Theme Article

Security Verification via Automatic Hardware-Aware Exploit Synthesis: The CheckMate Approach

Caroline Trippel
Princeton University

Daniel Lustig
NVIDIA

Margaret Martonosi
Princeton University

***Abstract*—Many hardware security exploits result from the combination of well-known attack classes with newly exploited hardware features. CheckMate is an approach and automated tool for evaluating microarchitectural susceptibility to specified attack classes, and for synthesizing proof-of-concept exploit code for susceptible designs.**

■ **A RECENT WAVE** of hardware security exploits—Spectre,[1] Meltdown,[2] and variants thereof—has heightened concerns about how the effects of speculative execution on nonarchitectural state (e.g., CPU caches) can make sensitive information available for extraction via side-channel attacks. What is novel and surprising about these exploits is not the side-channel attacks they use, but rather their clever ability to create practical working exploits out of a variety of widely implemented hardware speculation features that have been considered safe for decades. As speculation is so fundamental to CPU performance, it has become a grand challenge in the computer architecture community to develop a general-purpose solution for preventing Spectre-like attacks while nevertheless permitting forms of speculation that are indeed safe.

More broadly, because a microarchitecture's state space is so large and designs are too complicated, hardware and system designers cannot possibly reason manually about all possible ways in which nonarchitectural side effects could be

Digital Object Identifier 10.1109/MM.2019.2910010
Date of publication 11 April 2019; date of current version 8 May 2019.

exploited via side channels. Instead, ideally, a tool should *automatically* analyze a particular microarchitecture early in the design process, thereby enabling designers to find and fix any vulnerabilities prior to their hardware being released into the wild.

This paper presents CheckMate, a formal approach and automated tool for evaluating the susceptibility of a microarchitecture to formally specified classes of security exploits. If CheckMate determines that a given hardware design is vulnerable to a particular class of attacks, it produces as output both proof-of-concept exploit code snippets and visual depictions of the specific execution interleavings exploited by the attack. Importantly, CheckMate's techniques are not specific to Spectre and Meltdown. In fact, the CheckMate project was started prior to the public announcement of the attacks in January 2018. Because of the tool's generality, CheckMate was able to automatically synthesize two new attack variants, "SpectrePrime" and "MeltdownPrime," that had not been identified prior.

This paper presents CheckMate, a formal approach and automated tool for evaluating the susceptibility of a microarchitecture to formally specified classes of security exploits.

A key insight powering CheckMate is the perhaps surprising overlap between microarchitecture-level analysis of memory consistency model (MCM) bugs and of security vulnerabilities. Namely, both can result from specific problematic interleavings of execution steps in the hardware implementation when an application executes. Based on this insight, we leveraged our earlier work on MCM verification to build enhanced microarchitecture models and formal analysis techniques that are more suitable for exploring the security domain.

No solution to the Spectre/Meltdown problem will be generally accepted unless it is accompanied by a rigorous demonstration of its safety. CheckMate makes it possible to rigorously and automatically analyze such proposals to determine if they indeed plug the vulnerabilities that they claim to fix. We hope that hardware systems designers will use our open-source CheckMate tool (publicly available at github.com/ctrippel/checkmate) to conduct early-stage security verification of their proposed designs.

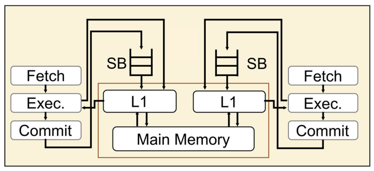
CHECKMATE: AN APPROACH AND TOOL FOR HARDWARE SECURITY VERIFICATION

CheckMate adds to the Check family of tools (see check.cs.princeton.edu) as the first hardware security verification tool in the suite; the others are designed for MCM verification. The Check tools feature a domain-specific language (DSL), μspec,[3] for encoding formal axiomatic specifications of microarchitectures and their relevant OS support called μspec models. A μspec model is comprised of axioms (essentially first-order logic statements) that define hardware-supported micro-operations (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 stage, out-of-order Execute stage, or FIFO Store Buffer). CheckMate conducts verification with respect to a μspec model of a hardware implementation (as do the other Check tools), evaluating its susceptibility to formally specified classes of security exploits that are provided in the same axiomatic format.

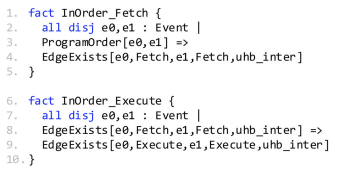
Microarchitecturally Happens-Before Graphs for Hardware Security Analysis

Automatic generation of potential vulnerabilities requires techniques for modeling and analyzing the particular behaviors that the attacks exploit. Given that all of the recent speculation-based attacks rely on leaking information via nonarchitectural state, any techniques to analyze security attacks must be able to account for speculation as well as other implementation-specific optimizations that might be exploited by side-channel attacks.

Many existing formal analysis techniques work at the level of an ISA or a programming language, since both frequently come with specifications (of varying degrees of formality) defining the set of legal behaviors, and the analysis tools can refer to the specifications as the arbiter of correctness. However, tools operating at these levels by definition have no chance of accounting for the types of hardware-specific optimizations that side-channels exploit. In contrast, CheckMate leverages techniques developed by other tools in the Check family[3]



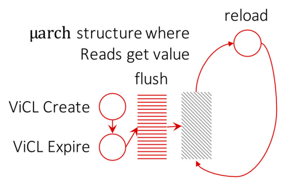
(a) Pedagogical 2-core, 3-stage, in-order μarch



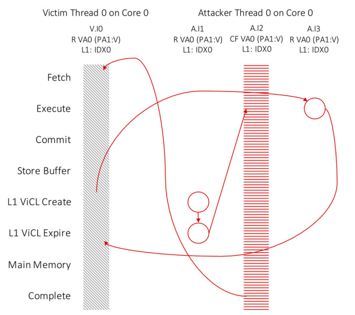
```
fact InOrder_Fetch {
  all disj e0,e1 : Event |
  ProgramOrder[e0,e1] =>
  EdgeExists[e0,Fetch,e1,Fetch,uhb_inter]
}

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

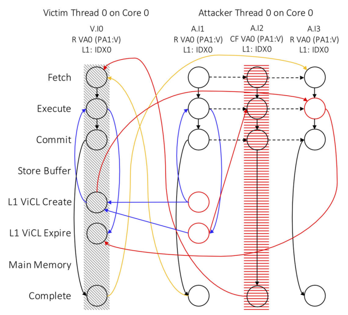
(b) μspec model excerpt corresponding to (a)



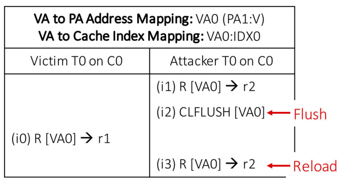
(c) FLUSH+RELOAD μhb pattern



(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.

Figure 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 subgraph, 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 synthesizes (e) and (f).

to build rigorous specifications of behaviors occurring at the microarchitecture level. With these implementation-aware specifications, CheckMate can conduct systematic analysis of the microarchitectural event orderings and interleavings that constitute a hardware security exploit.

More specifically, CheckMate adopts microarchitecturally happens-before (μhb) graphs from prior MCM work which were designed to model microarchitecture-specific program executions as directed graphs. Figure 1(e) gives an example. 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).

In this paper, 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. For example, both MCM and security analysis share two requirements: a way to determine if a specific program execution scenario is possible on a given microarchitecture, and a mechanism for analyzing microarchitectural event orderings and interleavings corresponding to a program's execution.

The first requirement above is met by a core principle of μhb graph analysis that cyclic μhb graphs represent impossible executions. For the second requirement, we extend and adapt μhb graphs in novel ways for security verification. Specifically, we introduce the concept of exploit patterns to represent hardware execution patterns indicative of security exploits as μhb subgraphs. When a μhb graph contains an exploit pattern, we say that the μhb graph represents an exploit program execution. We also leverage relational model finding (RMF) techniques rather than the custom solvers used in prior Check tools in order to enable the broader analysis that CheckMate needs to perform.

RMF for Efficient Hardware-Aware Exploit Program Synthesis

As described above, CheckMate conducts implementation-aware exploit program synthesis via synthesis of μhb graphs, where a μhb graph represents a specific execution of an exploit program. These synthesized program executions must satisfy two primary constraints: the μhb graph is acyclic, meaning it represents an observable program execution, and the μhb graph contains the user-specified exploit pattern. Figures 1(e), 2, and 3 give examples of CheckMate-generated μhb graphs.

RMF is a natural fit for implementation-aware program synthesis. 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.[4] 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 subgraph or pattern.

CheckMate stands out in the Check suite as the first security verification tool, but also in its use of RMF for conducting microarchitectural analysis. Thus, CheckMate's inputs—a μspec model and a formal description of class of security exploits, called an *exploit pattern*—are both provided in an embedding of the μspec DSL in the Alloy RMF DSL.[5] Using Alloy's RMF backend, CheckMate transforms the microarchitecture and exploit pattern specification inputs into hardware-specific exploit programs when the input microarchitecture is susceptible to the input vulnerability. Figure 1 gives an example of this transformation.

While RMF naturally aligns with CheckMate's program synthesis approach, a naïve embedding of μhb in the Alloy DSL was not sufficient to conduct program synthesis that terminates (within days) on even trivially simple designs. RMF can be challenged by huge search spaces that are infeasible to analyze in terms of time or memory. Thus, one of our contributions was making implementation-aware program synthesis *tractable*. Our optimized embedding of CheckMate in Alloy, detailed in Section V of the original conference paper,[6] prunes large portions of the search space during synthesis if found to be redundant; this makes implementation-aware program synthesis feasible. Specifically, CheckMate runtimes for the experiments in our case study range from minutes (for generating the first exploit on a susceptible microarchitecture) to hours (for generating all exploits within a user-provided bound).

Security Litmus Tests

CheckMate conducts bounded verification, meaning the user must specify a maximum exploit program size for synthesis (in terms of parameters such as the number of physical cores, threads, instructions, and processes). Ultimately, CheckMate outputs μhb graphs that represent executions of, *security litmus tests* to adapt a term from the MCM literature.[7–10] Security litmus tests are the most compact representation of an exploit program, meaning they contain the minimal number of operations necessary to produce the exploit pattern of interest. They are useful to output because: 1) they are much more practical to analyze with formal techniques than a full program due to their compact nature; and 2) they are nevertheless easily transformed into full executable programs when necessary.

CheckMate can automatically generate a large volume of security litmus test programs so that the user can identify all vulnerable hardware features. For example, given a Flush+Reload pattern [see Figure 1(c)], CheckMate effectively generates all possible ways in which an input microarchitecture could render the reload access a hit. Each generated program differs in some way, such as *how* the attack is performed. For example, in our case study (later in this paper), CheckMate-synthesized Meltdown and Spectre attack variants 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 litmus test pattern is also sufficiently general that CheckMate also 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).

SYNTHESIZING REAL-WORLD SECURITY EXPLOITS WITH CHECKMATE

To showcase CheckMate's applicability 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 and Spectre attacks. Upon switching the Flush+Reload pattern to a Prime+Probe pattern, CheckMate synthesized *new attacks* related to Meltdown and Spectre, yet distinct.[11]

CheckMate augments existing μspec modeling techniques 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 five-stage pipeline—Fetch, Execute, Reorder Buffer, Permission Check (PC), Commit—where processor cores have FIFO store buffers and private L1 caches connected to main memory. We note, however, that exploits generated for this simple pipeline have also been demonstrated to succeed on more complex real-world microarchitectures as well.

The μhb graphs in Figures 2 and 3 reflect the five-stage design. The μhb graphs in Figure 3 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 Figure 2 μhb graphs 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 (which, like the x86 CLFLUSH instruction, flushes a line from the cache), conditional branches, and full fences. The pipeline implements the Total Store Order MCM. Other micro-ops and/or MCMs are easy to add or implement as desired; the CheckMate approach is easily extensible.

Automatic Synthesis of Meltdown and Spectre

Figure 2 features μhb graphs synthesized by CheckMate, which correspond to security litmus test programs representative of the publicly disclosed Meltdown and Spectre attacks. The pattern from Figure 1(c) 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. We also note that CheckMate outputs detailed metadata such as the

1) index that each virtual (or physical, if physically mapped) address maps to in each cache;
2) physical address that each virtual address maps to;
3) physical core that a micro-op executes on;
4) process access permissions for each address;
5) cacheability attributes of virtual addresses. For clarity, Figure 2 includes a simplified subset.

The Meltdown graph in Figure 2 demonstrates how the lack of synchronization between the PC of a memory access and the fetching of said memory location into the cache can induce the Flush+Reload pattern [see Figure 1(c)]. The Spectre graph in Figure 2 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.

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. CheckMate also generated variants

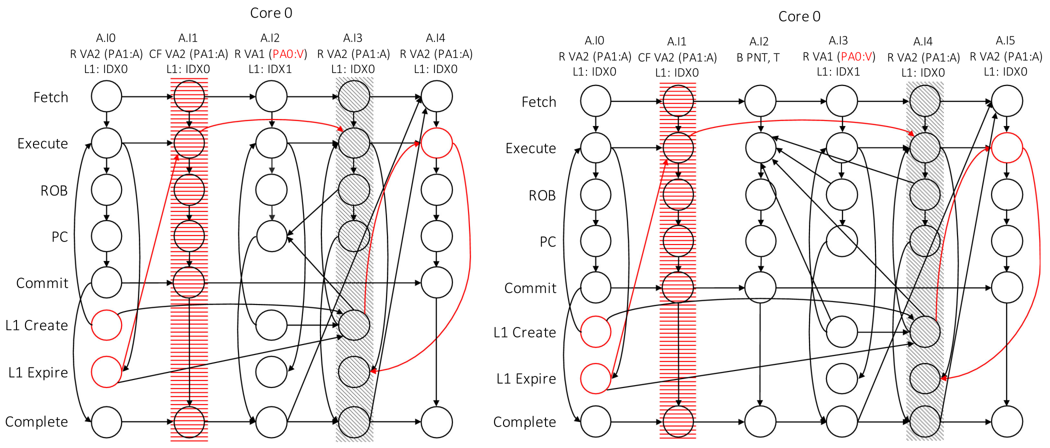

Core 0
A.I0 R VA2 (PA1:A) L1: IDX0
A.I1 CF VA2 (PA1:A) L1: IDX0
A.I2 R VA1 (PA0:V) L1: IDX1
A.I3 R VA2 (PA1:A) L1: IDX0
A.I4 R VA2 (PA1:A) L1: IDX0
Fetch
Execute
ROB
PC
Commit
L1 Create
L1 Expire
Complete
Core 0
A.I0 R VA2 (PA1:A) L1: IDX0
A.I1 CF VA2 (PA1:A) L1: IDX0
A.I2 B PNT, T
A.I3 R VA1 (PA0:V)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

Figure 2. Synthesized μhb graphs showing selected security litmus test executions for conducting Meltdown (left) and Spectre (right). Both exploit the Flush+Reload pattern in Figure 1(c). Some hardware locations have been omitted from the graphs for clarity as they do not contribute to the exploit. CF represents a CLFLUSH micro-op. B PNT, T represents a branch that is mispredicted as "not taken."

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.[3]

We make two key observations from these results. First, although manual efforts left Spectre and Meltdown undetected for decades, CheckMate was able to synthesize them automatically. We believe tools like CheckMate will be important tools in every architect's toolbox moving forward; it shifts security analysis from *ad hoc* inspection toward more thorough and automated techniques. Second, μhb graphs are instructive and can suggest edges whose addition mitigates an exploit by rendering the graph cyclic. This type of clear feedback will also be very useful for architects looking to patch vulnerabilities early, before the hardware is released into the wild.

Synthesizing New Exploits: MeltdownPrime and SpectrePrime

Figure 2 depicts μhb graphs synthesized by CheckMate which correspond to security litmus

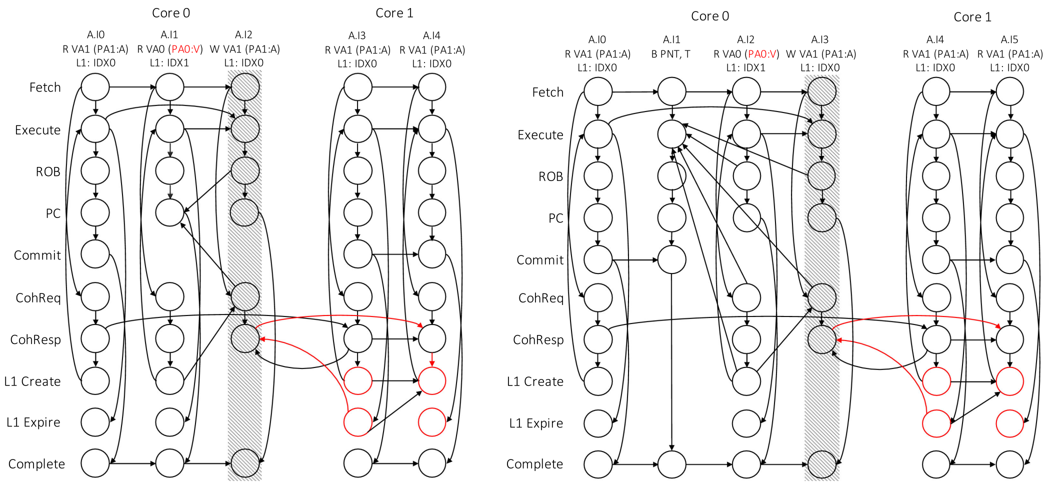

Core 0
A.I0 R VA1 (PA1:A) L1: IDX0
A.I1 R VA0 (PA0:V) L1: IDX1
A.I2 W VA1 (PA1:A) L1: IDX0
Core 1
A.I3 R VA1 (PA1:A) L1: IDX0
A.I4 R VA1 (PA1:A) L1: IDX0
Fetch
Execute
ROB
PC
Commit
CohReq
CohResp
L1 Create
L1 Expire
Complete
Core 0
A.I0 R VA1 (PA1:A) L1: IDX0
A.I1 B PNT, T
A.I2 R VA0 (PA0:V) L1: IDX1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

Figure 3. Synthesized μhb graphs showing selected security litmus test executions for conducting MeltdownPrime (left) and SpectrePrime (right). Both exploit the Prime+Probe pattern in Figure 4(b) of the original paper.[11] As in Figure 2, some hardware locations have been omitted for clarity. CF represents a CLFLUSH micro-op. B PNT, T represents a branch that is mispredicted as "not taken."

test programs representative of our new MeltdownPrime and SpectrePrime attacks. 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 Figure 4(b) in the original conference paper[6] 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. Nevertheless, the synthesis of all of these attack variants is a testament to the generality of the CheckMate tool.

Early-Stage Verification With Interactive Runtimes

The axiomatic microarchitecture models used by CheckMate can be constructed early in the hardware design process. As demonstrated by our case study, arbitrary complexity in the design specification is not necessary for synthesizing real-world exploits. In other words, security vulnerabilities that affect complex processor designs can be synthesized from rather abstract axiomatic representations of hardware. This feature enables hardware security verification to be moved much earlier in the design cycle of a microarchitecture so that vulnerabilities can be preemptively mitigated.

Furthermore, our optimizations (discussed in Section V of the original paper[6]) enable exploit program synthesis with CheckMate to achieve runtimes on the order of minutes for synthesizing the first security litmus test for a vulnerable microarchitecture and on the order of minutes to hours for synthesizing all possible security litmus tests within a user-specified bound. These interactive runtimes are also important for the type of early-stage analysis CheckMate is intended to conduct. Specifically, they enable microarchitects and other computer systems designers to iterate on their designs and specifications to achieve security.

BROADER IMPACT OF CHECKMATE: A “FISHING POLE” FOR IDENTIFYING HARDWARE SECURITY VULNERABILITIES

Over the past few decades, performance and power have become first-class design metrics that architects assess and optimize for early in the design process. In other words, rather than waiting to have a final working prototype, tools exist for conducting early-stage analysis. CheckMate demonstrates that security can be viewed in a similar way. In particular, CheckMate provides hardware systems architects with an automated tool for conducting early-stage hardware security verification. Despite abstract models that enable fast runtimes, CheckMate has demonstrated its value by identifying vulnerabilities that affect commercial processors.

CheckMate provides hardware designers with a new way to rigorously and systematically evaluate susceptibility of a microarchitecture to specified exploit classes. Hardware designs are complex and support their architectural specifications through a range of hardware-specific orderings and optimizations. Without 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. Rather than relying on *ad hoc* “brainstorming” approaches for discovering new implementation-specific security vulnerabilities, CheckMate provides a principled alternative rooted in techniques that have proven useful in the MCM verification space.

In addition to enabling early-stage hardware vulnerability detection, CheckMate can be used to evaluate both hardware and software mitigation strategies for identified exploits. As an example, the computer architecture community is working to develop optimal mitigation techniques for Spectre-like attacks: prohibiting speculation when it is potentially harmful while permitting forms of speculation that are indeed safe. Whether such mitigations are implemented in hardware, software, or some combination thereof, CheckMate can be used to determine if the target vulnerabilities are indeed mitigated.

Drawing from composable axiomatic specifications of microarchitecture and systems features, CheckMate integrates analysis across different modules to be more comprehensive than manual or prior approaches. Hardware designers, systems designers, and security experts can collectively use CheckMate to verify the security of computing systems. Overall, our work showcases the power and applicability of CheckMate for analyzing and protecting against a wide range of security vulnerabilities. In the future, we envision the CheckMate approach serving as the primary mechanism by which industrial-scale processor designs are verified secure against the wide range of confidentiality and integrity attacks rooted in event ordering issues.

SIDEBAR: HARDWARE SECURITY EXPLOITS

Side-channel attacks have demonstrated that observable state is not limited to architectural state; rather, it also includes nonarchitectural state (e.g., CPU caches). Nonarchitectural state cannot be directly accessed by user-facing instructions, but nevertheless can be *detected* (e.g., with a simple side-channel attack) and lead to information leakage. Furthermore, a new wave of side-channel attacks has revealed that nonarchitectural state can be modified by speculatively executed instructions in addition to those that commit. Thus, CheckMate supports modeling of nonarchitectural state as well as speculation. This sidebar gives an overview of how some hardware features (specifically, those most relevant to our case study) can be leveraged to induce information leakage.

Cache Timing Side-Channel Attacks

Side-channel attacks threaten confidentiality by exploiting implementation-specific behaviors with measurable dynamic state, for example, execution time, updates to storage elements, power consumption, resource sharing, acoustics, and radiation. *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.[12]

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, whereas timing-driven attacks measure timing of an entire security-critical operation. While the CheckMate approach can handle any security exploit scenario 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. Flush+Reload is the exploit pattern leveraged by the original Meltdown and Spectre attacks, and Prime+Probe is used by our case study later in this paper.

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 reaccessing 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 preloaded 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 the recent wave of speculation-based side-channel attacks, such as Meltdown and Spectre, is that microarchitectural speculation can be used to construct a side-channel attack that does not require shared virtual memory between the attacker and the victim.

One fundamental insight of the recent wave of speculation-based side-channel attacks, such as Meltdown and Spectre, is that microarchitectural speculation can be used to construct a side-channel attack that does not require shared virtual memory between the attacker and the victim. We describe this next.

Why Speculation Matters

Many processors employ hardware optimizations such as speculative execution 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 *nonarchitectural* state to leak-sensitive information into a side channel for extraction via some well-known cache side-channel attack. As a specific example, Meltdown and Spectre leverage the effects of speculative execution on cache state in combination with a Flush+Reload attack. 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 (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 versus outside a sandbox). To make the distinction clear, we define *sensitive* data as that which should only be accessible by the victim, and *nonsensitive* data as that which is accessible by the attacker. Meltdown and Spectre perform this step differently (see next).
3) While in the window of speculative execution, the attacker accesses nonsensitive data whose address is dependent (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 nonsensitive memory access from step 3.
5) From the address result of step 4, the attacker determines the sensitive data that were used to calculate it in step 3.

REFERENCES

1. P. Kocher *et al.*, “Spectre attacks: Exploiting speculative execution,” Jan. 2018. [Online]. Available: https://arxiv.org/abs/1801.01203
2. M. Lipp *et al.*, “Meltdown,” Jan. 2018. [Online]. Available: https://arxiv.org/abs/1801.01207
3. M. Martonosi *et al.*, “Check: Research tools and papers,” 2017. [Online]. Available: http: //check.cs.princeton.edu

4. E. Torlak and D. Jackson, “Kodkod: A relational model finder,” in *Proc. 13th Int. Conf. Tools Algorithms Construction Analysis Syst.*, 2007.
5. D. Jackson, “Alloy analyzer website,” 2012. [Online]. Available: http://alloy.mit.edu/
6. C. Trippel, D. Lustig, and M. Martonosi, “CheckMate: Automated exploit program generation for hardware security verification,” in *Proc. 51st Int. Symp. Microarchit.*, 2018.
7. J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, “Litmus: Running tests against hardware,” in *Proc. 17th Int. Conf. Tools Algorithms Construction Analysis Syst., Part Joint Eur. Conf. Theory Practice Software*, 2011.
8. S. Hangal, D. Vahia, C. Manovit, and J.-Y. J. Lu, “TSOtool: A program for verifying memory systems using the memory consistency model,” in *Proc. 31st Annu. Int. Symp. Comput. Archit.*, 2004.
9. D. Lustig, A. Wright, A. Papakonstantinou, and O. Giroux, “Automated synthesis of comprehensive memory model litmus test suites,” in *Proc. 22nd Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2017.
10. S. Mador-Haim, R. Alur, and M. M. K. Martin, “Generating litmus tests for contrasting memory consistency models,” in *Proc. 22nd Int. Conf. Comput. Aided Verification*, 2010.
11. C. Trippel, D. Lustig, and M. Martonosi, “MeltdownPrime and SpectrePrime: Automatically-synthesized attacks exploiting invalidation- based coherence protocols,” arxiv:1802.03802, 2018. [Online]. Available: http://arxiv.org/abs/1802.03802
12. Q. Ge, Y. Yarom, D. Cock, and G. Heiser, “A survey of microarchitectural timing attacks and countermeasures on contemporary hardware,” *J. Cryptogr. Eng.*, vol. 8, pp. 1–27, 2016.

Caroline Trippel is currently a PhD student in the Computer Science Department, Princeton University. Her research focuses on computer architecture, with a particular emphasis on concurrency and security verification in heterogeneous parallel systems. She has an MS in computer science from Princeton University. She is a student member of the Association for Computing Machinery (ACM). Contact her at ctrippel@princeton.edu.

Daniel Lustig is a senior research scientist at Nvidia. His research focuses on architecting efficient and correct memory systems, with a particular focus on memory consistency models. He has a PhD in electrical engineering from Princeton University. He is a member of the Association for Computing Machinery (ACM) and the IEEE. Contact him at dlustig@nvidia.com.

Margaret Martonosi is the Hugh Trumbull Adams ’35 Professor of Computer Science at Princeton University. Her research focuses on computer architecture and mobile systems, with a particular emphasis on verification, performance, and power efficiency in heterogeneous systems. She has a PhD in electrical engineering from Stanford University. She is a Fellow of the IEEE and the Association for Computing Machinery (ACM). Contact her at mrm@princeton.edu.