer) with a target of their choosing. To do this reliably, attackers must carefully craft an attack payload using knowledge of implementation information. Specifically, all control-flow attacks require knowledge of the data layout to accurately corrupt the intended information. Additional implementation information that is needed is dependent on the chosen target; code injection necessitates knowledge of the representation of data pointers and code, whereas code reuse necessitates the location of a target function. We provide a chronological listing of attacks that have leveraged novel vulnerability sources in Table 3 and prominent control-flow attacks and defenses in Table 4. Below, we detail how these attacks

Table 3. Novel Vulnerability Sources in Control-flow Attacks

	Attacks	Vulnerability/Implementation Info.	Source	Acquisition
1996	**Stack Smashing (Stack Buffer Overflow) [8]**	Memory Access Error *(Corrupt Code Ptr.)*	Lang-level Und. Sem.	—
		Order of Data *(Stack)*	Compiler Choices	DOC, ENGR
		Location of Data *(Stack)*	Compiler Choices	ENGR
		Representation of Code	Compiler Choices	DOC
		Representation of Code Pointers	Compiler Choices	DOC
1997	**Return-into-libc**	Location of Code	Compiler Choices	ENGR
1999	**Heap Smashing (Heap Overflows) [98]**	Order of Data *(Heap)*	Compiler choices	DOC, ENGR
2001	**Heap Spray [40]**	Location of Data *(Heap)*	Compiler Choices	ENGR
2016	**Data-oriented Programming [46]**	Memory Access Error *(Corrupt Data)*	Lang-level Und. Sem.	—
		Representation of Data Pointers	Compiler Choices	DOC

We enumerate control-flow attacks that have leveraged a novel vulnerability or piece of implementation information (i.e., one that has not been used previously). We classify these vulnerability sources by their source and method of acquisition.

Table 4. Chronological Ordering of Control-flow Attacks and Defenses

	Attacks		MAE for CP	MAE for D	Code Contents	Order of Data	Loc. of Data	Loc. of Code	Rep. of Code	Rep. of CP	Rep. of DP
1996	**Stack Smashing (Stack Buffer Overflow) [8]**		●	○	◐	◐	◐	○	●	●	○
1997	**Return-into-libc [92]**		●	○	◐	◐	○	●	○	●	○
1999	**Heap Smashing (Heap Overflows) [98]**		●	○	◐	◐	◐	○	●	●	○
2001	**Heap Spray [40]**		●	○	◐	◐	◐	○	●	●	○
2007	**Return Oriented Programming (ROP) [81]**		●	○	◐	◐	○	●	○	●	○
2011	**Jump Oriented Programming (JOP) [18]**		●	○	◐	◐	○	●	○	●	○
2014	**Call Oriented Programming [24]**		●	○	◐	◐	○	●	○	●	○
2015	**Control-Flow Bending [23]**		●	○	◐	◐	○	●	○	●	○
2016	**Data Oriented Programming [46]**		○	●	◐	◐	○	○	○	○	●
	Defenses										
1997	**NX-Stack [91]**	*Runtime Enforcement*	◐	○	○	○	○	○	○	○	○
1998	**Stack Canaries [28]**	*Load-Time Obfuscation, Low Entropy*	◐	○	○	○	○	○	○	○	○
2003	**NX-Bit [87]**	*Runtime Enforcement*	◐	○	○	○	○	○	○	○	○
	Instruction Set Randomization (ISR) [14, 50]	*Compile-Time Obfuscation, Low Entropy*	○	○	◐	○	○	○	●	○	○
	Addr. Space Layout Randomization (ASLR) [75]	*Load-Time Obfuscation, Low Entropy on 32-bit*	○	○	○	○	●	●	○	○	○
	PointGuard [27]	*Load-Time Obfuscation, Low Entropy*	○	○	○	○	○	○	○	●	●
2009	**Control Flow Integrity (CFI) [4]**	*Runtime Enforcement*	●	○	○	○	○	○	○	○	○
2013	**kBouncer [74], ROPecker (2014) [26]**	*Runtime Enforcement*	◐	○	○	○	○	○	○	○	○
2014	**Code-Pointer Integrity (CPI) [60]**	*Runtime Enforcement*	●	○	○	○	○	○	○	○	○
	CHERI [104]	*Runtime Enforcement*	●	●	○	○	○	○	○	○	○
2018	**Random Embedded Secret Tokens (REST) [89]**	*Runtime Obfuscation, High Entropy*	●	●	○	○	○	○	○	○	○

We enumerate the vulnerabilities and implementation information leveraged/protected by prominent control-flow attacks and defenses. Each asset is either leveraged/protected fully (●), partially (◐), or not at all (○). For space driven brevity, we adopt the following abbreviations: memory access error (MAE), code pointer (CP), and data pointer (DP).

have evolved, from early work focused on exploiting the stack, to advanced attacks that leveraged existing code and non-control data to redirect execution.

5.1.1 Code Injection. The entire class of control-flow exploits originated from the first stack-smashing attack in the late 1990s [8]. Within this exploit, an attacker leverages a buffer overflow vulnerability on the stack to influence unintended state in two ways: to inject malicious code into the data segment, and to overwrite the return address with a pointer to the injected code. The target processor proceeds from this corrupt state, popping the return address and executing the malicious code to conclude the exploit. To craft a payload to synthesize this exploit successfully, the attacker requires the following knowledge of implementation information from compilation: *(i)* The order of data on the stack (to overflow the buffer), *(ii)* The representation of code (to inject code), *(iii)* The location of data on the stack (to redirect to injected code), and *(iv)* The representation of code pointers (to overwrite the return address). Additionally, unintended states in the implementation FSM must permit the execution of stack data. Attackers can derive knowledge of this implementation information from documentation of the system and target processor.

Two novel protections were promptly adopted to prevent stack smashing attacks. In 1997, the non-executable stack patch forbid the execution of data on the stack, effectively enforcing intended states in the implementation FSM after a memory access error occurs [91]. A year later, stack canaries leveraged obfuscation to detect linear buffer overflows by asserting that a secret value on the stack had not been corrupted during runtime [28]. While these prominent defenses protected the call stack, they were not comprehensive of all memory. Attackers quickly subverted these defenses by exploiting buffer overflows and performing code injection within the heap [40, 98]. Shortly afterward, the protections of the non-executable stack expanded to all data to fully eradicate code injection. Specifically, Intel introduced the **no-execute bit (NX-bit)** [87] to its X86 instruction-set architecture to forbid data execution, effectively preventing these unintended states from being entered due to memory corruption. In response, attackers adapted their techniques to exploit other unintended states and transitions still present on the target machine.

5.1.2 Code Reuse. With the demise of code injection, exploits morphed to *reuse* existing code on the victim device. Code reuse attacks originated from return-into-libc [92], a variant of the stack smashing exploit that redirects program execution to a library function. This attack is identical to the stack smashing exploit above except that, rather than requiring knowledge of data location and code representation, knowledge of *code location* is leveraged to redirect execution to an unintended library function. While library functions do exist in the implementation, the states used in return-into-libc are unintended, because this specific stack layout and execution pattern is not permitted by the intended FSM.

A decade later, **Return-oriented Programming (ROP)** [81] revisited code reuse attacks, exploiting existing code at a finer granularity. Rather than executing a single library function, ROP stitched together short code gadgets to form arbitrary programs that were just as expressive as what code injection achieved. This attack required identical implementation information as return-into-libc, relying heavily on *code contents* and *code location* to identify gadgets. Early enforcement-based protections profiled the call patterns of ROP attacks, triggering security exceptions when malicious execution was detected [26, 74]. However, these defenses were easily subverted by attacks that used call patterns outside of the assembled profile, for example, by executing gadgets that ended in jump instructions rather than returns [18, 24]. Other defenses attempted to fortify the code space, such as by preventing memory access errors from corrupting return

addresses (discussed below). In response, ROP evolved to exploit data. **Data-oriented Programming (DOP)** [46] manipulates non-control data, instead of function pointers, to influence indirect control flow and produce unexpected behavior. Hence, rather than requiring knowledge of the location of code or representation of code pointers, DOP solely relies on implementation information related to data and data pointers.

5.1.3 Defenses Against Control-flow Attacks. As shown in Table 4, defenses for control-flow attacks primarily fall into two categories: *(i)* Obfuscation of critical implementation information (e.g., location of code/data, representation of code, and representation of code/data pointers), and *(ii)* Enforcement of memory access errors. Obfuscation-based mitigations looked to fortify access to data and code locations, and the representation of pointers and code. Early obfuscation-based defenses aimed to obfuscate memory layout to prevent attackers from crafting pointers to injected code or existing code gadgets. ASLR [75] shifts the location of the stack, heap, and code in memory. Since this technique randomizes once, sophisticated implementation acquisition methods can be used to locate the obfuscated segments at runtime. This arms race led to an entire new genealogy of attacks, termed derandomization attacks, detailed in Section 5.2. Finer-grained randomization techniques [44, 53, 73] and protections that re-randomize at runtime [16] have been developed to combat these derandomization attempts. Obfuscation defenses also randomized the *representation of code* at either compile-time or load-time to thwart code injection and gadget discovery [14, 50, 72, 88]. Other techniques fortified code pointers by encrypting the *representation of pointers* [27].

Other defenses employ enforcement techniques to detect memory access errors and thwart control-flow attacks. **Control-flow Integrity (CFI)** [4] analyzes the trusted program for all possible, programmer-intended execution paths in the control-flow graph and restricts execution to this graph during runtime, effectively preventing memory access errors by restricting the acceptable values of code pointers. Although CFI is a robust enforcement mechanism, it comes with high overheads. Thus, adopted CFI implementations only trace a subset of control-flow edges [31, 51, 67], leaving them vulnerable to subversion [23, 35, 39, 94]. Subsequent defenses continued to fortify control data. Code Pointer Integrity [60] uses enforcement to stop memory access errors from influencing code pointers by isolating pointers in a protective memory region. Control-data Isolation [10] took a subtractive approach to achieve the same goal, removing all indirect control-flow instructions that can be influenced by user data (i.e., indirect jumps, calls, and returns). Other protections like CHERI [104] and REST [89] employed runtime mechanisms to detect memory access errors quickly.

5.2 Case Study: Derandomization Attacks

As we saw in Section 5.1, control-flow attacks leverage knowledge of compiler choices related to code and data layout to craft attack payloads. Mitigations have focused on *obfuscating* this critical implementation information to prevent attacks. ASLR [75] is one of few widely adopted randomization defenses. ASLR obfuscates memory layout at load-time, changing the position of the data and code segments to hide location information. However, while ASLR is effective, sophisticated implementation acquisition methods can be used to gain knowledge of the randomized implementation at runtime and subvert the defense. Thus, exploits have prevailed in the face of ASLR by using these derandomization tactics. This genealogy expands upon control-flow attacks and defenses to explore the advanced methods of implementation information acquisition used to evade obfuscation protections. We provide a chronological listing of derandomization attacks that have leveraged novel vulnerability sources in Table 5 and prominent attacks and defenses in Table 6. Below, we detail how these derandomization attacks have evolved, from exploiting memory leakage to advanced probing tactics that leverage timing side-channels.

Table 5. Novel Vulnerability Sources and Acquisition Methods in Derandomization Attacks

	Attacks	Vulnerability/Implementation Info.	Source	Acquisition
2013	**Just-in-time Code Reuse (JIT-ROP) [90]**	Location of Code	OS Choices	ENGR*
2014	**Blind ROP [17]**	Location of Code	OS Choices	PROBE*
2016	**Crash-resistant-oriented Programming [38]**	Location of Code	OS Choices	ENGR*

We enumerate derandomization attacks that have leveraged a novel vulnerability source or method of implementation information acquisition (denoted by *). We classify these assets by their source and method of acquisition. In this genealogy, code location originates from OS Choices, because ASLR [75] is in place.

Table 6. Chronological Ordering of Derandomization Attacks and Defenses

	Attacks		MAE for CP	Code Contents	Order of Data	Loc. of Data	Order of Code	Loc. of Code	Rep. of Code	Rep. of CP	Rep. of DP
2004	**Brute Force Attacks on ASLR [85]**		●	◐	◐	◐	○	●	○	●	○
2013	**Just-in-Time Code Reuse (JIT-ROP) [90]**		●	◐	◐	◐	○	●	●	●	○
2014	**Blind ROP [17]**		●	◐	◐	◐	○	●	○	●	○
2016	**Crash-Resistant Oriented Programming [38]**		●	◐	◐	◐	○	●	○	●	○
	Defenses										
2006	**Address Space Layout Permutation [53]**	*Load-Time Obfuscation, Low Entropy on 32-bit*	○	○	●	●	●	●	○	○	○
2014	**Oxymoron [12], Isomeron (2015) [32]**	*Load-Time Obfuscation, Low Entropy*	○	○	○	○	○	◐	○	●	○
	Execute-no-Read (XnR) [11]	*Runtime Enforcement*	○	○	○	○	○	○	○	●	●
2015	**Readactor [29]**	*Load-Time Obfuscation, Low Entropy*	○	○	○	○	○	◐	○	●	○
		Runtime Enforcement	○	○	○	○	○	○	○	●	◐
	Timely Addr. Space Randomization (TASR) [16]	*Runtime Obfuscation, Low Entropy*	○	○	○	●	○	●	○	○	○
2016	**RuntimeASLR [64]**	*Runtime Obfuscation, High Entropy*	○	○	○	●	○	●	○	○	○
	Remix [25]	*Runtime Obfuscation, Low Entropy*	○	○	○	○	◐	◐	○	○	○
	Shuffler [103]	*Runtime Obfuscation, Low Entropy*	○	○	○	○	●	●	○	◐	○
2019	**Smokestack [7]**	*Runtime Obfuscation, Low Entropy*	○	○	◐	◐	○	○	○	○	○
	Morpheus [37]	*Runtime Obfuscation, High Entropy*	○	○	○	●	○	●	●	●	●

We enumerate the vulnerabilities and implementation information leveraged/protected by prominent attacks and defenses. Each asset is either leveraged/protected fully (●), partially (◐), or not at all (○). For space driven brevity, we adopt the following abbreviations: memory access error (MAE), code pointer (CP), and data pointer (DP).

5.2.1 Disclosing Code Location via Memory Leaks. Attacks responded to the adoption of ASLR by derandomizing code location before synthesizing an exploit. Initial implementations of ASLR on 32-bit systems had low entropy and were trivially defeated by control-flow attacks that brute-force guessed the randomized memory layout [85]. With the deployment of ASLR on 64-bit systems, attacks needed more sophisticated derandomization methods to overcome the increased entropy. Exploits began to use the limited coverage of ASLR to subvert the defense. In particular, while ASLR randomizes the location of objects, it does not randomize the distance between objects in a particular segment. Thus, given a leaked pointer and knowledge of *relative addressing*, attackers can easily derandomize the address space. Variants of ROP integrated these derandomization techniques to subvert ASLR prior to an attack. JIT-ROP [90] uses a memory disclosure vulnerability to leak the *absolute* location of a code object, then uses this address to read code pages and follow

direct jumps to discover other parts of the code segment. Once enough of the code contents and location is known, an ROP payload is synthesized using just-in-time compilation. In addition to the assets leveraged by ROP [81] (discussed in Table 4), JIT-ROP uses knowledge of the *representation of code* to read code pages to uncover the locations of code gadgets at runtime.

To mitigate JIT-ROP, Oxymoron obfuscated direct branch targets to prevent code pages from leaking code locations [12]. Attacks that used indirect branch targets to infer code locations bypassed this protection [32]. Subsequent work extended randomization to all jump targets [32], adding a layer of indirection between code pointers and code location to confuse attackers. Other JIT-ROP mitigations worked to make code pages unreadable by enforcing different permissions for code pointers and data pointers (e.g., load/store operations can only use data pointers) [11]. These randomization and enforcement techniques have been combined in Readactor [29] to efficiently mitigate both direct and indirect forms of code page discovery with low performance overheads.

With the rise of new protections, attacks turned to more advanced acquisition techniques to learn code locations. Crash-resistant-oriented Programming [38] demonstrated techniques to disclose code pointers without crashing the victim process, allowing this attack to sidestep enforcement protections that triggered exceptions. To sidestep protections that prevented reading code pages, Blind-ROP [17] employed sophisticated probing techniques to learn pointer values. Specifically, Blind-ROP used a buffer overflow vulnerability to repeatedly overwrite a single byte of the return address on the stack, using the execution of the child process as an oracle to determine if the guessed byte was correct. Repeated probing of subsequent bytes disclosed the complete return address, derandomizing the ASLR offset and enabling a ROP attack. Some derandomization techniques leveraged timing side-channels to recover the ASLR offset. We discuss these attacks in Section 5.3.4 as they leverage microarchitectural vulnerabilities that are common to the timing side-channel attack genealogy.

5.2.2 Defenses with Re-Randomization. In response to derandomization attacks, defenses have employed re-randomization of critical implementation information, including re-randomization of the location and order of code/data objects, the representation of code, and the representation of pointers. Although the re-randomization of defenses provably adds one bit of entropy to existing systems [85], recent work has relied heavily on this technique to mitigate derandomization attacks. Re-randomization elevates the difficulty of obtaining implementation information by making it transient, forcing runtime acquisition techniques to learn and exploit information before it changes. New ASLR-variants re-randomized the *location of objects* dynamically, such as after every kernel I/O operation (TASR [16]) or processes fork (RuntimeASLR [64]). Both approaches serve to destroy leaked pointers before they are injected back into the system. Periodic re-randomization has also proven valuable. Morpheus [37] reduces performance overheads by leveraging hardware support to re-randomize memory during runtime, but only obfuscates the absolute location of code and data. Remix [25] obfuscates relative addressing in the code segment by reordering basic blocks within their respective functions, but does not randomize the location of function entries. Conversely, Shuffler [103] obfuscates the relative distance between functions by reordering them at runtime, but does not address intra-function relative distance. Smokestack [7] focuses solely on data, randomizing the stack layout to mitigate memory access errors. Ultimately, runtime re-randomization introduces chaos for attackers attempting to execute code gadgets.

5.3 Case Study: Timing Side-channel Attacks

Timing side-channel attacks leverage implementation information introduced during hardware realization to leak confidential data. These attacks abuse a *side channel*-a vulnerability originating from the physical processor model where information propagates to externally observable states

in an unintended manner. These observable states include shared state (e.g., cache contents) and readable auxiliary state (e.g., program timing). Attackers must have intimate knowledge of code contents to decipher how side channels reveal program secrets during execution. Furthermore, a significant number of timing side-channel attacks directly interfere with the victim program, leveraging *sharing-dependent program timing* to strategically reveal confidential information. We provide a chronological listing of attacks that leverage novel vulnerability sources in Table 7, as well as prominent attacks and defenses in Table 8. Below, we detail how these attacks have evolved, from early work that exploited program timing, to advanced attacks that leveraged external interference to leak secret information. We also discuss timing side-channel attacks that have sought to derandomize ASLR rather than leak cryptographic keys.

5.3.1 Timing Secret-dependent Operations. In 1996, Kocher observed that the control flow of cryptographic algorithms was key-dependent. Bits of the secret key influenced readable auxiliary states (e.g., program timing) in the implementation, enabling an observer with knowledge of the algorithm to recover these key bits. While the cryptographic primitives were sound, the naive source code implementations proved to be dangerous due to unintended fluctuations in execution time. Kocher demonstrated that secret key information could be obtained by feeding the program carefully designed inputs and measuring the *execution time* of the algorithm [58]. To prevent this exploit, constant-time algorithms with key-independent control flow were developed, effectively *avoiding* side channels in the implementation. In many cases, it was up to the programmer to develop constant-time source code and assure that compilation and hardware realization upheld their intent. This task was incredibly challenging as it required significant expertise to model the complexities of the compiler and hardware in one's notional computer model. Other defenses sought to automate this process [70]. While many constant-time implementations were developed, they were fragile as any small fluctuations in execution could break them.

Although cryptographic algorithms adopted data-independent control flow, some performed key-dependent memory accesses to critical data structures (e.g., substitution boxes), leaving them vulnerable to attack. Nearly ten years after Kocher's original work, Bernstein published a timing side-channel attack on AES that instead measured timing variations of *memory accesses* using knowledge of *microarchitectural structures* [15]. Bernstein observed capacity misses during key-dependent table lookups to infer which table entries were accessed. By iteratively choosing program input and observing memory composition via timing information, he was able to recover the secret key. This attack specifically observed differences in the timing of memory accesses due to *internal interference* in the victim program. Bernstein's attack inspired future side-channel exploits that observed timing differences caused by *external interference* provoked by the attacker.

5.3.2 Timing Memory Accesses. As few algorithms exhibited internal interference, attacks began to instigate timing differences by directly interfering with the victim program. This trend was shaped by widespread support for **simultaneous multithreading (SMT)** [95], which caused threads to share *all* microarchitecture structures in a processor. Thus, attackers could modify shared state in the implementation FSM by leveraging knowledge of microarchitectural sharing and the location of critical data. Then, attackers could use *sharing-dependent program timing* to observe the result of this external interference in one of two ways: via timing-based or access-based attacks.

In timing-based attacks, attackers modify state shared between the victim's and attacker's implementation FSM, then time the victim's execution to see the effects of their malicious interference. These attacks specifically exploit readable auxiliary state in the victim's implementation FSM to leak secrets. In access-based attacks, attackers similarly influence shared state, then allow

Table 7. Novel Vulnerability Sources in Timing Side-channel Attacks

	Attacks	Vulnerability/Implementation Info.	Source	Acquisition
1996	**Kocher's Timing Attacks... [58]**	Side Channels	Phys. Proc. Model	–
		Timing of Operations *(Execution Time)*	Microarch. Choices	PROBE
2005	**Bernstein's Timing Attack on AES [15]**	Microarch. Sharing *(L1 Cache)*	Microarch. Choices	ENGR, PROBE
		Timing of Operations *(Mem. Accesses)*	Microarch. Choices	PROBE
	Cache Missing, Prime+Probe, Evict+Time [71, 77]	Sharing-dependent Program Timing	Phys. Proc. Model	–
2007	**BTB Side Channel [6]**	Microarch. Sharing *(BTB)*	Microarch. Choices	ENGR, PROBE
	I-Cache Side Channel [5]	Microarch. Sharing *(I Cache)*	Microarch. Choices	ENGR, PROBE
2014	**Flush+Reload [107]**	Microarch. Sharing *(LLC)*	Microarch. Choices	ENGR, PROBE

We enumerate timing side-channel attacks that leverage a novel vulnerability or piece of implementation information (i.e., one that has not been used previously). We classify these vulnerability sources by their source and method of acquisition.

Table 8. Chronological Ordering of Timing Side-Channel Attacks and Defenses

			Side Channels	SD Prog. Timing	Code Contents	Timing of Op.	Loc. of Data	Loc. of Code	Microarch. Sharing
	Attacks								
1996	**Kocher's Timing Attacks... [58]**		●	○	◐	●	○	○	○
2005	**Bernstein's Timing Attack on AES [15]**		●	○	◐	●	○	○	◐
	Cache Missing, Prime+Probe, Evict+Time [71, 77]		●	●	◐	●	◐	○	◐
2007	**BTB Side Channel [6]**		●	●	◐	●	○	◐	◐
	I-Cache Side Channel [5]		●	●	◐	●	○	◐	◐
2013	**Attacks Against Kernel Space ASLR [47]**		●	●	◐	●	○	◐	◐
2014	**Flush+Reload [107]**		●	●	◐	●	◐	○	◐
2016	**Jump over ASLR [36]**		●	●	◐	●	○	◐	◐
2017	**ASLR⊕Cache (AnC) [42]**		●	●	◐	●	○	◐	◐
	Defenses								
2005	**Automated Constant-Time Programs [70]**	*Program-Time Avoidance*	◐	○	○	○	○	○	○
2007	**Random Permutation Cache [100]**	*Runtime Obfuscation, Low Entropy*	○	○	○	○	○	○	◐
	Partition Locked Cache [100]	*Runtime Enforcement*	○	◐	○	○	○	○	◐
2016	**Cache Allocation Technology (CAT) [43]**	*Runtime Enforcement*	○	◐	○	○	○	○	◐
2018	**Dynamically Allocated Way Guard (DAWG) [55]**	*Runtime Enforcement*	○	◐	○	○	○	○	●
	CEASER [78]	*Runtime Obfuscation, High Entropy*	○	○	○	○	○	○	◐
2019	**Data-Oblivious ISA Extensions (OISA) [108]**	*Runtime Enforcement*	●	○	○	○	○	○	○
2019	**Skewed-CEASER (CEASER-S) [79]**	*Runtime Obfuscation, High Entropy*	○	○	○	○	○	○	◐

We enumerate the vulnerabilities and implementation information leveraged/protected by prominent side channel attacks and defenses. Each asset is either leveraged/protected fully (●), partially (◐), or not at all (○). For space driven brevity, we adopt the following abbreviation: sharing-dependent (SD).

the victim program to execute. However, the attacker does not observe the victim's execution via auxiliary state. Rather, the attacker executes and times their own program (i.e., the attacker's implementation FSM) to see how the victim has modified shared state. Because the attacker

measures their own memory accesses, they can obtain timing information at a much finer granularity and higher fidelity than observing the victim.

In 2005, Percival and Osvik et al. independently formalized access-based attacks on RSA and AES that leveraged sharing in the L1 cache [71, 77]. These exploits first primed the cache by filling it with attacker data, essentially providing attackers with complete knowledge of the shared cache state. Afterwards, attackers surrendered execution to the victim process. The victim process loads confidential data into the cache, modifying the shared state in the attacker's implementation FSM. Once the attacker resumes control of the processor, they can recover these traces by timing their own memory accesses to infer shared state. Slow accesses signified a conflict miss, revealing that the victim program accessed data mapped to the same cache block. By compiling these access patterns, the secret key was successfully recovered. Osvik published a second attack, termed Evict+Time, that took a timing-based approach to break AES. Rather than probing memory and measuring the time of each access, the victim's overall execution time was measured after the cache was evicted. Slowed execution signified that the cryptographic algorithm used the evicted data, revealing the memory access patterns of the algorithm.

5.3.3 Beyond the L1 Cache. Attackers continued to uncover alternative side channels to observe a victim program's execution. Given implementation information about *microarchitectural sharing* in the **Branch Target Buffer (BTB)**, Aciicmez et. al observed the branching patterns of a cryptographic process [6]. Aciicmez also extended this technique to the I-Cache to similarly monitor the control flow of the victim process [5]. In both cases, the attacker was able to derive the outcome of a key-dependent branch condition, leaking information about the secret key. Timing side-channel exploits also expanded into lower levels of the memory hierarchy. A new cache attack called Flush+Reload [107] achieved high resolution in the **last-level cache (LLC)** by exploiting shared memory between two cores. This technique can be used to break process isolation and trace memory accesses of a victim. More importantly, this exploit broadened the attack surface from processes running on the same core to processes running on separate cores. Thus, disabling SMT could not stop this attack. Subsequent work demonstrated that Prime+Probe is also practical on the LLC [63].

5.3.4 Side-channel-based Derandomization. Timing side-channel attacks have also been used to derandomize ASLR [75]. In 2013, the I-Cache side-channel attack [5] was used to dismantle kernel-space ASLR [47]. Collisions in the I-Cache revealed the address of the system call handler and, through repetition, this information was combined to uncover bits of the KASLR offset. Jump-over-ASLR exploited a timing-side channel in the BTB to monitor collisions between the attacker's branch instructions and the victim program, eventually derandomizing the lower bits of ASLR [36]. The following year, AnC used Evict+Time [71] to observe the *page table walk* of the memory management unit. Ultimately, this attack was able to derive 28 bits of the ASLR offset in only 150 seconds [42]. This attack combined traditional side channels with a unique probing technique to recover more bits of ASLR than previous exploits in a short amount of time.

5.3.5 Defenses for Timing Side Channels. As shown in Table 8, defenses for timing side-channel attacks have broadly taken one of the following approaches: *(i)* Randomization of microarchitectural sharing, *(ii)* Enforcement of microarchitectural sharing/sharing-dependent behaviors via the partitioning of shared structures, *(iii)* Randomization of program timing via adding noise to timing infrastructures, or *(iv)* Avoidance/enforcement of side channels via the use of constant-time or data-oblivious programs. Novel cache architectures obfuscate microarchitectural sharing by

randomizing memory mappings within shared structures [100, 101]. CEASER [78] and Skewed-CEASER [79] recently further expanded obfuscation of sharing by using encrypted addresses for cache indexing. These two techniques made it improbable to craft an eviction set by dynamically remapping the caches with high entropy and were applicable for the LLC.

Other protections use enforcement to protect microarchitectural sharing and sharing-dependent program timing by prohibiting external interference in the cache. These mitigations include PL-Cache [100], which proposed secure cache replacement protocols that prevented confidential data from being evicted by other processes. Subsequent protections continued to address inter-process conflicts. **Cache Allocation Technology (CAT)** [43] dynamically partitioned the LLC, enforcing strict bounds between two processes to mitigate Flush+Reload attacks. This defense was extended to all set associative structures, including caches, in **Dynamically Allocated Way Guard (DAWG)** [55]. Recent work has revisited mitigating side channels via constant-time programs, instead enforcing these guarantees within the instruction-set architecture directly. In particular, **Data-oblivious ISA Extensions (OISA)** [108] detects and prevents confidential data from propagating to side channels at runtime, providing programmers with a direct mechanism to write constant-time code.

5.4 Case Study: Transient Execution Attacks

A particularly dominant class of timing side-channels exfiltrate secrets by observing the side-effects of transient instructions that operate on sensitive information. These attacks abuse *mis-peculation*-the spurious execution of an instruction that is eventually deemed unnecessary due to the emergence of a misprediction or fault. When mispeculation occurs, modern processors squash all in-flight instructions and revert the processor state. However, not *all* state is reverted. Residues in the caches and other shared microarchitectural structures remain, as these states do not affect program correctness. This oversight establishes a *side channel* for third parties with adequate microarchitectural knowledge to observe program execution. Transient execution attacks are broadly categorized into two classes: those that cause mispredictions by polluting the contents of the branch predictor, and those that leverage faults to indirectly squash instructions [22]. We provide a chronological listing of attacks that have leveraged novel vulnerability sources in Table 9 and prominent transient execution attacks and defenses in Table 10. Below, we detail how this important new class of attacks exploits microarchitectural design choices in sophisticated ways.

5.4.1 Prediction-based Attacks. Spectre [57] directly influences the misprediction of instructions by exploiting *sharing-dependent program execution* in the BTB. Leveraging knowledge of *microarchitectural sharing*, the attacker trains the BTB to poison the shared predictor and force the misprediction of a specific branch in the victim's program. In Spectre V1, the victim mispredicts a conditional branch that makes a secret-dependent memory access, leaving residues in the cache. The evidence of this transient execution is later recovered by the attacker via either Prime+Probe [77] or Flush+Reload [107]. We show the vulnerabilities and implementation information leveraged by Spectre V1 in Table 10, which is comprised of the aforementioned assets, as well as the location of the spectre gadget and the assets used by Prime+Probe/Flush+Reload. Spectre V2 forces a misprediction on an indirect branch, enabling the attacker to redirect program execution to *any* unintended code of their choosing. This exploitation mimics ROP [81] and would require a similar set of attack assets.

Mispeculation provides attackers an avenue to directly influence the execution of a victim program without requiring program-level vulnerabilities. However, because mispeculated

Table 9. Novel Vulnerability Sources in Transient Execution Attacks

	Attacks	Vulnerability/Implementation Info.	Source	Acquisition
2018	**Spectre [57]**	Mispeculation *(Prediction-based)*	Phys. Proc. Model	–
		Sharing-dependent Prog. Exec. *(BTB)*	Phys. Proc. Model	–
	Meltdown [62]	Mispeculation *(Fault-based)*	Phys. Proc. Model	–
	Foreshadow [21]	Microarch. Sharing *(SGX Cache)*	Microarch. Choices	ENGR, PROBE
2019	**RIDL [96], Fallout [69]**	Microarch. Sharing *(Store Buffer)*	Microarch. Choices	ENGR, PROBE

We enumerate transient execution attacks that have leveraged a novel vulnerability or piece of implementation information (i.e., one that has not been used previously). We classify these vulnerability sources by their source and method of acquisition.

Table 10. Chronological Ordering of Transient Execution Attacks and Defenses

	Attacks		Mispeculation	SD Prog. Exec.	Side Channels	SD Prog. Timing	Code Contents	Timing of Op.	Loc. of Data	Loc. of Code	Microarch. Sharing
2018	**Spectre V1 [57]**		●	●	●	●	◐	●	◐	◐	◐
	Meltdown [62]		●	○	●	●	◐	●	◐	○	◐
	Foreshadow [21]		●	○	●	●	◐	●	◐	○	◐
2019	**RIDL [96], Fallout [69]**		●	○	●	●	◐	●	◐	○	◐
	Defenses										
2018	**Retpoline [41, 48]**	*Program-Time Enforcement*	◐	○	○	○	○	○	○	○	○
	InvisiSpec [105]	*Runtime Enforcement*	◐	○	◐	○	○	○	○	○	○
2019	**Speculative Taint Tracking [82], NDA [102]**	*Runtime Enforcement*	●	○	◐	○	○	○	○	○	○
	CleanupSpec [82]	*Program-Time Enforcement*	◐	○	◐	○	○	○	○	○	○

We enumerate the vulnerabilities and implementation information leveraged/protected by prominent transient execution attacks and defenses. Each asset is either leveraged/protected fully (●), partially (◐), or not at all (○). For space driven brevity, we adopt the following abbreviation: sharing-dependent (SD).

instructions are eventually squashed, this hardware-level vulnerability can only be used to execute short instruction sequences. Spectre-style exploits have since expanded to other microarchitectural structures, including the pattern history table, return stack buffer, and memory disambiguation logic [45, 56, 59, 65].

5.4.2 Fault-based Attacks. Rather than influencing the misprediction of an instruction, Meltdown-style attacks rely on faulty operations that squash transient instructions [22]. Meltdown abuses the fact that memory permissions in specific Intel processors are not checked until instruction commitment [62]. Thus, memory accesses and other invalid operations are performed long before they are deemed faulty. Meltdown leverages this property to leak kernel memory by performing transient kernel reads, which are later recovered through cache timing side-channels like FLUSH+RELOAD [107]. Because Meltdown does not influence program execution via shared state, it does not leverage sharing-dependent program execution like Spectre [57].

Foreshadow [21] similarly exploited delayed permissions checks to subvert Intel SGX [68]. Specifically, Foreshadow makes a transient access to enclave memory during an address

translation with a faulty page-table lookup. By using timing side-channels to observe the cache state after the fault, attackers were able to derive SGX enclave secrets. Fault-based attacks expanded to include variants that raise exceptions due to accesses to system registers [2] or lazy floating-point register state switching [93], and have extended into the store buffer [69, 96].

5.4.3 Defenses for Transient Execution Attacks. To combat transient execution attacks, defenses have focused on addressing the hardware-level constructs that permit the exploit. Specifically, defenses have broadly taken two approaches: *(i)* Limiting the speculative execution of instructions, classified as either avoidance or enforcement of mispeculation, or *(ii)* Eliminating the side-effects of transient instructions, classified as the enforcement of mispeculation and side channels.

Retpoline [41, 48] uses software enforcement techniques to prevent the speculative execution of vulnerable instructions. Specifically, Retpoline uses a return trampoline to prevent the processor from speculating on an indirect jump, thus prohibiting attackers from steering code during speculation. Other mitigations address the microarchitectural side-effects left by transient execution, preventing this unintended information from propagating to shared structures. We classify this technique as the enforcement of mispeculation and side-channels, as it prevents sensitive information from propagating to unintended outlets (i.e., side channel vulnerability) during mispeculation. InvisiSpec [105] prevents transient loads from propagating to the cache state, whereas similar defenses monitor and restrict the propagation of speculative data from reaching any known covert channels [102, 109]. Other protections try to address covert channels through enforcement and partitioning [52, 55, 83]. Rather than preventing the propagation of sensitive information, CleanupSpec [82] *undoes* microarchitectural residues that occur during mispeculation, essentially reverting to the same state in the implementation FSM before mispeculation. Thus, even though transient execution occurs, the processor truly reverts to the previous state, preventing attackers from exfiltrating secrets.

6 CONCLUDING THOUGHTS

In this systematization-of-knowledge article, we examined how vulnerability sources are introduced throughout the program life cycle, and how these sources, along with implementation information, are used by attackers to construct security exploits. These vulnerability sources arise from programming, compilation, and hardware realization, and introduce states in the implementation FSM that are *unintended* relative to the programmer's intended FSM. Security exploits leverage these sources to transition the implementation FSM to an unintended state to carry out an attack. Further, through the case studies of four important genealogies, we saw the evolution and increasing sophistication of the implementation information used for attacks and the methods to acquire this information, alongside a corresponding increase in the sophistication of defenses. We conclude by drawing on our framework and genealogies to point to key trends in defenses that hold increasing promise for the future.

6.1 Uses and Limitations of this Framework

This article aims to address the significant challenge of understanding the large space of security exploits and defenses by presenting an organization of security vulnerabilities as they arise throughout the program life cycle. This framework is designed with the purpose of conceptual modeling to aid researchers in developing more durable defenses that protect against wider classes of attacks. Specifically, this framework provides researchers a way to organize vulnerability sources across phases and sources within the program life cycle, identify similarities across different exploits, and model the ways in which defenses address these vulnerability sources. We do not intend for this framework to be used to model security attacks and defenses using design exploration tools. In

fact, given the complexity of the state space of our proposed FSM framework, such modeling may very well be infeasible and thus is not the intent of this work.

As presented in this article, the program life cycle framework captures security exploits that compromise program confidentiality or integrity. Availability exploits, such as denial-of-service attacks, are beyond the scope of our current framework. However, some denial-of-service attacks are easy to consider in this framework. For example, a denial-of-service attack that uses a malicious packet to crash a program is clearly transitioning the program into an unintended state, in alignment with our proposed framework. Other availability exploits are more complex and require additional consideration, such as those that flood a server with packets to oversaturate server capacity and hinder performance. While the results of such an attack are also unintended by the programmer, they require more careful consideration of the programmer's intended FSM to fully model how these exploits violate unintended states and transitions.

Finally, our framework uses auxiliary state (Section 2.3.3) to capture time and occurrence count, enabling us to model analog effects. However, auxiliary state is hard to discover in advance of deploying a system, as these effects are not intentional. Rather, these states are often determined *post-facto*, only after their value has been demonstrated. This makes it difficult to apply this modeling ability to the practical identification of vulnerability sources and defenses.

6.2 Trends and Suggestions for Defenses

Mitigating attacks through the *enforcement* of vulnerabilities remains a popular approach. Certainly, many solutions exist to enforce memory safety through the detection of memory access errors [60, 104]. Additionally, the enforcement of memory permissions has seen steady advancements, from the NX-bit stack protections [91], to x86 SMAP and SMEP [49], and more recently Intel SGX [68]. Remaining program-level vulnerabilities (e.g., uninitialized values and integer overflow) can be detected using generalized debugging tools, like as UBSan [9] and ASan [84]. However, unless these enforcement mechanisms are supported by specialized hardware, performance overheads become high, leaving most code running in the wild with few enforcement protections against the exercising vulnerabilities.

For *obfuscation* defenses, two trends have emerged: *(i)* The movement toward the randomization of multiple pieces of implementation information, and *(ii)* The application of finer-grained randomization defenses. Initially, randomization was reserved for the location of objects (e.g., ASLR [75]). But, over time, randomization defenses have expanded to include the representation of code [14], representation of pointers [27, 37], microarchitectural sharing [78], timing of operations [66], relative location of functions [103], and the order of stack variables [7]. These techniques garner broad appeal, since they (at least partially) address a wide range of attacks that utilize the randomized implementation information. Furthermore, defenses have also trended toward more fine-grained, runtime randomization of implementation information. In its most course-grained form, randomization is applied at compile-time or load-time. Thus, long periods are available to probe these randomized systems, giving attackers a foothold to discover the obfuscated implementation information. Derandomization attacks of this nature have been extremely successful (e.g., Blind ROP [17] and AnC [42]). Consequently, defenses have begun to aggressively re-randomize during execution in hopes that this approach will defeat malicious attempts to acquire implementation information at runtime.

6.2.1 Cost Trade-offs. While defenses may appear promising or protect against a wide array of vulnerability sources, the costs of these defenses must also be considered. In this context, cost may refer to runtime performance overheads, complexity of software modifications, complexity of hardware modifications, silicon area and power overheads, cost to the programmer to use the defense,

and so on. Existing security solutions that are deployed in practice are those that presented the best cost trade-offs at that time, otherwise they would have been dismissed. For example, ASLR [75] was one of the first widely deployed control-flow defenses and had very low cost. However, ASLR has since proven to be non-durable, as many exploits have prevailed despite it by leveraging advanced derandomization attacks (Section 5.2). Many ASLR variants have since been proposed, but none have been widely deployed.

Some security solutions are known but deemed impractical due to their high costs, including some discussed in this article (e.g., constant-time execution or complete isolation for timing side channels). It is possible that these high-cost defenses will never be deployed, or will only be deployed when there is an attack so pervasive that going without them is intolerable. It is often the case that security can be sacrificed to lessen costs, or that certain costs can be used to alleviate others. For example, modified computer hardware is often leveraged to reduce the performance overheads of runtime defenses [37]. In this case, hardware complexity and silicon area and power overheads are increased to lessen performance overheads. The framework proposed in this article serves to aid the development of efficient, comprehensive defenses by modeling similarities in vulnerability sources across prominent security attacks. To assess the cost trade-offs of new defenses, it is also critical to understand what threat model is necessary and what costs are tolerable to developers and consumers. Both are incredibly challenging tasks that warrant future work.

6.2.2 Suggestions for Future Defenses. Our study reveals two key properties that can inform how to best approach secure system design in the future: *(i)* A small amount of implementation information is used widely across numerous security attacks, and *(ii)* The classes of vulnerabilities grow very slowly, with only a few new vulnerabilities introduced per decade. Given these observations, defenses that focus on enforcing or obfuscating *multiple* pieces of implementation information have the potential to be both effective and durable, as broad swaths of the attack landscape will be disrupted. In contrast, avoidance defenses that focus on patching vulnerabilities will become increasingly ineffective and non-durable. Their ineffectiveness arises from the challenge of finding all vulnerabilities, which requires formal methods that do not scale with large systems. Additionally, the introduction of new code can introduce a new vulnerability, and no strategies exist for finding yet-undiscovered vulnerabilities.

While both enforcement and obfuscation are appropriate techniques for protecting vulnerabilities, there is no clear winning approach in our mind. Enforcement defenses can be made sound and complete, unlike obfuscation defenses, which provide only probabilistic security guarantees. Yet, enforcement defenses may conflict with non-malicious programs that use undefined semantics [104], while obfuscation techniques can often comply with these requirements [37]. Perhaps the defense approach that emerges predominantly will depend on the efficiency of the defense and to what extent they impact programmers that utilize these machines.

For vulnerabilities introduced during hardware realization, protections have focused largely on randomization-based techniques. We see randomization defenses trending toward higher-entropy implementations to increase their overall strength against brute-force and side-channel attacks. As detailed in Section 5.2, derandomization attacks have progressed to the point where randomized semantics can be recovered with mere minutes of probing. Higher entropy protections will diminish the likelihood of guessing randomized information and, while derandomization attacks are still be possible, they will take much more time. Thus, combining high-entropy defenses with fast rates of re-randomization will undercut the expected time of derandomization attacks and ensure that these systems can never be successfully penetrated through accessing implementation information.

Finally, our framework highlights the importance of modeling auxiliary states, such as observable timing or occurrence counters, to capture analog effects that affect visible state, like charge loss. Thus far, such auxiliary states have been determined *post-facto*, only after their value has been demonstrated. An important defense strategy would be to *anticipate* auxiliary states that could be exploited in the future-a very hard problem-but one that must be tackled in this ever-evolving arms race between attacks and defenses.

REFERENCES

[1] Mitre. 2001. CVE-2001-0144: SSH CRC-32 Compensation Attack Detector Vulnerability. Available from MITRE, CVE-ID CVE-2001-0144. Retrieved from cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2001-0144.

[2] Intel. 2018. Intel Analysis of Speculative Execution Side Channels. Retrieved from https://software.intel.com/security-software-guidance/api-app/sites/default/files/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf.

[3] Polyverse. 2020. Polyverse. Retrieved from https://polyverse.com.

[4] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*. ACM, New York, NY, 340–353. https://doi.org/10.1145/1102120.1102165

[5] Onur Aciiçmez. 2007. Yet another microarchitectural attack: Exploiting I-cache. In *Proceedings of the ACM Workshop on Computer Security Architecture*. ACM, 11–18.

[6] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. Predicting secret keys via branch prediction. In *Cryptographer's Track at the RSA Conference*. Springer, 225–242.

[7] Misiker Tadesse Aga and Todd Austin. 2019. Smokestack: Thwarting DOP attacks with runtime stack layout randomization. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO'19)*. 26–36.

[8] Aleph One. 1996. Smashing the Stack for Fun and Profit. Retrieved from http://phrack.org/issues/49/14.html.

[9] Apple Corporation. 2018. Undefined Behavior Sanitizer. Retrieved from https://developer.apple.com/documentation/code_diagnostics/undefined_behavior_sanitizer.

[10] William Arthur, Ben Mehne, Reetuparna Das, and Todd Austin. 2015. Getting in control of your control flow with control-data isolation. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO'15)*. 79–90. https://doi.org/10.1109/CGO.2015.7054189

[11] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. 2014. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1342–1353.

[12] Michael Backes and Stefan Nürnberger. 2014. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *Proceedings of the USENIX Security Symposium*. 433–447.

[13] Julian Bangert, Sergey Bratus, Rebecca Shapiro, and Sean W. Smith. 2013. The page-fault weird machine: Lessons in instruction-less computation. In *Proceedings of the 7th USENIX Workshop on Offensive Technologies*.

[14] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. 2003. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS'03)*. ACM, New York, NY, 281–289. https://doi.org/10.1145/948109.948147

[15] Daniel J. Bernstein. 2005. Cache-timing attacks on AES. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[16] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 268–279.

[17] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking blind. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'14)*. IEEE, 227–242.

[18] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 30–40.

[19] Hans-Juergen Boehm. 2011. How to miscompile programs with "Benign" data races. In *Proceedings of the USENIX Workshop on Hot Topics in Parallelism (HotPar'11)*. 3–3.

[20] Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, and Anna Shubina. 2011. Exploit programming: From buffer overflows to weird machines and theory of computation. *USENIX; login* 36, 6 (2011).

[21] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with

Transient Out-of-Order Execution. In *Proceedings of the 27th USENIX Security Symposium (USENIXSecurity'18)*. USENIX Association, Baltimore, MD, 991–1008.

[22] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A systematic evaluation of transient execution attacks and defenses. In *Proceedings of the 28th USENIX Security Symposium (USENIXSecurity'19)*. 249–266.

[23] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Security Symposium (USENIXSecurity'15)*. 161–176.

[24] Nicholas Carlini and David Wagner. 2014. ROP is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium (USENIXSecurity'14)*. 385–399.

[25] Yue Chen, Zhi Wang, David Whalley, and Long Lu. 2016. Remix: On-demand live randomization. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY'16)*. ACM, New York, NY, 50–61. https://doi.org/10.1145/2857705.2857726

[26] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H. Deng. 2014. ROPecker: A generic and practical approach for defending against ROP attacks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'14)*. Internet Society, Reston, VA. https://doi.org/10.14722/ndss.2014.23156

[27] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. 2003. Pointguard TM: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th conference on USENIX Security Symposium*, Vol. 12. USENIX Association, Berkeley, CA, 91–104.

[28] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. 1998. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium (SSYM'98)*. 5–5.

[29] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical code randomization resilient to memory disclosure. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'15)*. IEEE, 763–780.

[30] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. 2019. A complete formal semantics of x86-64 user-level instruction set architecture. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1133–1148.

[31] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. 2015. HAFIX: Hardware-assisted flow integrity extension. In *Proceedings of the 52nd Annual Design Automation Conference (DAC'15)*. ACM, New York, NY, Article 74, 6 pages. https://doi.org/10.1145/2744769.2744847

[32] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, and Fabian Monrose. 2015. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'15)*.

[33] Vijay D'Silva, Mathias Payer, and Dawn Song. 2015. The correctness-security gap in compiler optimization. In *Proceedings of the IEEE Security and Privacy Workshops*. IEEE, 73–87.

[34] Thomas F. Dullien. 2020. Weird machines, exploitability, and provable unexploitability. *IEEE Trans. Emerg. Topics Comput.* 8, 2 (2020), 391–403.

[35] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 901–913.

[36] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 40.

[37] Mark Gallagher, Lauren Biernacki, Shibo Chen, Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Austin Harris, Zhixing Xu, Baris Kasikci, Valeria Bertacco, Sharad Malik, Mohit Tiwari, and Todd Austin. 2019. Morpheus: A vulnerability-tolerant secure architecture based on ensembles of moving target defenses with churn. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. ACM, New York, NY, 16. https://doi.org/10.1145/3297858.3304037

[38] Robert Gawlik, Benjamin Kollenda, Philipp Koppe, Behrad Garmany, and Thorsten Holz. 2016. Enabling client-side crash-resistance to overcome diversification and information hiding. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'16)*.

[39] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of control: Overcoming control-flow integrity. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 575–589.

[40] Shashank Gonchigar. 2007. ANI vulnerability: History repeats. *SANS Institute Information Security Reading Room* (Oct. 2007), 51. Retrieved from https://www.sans.org/reading-room/whitepapers/threats/ani-vulnerability-history-repeats-1926.

[41] Google [n.d.]. *Retpoline: A software construct for preventing branch-target-injection.* Google. Retrieved from https://support.google.com/faqs/answer/7625886.
[42] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the line: Practical cache attacks on the MMU. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'17)*, Vol. 17. 13.
[43] Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. 2016. Cache QoS: From concept to reality in the Intel Xeon processor E5-2600 v3 product family. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. IEEE, 657–668.
[44] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. 2012. ILR: Where'd my gadgets go? In *Proceedings of the IEEE Symposium on Security and Privacy*. 571–585. https://doi.org/10.1109/SP.2012.39
[45] Jann Horn. 2018. speculative execution, variant 4: Speculative store bypass. Retrieved from https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.
[46] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-oriented programming: On the expressiveness of non-control data attacks. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'16)*. IEEE, 969–986.
[47] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 191–205.
[48] Intel [n.d.]. *Retpoline: A Branch Target Injection Mitigation.* Intel. Retrieved from https://software.intel.com/sites/default/files/managed/1d/46/Retpoline-A-Branch-Target-Injection-Mitigation.pdf.
[49] Intel Corporation. 2019. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1.* Intel Corporation, Santa Clara, CA. Order No. 253668-070US.
[50] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. 2003. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS'03)*. ACM, New York, NY, 272–280. https://doi.org/10.1145/948109.948146
[51] John Kennedy and Michael Satran. 2018. Control Flow Guard—Windows Applications | Microsoft Docs. Retrieved from https://docs.microsoft.com/en-us/windows/desktop/secbp/control-flow-guard.
[52] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2019. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. In *Proceedings of the 56th ACM/IEEE Design Automation Conference (DAC'19)*. IEEE, 1–6.
[53] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. 2006. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE, 339–348.
[54] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA'14)*. IEEE Press, Piscataway, NJ, 361–372. Retrieved from http://dl.acm.org/citation.cfm?id=2665671.2665726.
[55] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A defense against cache timing attacks in speculative execution processors. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*. IEEE, 974–987.
[56] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative buffer overflows: Attacks and defenses. Retrieved from https://arXiv:cs/1807.03757.
[57] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19)*.
[58] Paul C. Kocher. 1996. Timing attacks on implementations of Die-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology| Crypto*, Vol. 96. 104113.
[59] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks Using the Return Stack Buffer. In *Proceedings of the 12th USENIX Conference on Offensive Technologies (WOOT'18)*. USENIX Association, Berkeley, CA, 3–3.
[60] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-pointer integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. 147–163.
[61] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming undefined behavior in LLVM. *ACM SIGPLAN Notices* 52, 6 (2017), 633–647.
[62] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*. USENIX Association, Baltimore, MD, 973–990. Retrieved from https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[63] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-level cache side-channel attacks are practical. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 605–622.

[64] Kangjie Lu, Wenke Lee, Stefan Nürnberger, and Michael Backes. 2016. How to make ASLR win the clone wars: Runtime re-randomization. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'16)*.

[65] Giorgi Maisuradze and Christian Rossow. 2018. Ret2Spec: Speculative execution using return stack buffers. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*. ACM, New York, NY, 2109–2122. https://doi.org/10.1145/3243734.3243761

[66] Robert Martin, John Demme, and Simha Sethumadhavan. 2012. TimeWarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12)*. 118–129.

[67] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. 2015. CCFI: Cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*. ACM, New York, NY, 941–951. https://doi.org/10.1145/2810103.2813676

[68] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13)*. ACM, New York, NY, Article 10, 8 pages. https://doi.org/10.1145/2487726.2488368

[69] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. 2019. Fallout: Leaking Data on Meltdown-resistant CPUs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS'19)*. Association for Computing Machinery, New York, NY, USA, 769–784. DOI: https://doi.org/10.1145/3319535.3363219

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

[71] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: The case of AES. In *Cryptographer's Track at the RSA Conference*. Springer, 1–20.

[72] Antonis Papadogiannakis, Laertis Loutsis, Vassilis Papaefstathiou, and Sotiris Ioannidis. 2013. ASIST: Architectural support for instruction set randomization. In *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security (CCS'13)*. ACM, New York, NY, 981–992. https://doi.org/10.1145/2508859.2516670

[73] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2012. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the IEEE Symposium on Security and Privacy*. 601–615. https://doi.org/10.1109/SP.2012.41

[74] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2013. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security'13)*. 447–462.

[75] PaX Team. 2003. PaX address space layout randomization (ASLR). Retrieved from http://pax.grsecurity.net/docs/aslr.txt.

[76] Jennifer Paykin, Eric Mertens, Mark Tullsen, Luke Maurer, Benoît Razet, Alexander Bakst, and Scott Moore. 2019. Weird machines as insecure compilation. Retrieved from https://arXiv:1911.00157.

[77] Colin Percival. 2005. Cache missing for fun and profit. Retrieved from http://css.csail.mit.edu/6.858/2014/readings/ht-cache.pdf.

[78] Moinuddin K. Qureshi. 2018. CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*. IEEE, 775–787.

[79] Moinuddin K. Qureshi. 2019. New attacks and defense for encrypted-address cache. In *Proceedings of the 46th International Symposium on Computer Architecture*. 360–371.

[80] John Regehr. 2010. A Guide to Undefined Behaviour in C and C++. Retrieved from https://blog.regehr.org/archives/213.

[81] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Info. Syst. Secur.* 15, 1, Article 2 (Mar. 2012), 34 pages. https://doi.org/10.1145/2133375.2133377

[82] Gururaj Saileshwar and Moinuddin K. Qureshi. 2019. CleanupSpec: An undo approach to safe speculation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 73–86.

[83] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. 2019. Efficient invisible speculative execution through selective delay and value prediction. In *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 723–735.

[84] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'12)*. 309–318.

[85] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*. ACM, 298–307.

[86] Rebecca Shapiro, Sergey Bratus, and Sean W. Smith. 2013. "Weird Machines" in ELF: A spotlight on the underappreciated metadata. In *Proceedings of the 7th USENIX Workshop on Offensive Technologies*.

[87] Gennadiy Shvets. 2018. Enhanced Virus Protection/Execute Disable Bit. Retrieved from http://www.cpu-world.com/Glossary/E/EVP_XD.html.

[88] Kanad Sinha, Vasileios P. Kemerlis, and Simha Sethumadhavan. 2017. Reviving instruction set randomization. In *Proceedings of the IEEE International Symposium on Hardware-oriented Security and Trust (HOST'17)*. IEEE Press, Piscataway, NJ, 21–28. https://doi.org/10.1109/HST.2017.7951732

[89] Kanad Sinha and Simha Sethumadhavan. 2018. Practical memory safety with REST. In *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*. 600–611. https://doi.org/10.1109/ISCA.2018.00056

[90] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'13)*. IEEE, 574–588.

[91] Solar Designer. 1997. Linux Kernel Patch from the Openwall Project: README. Retrieved from https://www.openwall.com/linux/README.shtml.

[92] Solar Designer. 1997. lpr LIBC RETURN exploit. Retrieved from http://insecure.org/sploits/linux.libc.return.lpr.sploit.html.

[93] Julian Stecklina and Thomas Prescher. 2018. LazyFP: Leaking FPU Register State Using Microarchitectural Side-Channels. Retrieved from https://arXiv:cs/1806.07480.

[94] M. Theodorides and D. Wagner. 2017. Breaking active-set backward-edge CFI. In *Proceedings of the IEEE International Symposium on Hardware-oriented Security and Trust (HOST'17)*. 85–89. https://doi.org/10.1109/HST.2017.7951803

[95] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. 1998. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of 25 years of the International Symposia on Computer Architecture (ISCA'98)*. ACM, New York, NY, 533–544. https://doi.org/10.1145/285930.286011

[96] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue in-flight data load. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'19)*.

[97] Julien Vanegue. 2014. The weird machines in proof-carrying code. In *Proceedings of the IEEE Security and Privacy Workshops*. IEEE, 209–213.

[98] w00w00. 1999. w00w00 on Heap Overflows. Retrieved from https://www.cgsecurity.org/exploit/heaptut.txt.

[99] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. Undefined behavior: What happened to my code? In *Proceedings of the Asia-Pacific Workshop on Systems*. ACM, 9.

[100] Zhenghong Wang and Ruby B. Lee. 2007. New cache designs for thwarting software cache-based side channel attacks. *ACM SIGARCH Comput. Architect. News* 35, 2 (2007), 494–505.

[101] Zhenghong Wang and Ruby B. Lee. 2008. A novel cache architecture with enhanced performance and security. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 83–93.

[102] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. 2019. NDA: Preventing speculative execution attacks at their source. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 572–586.

[103] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and deployable continuous code re-randomization. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 367–382.

[104] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA'14)*. IEEE Press, Piscataway, NJ, 457–468. Retrieved from http://dl.acm.org/citation.cfm?id=2665671.2665740.

[105] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. 2018. Invisispec: Making speculative execution invisible in the cache hierarchy. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*. IEEE, 428–441.

[106] Kaiyuan Yang, Matthew Hicks, Qing Dong, Todd Austin, and Dennis Sylvester. 2016. A2: Analog malicious hardware. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'16)*. IEEE, 18–37.

[107] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium (USENIXSecurity'14)*. 719–732.

[108] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. 2019. Data oblivious ISA extensions for side channel-resistant and high performance computing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'19)*.

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

Received June 2020; revised January 2021; accepted March 2021