2018 51st Annual IEEE/ACM International Symposium on Microarchitecture

CheckMate: Automated Synthesis of Hardware Exploits and Security Litmus Tests

Caroline Trippel
Princeton University
ctrippel@princeton.edu

Daniel Lustig
NVIDIA
dlustig@nvidia.com

Margaret Martonosi
Princeton University
mrm@princeton.edu

***Abstract*—Recent research has uncovered a broad class of security vulnerabilities in which confidential data is leaked through programmer-observable microarchitectural state. In this paper, we present CheckMate, a rigorous approach and automated tool for determining if a microarchitecture is susceptible to specified classes of security exploits, and for synthesizing proof-of-concept exploit code when it is. Our approach adopts "microarchitecturally happens-before" (μhb) graphs which prior work designed to capture the subtle orderings and interleavings of hardware execution events when programs run on a microarchitecture. CheckMate extends μhb graphs to facilitate modeling of security exploit scenarios and hardware execution patterns indicative of classes of exploits. Furthermore, it leverages relational model finding techniques to enable automated exploit program synthesis from microarchitecture and exploit pattern specifications.**

As a case study, we use CheckMate to evaluate the susceptibility of a speculative out-of-order processor to FLUSH+RELOAD cache side-channel attacks. The automatically synthesized results are programs representative of Meltdown and Spectre attacks. We then evaluate the same processor on its susceptibility to a different timing side-channel attack: PRIME+PROBE. Here, *CheckMate synthesized new exploits* that are similar to Meltdown and Spectre in that they leverage speculative execution, but unique in that they exploit distinct microarchitectural behaviors—speculative cache line invalidations rather than speculative cache pollution—to form a side-channel. Most importantly, our results validate the CheckMate approach to formal hardware security verification and the ability of the CheckMate tool to detect real-world vulnerabilities.

***Index Terms*—hardware security, automated verification, relational model finding, exploit synthesis, side-channel attacks**

I. INTRODUCTION

Starting with the January announcement of Meltdown [1] and Spectre [2], 2018 has been the year of the hardware security exploit. Meltdown and Spectre effectively enable an adversarial process running on a susceptible microarchitecture to leak privileged data (e.g., private kernel memory) with high accuracy. Both attacks hinge on the fact that speculatively executed instructions are capable of polluting CPU caches. By inducing speculative execution and subsequently performing the well-known cache timing side-channel attack, FLUSH+RELOAD, Meltdown and Spectre can leak data that was accessed while a processor was speculating.

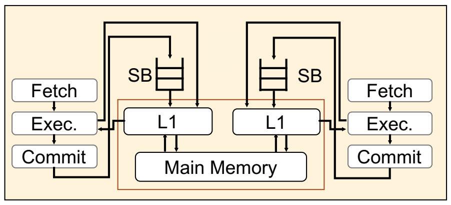
A steady stream of speculation-based attacks have been reported since the announcement of Meltdown and Spectre [3–12]. All of these attacks are structured similarly in that they leverage the effects of speculative execution on non-architectural state to make sensitive information available to software for extraction via some well-known side-channel attack (e.g., FLUSH+RELOAD). What is novel and surprising about these attacks is not the side-channel attack component, but rather their clever ability to create practical working exploits out of a variety of widely-implemented microarchitectural features.

This observation highlights the importance of automated verification techniques for identifying hardware behaviors that can be exploited to leak sensitive data into a side-channel. Because the state space is so large and designs are too complicated to reason about manually, hardware and system designers need the ability to reason rigorously about, and ideally even automatically generate, all possible ways in which microarchitectural features could be used to induce a side-channel on a given microarchitecture.

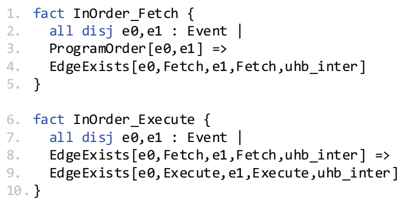
Auto-generating exploit scenarios requires techniques for modeling and analyzing them. Given that all of these speculation-based attacks rely on leaking information via non-architectural state (e.g., CPU caches), any techniques to analyze them must be able to account for implementation-specific optimizations that may not affect architecturally-visible state but that nevertheless result in variability across underlying microarchitectural executions. This variability is what can be detected with a simple side-channel attack. Thus, our approach, named CheckMate[1], adopts "microarchitecturally happens-before" (μhb) graphs from prior memory consistency model (MCM) work [13–17]. Originally, μhb graphs were designed to model microarchitecture-specific program executions as directed graphs. Nodes represent microarchitectural events of interest, such as a micro-op reaching some particular point in the microarchitecture (e.g., a store entering or exiting a store buffer); directed edges represent temporal "happens-before" relationships between nodes (e.g., a store enters the store buffer before it writes to the L1 cache).

CheckMate extends and adapts μhb graph analysis for security in new ways. To facilitate modeling of security exploit scenarios, we introduce the concept of an *exploit pattern*, which we formulate as a μhb sub-graph indicative of some class of exploits. Additionally, we leverage relational model finding (RMF) techniques to facilitate automated exploit program synthesis from CheckMate's inputs, which are shown in

[1]CheckMate is open source and publicly available at github.com/ctrippel/checkmate.



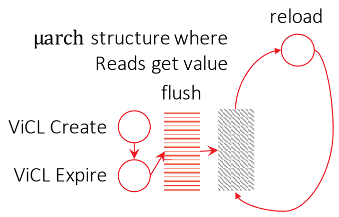
(a) Pedagogical 2-core, 3-stage, in-order μarch



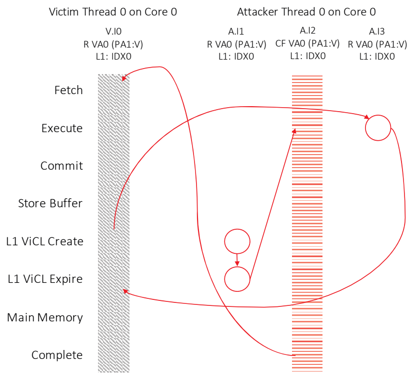
```
fact InOrder_Fetch {
  all disj e0,e1 : Event |
  ProgramOrder[e0,e1] =>
  EdgeExists[e0,Fetch,e1,Fetch,uhb_inter]
}

fact InOrder_Execute {
  all disj e0,e1 : Event |
  EdgeExists[e0,Fetch,e1,Fetch,uhb_inter] =>
  EdgeExists[e0,Execute,e1,Execute,uhb_inter]
}
```

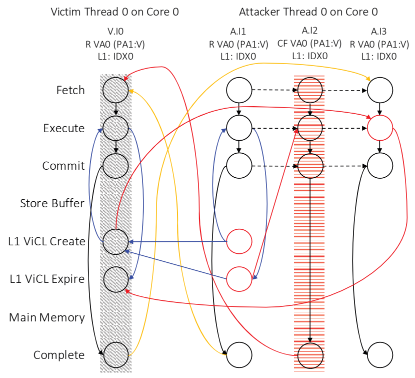
(b) μspec model excerpt corresponding to (a)



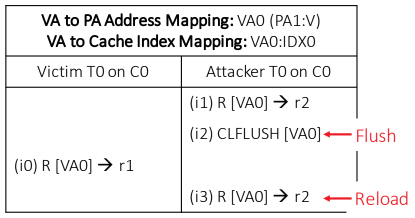
(c) FLUSH+RELOAD μhb pattern



(d) Exploit patterns are design-agnostic. For example, the exploit pattern in (c) specifies a malicious event sequence that can be attached to (i.e., be superimposed on) a μhb graph as here in (d). The corresponding full μhb graph is shown in (e).



(e) CheckMate-synthesized μhb graph exploiting (c)'s FLUSH+RELOAD pattern that only requires the presence of caches or similar structures (e.g., TLBs) that can by modeled with ViCLs (§III-A2). The μarch structure where reads bind their value in (a) is the Execute stage.



VA to PA Address Mapping: VA0 (PA1:V) VA to Cache Index Mapping: VA0:IDX0	
Victim T0 on C0	Attacker T0 on C0
	(i1) R [VA0] → r2
	(i2) CLFLUSH [VA0] ← Flush
(i0) R [VA0] → r1	
	(i3) R [VA0] → r2 ← Reload

(f) CheckMate-synthesized FLUSH+RELOAD security litmus test corresponding to the μhb graph in (e). The litmus test is extracted from the top of (e) and contains two threads from different (attacker and victim) processes that are time-multiplexed on the same physical core.

Fig. 1: CheckMate requires two inputs: (i) a microarchitecture specification, as in (b), which is an axiomatic description of a hardware design, as in (a), and its related OS support, and (ii) an axiomatic exploit pattern specification which can be thought of as a μhb sub-graph, as in (c). CheckMate evaluates the microarchitecture's susceptibility to the class of exploits and outputs μhb graphs representative of implementation-aware exploit program executions. Given (b) and (c) as inputs, CheckMate synthesized (e) and (f). CheckMate's inputs and outputs are explained in more detail in §III.

Fig. 1. CheckMate requires two inputs: a formal specification of a microarchitecture and its related OS support (Fig. 1b), and a formal description of an exploit pattern (Fig. 1c). Both are provided in an embedding of the μspec [15] domain-specific language (DSL) in the Alloy DSL [18]. From these inputs, CheckMate uses Alloy's RMF backend to synthesize programs that can induce the exploit pattern on the microarchitecture (Fig. 1f). CheckMate synthesizes small hardware-specific programs which represent attacker programs in their most abstracted form—i.e., *security litmus tests*, to borrow a term from the MCM literature. In §VII-C, we demonstrate the ease with which compact security litmus tests can be analyzed, and how they can be extended to full exploits when necessary. We summarize our contributions in the following paragraphs.

CheckMate: We develop CheckMate, an approach and automated tool for determining whether a microarchitecture is susceptible to a given class of security vulnerabilities.

μhb graphs for hardware security analysis: We make the important and non-obvious observation that the event ordering issues present in hardware MCM analysis are similar to those relevant for hardware security analysis. This enables us to re-purpose and augment μhb graphs (originally proposed for verification of MCM implementations) for modeling hardware-specific security exploit scenarios.

Security litmus tests: We propose security litmus tests as a means of representing exploit programs in a form that is abstracted for efficiency, but useful for security analysis. Their compact nature enables efficient and interactive analysis with formal techniques, yet they are easily transformed into full executable programs when necessary.

Efficient hardware-aware exploit program synthesis: Given only microarchitecture and exploit pattern specifications, CheckMate efficiently and automatically synthesizes relevant μhb graphs and then in turn actual exploit programs. To showcase the applicability of CheckMate to real-world hardware security vulnerability detection, we conduct a case study by first supplying CheckMate with a speculative out-of-order (OoO) processor and a FLUSH+RELOAD cache side-channel attack exploit pattern. From these inputs, CheckMate synthesizes programs representative of Meltdown and Spectre attacks. Next, holding the microarchitecture constant, we replace the FLUSH+RELOAD exploit pattern in our case study with a PRIME+PROBE exploit pattern. Here, *CheckMate generated new attacks*—MeltdownPrime and SpectrePrime—which leverage invalidation messages sent to sharer cores on a write request (even if the write is speculative) in many cache coherence protocols. As a proof of concept, we implemented SpectrePrime as a C program and ran it on an Intel Core i7 processor; it achieved 99.95% accuracy in leaking private information over 100 runs. This result validates the CheckMate approach to automated synthesis of real-world exploits.

II. BACKGROUND

Since attacks like Meltdown and Spectre exploit non-architectural state updates and observable data-dependent variability across different executions of the same program, it is important for hardware security verification techniques to take into account the subtle orderings and interleavings of microarchitectural events when a program executes. This requires modeling features including (but not limited to) caches, branch predictors, and speculative memory accesses. This section gives an overview of how some hardware features (specifically, those most relevant to our case study in §VI) can be leveraged to induce information leakage.

A. Cache Timing Side-Channel Attacks

Side-channel attacks threaten confidentiality by exploiting implementation-specific behaviors with measurable dynamic state: for example, execution time [19], updates to storage elements [20], power consumption [21], resource sharing [22], acoustics [23], and radiation [24]. *Cache-based side-channel attacks* specifically target cache occupancy and rely on the attacker being able to differentiate between cache hits and misses.

Most cache side-channel attacks leverage timing as the key mechanism for distinguishing cache hits from cache misses [25]. Attackers monitor access times of their own or the victim's memory accesses in order to infer information about victim memory. "Access-driven" and "timing-driven" attacks both traditionally measure differences in access time. Access-driven attacks measure timing of a single memory operation [26], whereas timing-driven attacks measure timing of an entire security-critical operation. While the CheckMate approach can handle any security exploit scenarios resulting from hardware-specific event orderings and interleavings during a program's execution, our case study focuses on two categories of access-driven cache side-channel attacks: PRIME+PROBE and FLUSH+RELOAD [25]. FLUSH+RELOAD is the exploit pattern leveraged by the original Meltdown and Spectre attacks, and PRIME+PROBE is used by our case study in §VI.

In traditional PRIME+PROBE attacks, the attacker first primes the cache by populating one or more sets with its own lines, and then it allows the victim to execute. After the victim has executed, the attacker probes the cache by re-accessing its previously-primed lines, timing the accesses for classification as a cache hit or a cache miss. Longer access times (i.e., cache misses) indicate that the victim must have touched an address mapping to the same cache set as a primed location, thereby evicting the attacker's line.

Traditional FLUSH+RELOAD attacks have a similar goal to PRIME+PROBE, but rely on shared virtual memory between the attacker and victim (e.g., shared read-only libraries or page deduplication), and the ability to flush by virtual address (e.g., with the x86 `clflush` instruction). The advantage of FLUSH+RELOAD attacks is that the attacker can identify a specific line accessed by a victim rather than just a cache set. The attacker initiates FLUSH+RELOAD by flushing one or more shared lines of interest, and subsequently allows the victim to execute. After the victim has executed, the attacker reloads the previously flushed lines, timing the accesses to determine if said lines were pre-loaded by the victim. A similar attack, EVICT+RELOAD, does not rely on a special flush instruction, but instead on evictions caused by cache collisions; consequently the attacker must be able to reverse-engineer the cache-replacement policy.

One fundamental insight of Meltdown and Spectre is that microarchitectural speculation can be used to construct a FLUSH+RELOAD attack that does not require shared virtual memory between the attacker and victim. We describe this next.

B. Vulnerabilities Caused by Speculation

Many processors employ hardware optimizations such as speculation to improve performance. Speculative execution permits instructions to initiate execution before it is known that they will commit. As such, incorrectly speculated instructions will be squashed after they have begun executing. Until recently, it was assumed that "erasing" all *architecturally-visible* effects of squashed instructions was sufficient to ensure that speculation would not lead to any harmful side effects.

Unfortunately, 2018's series of speculation-based attacks leverage the effects of speculative execution on *non-architectural* state. As a specific example, Meltdown and Spectre leverage the effects of speculative execution on cache state. Since a CPU cache can be polluted by instructions that are eventually squashed, even if all architecturally-visible effects are erased, microarchitectural effects remain that can be observed. This can result in the leakage of privileged data via the following steps:

1) The attacker sets up its Meltdown/Spectre exploit by performing the Flush step of a FLUSH+RELOAD attack.
2) The attacker induces speculative execution of a read instruction that accesses sensitive[2] data. Meltdown and Spectre perform this step in different ways; see below.
3) While in the window of speculative execution, the attacker accesses non-sensitive data whose address is de-

[2]In some cases (e.g., Meltdown), the data being leaked lives in a different architectural privilege level. In other cases (e.g., Spectre v2), both attacker and victim data live in the same architectural privilege level, but each may be accessible only by certain parts of the program (e.g., from within vs. outside a sandbox). To make the distinction clear, we define *sensitive* data as that which should only be accessible by the victim, and *non-sensitive* data as that which is accessible by the attacker.

pendent (via address calculation) on the sensitive data returned by Step 2's read access.
4) The attacker performs the Reload step of a FLUSH+RELOAD attack to determine the address of the non-sensitive memory access from Step 3.
5) From the address result of Step 4, the attacker determines the sensitive data that was used to calculate it in Step 3.

Meltdown and Spectre achieve speculative cache pollution in different ways. If a user process accesses kernel memory, the permission check will eventually fail and cause the CPU to trigger a fault. Meltdown exploits the fact that speculative execution in some processors continues to execute subsequent program instructions, and consequently modify cache state, in the short window of time between the illegal memory access and the corresponding CPU fault. Spectre induces a victim (e.g., the operating system), via a mis-speculation past a branch, to speculatively execute instructions that would not have been executed during correct program execution.

Meltdown and Spectre provide a couple additional insights, along with exposing vulnerabilities related to speculative execution. First, unlike traditional FLUSH+RELOAD attacks, Meltdown and Spectre demonstrate that the victim is not necessarily required to execute between the flush and reload phases. Second, Meltdown and Spectre demonstrate that an attacker can leak data from *any memory location* (rather than only shared memory [25]).

The above insights are a prime example of why automated security verification with CheckMate can be so powerful. Consider the first insight above. CheckMate enables the user to define a single FLUSH+RELOAD attack pattern that is simultaneously capable of synthesizing exploits involving multiple processes (e.g., attacker and victim processes interleaved as in traditional FLUSH+RELOAD attacks) *and* a single process (e.g., a single attacker process as in some speculation-based attacks). This generality is not limited to processes; with the same exploit pattern, CheckMate considers a wide range of system and execution scenarios. For instance, synthesized exploit programs may vary in their instruction composition, number of physical hardware cores, and number of threads of execution. The CheckMate-generated MeltdownPrime and SpectrePrime attacks are examples of two-core exploits.

Regarding the second insight, the recent wave of speculation-based attacks highlight that a variety of subtle execution orderings in a program execution (e.g., address dependencies between instructions) can lead to variability across microarchitectural executions and thus induce a side-channel; these are precisely the types of ordering relationships that the CheckMate approach seeks to model using μhb graphs. Furthermore, CheckMate evaluates a range of memory partitioning and data sharing configurations that can indicate the conditions under which certain attack scenarios involving memory dependencies are possible.

III. CHECKMATE: μHB ANALYSIS FOR SECURITY

This paper leverages the observation that hardware security analysis is actually in many ways similar to analysis of MCM implementations. Specifically, both share two requirements: (i) a way to determine if a specific program execution scenario is possible on a given microarchitecture, and (ii) a mechanism for analyzing microarchitectural event orderings and interleavings corresponding to a program's execution. The first requirement is met by a core principle of μhb graph analysis that cyclic μhb graphs represent impossible executions (i.e., executions that are *unobservable* on the target microarchitecture). Intuitively, a cycle in a μhb graph represents a scenario in which a physical event happens before itself; i.e., a proof by contradiction that the proposed execution is impossible. Similarly, acyclic μhb graphs represent *observable* executions.

For the second requirement, we adopt μhb graphs from prior MCM verification work, but extend and adapt them in interesting ways for security verification. Specifically, we first introduce the concept of exploit patterns to represent hardware execution patterns indicative of security exploits as μhb sub-graphs. Second, we leverage RMF techniques to facilitate implementation-aware exploit program synthesis. The remainder of this section details how CheckMate transforms the inputs of Figs. 1b and 1c into the outputs of Figs. 1e and 1f.

A. CheckMate Inputs

CheckMate requires two inputs: a microarchitecture specification and specification of a class of exploits. Fig. 1 contains examples that are referenced throughout this section.

1) Microarchitecture Specification: As prior work has demonstrated, a microarchitecture and its related OS support can be modeled axiomatically [13, 15]. An axiomatic microarchitecture specification defines hardware-supported micro-ops, microarchitectural structures that micro-ops pass through at various points of execution, and any hardware-specific execution event orderings (e.g., in-order Fetch or OoO Execute). To encode microarchitecture specifications (i.e., *μspec models*), CheckMate uses a μspec-like DSL [15], that is augmented for security modeling and embedded within the Alloy DSL [18]. The μspec models used by CheckMate support descriptions of complex microarchitectural features, such as branch prediction, speculation, virtual memory, and user-level processes.

Fig. 1b provides an excerpt of a μspec model corresponding to Fig. 1a's pedagogical two-core, three-stage, in-order hardware design. μspec models are essentially first-order logic formulations of hardware designs, built on top of μhb graph-related predicates. Examples of such predicates include statements like `ProgramOrder` which evaluates to `True` if the two micro-ops passed to it are in order in the instruction stream, or `EdgeExists` which evaluates to `True` if there exists a happens-before edge between the two nodes passed to it (where a node is an $\langle Event, Location\rangle$ pair).

2) Exploit Pattern Specification: Exploit patterns are formalizations of hardware execution patterns indicative of security exploit classes. Most basically, they are μhb sub-graphs. For input into CheckMate, they are expressed using the same DSL that is used for the microarchitecture specification input.

Fig. 1c illustrates the exploit pattern we constructed for FLUSH+RELOAD attacks[3]. The Value in Cache Lifetime (ViCL) abstraction referenced in the figure is detailed in §VI-A1. For the moment, "ViCL Create" and "ViCL Expire" can be intuitively understood as "cache line create" and "cache line expire," respectively. The first pair of ViCL Create and Expire nodes in Fig. 1c represent the attacker *possibly* having the exploit's line of interest residing in its cache at the beginning of the attack. To officially start the attack, the attacker uses an explicit flush instruction (or causes a cache collision), to evict a virtual address of interest. This flush/evict event is represented by the rectangle shaded with horizontal red lines. If the first pair of ViCL Create and Expire nodes correspond to the same virtual address that the flush/eviction is targeting, we can draw a happens-before edge from the first ViCL Expire node to the flush/evict event.

In the absence of any instructions between the flush and reload events, FLUSH+RELOAD attacks *expect* to observe a cache miss on the reload access, resulting in new ViCL Create and Expire nodes. If, in the rectangle shaded with diagonal gray lines, the evicted location was brought into the cache by either (i) the victim accessing the same address (e.g., a via a shared library) or (ii) a speculative operation that is dependent on victim memory, the attacker will observe a cache hit on its reload access and have the potential to infer victim information it does not have permissions to access. The cache hit is illustrated by the *absence* of ViCL Create and Expire nodes for the reload access.

Another key insight of our approach is that we can repurpose the axiomatic modeling technique used to encode μspec models in order to *abstract away some implementation-specific features* and create more portable exploit specifications. The FLUSH+RELOAD exploit pattern is general to the degree that it only relies on the presence of caches or similar structures (e.g., TLBs) that can be modeled with ViCLs and a particular microarchitectural structure (e.g., the Execute stage of the pipeline in Fig. 1a), where reads from said structure bind their value. This pattern is portable and can be applied to a wide variety of microarchitectures or systems. When combined with a microarchitecture specification, this pattern will generate all program scenarios realizable on the microarchitecture (up to a user-specified program size) that can induce a hit on the reload access. Fig. 1d and corresponding Fig. 1e show the FLUSH+RELOAD exploit pattern superimposed on the execution of a program (specifically the program in Fig. 1f) on Fig. 1a's microarchitecture. In §VI, the same pattern is used to produce Meltdown and Spectre attacks on a different hardware design.

B. CheckMate Outputs

1) μhb Graphs: The CheckMate approach ultimately transforms the microarchitecture and exploit pattern specification inputs into μhb graphs representative of hardware-specific exploit program executions when the input microarchitecture is susceptible to the input vulnerability. As in Fig. 1e, we depict μhb graph nodes in a grid format; a node's event (i.e., micro-op) is denoted by the column label and a node's location is denoted by the row label. We have highlighted the FLUSH+RELOAD exploit pattern from Fig. 1c in red nodes and edges, a rectangle shaded with diagonal gray lines, and a rectangle shaded with horizontal red lines.

Fig. 1e shows how exploit execution scenarios are represented as μhb graphs. Here, the Attacker (A) and Victim (V) are two distinct processes that are time-multiplexed on the same physical core and thus share an L1 cache. Yellow edges connecting Complete events to Fetch events (and one red edge from $\langle A.I2, Complete\rangle$ to $\langle V.I0, Fetch\rangle$) represent time-multiplexing; a micro-op from one process must complete before a micro-op from another process is fetched. Dashed edges show the order of the instruction stream through the pipeline; given the pipe-stages are fully in-order, A.I1 is in the Fetch stage before A.I2 is in the Fetch stage, and so on for Execute and Commit. Black solid edges represent a single micro-op's path through the pipeline; each micro-op is in the Fetch stage before it is in the Execute stage, etc. Blue edges and two red edges from ($\langle V.I0, L1\ ViCL\ Create\rangle$ to $\langle A.I3, Execute\rangle$ and from $\langle A.I3, Execute\rangle$ to $\langle V.I0, L1\ ViCL\ Expire\rangle$) are specific to L1 cache ViCL Create and ViCL Expire events; looking at A.I1, a cache line must be brought into the L1 cache (L1 ViCL Create) before it is read in the Execute stage, and the read of memory must complete in the Execute stage before the cache line is evicted from the L1 or invalidated (L1 ViCL Expire).

2) Security Litmus Tests: CheckMate conducts bounded verification, meaning the user must specify a maximum program size for synthesis (in terms of parameters such as the number of physical cores, threads, instructions, and processes). Ultimately, CheckMate outputs μhb graphs (Fig. 1e) that represent executions of *security litmus tests* (in Fig. 1f). Security litmus tests are the most compact representation of an exploit program, meaning they contain the minimal number of micro-ops necessary to produce the exploit pattern of interest. They are similar in concept to MCM litmus tests for concurrent programs [27–30]. Security litmus tests are useful to output because: (i) they are much more practical to analyze with formal techniques than a full program due to their compact nature, and (ii) they are nevertheless easily transformed into full executable programs when necessary [27, 31, 32].

Consider the security litmus test in Fig. 1f which corresponds to the μhb graph in Fig. 1e and which represents a traditional FLUSH+RELOAD attack. The attacker performs two reads and an intervening CLFLUSH operation all with the same effective address and experiences a cache hit on the second read due to a victim access that brought the memory location back into the physically shared L1 cache. This litmus test performs the attack on a single address, whereas a full FLUSH+RELOAD attack would require scanning the entire cache for the flush and reload accesses. Furthermore, the litmus test assumes that cache is direct mapped. We choose

[3]Our FLUSH+RELOAD exploit pattern is general enough to additionally capture EVICT+RELOAD attacks.

to handle set-associativity with litmus test post-processing that accounts for the cache replacement policy of the target microarchitecture.

CheckMate can automatically generate a large volume of tests so that the user can identify all of the vulnerable hardware features. Given a FLUSH+RELOAD pattern, CheckMate effectively generates all possible ways in which an input microarchitecture could render the reload access a hit. Each generated litmus test differs in some way, such as *how* the attack is performed. For example, in our case study, synthesized Meltdown and Spectre attacks exploit speculative cache pollution whereas synthesized traditional FLUSH+RELOAD attacks exploit the combination of shared read-only memory and physical resource sharing between Attacker and Victim. Our FLUSH+RELOAD pattern is also sufficiently general such that in our experiments CheckMate generates alternative attacks where the CLFLUSH instruction is another memory access mapping to the same L1 cache line as the exploit's target address thereby evicting it (i.e., EVICT+RELOAD).

While the security community has historically placed emphasis on ad hoc discovery of concrete working examples of exploits, we see benefits in automatically generating litmus test abstractions of exploits that aid microarchitects in designing secure hardware. §VII-C shows how security litmus tests make the path to a full exploit clear.

IV. RELATIONAL MODEL FINDING FOR IMPLEMENTATION-AWARE PROGRAM SYNTHESIS

CheckMate automatically synthesizes microarchitecture-aware programs that feature user-specified exploit patterns of interest. To implement this, we leverage RMF techniques. This section introduces terminology and presents an unoptimized version of CheckMate that we implement using the Alloy RMF language [18]. §V contains the optimizations that make CheckMate efficient.

A. Why Relational Model Finding?

Most basically, a relational model is a set of constraints on an abstract system of atoms (basic objects) and relations, where an N-dimensional relation defines some set of N-tuples of atoms [33]. For example, a μhb graph is a relational model: the nodes of the μhb graph are atoms, and the edges in the μhb graph form a two-dimensional relation over the set of nodes (with one source node and one destination node for each edge). A constraint for a μhb graph might state that the set of edges in any satisfying instance (i.e., any satisfying μhb graph) is acyclic. Another constraint might state that the set of nodes and edges in any instance must contain a specific μhb sub-graph or pattern.

Finding instances of an exploit on a microarchitecture is clearly model finding, and the use of μhb graphs is a good fit for relational models; together that makes RMF a good fit. Fortunately, optimized tools for efficient RMF already exist. We use Alloy [18] as the language in which we implement CheckMate, due to its easy-to-use DSL and efficient mapping into SAT via its Kodkod backend [33]. Any solutions found by the SAT solver are then translated back into the corresponding relations in the original Alloy model so that they can be analyzed by the user. The generality of this approach stands in contrast to previous work on μhb graphs [15], which used a custom solver incapable of capturing all of the features needed for CheckMate.

B. Unoptimized Formulation of μspec Primitives in Alloy

Fig. 2 gives an overview of the CheckMate toolflow. CheckMate conducts microarchitecture-aware program synthesis in effectively two stages. First, given a set of all available micro-ops (as part of the μspec model) and a synthesis bound, CheckMate deduces the set of all possible program executions on the input microarchitecture. We refer to this set of program executions as *candidate executions* [34]. Second, CheckMate prunes the set of candidate executions to only those which feature the desired exploit execution pattern. The result is a set of all possible security litmus test programs (within the synthesis bound) and all possible executions of those programs (i.e., all interleavings of hardware execution events) that can expose the exploit pattern on the input microarchitecture.

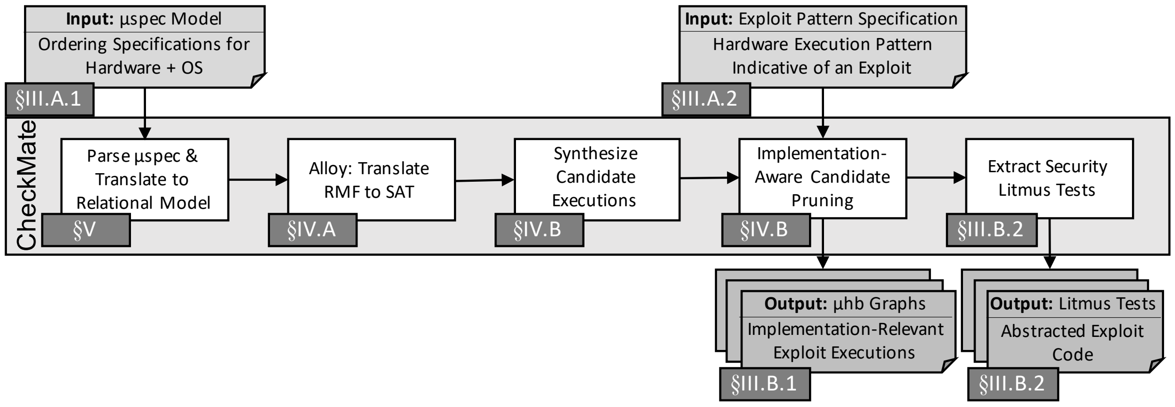
While CheckMate moves beyond MCM verification, some *MCM relations* are relevant when generating candidate executions. Specifically, MCMs define communication-based happens-before relationships that order micro-ops operating on the same effective address; we refer to μhb edges reflecting such relationships as `com` (or "communication") edges. MCMs also define dependency happens-before relationships (`addr`, `data`, and `ctrl`) that affect ordering of dependent micro-ops and a program order (or `po`) happens-before relationship that orders micro-ops with other micro-ops that occur later in the instruction stream.

As discussed in §III-A, the microarchitecture and exploit pattern specifications supplied to CheckMate are expressed in the (augmented) μspec-like DSL embedded in Alloy. In order to interpret μspec models and leverage Alloy's RMF backend, CheckMate requires an Alloy formulation of the following μspec primitives: addressable memory locations, micro-ops (i.e., "events"), micro-ops that access memory (i.e., "memory events"), hardware locations, MCM relations (`com` and `po`), μhb nodes, and μhb edges. These μspec primitives are then used to construct μspec predicates such as the `ProgramOrder` and `EdgeExists` predicates in Fig. 1b.

Fig. 3a presents an unoptimized formulation of μspec primitives in Alloy. The figure shows four high level atoms or "signatures" (`sig`) in Alloy syntax, along with other `sigs` that extend from them. Each `sig` is essentially a set in Alloy. Fig. 3b summarizes the set contents of the `sigs` we define. Unfortunately, this naive approach suffers from inefficiencies and poor scalability for our application scenarios, so CheckMate addresses these issues (see §V).

V. CHECKMATE TOOL: KEEPING IMPLEMENTATION-AWARE PROGRAM SYNTHESIS TRACTABLE

The key to making CheckMate useful is keeping it efficient; RMF is challenged by huge search spaces that are infeasible



Input: μspec Model
Ordering Specifications for Hardware + OS
§III.A.1
Input: Exploit Pattern Specification
Hardware Execution Pattern Indicative of an Exploit
§III.A.2
CheckMate
Parse μspec & Translate to Relational Model
§V
Alloy: Translate RMF to SAT
§IV.A
Synthesize Candidate Executions
§IV.B
Implementation-Aware Candidate Pruning
§IV.B
Extract Security Litmus Tests
§III.B.2
Output: μhb Graphs
Implementation-Relevant Exploit Executions
§III.B.1
Output: Litmus Tests
Abstracted Exploit Code
§III.B.2

Fig. 2: Overview of the CheckMate toolflow. Inputs are listed across the top with arrows depicting when given inputs are used. Outputs are listed across the bottom with arrows originating from the steps that produce them.

to analyze in terms of time or memory. Thus, when building CheckMate, we paid close attention to constraining the solution space so as to minimize time wasted exploring redundant solutions. Our techniques are not specific to Alloy and could be used to improve the scalability of other RMF- and SAT-based techniques for μhb analysis.

Although μhb analysis covers large search spaces, huge portions of the space can be pruned quickly. Microarchitecture specifications define concrete hardware locations and hardware-enforced orderings, enabling us to frame the problem intelligently to keep runtimes tractable. For example, the μspec model specifies the set of locations that a specific type of micro-op must pass through (e.g., during its path through the pipeline). The μspec model can therefore statically determine which nodes should be present. The key is to ensure that the underlying tools (Alloy in this case) have the information they need to perform this pruning.

With a naive node implementation like that in Fig. 3a, Alloy will analyze many instances of the model that are repetitive or symmetric. For example some instances/solutions might be isomorphic to others except for arbitrary node relabeling. Consider the security litmus test in Fig. 1f and its corresponding μhb graph in Fig. 1e that contains 20 nodes. With no symmetry-breaking, a naive Alloy encoding would cause Kodkod to generate 20! variants of this single μhb graph corresponding to each of the different ways that nodes could be assigned to the event location pairs. This does not even include the number of ways in which edges can be assigned to node pairs. While Alloy does have some symmetry-breaking built in, its heuristics are not sufficient to prune enough of search space to make microarchitecture-aware program synthesis feasible. Fig. 3c shows the unoptimized runtime explodes for practical microarchitectures.

A. Avoiding Re-Analysis of Isomorphic Graph Nodes

Problem sizes quickly become intractable without a way to constrain nodes. Fig. 3a shows a naive way to represent nodes would be as a new `sig`. In that case, we also need two new relations describing the micro-op and location assigned to each node. The exact μhb graph layout is known a priori (i.e., a regular grid), but unfortunately, Alloy can only express relations as SAT expressions to be concretized later by the SAT solver. Thus, such an approach introduces two new large degrees of freedom that do not even carry any semantic content, resulting in a tremendous waste of computational resources.

A more efficient mapping is to simply encode nodes as a relation `NodeRel`, of type `Event`→`Location`. In this way, the necessary mapping information is encoded directly, reducing wasteful compute. Consequently, we can instantiate a constrained and relevant set of μhb nodes. `NodeRel` maps `Event` atoms to each of the specific `Location` atoms that they must pass through in a valid execution. That is, the instructions flow through the pipestages in a familiar way. For Fig. 1e's μhb graph: `NodeRel` = $\{\langle V.I0, Fetch\rangle$, $\langle V.I0, Execute\rangle$, $\langle V.I0, Commit\rangle$, ...$\}$.

B. Avoiding Re-Analysis of Isomorphic Graph Edges

Since μhb nodes are represented by the `NodeRel` relation of type `Event`→`Location`, μhb edges have the type `(Event`→`Location)`→`(Event`→`Location)`. An edge of this type implies a happens-before edge from an instruction at one location to a possibly different instruction at a possibly different location.

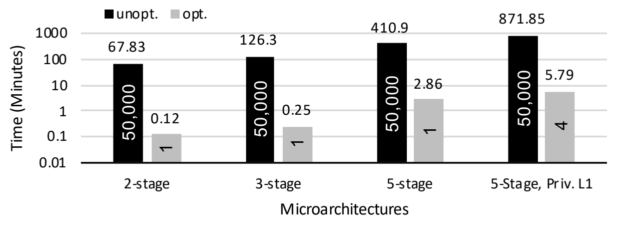
Once all required edges have been added to a μhb graph, cycle checking is performed by taking the transitive closure of all edges and checking for reflexive edges (i.e., edges that start and end at the same node). To constrain the model finding problem to focus only on edges of interest, we created various categories of edge relations that are ultimately composed into a single relation, `sub_uhb`. For example, two subsets of `sub_uhb` include: `uhb_intra`, which describes intra-instruction edges, and `uhb_inter`, for inter-instruction edges. By dividing `sub_uhb` into sub-relations, we drastically reduce the exploration of graphs that result from adding edges that would already be included in the transitive closure of

```
sig Address { }
abstract sig Event { po: lone Event }
abstract sig MemoryEvent extends Event { address: one Address }
sig Write extends MemoryEvent { rf : set Read, co : set Write }
sig Read extends MemoryEvent { fr : set Write }
fun com : MemoryEvent->MemoryEvent { rf + fr + co }
abstract sig Location { }
sig Node {
  event: one Event,
  loc: one Location,
  uhb: set Node
}
```

(a) Unoptimized Alloy formulation of μspec primitives.

Alloy Signature	**Set Contains All...**
sig Address	addressable memory locations
abstract sig Event	micro-ops
abstract sig MemoryEvent extends Event	micro-ops that access memory
sig Write extends MemoryEvent	micro-ops that write memory
sig Read extends MemoryEvent	micro-ops that read memory
abstract sig Location	microarchitectural structures
sig Node	nodes in a μhb graph

(b) Contents of Alloy `sigs` (i.e., Alloy sets) from (a).



(c) Performance: This chart illustrates the benefits of optimized (opt.) CheckMate (§V) over unoptimized (unopt.) CheckMate (§IV-B). Runtimes reflect the time to generate all satisfying μhb graphs for a synthesis problem that has only one solution. Unoptimized CheckMate generates 10s-100s of thousands of duplicate or isomorphic μhb graphs without terminating (we did not observe termination within a 24 hour limit), so we cap synthesis for those cases at 50,000 graphs. The number of synthesized examples is noted inside the bars; numbers greater than one indicate duplicate or isomorphic examples that we filter (§V-C). CheckMate's optimizations enable more targeted and efficient program synthesis that terminates.

Fig. 3: Compared with §IV's unoptimized CheckMate implementation, §V's optimizations enable significantly improved scalability with increasing hardware complexity.

edges. However, despite the distinct names assigned to each category, all μhb edges are still treated equivalently by the cycle checking that is ultimately performed to categorize a potential solution program as observable or unobservable.

C. *Constraining Solutions*

In addition to node and edge optimizations, a third optimization pertains to solution constraints. During the course of each run, CheckMate generates μhb graphs representing each of the synthesized program executions. If isomorphic μhb graphs are reproduced with different labels (V-A), the same security litmus test can be reproduced multiple times for the same run of CheckMate. Filtering duplicate solutions produces a more concise set of results.

Furthermore, there are cases where programs might be symmetric or differ only in addresses being swapped. We consider two results with this type of symmetry to be the same, and filter one. Another issue arises with unbounded relationships. For example, when modeling caches, there might be a large or unbounded number of ways in which a system's caches could issue and respond to coherence messages. In this case, the user can constrain the number of μhb edges corresponding to the cache coherence activity to be a finite number. This bounds the number of example programs that are generated. If the user can identify a true upper bound to specify as the constraint, then the generated set is still complete. If a bound is set without knowing the true upper bound, then the generated output programs may be an incomplete set, but this is a performance vs. coverage tradeoff. Our experience with litmus test symmetries is not unique, and other related work employs various work-arounds [28, 29]. We use a simple heuristic for eliminating duplicate security litmus test produced by CheckMate, but techniques from prior work are also applicable.

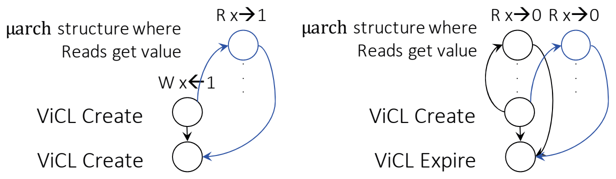
VI. Case Study: Synthesizing Real Attacks

To showcase the applicability of CheckMate to modern secure processor and systems design, we conducted a case study to evaluate the susceptibility of a speculative OoO processor to both FLUSH+RELOAD and PRIME+PROBE cache timing side-channel attacks. When supplying CheckMate with our microarchitecture and FLUSH+RELOAD exploit pattern, CheckMate automatically generated security litmus test programs representative of Meltdown [1] and Spectre [2] attacks. Upon switching the FLUSH+RELOAD pattern to a PRIME+PROBE pattern, CheckMate synthesized *new attacks* related to Meltdown and Spectre, yet distinct.

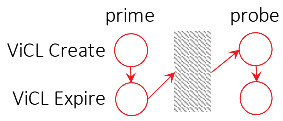
A. *Specifying Attack Patterns*

§III-A2 explains the exploit pattern we constructed for FLUSH+RELOAD cache side-channel attacks. This section describes the ViCL abstraction that was used to construct that pattern and presents another exploit pattern we formulated, specifically for PRIME+PROBE attacks.

1) Value in Cache Lifetime (ViCL): Modeling any type of cache side-channel attack necessitates modeling cache occupancy. To model cache occupancy, we use the ViCL abstraction from prior μhb analysis work [14]. As Fig. 4 shows, a ViCL seeks to abstract the lifetime of a cache line into two main events: a "Create" event and an "Expire" event, which can then be used to reason about event orderings and interleavings. A ViCL Create occurs when either (i) a cache line enters a usable state from a previously unusable state, or (ii) when a new value is written into a cache line. A ViCL Expire occurs when (i) its cache line enters an unusable state from a previously usable state, or (ii) a value in a cache line is overwritten and no longer accessible. For read accesses, ViCL Create and Expire nodes are not instantiated if the read experiences a cache hit. In that case, the read is "sourced" from a pre-existing ViCL. That is, the read receives its value from another micro-op that has brought/written the location/value into the cache.


μarch structure where Reads get value
R x→1
W x←1
ViCL Create
ViCL Create
μarch structure where Reads get value
R x→0 R x→0
ViCL Create
ViCL Expire

(a) How ViCLs Work: On the left, the write creates a new ViCL pair which sources the subsequent read. On the right, the first read misses creating a new ViCL pair, which sources the following read, which hits.



(b) PRIME+PROBE μhb exploit pattern; rectangles can contain any combination of nodes and edges.

Fig. 4: Modeling cache side-channel attacks with ViCLs.

Both of the cache side-channel attacks we consider in this paper—PRIME+PROBE and FLUSH+RELOAD—fit a similar format where the attacker conducts two primary accesses to the same target address. The first access—the prime (resp. flush) access in a PRIME+PROBE (resp. FLUSH+RELOAD) attack—sets up the attack. The second (and subsequent) access—the probe (resp. reload) access in a PRIME+PROBE (resp. FLUSH+RELOAD) attack—completes the attack and is timed for classification as a cache hit or miss. Cache hits and misses for a load can largely be distinguished by the presence or absence of new ViCL Create and Expire nodes, respectively, in a μhb graph. This is not strictly true in all cases, since for example a load may suffer a cache miss but a ViCL hit if it accesses a line that already has a pending fill outstanding. However, this situation will be uncommon in the more controlled scenarios of interest in this paper, and hence we simply consider it to be part of the noise in the signal. Although writes always inherently produce new ViCLs, we analyze them the same way we do reads, and we post-process them to generate analogous cache-based timing attacks with a write rather than a read as the second access.

2) PRIME+PROBE *Exploit Pattern:* Fig. 4b depicts the PRIME+PROBE exploit pattern we constructed in an effort to synthesize new exploits related to Meltdown and Spectre, but leveraging a different side-channel attack. This pattern consists of two consecutive memory accesses to the same address, and new ViCL Create and ViCL Expire nodes for the second access. To the extent that clean lines will not otherwise be evicted (causing noise in the signal), this pattern signifies a measurable timing difference and the potential for an attacker to infer victim information it does not have permissions to access when (i) the victim, evicts the attacker's line (e.g., by accessing a memory location that maps to the same spot in the cache, causing a collision) or (ii) a speculative operation that is dependent on victim memory evicts the line.

B. Experimental Setup

CheckMate augments μspec modeling with additional capabilities and features including: distinct processes (e.g., attacker and victim processes), private and shared address spaces, memory access permissions, cache indices, coherence protocol invalidation messages, speculation, and branch prediction. The hardware design in our experiments is a 5-stage pipeline—Fetch, Execute, Reorder Buffer (ROB), Permission Check (PC), Commit—where processor cores have FIFO store buffers and private L1 caches connected to main memory. The μhb graphs in Fig. 5 reflect this design. The μhb graphs in Figs. 5c and 5d additionally feature RWReq/RWResp execution events, which correspond to the points at which coherence requests/responses are made/received for a given memory access. We omit these locations from Figs. 5a and 5b since they are not relevant for the Meltdown and Spectre security litmus tests. The supported micro-ops in our μspec model are reads, writes, CLFLUSH (analogous to x86's `clflush`), conditional branches, and full fences. The pipeline implements the Total Store Order (TSO) MCM. Other micro-ops and/or MCMs are easy to add or implement as desired; the CheckMate approach is easily extensible.

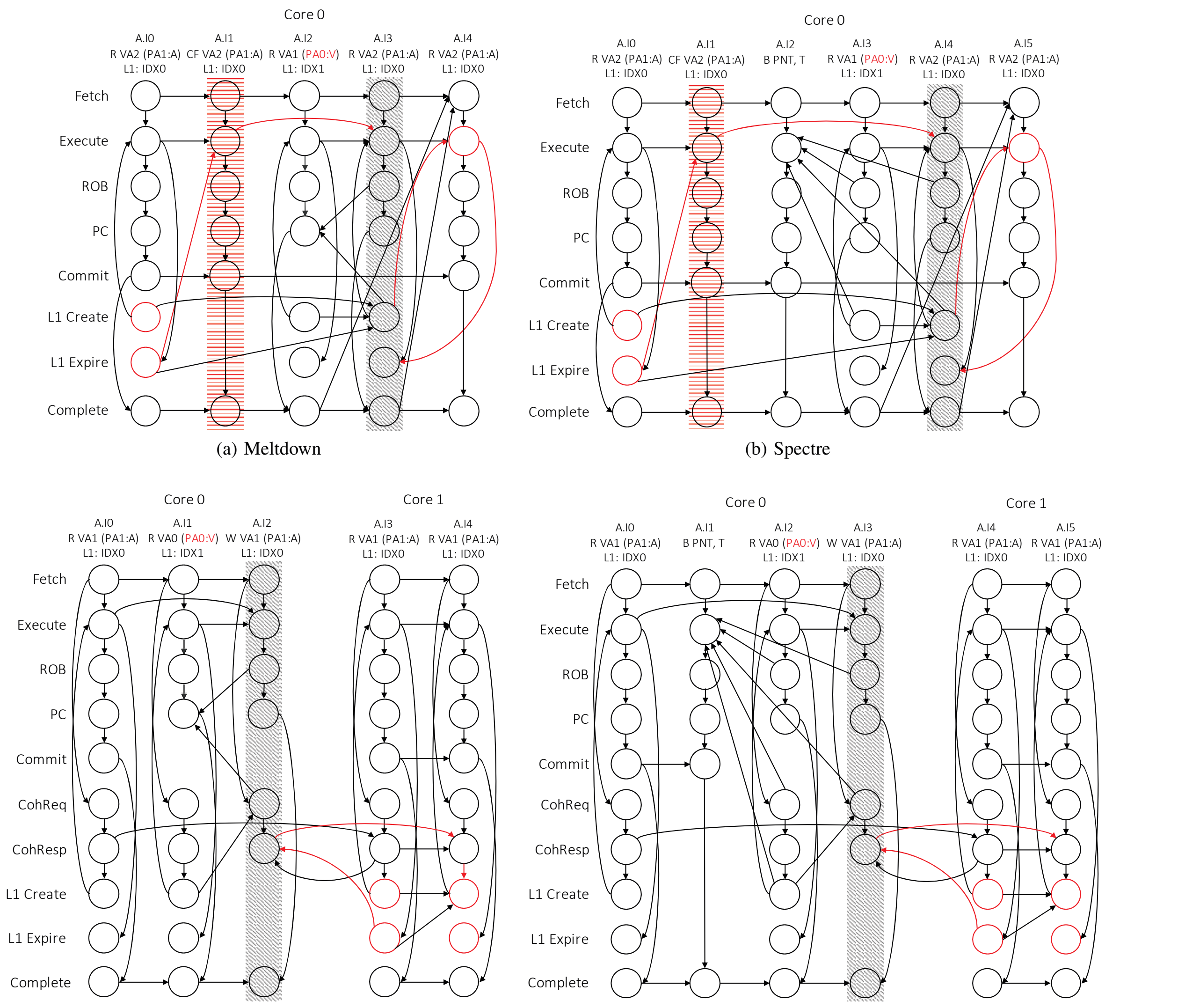
In our runs of CheckMate, we take explicit steps to reduce noise in the synthesized outputs. First, we make an *attacker assumption* which mandates that the attacker will not cause noise in our experiments (i.e., the attacker will not void its own exploit). We assume for convenience that collisions are the only mechanism by which cache lines can be evicted. In other words, we categorize any evictions not due to collisions as noise in the signal. This filtering also helps us avoid false positives. Finally, we supply CheckMate with an additional constraint that requires attacker programs end after they have acquired the desired information from the victim (e.g., after the probe step of a PRIME+PROBE attack).

Between the one processor input and two exploit pattern inputs (i.e., FLUSH+RELOAD and PRIME+PROBE), we tested two total (processor, exploit pattern) input combinations. For these inputs, we ran CheckMate with increasing bounds until an attack was found. We ran our experiments using Alloy Version 4.2 [18] and Kodkod Version 2.1 [33], both of which run as Java applications.

VII. RESULTS

A. Automatic Synthesis of Meltdown and Spectre

Figs. 5a and 5b depict μhb graphs synthesized by CheckMate which correspond to security litmus test programs representative of the publically disclosed Meltdown and Spectre attacks, respectively. The pattern from Fig. 1c that seeded synthesis is highlighted in red nodes and edges and rectangles shaded with horizontal red lines and diagonal gray lines in each graph. The security litmus test itself is listed at the top of each graph with per-core micro-op sequencing from left to right. As the figures show, the security litmus test is the most abstracted form of each attack; it only applies to a single virtual address (see §III-B2). We also note that


Core 0
A.I0 R VA2 (PA1:A) L1: IDX0
A.I1 CF VA2 (PA1:A) L1: IDX0
A.I2 R VA1 (PA0:V) L1: IDX1
A.I3 R VA2 (PA1:A) L1: IDX0
A.I4 R VA2 (PA1:A) L1: IDX0
Fetch
Execute
ROB
PC
Commit
L1 Create
L1 Expire
Complete
Core 0
A.I0 R VA2 (PA1:A) L1: IDX0
A.I1 CF VA2 (PA1:A) L1: IDX0
A.I2 B PNT, T
A.I3 R VA1 (PA0:V) L1: IDX1
A.I4 R VA2 (PA1:A) L1: IDX0
A.I5 R VA2 (PA1:A) L1: IDX0
Core 0
A.I0 R VA1 (PA1:A) L1: IDX0
A.I1 R VA0 (PA0:V) L1: IDX1
A.I2 W VA1 (PA1:A) L1: IDX0
Core 1
A.I3 R VA1 (PA1:A) L1: IDX0
A.I4 R VA1 (PA1:A) L1: IDX0
CohReq
CohResp
Core 0
A.I0 R VA1 (PA1:A) L1: IDX0
A.I1 B PNT, T
A.I2 R VA0 (PA0:V) L1: IDX1
A.I3 W VA1 (PA1:A) L1: IDX0
Core 1
A.I4 R VA1 (PA1:A) L1: IDX0
A.I5 R VA1 (PA1:A) L1: IDX0

(a) Meltdown (b) Spectre

(c) MeltdownPrime (d) SpectrePrime

Fig. 5: Synthesized μhb graphs showing selected security litmus test executions for conducting Meltdown, Spectre, Meltdown-Prime, and SpectrePrime attacks. Both (a) and (b) exploit the pattern in Fig. 1c, while (c) and (d) exploit the pattern in Fig. 4b. The Store Buffer and Main Memory stages have been removed for clarity as these particular μhb graphs do not contain write micro-ops. B PNT, T represents a branch that is mispredicted as "not taken." CF represents a CLFLUSH micro-op. Table I shows that (c) and (d) were synthesized with instruction bounds of 4 and 5, respectively. §VII-C explains why we include an extra initial instruction for each here.

CheckMate outputs detailed meta-data such as (i) the index that each virtual (or physical, if physically mapped) address maps to in each cache, (ii) the physical address that each virtual address maps to, (iii) the physical core that a micro-op executes on, (iv) process access permissions for each address, and (v) cacheability attributes of virtual addresses. For clarity, Fig. 5 includes a simplified subset.

Fig. 5a demonstrates how the lack of synchronization between the permission check of a memory access and the fetching of said memory location into the cache can result in the FLUSH+RELOAD pattern of Fig. 1c; μhb graphs are instructive and can suggest edges whose addition mitigate an exploit by rendering the graph cyclic. Fig. 5b demonstrates a similar scenario, but the lack of synchronization is between the evaluation of the branch outcome in the Execute stage of the branch and any subsequent fetching of cache lines. We note that in our synthesized exploits, an Attacker (A) process represents the Attacker executing instructions *or* a Victim (V) executing Attacker-influenced instructions due to a branch or jump misprediction.

Exploit Pattern	Inst.	Output Attack	Min. to Synth. 1	Min. to Synth. All	Unique Litmus Tests
FLUSH+RELOAD	4	FLUSH+RELOAD	3.91	6.32	8
	5	Meltdown	19.53	55.48	6
	6	Spectre	79.83	215.11	12
PRIME+PROBE	3	PRIME+PROBE	3.27	4.14	6
	4	MeltdownPrime	15.73	16.78	4
	5	SpectrePrime	64.87	67.27	8

TABLE I: Sample runtimes (averaged over 10 runs) for generating various exploits. For both exploit patterns, we ran CheckMate with increasing bounds and recorded the time to synthesize the first exploit and all exploits within the bound. For runtimes related to the FLUSH+RELOAD exploit pattern, we omit RWReq/RWResp modeling as it does not produce distinct results. The number of unique litmus tests reflects the post-processing removal of duplicate and isomorphic results described in §V-C. We do not include the post-processing mentioned in §VI-A1. Lastly, for FLUSH+RELOAD attacks, the filtered results only include those with a read preceding the flush as the access that *could have* brought the target virtual address into the cache initially.

Table I shows that CheckMate synthesized the first exploit variant of both Meltdown and Spectre (and the other output attacks) on the order of minutes. After generating the first variant, CheckMate continually identifies others within the user-provided verification bounds; CheckMate synthesized Meltdown at an instruction bound of 5 and Spectre at an instruction bound of 6. In addition to the instruction bounds listed in the table, we also bound the number of virtual and physical addresses to reduce the number of symmetric results produced.

Other significant Meltdown and Spectre variants synthesized by CheckMate include those which have a write instead of a read for the speculative attacker access which brings the flushed address back into the cache. This is due to modeling a write-allocate cache. We have modeled the allocate portion of a write instruction in two ways: using a read micro-op and using a write-allocate micro-op. The results in Table I use the former implementation. CheckMate also generated variants representative of EVICT+RELOAD attacks—rather than a flush instruction, they use a colliding memory operation to evict a line of interest from the cache to initiate the attack. Our additional synthesized security litmus tests are provided online [35].

For each output attack listed in Table I, CheckMate generated tens to hundreds of security litmus tests. CheckMate synthesized all satisfying μhb graphs within the search space and terminated after a reasonable duration (unlike unoptimized CheckMate in Fig. 3c), which is noted in the table. Of the large number of μhb graphs generated by CheckMate, a sometimes significant portion, depending on the verification case, correspond to duplicate or isomorphic results. Duplicates can result from multiple encodings of the SAT problem by Kodkod which happen to produce the same result modulo an internal labeling of solver variables. Isomorphic results might feature the attack targeting a different address. We post-process CheckMate output to analyze only unique exploit variants. The number of unique variants we identified is presented in Table I for each output attack.

B. *Automatic Synthesis of New Exploits: MeltdownPrime and SpectrePrime*

Figs. 5c and 5d depict μhb graphs corresponding to the programs CheckMate synthesized representative of our new MeltdownPrime and SpectrePrime attacks, respectively. These new exploits rely on invalidation-based coherence protocols in combination with PRIME+PROBE attacks. In particular, by exploiting speculative cache invalidations, MeltdownPrime and SpectrePrime can leak victim memory at the same granularity as Meltdown and Spectre while using a PRIME+PROBE timing side-channel. The pattern from Fig. 4b that seeded synthesis is highlighted in red nodes and edges and a rectangle shaded with diagonal gray lines in each of the generated examples. The security litmus test is again listed at the top of each graph.

In the input microarchitecture used to synthesize these attacks, we model the sending and receiving of coherence request and response messages that enable a core to gain write and/or read permissions for a memory location. Due to this level of modeling detail we are able to capture perhaps surprising coherence protocol behavior. Specifically, the coherence protocol may invalidate cache lines in sharer cores as a result of a speculative write access request even if the write is eventually squashed. These CheckMate-generated attacks are split across two cores to make use of coherence protocol invalidations.

Some other notable CheckMate-synthesized variants of our Prime attacks featured a CLFLUSH instruction instead of the write access for the mechanism by which an eviction is caused on another core. This is under the assumption of cache inclusivity, that such a flush instruction exists, and that virtual addresses can be speculatively flushed. We have not observed this speculative flushing variant on real hardware. Given that, the microarchitecture used to gather the performance results in Table I does not implement speculative flushes.

C. *From SpectrePrime Security Litmus Test to Real Exploit*

To demonstrate our coherence protocol invalidation-based attack on real hardware, we expanded the SpectrePrime security litmus test of Fig. 5d to a full attack program. It is possible to automate the process of expanding security litmus test to full exploit programs. However, our intention is for CheckMate to serve as a hardware designer's assistant for evaluating the resilience of their designs to attacks, rather than an attack generator.

The synthesized SpectrePrime litmus test exemplifies the attack on a single address. We extended the litmus test according to the L1 cache specifications of the Intel Core i7 Processor on which we ran our experiments; our experimental setup consisted of a Macbook with a 2.4 GHz Intel Core i7 Processor running macOS Sierra, Version 10.12.6. We then used the original Spectre proof-of-concept C code [2] as a template to create an analogous SpectrePrime attack [12]. In our experiments, we observed 99.95% accuracy in leaking private information when running SpectrePrime on our hardware setup, where this accuracy percentage refers to the percentage of correctly leaked characters in the secret message averaged over the course of 100 runs.

As we have noted, CheckMate synthesizes multiple potential exploit variants. For example, in the originally synthesized SpectrePrime variant (with an instruction bound of 5), the first read instruction on Core 0 in Fig. 5d was eliminated entirely. This alternate attack mostly still worked, but with much lower accuracy. Thus, single-writer permission is more quickly returned to a core when it already holds the location (VA1 in Fig. 5d) in the shared state.

D. Mitigations

After testing SpectrePrime, we evaluated the exploit with a barrier between the condition for the branch that is speculated incorrectly and the body of the conditional. We found that both Intel's `mfence` and `lfence` instructions were sufficient to prevent the attack. Since Intel's `mfence` is not a serializing instruction intended to prevent speculation, it is possible that the fence simply skewed other subtle event timings on which our attack relies. It is also possible that the `mfence` was implemented in a way that enforces more orderings than required on our tested hardware. We did not investigate further.

Given our observations, and as confirmed by relevant companies, current software techniques that mitigate Meltdown and Spectre will also mitigate MeltdownPrime and SpectrePrime. On the other hand, *microarchitectural mitigation* of our Prime variants will require new considerations. Meltdown and Spectre arise by polluting the cache during speculation; MeltdownPrime and SpectrePrime are caused by speculative write requests triggering cache invalidation requests in a system that uses an invalidation-based coherence protocol. We expect that speculation-based security attacks will be a major area of study in the coming years, and only with the rigor and automation of a tool like CheckMate will we be able to gain confidence in our ability to one day declare a speculative microarchitecture provably secure.

VIII. Related Work

Prior axiomatic MCM analyses at the software, ISA, and implementation levels rely on graph-based happens-before modeling and cycle checks [29, 34, 36–38]. Some of these tools leverage RMF for directly comparing ISA MCMs, synthesizing litmus test suites, and synthesizing MCMs. Other work has looked at improving RMF techniques by modifying Kodkod (Alloy's backend) to handle higher-order relational models [39]. We tested CheckMate with this Alloy variant, but did not reap performance benefits. Furthermore, the security litmus tests we advocate for here are related to analysis techniques common in the MCM world [28, 30, 34, 36].

Many researchers have studied and implemented cache-based, timing-driven side-channel attacks. Early exploits targeted L1 data caches [40–44] and L1 instruction caches [45–48], with more recent exploits focusing attack efforts on last-level caches [19, 49–56] and even TLBs and page tables [57]. Related to how our Prime attacks use Prime+Probe to "re-implement" the Flush+Reload-based Meltdown and Spectre attacks, prior work has used Prime+Probe to improve the resolution of LLC Flush+Reload attacks [58]. Recent work also demonstrated cache-based, storage-driven attacks [20] and attacks on microarchitectural structures other than caches and TLBs, such as branch predictors [59, 60]. CheckMate is capable modeling and analyzing the effects of such hardware features on security. While our new attacks are the first proposed speculation-based attacks which leverage two cores and cache coherence protocol invalidations as part of the covert channel, various aspects of coherence protocols have been exploited for conducting different attacks [52, 61–63].

Some prior work aims to automate cache attacks, called *cache template attacks* [44, 53], but requires profiling application's executions. We aim to conduct early-stage verification. The primary contribution of CheckMate is a new approach and tool for evaluating the security of microarchitectures early in the design process. Other work advocates for using model checking to search for security vulnerabilities (particularly time-of-check to time-of-use) in protocols [64]. Similar in vein to CheckMate, CacheD [65] seeks to analyze programs to identify memory accesses that are vulnerable to timing side-channels. Instead, we identify vulnerable microarchitectural components. Finally, recent work calculates probabilities of various cache-based attacks on different system configurations [66]. In the future, CheckMate could aid in this type of analysis by focusing on the number of ways (false positives included) in which an exploit scenario could occur.

IX. Conclusion

CheckMate is a formal methodology and automated tool for efficiently and automatically synthesizing hardware- and system-aware exploit programs. Microarchitecture designs are complex and support their architectural specifications through a range of hardware-specific orderings and optimizations. In the absence of formal and automated techniques, this hardware complexity in combination with process- and system-level implementation detail significantly complicates the task of achieving full-system security. Drawing from composable axiomatic specifications of microarchitecture and systems features, CheckMate can integrate analysis across different modules to be more comprehensive than manual or prior approaches. This enables experts in hardware design, systems design, and security to collectively contribute their expertise to verifying the security of computing systems.

Our work showcases the power and applicability of CheckMate for analyzing and protecting against a wide range of security vulnerabilities. CheckMate runs on specifications of real processors and generates many variations of exploits without false positives. As one example, out-of-the box CheckMate generated over a dozen unique security litmus test variants representative of the publicly disclosed Meltdown and Spectre attacks. CheckMate also synthesized new exploits related to Meltdown and Spectre, but distinct—MeltdownPrime and SpectrePrime. Overall, CheckMate represents an important application of formal analysis techniques to hardware security verification.

Acknowledgment

This work was supported in part by U.S. NSF grants including SHF 1763838 and XPS 1533837. We also gratefully acknowledge an NVIDIA Graduate Research Fellowship, and a research gift from Intel Corporation.

References

[1] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *ArXiv e-prints*, Jan. 2018. [Online]. Available: https://arxiv.org/abs/1801.01207

[2] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *ArXiv e-prints*, Jan. 2018. [Online]. Available: https://arxiv.org/abs/1801.01203

[3] Intel, "Q2 2018 speculative execution side channel update," 2018, https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00115.html.

[4] J. Horn, "speculative execution, variant 4: speculative store bypass," 2018, https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.

[5] J. Stecklina and T. Prescher, "Lazyfp: Leaking fpu register state using microarchitectural side-channels," *ArXiv e-prints*, 2018. [Online]. Available: https://arxiv.org/abs/1806.07480

[6] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," *CoRR*, vol. abs/1807.03757, 2018. [Online]. Available: http://arxiv.org/abs/1807.03757

[7] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[8] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, "Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution," *Technical report*, 2018, see also USENIX Security paper Foreshadow [7] (https://foreshadowattack.eu).

[9] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, "Netspectre: Read arbitrary memory over network," *CoRR*, vol. abs/1807.10535, 2018. [Online]. Available: http://arxiv.org/abs/1807.10535

[10] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," *CoRR*, vol. abs/1807.10364, 2018. [Online]. Available: http://arxiv.org/abs/1807.10364

[11] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. B. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th USENIX Workshop on Offensive Technologies, WOOT 2018, Baltimore, MD, USA, August 13-14, 2018.*, 2018. [Online]. Available: https://www.usenix.org/conference/woot18/presentation/koruyeh

[12] C. Trippel, D. Lustig, and M. Martonosi, "MeltdownPrime and SpectrePrime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols," *CoRR*, vol. abs/1802.03802, 2018. [Online]. Available: http://arxiv.org/abs/1802.03802

[13] D. Lustig, M. Pellauer, and M. Martonosi, "PipeCheck: Specifying and verifying microarchitectural enforcement of memory consistency models," in *47th International Symposium on Microarchitecture (MICRO)*, 2014.

[14] Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi, "CCICheck: Using μhb graphs to verify the coherence-consistency interface," in *48th International Symposium on Microarchitecture (MICRO)*, 2015.

[15] D. Lustig, G. Sethi, M. Martonosi, and A. Bhattacharjee, "COATCheck: Verifying memory ordering at the hardware-OS interface," in *21st International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[16] C. Trippel, Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi, "TriCheck: Memory model verification at the trisection of software, hardware, and ISA," in *Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17, 2017.

[17] Y. A. Manerkar, D. Lustig, M. Martonosi, and M. Pellauer, "RTLCheck: Verifying the memory consistency of RTL designs," in *50th International Symposium on Microarchitecture (MICRO)*, 2017.

[18] D. Jackson, "Alloy analyzer website," 2012, http://alloy.mit.edu/.

[19] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games – bringing access-based cache attacks on AES to practice," in *2011 IEEE Symposium on Security and Privacy*, ser. SP '11, 2011.

[20] R. Guanciale, H. Nemati, C. Baumann, and M. Dam, "Cache storage channels: Alias-driven attacks and verified countermeasures," in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016.

[21] S. Mangard, "A simple power-analysis (spa) attack on implementations of the aes key expansion," in *5th International Conference on Information Security and Cryptology*, ser. ICISC'02, 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=1765361.1765392

[22] D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Side channels in cloud services: Deduplication in cloud storage," *IEEE Security and Privacy*, vol. 8, Nov. 2010.

[23] M. Backes, M. Dürmuth, S. Gerling, M. Pinkal, and C. Sporleder, "Acoustic side-channel attacks on printers," in *19th USENIX Conference on Security*, ser. USENIX Security'10, 2010.

[24] N. Homma, T. Aoki, and A. Satoh, "Electromagnetic information leakage for side-channel analysis of cryptographic modules," in *2010 IEEE International Symposium on Electromagnetic Compatibility*, July 2010.

[25] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, 2016.

[26] M. Neve and J.-P. Seifert, "Advances on access-driven cache attacks on aes," in *13th International Conference on Selected Areas in Cryptography*, ser. SAC'06, 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1756516.1756531

[27] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Litmus: Running tests against hardware," in *17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ser. TACAS'11/ETAPS'11, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1987389.1987395

[28] S. Mador-Haim, R. Alur, and M. M. K. Martin, "Generating litmus tests for contrasting memory consistency models," in *22nd International Conference on Computer Aided Verification*, ser. CAV'10, 2010. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14295-6_26

[29] D. Lustig, A. Wright, A. Papakonstantinou, and O. Giroux, "Automated synthesis of comprehensive memory model litmus test suites," in *Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17, 2017.

[30] S. Hangal, D. Vahia, C. Manovit, and J.-Y. J. Lu, "Tsotool: A program for verifying memory systems using the memory consistency model," in *31st Annual International Symposium on Computer Architecture*, ser. ISCA '04, 2004.

[31] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Fences in weak memory models," in *22nd International Conference on Computer Aided Verification*, ser. CAV'10, 2010. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14295-6_25

[32] C. Manovit, S. Hangal, H. Chafi, A. McDonald, C. Kozyrakis, and K. Olukotun, "Testing implementations of transactional memory," in *15th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '06, 2006.

[33] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'07, 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1763507.1763571

[34] J. Alglave, L. Maranget, and M. Tautschnig, "Herding cats: Modelling, simulation, testing, and data mining for weak memory," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 36, Jul. 2014.

[35] Caroline Trippel and Daniel Lustig and Margaret Martonosi, "CheckMate," 2018, https://github.com/ctrippel/checkmate.

[36] M. Martonosi *et al.*, "Check: Research tools and papers," 2017, http://check.cs.princeton.edu.

[37] J. Wickerson, M. Batty, T. Sorensen, and G. A. Constantinides, "Automatically comparing memory consistency models," *44th Symposium on Principles of Programming Languages (POPL)*, 2017.

[38] J. Bornholt and E. Torlak, "Synthesizing memory models from framework sketches and litmus tests," in *38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017, 2017.

[39] A. Milicevic, J. P. Near, E. Kang, and D. Jackson, "Alloy*: A general-purpose higher-order relational constraint solver," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, May 2015.

[40] C. Percival, "Cache missing for fun and profit," in *Proc. of BSDCan 2005*, 2005.

[41] M. Neve and J.-P. Seifert, "Advances on access-driven cache attacks on aes," in *13th International Conference on Selected Areas in Cryptography*, ser. SAC'06, 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1756516.1756531

[42] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes," in *2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, ser. CT-RSA'06, 2006. [Online]. Available: http://dx.doi.org/10.1007/11605805_1

[43] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on aes, and countermeasures," *J. Cryptol.*, vol. 23, Jan. 2010. [Online]. Available: http://dx.doi.org/10.1007/s00145-009-9049-y

[44] B. B. Brumley and R. M. Hakala, "Cache-timing template attacks," in *15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ser. ASIACRYPT '09, 2009. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-10366-7_39

[45] O. Aciiçmez, "Yet another microarchitectural attack:: Exploiting i-cache," in *2007 ACM Workshop on Computer Security Architecture*, ser. CSAW '07, 2007.

[46] O. Aciiçmez and W. Schindler, "A vulnerability in rsa implementations due to instruction cache analysis and its demonstration on openssl," in *2008 The Cryptopgraphers' Track at the RSA Conference on Topics in Cryptology*, ser. CT-RSA'08, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1791688.1791711

[47] O. Aciiçmez, B. B. Brumley, and P. Grabher, "New results on instruction cache attacks," in *12th International Conference on Cryptographic Hardware and Embedded Systems*, ser. CHES'10, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1881511.1881522

[48] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *2012 ACM Conference on Computer and Communications Security*, ser. CCS '12, 2012.

[49] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *16th ACM Conference on Computer and Communications Security*, ser. CCS '09, 2009.

[50] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy*, ser. SP '15, 2015.

[51] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

[52] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[53] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[54] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in paas clouds," in *2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14, 2014.

[55] Y. Yarom and N. Benger, "Recovering openssl ecdsa nonces using the flush+reload cache side-channel attack," *IACR Cryptology ePrint Archive*, vol. 2014, 2014.

[56] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: Practical cache attacks in javascript and their implications," in *22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, 2015.

[57] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing smap and kernel aslr," in *2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, 2016.

[58] M. Kayaalp, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, "A high-resolution side-channel attack on last-level cache," in *53rd Annual Design Automation Conference*, ser. DAC '16, 2016.

[59] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49, 2016.

[60] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," in *Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18, 2018.

[61] J. Horn, "Cpu security bug: information leak using speculative execution," 2017, https://bugs.chromium.org/p/project-zero/issues/attachmentText?aid=287305.

[62] A. Fogh, "Row hammer, java script and mesi," 2016, http://dreamsofastone.blogspot.ch/2016/02/row-hammer-java-script-and-mesi.html.

[63] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are coherence protocol states vulnerable to information leakage?" in *24th International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[64] S. Krsti, J. Yang, D. W. Palmer, R. B. Osborne, and E. Talmor, "Security of soc firmware load protocols," in *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, May 2014.

[65] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "Cached: Identifying cache-based timing channels in production software," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[66] Z. He, F. Liu, and R. B. Lee, "How secure is your cache against side-channel attacks?" in *50th International Symposium on Microarchitecture (MICRO)*, 2017.