SpecSafe: Detecting Cache Side Channels in a Speculative World

ROBERT BROTZMAN*, Pennsylvania State University, USA
DANFENG ZHANG, Pennsylvania State University, USA
MAHMUT TAYLAN KANDEMIR, Pennsylvania State University, USA
GANG TAN, Pennsylvania State University, USA

The high-profile Spectre attack and its variants have revealed that speculative execution may leave secret-dependent footprints in the cache, allowing an attacker to learn confidential data. However, existing static side-channel detectors either ignore speculative execution, leading to false negatives, or lack a precise cache model, leading to false positives. In this paper, somewhat surprisingly, we show that it is challenging to develop a speculation-aware static analysis with precise cache models: a combination of existing works does not necessarily catch all cache side channels. Motivated by this observation, we present a new semantic definition of security against cache-based side-channel attacks, called Speculative-Aware noninterference (SANI), which is applicable to a variety of attacks and cache models. We also develop SpecSafe to detect the violations of SANI. Unlike other speculation-aware symbolic executors, SpecSafe employs a novel program transformation so that SANI can be soundly checked by speculation-unaware side-channel detectors. SpecSafe is shown to be both scalable and accurate on a set of moderately sized benchmarks, including commonly used cryptography libraries.

CCS Concepts: • **Security and privacy → Formal security models**; **Side-channel analysis and countermeasures**.

Additional Key Words and Phrases: side channel, cache, symbolic execution

ACM Reference Format:
Robert Brotzman, Danfeng Zhang, Mahmut Taylan Kandemir, and Gang Tan. 2021. SpecSafe: Detecting Cache Side Channels in a Speculative World. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 129 (October 2021), 29 pages. https://doi.org/10.1145/3485506

1 INTRODUCTION

Side-channel attacks have increasingly become a security concern due to their ability to leak sensitive data quickly and stealthily. They leverage information obtainable by observing the characteristics of a machine during a program's execution. These attacks have been demonstrated on modern machines using information from the CPU cache [Bernstein 2005; Osvik et al. 2006], execution time [Kocher 1996], power usage [Kocher et al. 1999], electromagnetic fields [Agrawal et al. 2003; Longo et al. 2015] and many more.

*The majority of this author's work was completed while a student at Pennsylvania State University and was finished while employed at Peraton Labs.

Authors' addresses: Robert Brotzman, robert.brotzman@peratonlabs.com, Pennsylvania State University, USA; Danfeng Zhang, zhang@cse.psu.edu, Pennsylvania State University, USA; Mahmut Taylan Kandemir, mtk2@psu.edu, Pennsylvania State University, USA; Gang Tan, gtan@psu.edu, Pennsylvania State University, USA.

2475-1421/2021/10-ART129
https://doi.org/10.1145/3485506

Among all side channels, cache side channels, the focus of this paper, tend to be one of *the most* dangerous since they can be launched without physical access to the victim and while maintaining high throughput. These attacks have been shown to be effective in cloud environments [Ristenpart et al. 2009; Wu et al. 2012; Xu et al. 2011] and even in Intel's SGX secure enclave [Brasser et al. 2017; Götzfried et al. 2017; Schwarz et al. 2017; Van Bulck et al. 2017; Xiao et al. 2017]. Most recently, Spectre [Kocher et al. 2019] and its variants have shown that speculative program execution creates a new dimension of side-channel attacks: cache side channels combined with speculative program execution allow an attacker to learn arbitrary data from a victim's memory space [Kocher et al. 2019].

To defend against cache side channels, one approach is to manually identify potential vulnerabilities and fix them in the source code. For example, well-studied cryptographic libraries such as OpenSSL and libgcrypt, follow this approach. This manual approach is time-consuming, and might miss vulnerable program points, especially those that elude known vulnerability patterns, such as Spectre [Kocher et al. 2019]. Another approach typically applied to Spectre attacks, is to use a compiler-aided patching system to protect every branch in a program, including those that may not be vulnerable. Unfortunately, blindly protecting every single branch will often result in prohibitive performance overheads. Moreover, conventional side channels remain.

In this paper, we develop a static program analysis that automatically detects cache side channels in the source code of unsafe languages such as C, with four important goals:

- Soundness: the analysis catches all cache side channels in a speculative world.
- Flexibility: the analysis reasons about security against a wide range of cache attacks.
- Precision: the analysis embodies cache models to accurately reason about cache status.
- Efficiency: the analysis is applicable to real-world applications.

In the past, much work has been done to automatically detect cache side channels. However, most of them ([Almeida et al. 2016; Brotzman et al. 2019; Doychev et al. 2013; Doychev and Köpf 2017; Wang et al. 2019a, 2017]) fall short in soundness: they only detect conventional cache side channels (i.e., in the absence of speculative execution). To detect code vulnerable to Spectre attacks, a recent tool oo7 [Wang et al. 2019b] searches for vulnerable code patterns—however, no rigorous definition of side-channel security is provided; as a result, the patterns might miss vulnerabilities.

More recently, sound static analysis [Cauligi et al. 2020; Cheang et al. 2019; Guarnieri et al. 2020] catching cache side channels in a speculative world have been developed. However, they fall short in flexibility and precision: they all use the program counter security model [Molnar et al. 2006] or the constant-time programming principle [Almeida et al. 2016; Cauligi et al. 2020].As a consequence, they disallow *any* branching on sensitive data. While these models are sound and they simplify static program analysis (as they over-approximate cache effects without modeling the cache), they also introduce false positives since branching on sensitive data does not necessarily introduce cache-based side channels. For example, the following program resembles preloading lookup tables, a permissive and efficient countermeasure against cache attacks on AES:

```
preload A[0] and A[512]; if (secret) then x:=A[0]; else x:=A[512];
```

In this example, the value of `secret` *does not* affect the final cache status, despite the branch on sensitive data. Note that such permissive countermeasures have been adopted as options in popular cryptographic implementations such as libgcrypt and OpenSSL [Doychev and Köpf 2017]. For conventional cache side channels, various cache side channel detectors (e.g., [Brotzman et al. 2019; Doychev et al. 2013; Doychev and Köpf 2017; Wang et al. 2019a]) are built on precise cache models to reduce false positives. However, to the best of our knowledge, no existing sound analysis in a speculative world meets these flexibility and precision goals.

In this paper, we first show that, with speculative execution, it is *subtle* to build a sound, flexible and precise analysis. In particular, the idea of *combining* analysis tools for conventional cache side channels and tools for speculative execution side channels (i.e., tools enforcing security definitions in [Cheang et al. 2019; Guarnieri et al. 2020]) *do not necessarily* cover all possible cache side channels. Indeed, we construct simple code patterns to demonstrate that the combination is sound only when the conventional side channel detector follows the rigid policy of disallowing any branching on sensitive data. In other words, it is unsound to combine tools enforcing security definitions in [Cheang et al. 2019; Guarnieri et al. 2020], for speculative side channels, and tools that use a precise cache model (e.g., [Brotzman et al. 2019; Doychev et al. 2013; Doychev and Köpf 2017; Wang et al. 2019a]), for conventional side channels. We also build covert channels on the code patterns to demonstrate their feasibility. These patterns demonstrate that the ignored attack surface in previous security definitions are realistic on commodity hardware. Although we have not detected these patterns in the wild in our experiments, we emphasize that our contribution is to identify the limitations of existing security definitions, and show that an attacker can utilize them to bypass state-of-the-art tools for detecting cache side channels.

Second, we propose a security definition for side-channel security against all forms of cache side channels (including conventional, Spectre, and the new Spectre instances introduced in this paper). Our security definition can be soundly applied to a variety of attack and cache models. Then, we propose SpecSafe to detect violations of the security definition in reasonably sized applications. At the core of SpecSafe is a novel and provably sound program transformations that allow us to soundly capture all cache side channels, including those manifesting from speculative execution, by reusing existing conventional side channel detectors. SpecSafe then performs fine-grained symbolic execution on the transformed code to identify vulnerable code regions.

In order to show the effectiveness of SpecSafe, we apply it to instances of the Spectre vulnerability by Kocher [2018], our newly discovered vulnerabilities, along with multiple crypto routines from libgcrypt 1.8.5.

To summarize, the contributions of this work include:

• **New Speculative Cache Side Channel:** We analyze and identify the limitations of existing security definitions used in state-of-the-art tools that detect cache-based side channels manifesting from speculative execution. Based on the limitations discovered in existing security definitions, we develop and show the practicality of a new kind of speculative cache attack that belongs to the Spectre Variant 1[1] attack family, but is more stealthy than existing ones since they cannot be detected by existing security definitions.

• **Speculative-Aware Noninterferece (SANI):** Motivated by the new vulnerabilities that elude existing security definitions, we propose SANI, a security definition that captures conventional cache attacks, Spectre attacks, and beyond. Unlike other existing security definitions, SANI is versatile enough to be applied to many different cache models (e.g., LRU and age-based) in a sound way. Hence, SANI rules out both conventional and speculative cache-based side channels. More importantly, it can reduce false positives in practical cases when compared with more rigid security definitions, due to the more accurate modeling of the cache.

• **Novel Program Transformation:** SpecSafe adopts a novel code transformation that assures the following property: if the transformed program is free of conventional side channels, then the original code satisfies SANI (i.e., is free of both conventional and speculative side channels). The novel design allows us to reuse existing conventional side channel detectors on the transformed code to enforce SANI on the original code. This is in contrast with other state-of-the-art tools that

[1] As in [Kocher et al. 2019], we use "Spectre Variant 1" to refer to Spectre attacks that exploit conditional branch misprediction. This is the same as "Spectre-PHT" in [Canella et al. 2019].

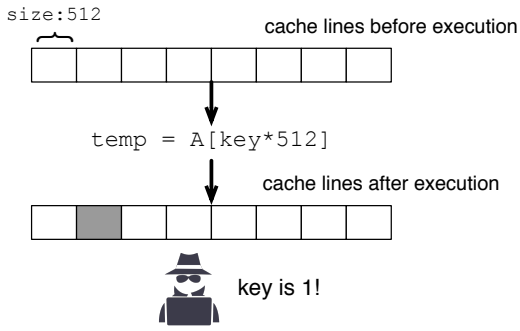

Fig. 1. An example of cache-based side channels.

are built from scratch to detect speculative side channels. Moreover, SpecSafe takes a more holistic approach by detecting both conventional and speculative side channels simultaneously, which avoids the potential pitfalls of checking incompatible properties in different tools to get a sound result. Moreover, we provide a formal proof of the soundness of the code transformation.

• **Speculative-Aware Symbolic Execution:** The core of SpecSafe is a symbolic execution engine that reasons about program variables, cache state, and speculative execution at the same time. After symbolic execution, SpecSafe emits verification conditions for the analyzed program and feeds them to an SMT solver; satisfiable verification conditions imply a violation of SANI in the source code.

• **Side-Channel Detection:** We analyze programs known to be vulnerable to Spectre-like attacks to demonstrate that SpecSafe is capable of detecting all of the vulnerabilities. We also show that SpecSafe is the only tool that can detect the new side channels discovered in this work. Lastly, we apply SpecSafe to a variety of crypto benchmarks from libgcrypt to analyze code used in practice. We find two *previously unknown* possible side channels, which to the best of our knowledge, have not been found before in libgcrypt ciphers.

2 BACKGROUND AND MOTIVATION

2.1 Conventional Cache Side Channels

A program might leak information via its secret-dependent footprints in the CPU cache, known as cache side channels. We use the simple example in Figure 1 to illustrate how confidential data, i.e., the value of `key`, affects cache. In this example, `A` is an array storing public data and we assume cache line size to be 512 bytes. Note that each possible value of *key* results in the access of a different cache line in this example; as a result, an attacker can learn the value of `key` by observing which cache line is being accessed, through techniques such as *prime and probe* [Bonneau and Mironov 2006; Liu et al. 2015; Osvik et al. 2006; Tromer et al. 2010; Zhang et al. 2012].

2.2 Speculative Cache Side Channels

Speculative Execution. Modern CPUs employ many optimizations to improve performance. One such optimization is *speculative execution*. Speculative execution executes instructions speculatively but ensures that the instructions are retired in sequential order. One example on modern CPUs that will cause speculative execution is when the CPU encounters a branch instruction. If the CPU does not yet know the result of the branch (e.g., waiting on a memory read), it will guess the condition using a branch predictor and speculatively jump to the instructions corresponding to the guessed branch. If the branch predictor is wrong, it would have to undo the changes made to the

architectural state to ensure correctness and then execute the correct branch. However, changes to the microarchitectural state, such as the CPU cache, are not rolled back. Most importantly, when the rolled back instructions leave secret-dependent footprints in the CPU cache, an attacker can reveal them since the microarchitectural state is not rolled back.

Spectre Attack. Spectre [Kocher et al. 2019] is a vulnerability that utilizes speculative execution to leak the content of an arbitrary location from a victim's memory space. This is possible since secret-dependent footprints left by speculative execution are not rolled back in architectural components such as the CPU cache. The example code fragment shown in Listing 1, taken from [Kocher et al. 2019], demonstrates a Spectre variant 1 vulnerability, the focus of prior static program analysis [Cheang et al. 2019; Guarnieri et al. 2020; Wang et al. 2019b]:

```
1  if (idx < b_size) {
2       temp = A[B[idx]*512]
3  }
```

Listing 1. Spectre Example

Assume both `A` and `B` are arrays of some size and contain only public information. Somewhere else in the victim's address space, there is some sensitive data that the adversary aims to learn. Further, assume that the adversary can control `idx`, and she can train the branch predictor such that it predicts that `idx < b_size` evaluates to `true`. Then, the adversary can supply an out-of-bounds `idx` value to make `B[idx]` access the sensitive data of interest. The speculative execution of the example code will then result in an out-of-bounds memory access to read the sensitive data and access a cache line that is determined by the value of the sensitive data. The adversary can then learn the value of the sensitive data by performing a cache attack to determine which cache line is accessed in array `A` (as we illustrated in Figure 1, except that the confidential data now is `B[idx]` instead of `key`).

2.3 Spectre Defenses

Many defenses have been proposed to prevent Spectre Variant 1 attacks. At the hardware level, one can partition the cache into security domains to prevent sharing (e.g., [Kiriansky et al. 2018]). Other systems [Yan et al. 2018] seek to prevent speculated memory accesses from being visible until the program uses them.

At the software level, there are two popular approaches. The first uses fences to prevent speculation. The other, known as speculative load hardening (SLH) [Carruth 2019], introduces artificial data dependencies and masking to secure data.

Fencing. The fencing defense places a special hardware instruction after conditional branches to prevent speculative execution from happening. For example, to protect the code in Listing 1, a *lfence* instruction would be placed between lines 1 and 2. The effectiveness of the defense assumes that the architecture prevents speculative execution past the *lfence* instruction, which has been confirmed by Intel on current Intel architectures [Intel 2018b]. A downside of the defense, however, is its high overhead when the defense is applied to all conditional branches (which essentially turns off speculative execution). One refinement is then to apply it only after branches that are potentially vulnerable to Spectre, which reduces overhead.

Speculative Load Hardening. Speculative Load Hardening, first described by Carruth [2019], uses two components to protect a conditional branch: creating a data dependence on the branch test and masking sensitive data during mis-speculation.

Typically, the implementation of SLH uses conditional moves to create a data dependency between potentially dangerous memory loads (i.e. line 2 of Listing 1) and the condition flags. The data dependency prevents the later dependent loads from executing before the branch condition has been resolved on current processors [Carruth 2019; Oleksenko et al. 2018]. Speculative load hardening also uses a mask as a fallback to ensure even if the load happens no sensitive data is revealed. The mask sets the address accessed during a mis-speculation to be a fixed value so no data can be leaked.

2.4 Threat Model

We assume an adversary who is co-located on the same physical machine as the victim. The adversary cannot directly observe the victim's program counter, memory bus and execution time. However, he can control the victim's branch predictors; he further shares a data cache with the victim, and can observe the state of the shared data cache. In the rest of the paper, whenever we mention cache, it refers to the data cache.

Note that our threat model is weaker than the attack model in constant-time programming [Almeida et al. 2016; Cauligi et al. 2020; Daniel et al. 2020], which also assumes that the adversary can directly observe the victim's program counter and execution time. However, our threat model more precisely reflects the threat models behind Spectre attacks [Kocher et al. 2019] and other cache attacks in cloud environments [Ristenpart et al. 2009; Wu et al. 2012; Xu et al. 2011]. The threat model is also consistent with the ones in prior work on conventional cache side channels [Brotzman et al. 2019; Doychev et al. 2013; Doychev and Köpf 2017; Wang et al. 2019a, 2017]. Moreover, we consider a trace-based attacker, in which an attacker can learn the shared cache state after each program point in the victim program; this models asynchronous cache attacks [Gullasch et al. 2011; Liu et al. 2015; Yarom and Falkner 2014].

Among all known variants of Spectre attacks, we focus on Spectre Variant 1 attacks [Kocher et al. 2019], also classified as Spectre-PHT attacks in [Canella et al. 2019], which utilize speculation due to conditional branches. Hence, our threat model not only covers "bounds check bypass" variants as described by Intel [2018a], but also other attacks exploiting conditional branch misprediction, including the new instances we present in Section 4.1. However, speculative execution caused by indirect branches [Kocher et al. 2019], return instructions [Koruyeh et al. 2018; Maisuradze and Rossow 2018] and speculative stores [Horn 2018] are outside the scope of this paper; we leave a static analysis targeting those Spectre variants as future work.

3 RELATED WORK

We describe previous work that identifies different classes of cache attacks using various program analyses.

3.1 Conventional Cache Attacks Detection

SpecSafe is built on symbolic execution. A few existing systems also use symbolic execution to model cache behavior in programs and detect potential side channels. One such approach is CacheD [Wang et al. 2017], which looks at memory accesses along a concrete program trace, and identifies a side channel when a memory access's address depends on a secret value. However, this approach analyzes only one concrete trace; it may miss side channels that can be exploited only by observing multiple execution paths. More relevant is CaSym [Brotzman et al. 2019], which employs a sound symbolic execution approach to verify the absence of cache side-channels using various abstract cache models. This approach is able to capture all program execution paths, and thus does not miss cache side channels for a given attack and cache model. However, like CacheD, CaSym [Brotzman et al. 2019] cannot detect speculative cache side channels.

Another common approach is to use abstract interpretation to reason about the cache state [Doychev et al. 2013; Doychev and Köpf 2017; Wang et al. 2019a]. CacheAudit [Doychev et al. 2013; Doychev and Köpf 2017] uses abstract interpretation to compute an upper bound on the amount of leakage possible via side channels using the CPU cache. This is achieved by using a variety of cache and attack models. Almeida et al. [2016] propose to use abstract interpretation in combination with a 2-safety property to verify programs to be constant-time. CacheS [Wang et al. 2019a] uses scalable abstract interpretation to pinpoint where in a program leakage happens. Besides using a different approach of static analysis, none of these systems can detect speculative cache side channels.

3.2 Spectre Side-Channel Detection

More recently, automatic static analysis for Spectre-style attacks have been proposed. One of the first such attempts is oo7 [Wang et al. 2019b]. It uses taint analysis along with predefined code patterns to detect potential Spectre variant 1 vulnerabilities in programs. However, it does not provide a security definition of what kinds of side channels it can identify, and as we will show, their pattern-based approach may miss new Spectre side channels. The most related work to ours is Spectector [Guarnieri et al. 2020], which offers both a formal security definition called speculative noninterference (SNI) and uses symbolic execution to detect violations of SNI. However, it assumes the program counter security model [Molnar et al. 2006] and hence, disallows *any* branching on sensitive data. Moreover, as we show in Section 5, when combined with other attack models, SNI might miss vulnerabilities caused by speculative execution. A parallel work to Spectector, Cheang et al. [2019] present a similar security definition, called trace property-dependent observational determinism (TPOD). While this definition has been specified as a 4-safety property, it is noted in [Cheang et al. 2019; Guarnieri et al. 2020] that it is similar to SNI. A recent work called InSpectre [Guanciale et al. 2020] builds on these ideas to detect multiple variants of Spectre vulnerabilities not captured by other existing works. Unlike all symbolic executors above that face the code coverage limitation (i.e., the code is not verified until all paths are analyzed by those tools), SpecSafe uses a novel program transformation that soundly removes loops in the original code.

Additionally, SpecFuzz [Oleksenko et al. 2020] uses dynamic symbolic execution to detect bounds-check bypass vulnerabilities using a technique called speculative exposure that modifies an IR to simulate branch mispredictions. Their technique only detects bounds-check bypass during a branch misprediction, a subclass of Spectre Variant 1 attacks. Hence, SpecFuzz cannot find the new Spectre instances we discover in this paper, since those instances do not rely on bounds-check bypass.

4 NEW SPECULATIVE SIDE CHANNELS

In this section, we first present examples for new instances of Spectre Variant 1 attacks shown in Figure 2 and show that such attacks are possible in practice. What makes the new attacks especially interesting is that, as we show in Section 5, existing security definitions of cache side channels, with or without speculative execution, fail to catch them. Our new security definition in Section 5, on the other hand, captures all avenues of cache-based leakage with or without speculative execution.

4.1 Attack Code Snippets

We present three code snippets that demonstrate new kinds of speculative cache attacks, which we call *conditional branch speculative side channels*, abbreviated as CB-channels.

To clearly show the leakage in each code snippet, we use "`begin...rollback`" to enclose memory addresses accessed by instructions that are *speculatively executed, but are rolled back due to misprediction.* Consider the code in Listing 1. With $idx \geq b\_size$ and the predictor predicting the true branch to

```
1 if (key == 0) {
2 temp=A[key*512];
3 }
4 else {
5 temp=A[key*0];
6 }
```

(a)

```
if (key%2 == 0) {
temp=A[((key+1)%2)*512];
}
else {
temp=A[(key%2)*512];
}
```

(b)

```
if (key == 0)
  d = a;
...
if (a == 0)  b = 0;
else c = 0;
```

(c)

Fig. 2. Cache side channels that elude existing security notions. key is a one-bit secret in all the examples.

be taken, the following memory accesses are generated:[2] begin $B + idx, A + B[idx] * 512$ rollback Due to the accessed memory address $A + B[idx] * 512$ (during speculative execution), the cache line being accessed reveals the value of $B[idx]$; hence, the program leaks the memory content at $B[idx]$.

Instance 1. The first instance of CB-channels is shown in Figure 2a. In this example, *key* is a one-bit secret and we assume $key * 512 < A_{size}$ (i.e., there is no out-of-bound access). Although the code always accesses $A[0]$ without speculation, it emits the following execution subtrace when $key = 1$, and the predictor predicts the true branch to be taken: begin $A + 512$ rollback. Therefore, the value of *key* directly affects the cache line being accessed; the value of *key* can be revealed via a cache attack.

Instance 2. The second instance, shown in Figure 2b, is more subtle than Instance 1. In this example, only *key* is confidential. Depending on the value of *key*, the following access subtraces will be generated when the predictor predicts the true branch to be taken:

$$\begin{cases} A + 512 & \text{when } key\%2 = 0 \\ \texttt{begin}\ A\ \texttt{rollback},\ A + 512 & \text{when } key\%2 \neq 0 \end{cases}$$

In other words, the cache line corresponding to address A is accessed *if and only if* the last bit of *key* is 1. Note that since the hardware does not roll back microarchitectural effects, including cache effect, a cache attack can reveal at least one bit of key. We note that in this example no information is leaked from memory accesses during speculation, but *whether speculation happens or not* leaks information. Leaking data solely based upon whether speculation occurs or not, to the best of our knowledge, has *not been previously explored* as a source of leakage. Technically, the new attacks can be classified as instances of Spectre Variant 1 attacks since they use conditional branches. We note that, while this sample code leaks one bit at a time, if it was in a loop, it could leak multiple sensitive bits.

Instance 3. The last instance, shown in Figure 2c, is similar to the example in Figure 2b since it also leaks data based on whether speculative execution occurs. However, the reason for the leakage is a bit different: in this instance, the leakage occurs as a result of whether or not the variable a is cached or not. This is because speculation will likely occur only when variable a is uncached. Depending on the value of *key*, the following two traces will be generated when the predictor predicts the false branch at line 5:

$$\begin{cases} key, a, d, \ldots, a, b & \text{when } key = 0 \text{ and } a = 0 \\ key, \ldots, a, \texttt{begin}\ c\ \texttt{rollback}, b & \text{when } key \neq 0 \text{ and } a = 0 \end{cases}$$

[2]We provide the formal program semantics that emits traces and cache models that compute cache states from traces in Appendix A.

Accuracy	>75%	>95%	>99%
Instance 1	64747	11181	2038
Instance 2	7040	2652	1630
Instance 3	5355	120	12
AES-128 (with Instance 1)	2456	1816	1604

Table 1. The throughput (bits per second) of the new side channels in Figure 2 as well as AES-128 embedded with Instance 1 at different levels of accuracy.

Note that when $key = 0$, speculative execution is unlikely to occur for any reasonable speculation depth since a is cached at line 4 due to an earlier read in `d=a`; the second trace has a uncached when reaching the second branch, and therefore there is speculative execution since the processor will need to wait for a to be fetched from memory.

Whether key is 0 or not is revealed by two side channels in the code: (1) whether the cache lines corresponding to a and d are accessed after the first branch, and (2) whether the cache line corresponding to variable c is accessed or not. We note that the first one is a conventional side channel, but the second one only shows up with speculative execution.

Although all examples in Figure 2 contain key-dependent branches, they are not essential. Consider the following variation of Figure 2c:

```
a = arr[key*512];
...
if (arr[0])  b = 0;
else c = 0;
```

In this case, we know that `arr[0]` is cached only if `key` is zero. As discussed previously, the presence or absence of speculation again leaks the value of `key` in this example.

4.2 Measuring Channel Throughput

For each of the samples in Figure 2, we construct an attack to reveal confidential data by mostly following the Spectre attack code presented in [Kocher et al. 2019] as a template. The only subtlety is for Instance 3; we need to ensure enough time elapses to allow the branch condition's data cached before line 5 while not allowing instructions to cause it to be evicted from the cache. We insert a sufficient number of `NOP` to tackle the issue.

Moreover, we demonstrate how to use these code snippets as gadgets when they are embedded inside sensitive applications to exfiltrate sensitive data. This is interesting since, as we will show in Section 5, state-of-the-art analysis for speculative side channels cannot detect them. In particular, we embedded Instance 1 code into the AES encryption routine so that it reveals one bit of the encryption key per AES encryption and obtain a reasonable amount of bandwidth as shown in the last row of Table 1.

Results. Table 1 shows the throughput of each code snippet with various accuracy rates. Here, accuracy refers to the percentage of bits correctly detected. Note that each of the code patterns presented in Figure 2 can leak thousands of bits per second with high accuracy.

The results indicate that CB-channels can leak significant amounts of data in a short period of time, even though they do it bit by bit. We also note that the throughput difference among the attacks largely depends on the number of operations in each code fragment.

Statements
$s ::= s_1; s_2 \mid \texttt{skip} \mid \texttt{fence} \mid X := E \mid A[X] := E \mid$
$\texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2 \mid \texttt{while } B \texttt{ do } S$
Expressions
$E ::= n \mid X \mid A[E] \mid E_1 \otimes E_2$
Boolean Expressions
$B ::= E_1 \odot E_2 \mid \neg B \mid B_1 \wedge B_2 \mid B_1 \vee B_2$

where $\otimes$ represents binary arithmetic operations, and $\odot$ represents binary comparison operators

Fig. 3. Language Syntax

5 SPECULATIVE-AWARE NONINTERFERENCE

We first formalize language, attack and cache models to reason about cache side channels. Then, we show that, except under the most conservative model, existing security definitions for cache side channels cannot identify the new CB-channels in Figure 2. Then, we introduce speculative-aware noninterference (SANI), a new semantic definition of cache side-channel security against a variety of attack and cache models.

Language Syntax. To formalize SANI and compare it with previous security definitions, we define a simple imperative language in Figure 3. This language models important features of the C language, such as sequential composition, assignments, branches, and loops. `skip` represents a no-op, which becomes handy to model single-sided branches, for instance. The only non-standard command is `fence`, which is a special command that will temporarily disable speculation.

For expressions, the language uses n as a numerical constant, X as a program variable, $A[E]$ an array with a base location of A at offset E. Lastly, the language allows both arithmetic operations using $\otimes$ and Boolean operations using $\odot$ on two expressions.

Speculative Program Semantics. Unlike standard semantics without speculation, the speculative semantics is parameterized on two features:

- *prediction oracle* O: a partial function that takes the id of a branch command and returns a queue of booleans that predict the outcomes of the branch.
- speculative transaction's length w: during speculation, instructions from the reorder buffer are executed speculatively; so the size of the reorder buffer on a processor serves as an upper bound on the maximum length of w [Doweck et al. 2017].

Since the semantics mostly follows prior work [Guarnieri et al. 2020], we omit the evaluation rules (formalized in Figure 7 in the Appendix). Given a program s, an oracle O and transaction length w, we use $[\![s]\!]^{O}_{\langle m,w\rangle}$ to denote the sequence of emitted events of s with initial memory m. Each event tracks the memory locations being accessed and program instructions being executed; we use $pc(i)$ to denote the latter. Finally, we use τ to denote event traces, and assume a trace view model `view` (to be defined next), which intuitively specifies the side channel information visible to the attacker. Hence, two events traces τ and τ' are indistinguishable for an attacker with `view` if $\texttt{view}(\tau) = \texttt{view}(\tau')$.

Attacker View and Cache Model. To reason about cache side channels, one crucial and challenging question is how to model attacker's view of the cache. To do so, we first assume *no cache model* and define attacker's view as done in [Doychev and Köpf 2017] – an attacker model can be abstracted as a projection of τ that is visible to an attacker. For instance, the most conservative model is when the attacker can *directly* view all the memory locations being accessed and program instructions

being executed (i.e., $\texttt{view}(\tau) = \tau$. A *block-trace observer* [Doychev and Köpf 2017] models attackers that can monitor memory accesses at the granularity level of cache lines; it can be formalized (on a 64-bit machine with cache line size of 512 bits) as $\tau \downarrow^{63:8}_{mem}$, a projection of τ that only contains bits 8 to 63 of each memory access, excluding *pc's*. A hierarchy of memory trace observers can be formalized in a similar way [Doychev and Köpf 2017].

The definition above implicitly uses trace projection to estimate cache effects. For better accuracy, we can tailor a more detailed cache model $\texttt{cacheM}(\tau)$ that computes the (abstract) cache state *at the end* of a subtrace τ, which captures the attacker view at the corresponding program point. Hence, we define the attacker view with projection $\downarrow$ and cache model $\texttt{cacheM}$ on the entire trace τ as:

$$\texttt{view}(\tau) \triangleq \texttt{cacheM}(\emptyset), \texttt{cacheM}(\tau \downarrow_{\preccurlyeq 1}), \cdots, \texttt{cacheM}(\tau \downarrow_{\preccurlyeq |\tau\downarrow|})$$

where $\tau \downarrow_{\preccurlyeq i}$ represents the prefix of τ up to position i. As another example, a more realistic cache model (e.g., LRU model) might only return the last memory location on τ when it is not among the most recently used locations. We refer to [Brotzman et al. 2019; Doychev et al. 2013] for various cache models ranging from very intricate models including cache size and replacement policies to abstract cache models that provide a good balance between analysis scalability and precision [Brotzman et al. 2019].

In this paper, we develop a general static analysis (Section 6) that can be parameterized on any attacker view defined as above. However, for simplicity, we will use a concrete and representative attacker view, called block trace view, for the rest of this section:

$$\texttt{view}_{\texttt{BT}}(\tau) \triangleq \tau \downarrow^{63:8}_{mem}$$

This attacker view is particularly interesting since (1) most known cache-based attacks exploit observations at the granularity of cache lines, (2) most tools for detecting conventional cache side channels use this attacker view[3], and (3) recall that we will show that conventional side channel detectors are unable to catch the new CB-channels in Figure 2; a tool based on more precise cache models [Brotzman et al. 2019; Doychev et al. 2013] will miss vulnerabilities if they are missed under $\texttt{view}_{\texttt{BT}}$.

5.1 Limitations of Existing Definitions

5.1.1 Conventional Cache Side Channels. Cache side-channel security can be formalized as a noninterference property [Goguen and Meseguer 1982]: confidential data does not affect the cache state. Let policy P be a set of public variables. We first define a low-equivalence relation on memory as follows:

Definition 1 (Low Equivalence). *Given a policy P, two memories m_1 and m_2 are low-equivalent according to P, written as $m_1 \simeq_P m_2$ if and only if $\forall x \in P.\ m_1(x) = m_2(x)$.*

To capture side channels without speculation, we note that given an oracle that *perfectly* predicts every single branch outcome and with $w = 0$, the program semantics coincides with standard semantics without speculation. Hence, we define the perfect oracle to be the one that correctly predicts every single branch outcome for the execution of program s on memory m.[4] To simplify notation, we use $[\![s]\!]^{perf}_{\langle m,w\rangle}$ to denote the execution of s under m, transaction length w, and the corresponding perfect oracle. Cache side-channel security without speculation is then formalized as follows:

[3]We note that this is the attack model assumed by CacheD [Wang et al. 2017] and CacheS [Wang et al. 2019a]; it is called "block-trace observer" in [Doychev and Köpf 2017].

[4]Note that since a program s might take different control flows during different executions, the perfect oracle is parameterized with both s and m.

Definition 2 (Cache noninterference (CNI)). *A program s with policy P satisfies CNI if for all O, $m_1 \simeq_P m_2$, we have*

$$\texttt{view}(\llbracket s \rrbracket^{perf}_{\langle m_1,0\rangle}) = \texttt{view}(\llbracket s \rrbracket^{perf}_{\langle m_2,0\rangle})$$

Consider the code in Figure 1 with policy $P \triangleq \{A_{\leq \texttt{a\_size}}, A\}$, meaning that the address of A and the value of A's elements within bound are public. Then, it is easy to show that with $m_1 \simeq_P m_2$ but $m_1(key) = 0$ and $m_2(key) = 1$, their memory trace projections are different between m_1 and m_2. That is, the program violates CNI, and hence, is vulnerable to traditional cache attacks with $\texttt{view}_{\texttt{BT}}$.

5.1.2 Speculative Side Channels. Definition 2 falls short for Spectre attacks since it completely ignores speculative execution. Recent work [Cheang et al. 2019; Guarnieri et al. 2020] proposed new security definitions to identify side channels that only exhibit themselves with speculative execution. In this paper, we focus on the definition called *speculative noninterference (SNI)* [Guarnieri et al. 2020] since both security definitions proposed concurrently by Guarnieri et al. [2020] and Cheang et al. [2019] are similar, despite different ways of formalism [Cheang et al. 2019].

Informally, SNI requires that, for every $m_1 \simeq_P m_2$, if their non-speculative traces *have exactly the same event traces*, including program counters, then their speculative traces will do the same. We present a version of SNI generalized with an attack view as follows and emphasize that the original SNI definition in [Guarnieri et al. 2020] assumes $\texttt{view}_{\texttt{SNI}}(\tau) = \tau$.

Definition 3 (SNI-GEN). *A program s with policy P satisfies SNI-GEN with transaction length w if for all $m_1 \simeq_P m_2$,*

$$\texttt{view}(\llbracket s \rrbracket^{perf}_{\langle m_1,0\rangle}) = \texttt{view}(\llbracket s \rrbracket^{perf}_{\langle m_2,0\rangle}) \implies \texttt{view}(\llbracket s \rrbracket^{mis}_{\langle m_1,w\rangle}) = \texttt{view}(\llbracket s \rrbracket^{mis}_{\langle m_2,w\rangle})$$

where $\llbracket s \rrbracket^{mis}_{\langle m,w\rangle}$ is the execution of s under m, transaction length w, and an oracle that always mispredicts (called the *always-mispredict speculative semantic* in [Guarnieri et al. 2020]). This approach proposes that the maximum leakage occurs when a mis-speculation occurs, which we show next may not always be the case.

5.1.3 Limitations of Existing Security Definitions. SNI is defined to detect side channels that only show up with speculative execution. However, as we show next, information leakage in Figure 2 remains "undetected" by both CNI and SNI. We will discuss instance 2 (Fig. 2b) in detail to identify the limitations; other variants are also vulnerable for similar reasons as instance 2.

With instance 2, consider two low-equivalent memories with $m_1(key) = 0$ and $m_2(key) = 1$, we have

$$\begin{aligned}
\llbracket s \rrbracket^{perf}_{\langle m_1,0\rangle} &= key, pc(2), key, A+512, temp \\
\llbracket s \rrbracket^{perf}_{\langle m_2,0\rangle} &= key, pc(5), key, A+512, temp \\
\llbracket s \rrbracket^{mis}_{\langle m_1,w\rangle} &= key, \texttt{begin}\ pc(5), key, A, temp\ \texttt{rollback}, \\
&\qquad pc(2), key, A+512, temp \\
\llbracket s \rrbracket^{mis}_{\langle m_2,w\rangle} &= key, \texttt{begin}\ pc(2), key, A, temp\ \texttt{rollback}, \\
&\qquad pc(5), key, A+512, temp
\end{aligned}$$

Limitation 1: SNI, which instantiates Definition 3 with $\texttt{view}_{\texttt{SNI}}(\tau) = \tau$, assumes a (conservative) cache side channel detector that rejects any program with a secret-dependent branch. This is dangerous since the assumption is implicit in the definition, and state-of-the-art detectors for conventional cache side channels do not necessarily follow this assumption: as discussed earlier,

most tools for detecting conventional cache side channels use the block trace view $\texttt{view}_{\texttt{BT}}$, or an even more precise cache model.

To see why a mismatch in attacker views is problematic, we first point out that with the block trace view $\texttt{view}_{\texttt{BT}}$ (which filters out PC values in traces), Definition 2 holds. Note that despite a secret-dependent branch in instance 2, there is no cache side channel in non-speculative execution, since the same sequence of memory addresses are accessed. Further, SNI holds since the assumption $\texttt{view}_{\texttt{SNI}}([\![s]\!]^{perf}_{\langle m_1,0\rangle}) = \texttt{view}_{\texttt{SNI}}([\![s]\!]^{perf}_{\langle m_2,0\rangle})$ is false due to $[\![s]\!]^{perf}_{\langle m_1,0\rangle} \neq [\![s]\!]^{perf}_{\langle m_2,0\rangle}$. Hence, both CNI (with $\texttt{view}_{\texttt{BT}}$ or a more precise cache model) and SNI (with $\texttt{view}_{\texttt{SNI}}$) fail to reject the vulnerable code of instance 2. But, as we showed in Section 4.1, the code leaks information via a cache side-channel only when speculative execution is enabled.

Limitation 2. Instance 2 also shows a more fundamental limitation of SNI and its generalized form SNI-GEN: they both ignore leakage resulting from whether or not to roll back a prediction transaction.

To see that more clearly, we use $\texttt{view}_{\texttt{BT}}$ for both CNI and SNI-GEN, which avoids the mismatching attacker view issue discussed in Limitation 1. It is easy to check that the traces satisfy both Definitions 2 and 3 (with $\texttt{view}_{\texttt{BT}}$) due to the same sequences of memory addresses being accessed under oracles *perf* and *mis*. Nevertheless, whenever the branch predictor O is trained to predict that the true branch should be taken, the program will produce the following two traces given m_1 and m_2 respectively:

$$
\begin{aligned}
[\![s]\!]^{O}_{\langle m_1,1\rangle} &= key, pc(2), key, A+512, temp \\
[\![s]\!]^{O}_{\langle m_2,1\rangle} &= key, \texttt{begin}\ pc(2), key, A, temp\ \texttt{rollback}, \\
&\qquad pc(5), key, A+512, temp
\end{aligned}
$$

Hence, the value of *key* is leaked by deciding whether memory location A is accessed during the execution.

With a more careful inspection, we observe that the issue here is that SNI only checks information leakage between two executions that *both* mispredict. However, the fact that misprediction might also leak information. Hence, SNI-GEN is insufficient under various attack models, such as $\texttt{view}_{\texttt{BT}}$: with the same predictor, one execution ($[\![s]\!]^{O}_{\langle m_1,1\rangle}$) might not have a misprediction while the other execution does ($[\![s]\!]^{O}_{\langle m_2,1\rangle}$). As we demonstrated in Section 4.1, an attacker can reveal at least one bit of *key* from Instance 2.

5.2 Speculative Aware Noninterference

Due to the limitations of SNI-GEN, we introduce a new noninterference definition called *Speculative Aware Noninterference* (SANI). The idea behind SANI is to compare the behavior of the program's memory access trace with any possible outcomes of the branch predictor, *without* assuming the original program is side-channel free. More formally:

Definition 4 (Speculative Aware Noninterference (SANI)). *A program s with policy P satisfies SANI w.r.t. attacker view* $\texttt{view}$ *and transaction length w if $\forall O, m_1 \simeq_P m_2$, we have*

$$\texttt{view}([\![s]\!]^{O}_{\langle m_1,w\rangle}) = \texttt{view}([\![s]\!]^{O}_{\langle m_2,w\rangle})$$

To see how the new definition correctly identifies CB-channels, consider Instance 2 again. Given an oracle that predicts the true branch, and two low-equivalent memories with $m_1(key) = 0$ and

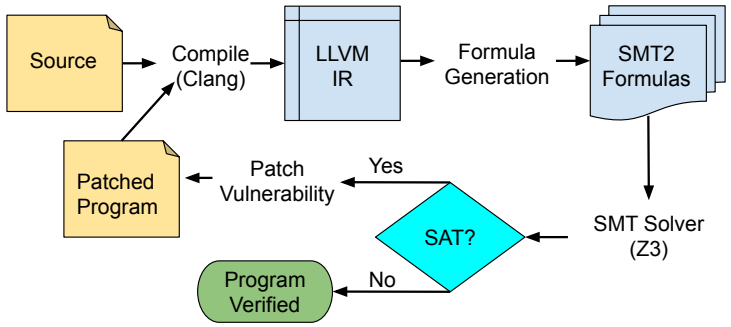

Fig. 4. SpecSafe's workflow.

$m_2(key) = 1$, we have

$$[\![s]\!]^{O}_{\langle m_1,1\rangle} = key, pc(2), key, A + 512, temp$$
$$[\![s]\!]^{O}_{\langle m_2,1\rangle} = key, \texttt{begin}\ pc(2), key, A, temp\ \texttt{rollback}, pc(5), key, A + 512, temp$$

Hence, SANI is violated since the cache line corresponding to address A is only accessed in the second trace.

Take Away. SNI (i.e., Definition 3 instantiated with $\texttt{view}_{\texttt{SNI}}$) is not compatible with many tools for detecting conventional cache side channels. Under other (more precise) cache models, however, SNI-GEN is unsound since it ignores leakage resulting from whether or not to roll back a prediction transaction, as illustrated by Instance 2. On the other hand, the SANI definition (Definition 4) is applicable to a variety of cache models, including the ones used by state-of-the-art cache side channel detectors. To classify conventional vs. Spectre side channels, we can use CNI (Definition 2) vs. SANI. For the examples presented in Figure 2, we can correctly classify Instance 1 and Instance 2 as Spectre side channels since they both violate SANI but not CNI.

6 SPECSAFE

In this section, we present a speculative-aware symbolic executor, SpecSafe, to enforce SANI. Compared with existing symbolic executors that are capable of detecting Spectre-like attacks (e.g., [Cauligi et al. 2020; Guarnieri et al. 2020]), SpecSafe adopts a novel code transformation, which benefits from the following property: if the transformed code is free of conventional side channels (i.e., satisfying CNI in Definition 2), then the original code satisfies SANI. The novel design allows us to reuse existing conventional side channel detectors on the transformed code to enforce SANI on the original code. Furthermore, the transformation soundly removes loops in the original code. Hence, SpecSafe avoids the code coverage limitation of other symbolic executors (i.e., the code is not verified until all paths are analyzed by those tools). Although soundly removing loops is only possible with the cost of some precision loss, we show that SpecSafe is still precise enough for real-world applications in Section 7.

SpecSafe's workflow is shown in Figure 4. SpecSafe performs code transformation on LLVM IR code and then symbolic execution on the transformed code. The symbolic execution reasons about architectural states (e.g. variable values), micro-architectural states (e.g. cache states), and speculative behavior. Security conditions needed for SANI are formalized as logical formulas, which are sent to an SMT solver (Z3). If the SMT solver determines the formula is satisfiable, it will also

$$\frac{s_1 \rightharpoonup s_1' \quad s_2 \rightharpoonup s_2' \quad K \text{ a set of sensitive variables at the last reset}}{s_1; s_2 \rightharpoonup s_1'; \texttt{check}\ K; s_2'} \text{(SE-Composition)}$$

$$\frac{id \text{ a unique integer} \qquad s_1 \rightharpoonup s_1' \quad s_2 \rightharpoonup s_2' \quad s_* \rightharpoonup s_*'}{\begin{array}{l}\texttt{if}\ B\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \rightharpoonup \texttt{capture}(id);\ \texttt{if}\ B\ \texttt{then}\ \big(\texttt{if}\ \neg pred_{id}\ \texttt{then}\ [s_2'; s_*']_w; \texttt{rollback}(id)\big); s_1' \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{else}\ \big(\texttt{if}\ pred_{id}\ \texttt{then}\ [s_1'; s_*']_w; \texttt{rollback}(id)\big); s_2';\end{array}} \text{(SE-IF)}$$

$$\frac{s \rightharpoonup s' \quad s_* \rightharpoonup s_*' \quad K_0 \text{ a set of sensitive variables at the beginning of loop}}{\begin{array}{l}\texttt{while}\ B\ \texttt{do}\ s \rightharpoonup \text{reset}; \texttt{capture}(id); \{\text{if}\ B\ \text{then}\ \big(\texttt{if}\ \neg pred_{id}\ \texttt{then}\ [s_*']_w; \texttt{rollback}(id)\big); s'; \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{else}\ \big(\texttt{if}\ pred'_{id}\ \texttt{then}\ [s'; (\texttt{while}\ B\ \texttt{do}\ s'); s_*']_w; \texttt{rollback}(id)\big)\}; \texttt{check}\ K_0; \text{reset};\end{array}} \text{(SE-WHILE)}$$

Fig. 5. Transformation rules for composition, `if` and `while` with transaction length parameter w. The helper function $[S]_w$ statically extracts w instructions from s, unless a `fence` is encountered during that. Furthermore, s_* is used to denote the sub-program right after the sub-program being analyzed (i.e., the extra instructions being padded in case of misprediction).

report line numbers identifying the vulnerabilities, making it easy to patch the program and restart verification. If the SMT solver reports the formulas are unsatisfiable, it is safe to conclude there is no cache-based leakage in the program under conventional or speculative execution.

6.1 Program transformation

In order to model the speculative behavior of a program, we introduce a program transformation that embeds the speculative behavior into the transformed program. The syntax of transformed programs includes two new primitives:

- `capture`(id): an instruction that records the current architectural state (i.e., memory and registers) of the program so that it can be later retrieved with id.
- `rollback`(id): an instruction that resets the program's architectural state (but not the microarchitectural state such as cache) to the recorded state with id.

To soundly remove loops, the transformed program also includes two security-related primitives that are introduced by Brotzman et al. [2019]:

- reset: an instruction that sets current architectural and microarchitectural state to an arbitrary value.
- check K: an instruction that checks cache side channels assuming only the secret-variable set K carries confidential data at the last reset.

The most interesting rules are listed in Figure 5; for all other instructions, the transformed program is the same as the original one.

Composition. A sequential composition $s_1; s_2$ is transformed to s_1; check K; s_2, where K is a set of sensitive variables at the last reset, provided by a sound taint analysis (described in Section 7). The inserted check ensures that the attacker view right after s_1 reveals no confidential data.

If transformation. Intuitively, programs during ordinary execution have two possible control flows when it encounters a branch; with speculation, there are four possible control flows due to the two extra branches on a fresh variable $pred_{id}$, which is true if and only if the predictor predicts

the true branch. For a misprediction, speculatively executed code up to transaction length w (i.e., $[s_2'; s_*']_w$, $[s_1'; s_*']_w$) are explicitly instrumented into the resulting code, followed by `rollback`(id), which resets the architectural state (but not the microarchitectural state) to the state before the branch.

Consider the example from Figure 2b. Following the transformation rule in Figure 5, the source code is transformed into a speculation-aware form, assuming $w = 1$:

```
capture(1);
if (key%2 == 0) {
  if (¬pred1){
    temp := A[(key%2)*512];
    rollback(1);
  }
  temp := A[((key+1)%2)*512];
}
else {
  if (pred1){
    temp := A[((key+1)%2)*512];
    rollback(1);
  }
  temp = A[(key%2)*512];
}
```

Hence, the transformation faithfully models four possible control flows of the example, assuming `E1=A[(key%2)*512]` and `E2=A[((key+1)%2)*512]`:

```
capture(1);temp:=E1;rollback(1);temp:=E2;
capture(1);temp:=E2;
capture(1);temp:=E2;rollback(1);temp:=E1;
capture(1);temp:=E1;
```

In practice, the program will speculate up to a certain depth w, specified as a parameter of SpecSafe. The transformation statically inserts $[s]_w$, which denotes the first w instruction of s. Note that with a `fence` command, the remaining commands are cut off in the construction of $[s]_w$.[5]

Loop transformation. The loop transformation is motivated by the transformation presented by Brotzman et al. [2019]. However, the latter faces two limitations when being applied to speculative execution. First, it does not consider speculative behavior at all. Second, it assumes that the loop condition is not sensitive. In comparison, our novel loop transformation is both speculative-aware and allows sensitive loop conditions. Moreover, we formally prove that our transformation soundly captures both conventional and speculative side channels, while a formal proof is absent in [Brotzman et al. 2019]. In fact, a sound loop transformation rule is subtle to develop; the soundness proof helped us to catch a few flaws in our early trials of the transformation.

The transformation, presented in Figure 5, performs the following major tasks. First, it conservatively resets the initial state to over-approximate both architectural and microarchitectural states right before each loop iteration.[6] Second, it introduces two new variables $pred_{id}$ and $pred'_{id}$ to model

[5] Although we formalize SANI and symbolic execution on a C-like source language for simplicity, SpecSafe is built on LLVM IR. The length of IR code is not the same as the length of micro-ops, but the gap can be reduced by a cost model that maps from IR instructions to the number of micro-ops.

[6] It might seem conservative to reset the state coming in so we know it is in an arbitrary low-equivalent state. We note that it is possible to make the transformation rule more accurate by requesting a loop invariant from the user and then use it to improve precision. However, we refrained from asking for a loop invariant to make the symbolic execution fully automatic.

the outcome of the branch predictor. Unlike in the if-statement case, two such variables are needed since the loop condition is checked multiple times during execution. Hence, each iteration might encounter different prediction outcomes. Third, under a mis-prediction, speculatively executed instructions are instrumented and then rolled back. Note that when b is `false` but $pred'_{id} = \texttt{true}$, the instrumented commands are $[s'; (\texttt{while } b \texttt{ do } s'); s'_*]_w$, which essentially unrolls the loop to extract the first w instructions.

The last component of the transformation: check K_0; $reset$; first ensures that there is no cache side channel after each loop iteration, forming a loop invariant, and then resets the final architectural and microarchitectural states to an arbitrary state as an over-approximation when checking the code after the loop.

Transformation Soundness. The novel program transformation in Figure 5 satisfies the following property: for any terminating program, oracle O and transaction length w, if the transformed code satisfies CNI (Definition 2), then the original code satisfies SANI (Definition 4). More formally:

Theorem 1 (Transformation Soundness).

$$\forall s, \underline{s}, w, O.\ s \textit{ terminates} \wedge s \rightharpoonup \underline{s} \implies$$
$$\forall m_1 \simeq_P m_2.\texttt{view}([\![\underline{s}]\!]^{perf}_{\langle m_1,0\rangle})) = \texttt{view}([\![\underline{s}]\!]^{perf}_{\langle m_2,0\rangle}) \implies$$
$$\forall m_1 \simeq_P m_2.\ \texttt{view}([\![s]\!]^{O}_{\langle m_1,w\rangle}) = \texttt{view}([\![s]\!]^{O}_{\langle m_2,w\rangle})$$

Proof. By induction on the structure of the original program s. In most cases, we can show that the transformed program approximates (in conventional semantics) speculative execution. The tricky case is for the while loop: the inserted check K_0 and reset intuitively enforces the following loop invariant: any two executions of the loop starting from the same attacker view results in the same final attacker view.

The full proof can be found in the appendix of [Brotzman 2021]. □

We note that the soundness result assumes that the addresses of data and instructions remain unchanged by the transformation. This is reasonable as the transformation is at source-code level and the transformed code is for verification purpose only (i.e., the extra variables and duplicated instructions in the transformed code are not physically present in memory). Hence, the analysis on the transformed code can assume that addresses of data and instructions remain unchanged, and that the duplicated instructions share the same addresses as their counterparts in the original program.

6.2 Cache-Aware Symbolic Execution

Our program transformation gives us the flexibility to use existing tools that detect conventional cache-based side channels. We choose to use CaSym [Brotzman et al. 2019] as the core symbolic execution engine for SpecSafe since it provides cache-aware symbolic execution with support for configurable attack and cache models. We first introduce the basics of CaSym and then show how SpecSafe improves it to reason about the new speculation-related primitives (i.e., `fence`, `capture`, and `rollback`).

Symbolic State. A symbolic state σ is comprised of three components: the program variables $\overline{X} = \overline{e}$, a cache variable $C = ce$, and a path condition Ψ. Here, symbolic program state $\overline{e}$ and path condition Ψ are mostly standard. ce represents a symbolic cache state, following the abstract cache models of CaSym. At a high-level, CaSym symbolically executes a program starting from an initial state σ_0 and computes a symbolic state σ_η at each program position η. To compute a sound symbolic

$$
\begin{array}{lll}
\mathsf{SE}[\![\texttt{fence}]\!]_\sigma & := & \sigma \\
\mathsf{SE}[\![\texttt{capture}(id)]\!]_\sigma & := & \sigma, \texttt{mems}[id] = \overline{e} \\
\mathsf{SE}[\![\texttt{rollback}(id)]\!]_\sigma & := & \overline{X} = \texttt{mems}[id] \wedge C = ce \wedge \Psi
\end{array}
$$

Fig. 6. Speculative-aware symbolic execution of SpecSafe assuming that the input σ is $\overline{X} = \overline{e} \wedge C = ce \wedge \Psi$. The symbolic execution also maintains a global list of symbolic memory snapshots mems, indexed by transaction identifiers.

state at each program point, CaSym uses novel techniques to merge execution paths. Notably, given two paths with path conditions Ψ_1 and Ψ_2, CaSym introduces a new logical variable Ψ and adds the following equation $\Psi = \Psi_1 \ \vee \ \Psi = \Psi_2$ along with other equations that merge cache and memory states. We refer to CaSym [Brotzman et al. 2019] for the technical details.

Verification Condition. To detect potential cache side channels, CaSym generates verification conditions (i.e., SMT constraints) in the following way.

CaSym starts with an initial symbolic state: $\overline{X} = \overline{x} \wedge C = c \wedge \texttt{true}$, where $\overline{x}$ are initial symbolic values of variables $\overline{X}$ and c is the initial cache state. Then, CaSym symbolically executes the program being analyzed (as sketched earlier) and computes the final symbolic state σ, which represents the final program and cache states, in the form of $\overline{X} = f(\overline{x}) \wedge C = g(c, \overline{x}) \wedge \Psi'$, where f and g are formulas using $\overline{x}$, c and possibly other fresh variables.

Let K be a set of sensitive variables and $\mathrm{eq}_C(c_1, c_2)$ be an equality check (details in [Brotzman et al. 2019]) on two cache states c_1 and c_2. For a trace-based model, CaSym generates a 2-safety verification condition at every program point as follows:

$$
\begin{array}{l}
VC \triangleq \exists \overline{x}, \overline{x}', c, c', \\
\quad \neg(\forall X_i \in K, x_i = x_i') \wedge (\forall X_j \notin K, x_j = x_j') \\
\quad \wedge c = c' \wedge \neg \mathrm{eq}_C(g(c, \overline{x}), g(c', \overline{x}'))
\end{array} \tag{1}
$$

When the verification condition is satisfiable, it is possible to run the program twice with two low-equivalent memories and the same initial cache state, but resulting in two different cache states (hence, a violation of CNI). On the other hand, the program being checked satisfies CNI when the verification condition is unsatisfiable, which implies that the original program before transformation satisfies SANI, based on Theorem 1.

Handling speculation-related instructions. On top of the symbolic execution rules of CaSym, Figure 6 formalizes SpecSafe's symbolic execution over the new speculation-related primitives. fence has no effects in symbolic execution since it only affects the number of instructions being speculatively executed (i.e., during the construction of $[S]_w$ in Figure 5). capture(id) saves the current architectural state (i.e., the symbolic state for variables) with id and rollback(id) recovers the architectural state from the stored states. We use a global list mems to store and retrieve such memory snapshots. Note that rollback only rolls back the architectural state but not the cache state and path condition[7].

[7]Rolling back the path condition is unnecessary because rollbacks are added to the end of the instrumented true/false branches of an if conditional during transformation; so the path condition will be rolled back automatically when symbolic execution merges path conditions after the if statement.

7 IMPLEMENTATION

We build SpecSafe on top of CaSym [Brotzman et al. 2019] (as an analysis pass in LLVM version 3.7).[8] Besides program transformation (Section 6.1) and reasoning about speculation-related instructions (Figure 6), SpecSafe also extends CaSym with a more robust reasoning of pointers. SpecSafe uses Z3 as the back-end for solving the generated verification conditions.

By default, SpecSafe instantiates the configurable transaction length parameter w to be 200 since this has been used by Guarnieri et al. [2020] and approximates the maximum speculation depth.

Identifying Interesting Functions. One bottleneck of symbolic execution is that it does not scale to large programs (e.g., the entire libgcrypt library). Hence, in order to analyze large programs, we first use a static taint analysis to provide a structured way to find intriguing functions where symbolic execution is subsequently applied. The taint analysis is implemented as an LLVM analysis pass, also using LLVM version 3.7 [Lattner and Adve 2004]. It is context-, field-, and flow-sensitive. The taint analysis looks for three patterns in programs that encompass both conventional and speculative cache side channels.

In particular, we follow patterns described in oo7 [Wang et al. 2019b] to identify interesting functions to evaluate SpecSafe. In particular, the pattern consists of three components: 1) there is a conditional branch that uses attacker-controllable data, 2) the attacker can control the index to an array within the speculation window of the attacker-controllable branch, and 3) the data read from the array in component 2 is used as an index for another array and the operation is also within the speculation window of the attacker controllable branch. Here, attacker controllable data means the data can be either directly or indirectly set by the attacker.

We note that the purpose of the taint analysis is to identify interesting functions in order to evaluate SpecSafe. However, SpecSafe is unable to detect potential cache side channels in code that is filtered out by the taint analysis (note that the patterns might miss vulnerabilities). Nevertheless, it does not undermine the soundness of SpecSafe, as any code that is examined by SpecSafe must satisfy the SANI requirements. We emphasize that the primary goal of identifying interesting functions is to evaluate SpecSafe in a *structured* manner, as opposed to manually identifying functions to analyze, as done in other works such as CaSym [Brotzman et al. 2019].

8 EVALUATION

We selected a variety of benchmarks to evaluate SpecSafe. These benchmarks include (1) 15 well-known samples vulnerable to Spectre Variant 1 attacks [Kocher 2018]; (2) the 2 samples of CB-channels from Figure 2; (3) a set of common ciphers from libgcrypt version 1.8.5. Our evaluation of SpecSafe on these benchmarks was run on Ubuntu 14.04 in a virtual machine with 30GB of RAM and 4 cores of an Intel i7-5820K CPU. We use the age cache model presented in CaSym [Brotzman et al. 2019] in our evaluation because it is the most conservative model available. In our experiments, we answer the following questions:

(1) Can SpecSafe detect cache side channels that are missed by other state-of-the-art tools?
(2) How effective can SpecSafe identify vulnerabilities?
(3) Does SpecSafe verify the absence of cache side channels manifesting from branch speculation?
(4) Can SpecSafe scale to programs of moderate sizes, by combining taint analysis with symbolic execution?

[8] SpecSafe is implemented as an analysis on LLVM IR code. Ideally, an analysis on lower-level code (e.g., binary) offers stronger security guarantee since it is immune to code transformations by later compiler stages [Daniel et al. 2020]. However, we note that even binary analysis faces similar limitation due to hardware features such as out-of-order execution. Moreover, this is largely an implementation limitation, since our methodology can generally be applied to any level of abstraction, from the source code to binary.

8.1 Micro-benchmarks

In order to provide a sanity check and also a comparison with other tools, we briefly discuss our findings on the 15 benchmarks from [Kocher 2018] and the 3 samples from Figure 2.

The second column of Table 2 presents the results of applying SpecSafe on two sets of micro-benchmarks, including 15 well-known samples that are vulnerable to Spectre Variant 1 attacks [Kocher 2018] and the examples in Figure 2. Each example is about 10 to 20 lines of C code. SpecSafe successfully detects all vulnerabilities in the micro-benchmarks, within a reasonable amount of time.

Once a vulnerability is detected, we use the locations of the failed checks reported by SpecSafe to place appropriate protection mechanisms such as *lfence* or *SLH* to fix the reported vulnerabilities. After applying a protection mechanism, we reran SpecSafe, which was able to verify that the samples with *lfence* and *SLH* defenses are free of cache side channels.

Interestingly, two of the new side channels proposed cannot be patched using SLH. This is because the leakage in them happens due to the *existence* of a misprediction. Since mitigation techniques such as SLH do not prevent mispredictions from occurring, it cannot completely protect all of the variants introduced in this paper.

We also compared SpecSafe with other state-of-the-art systems geared towards finding leakage caused by speculative execution (Table 3). We note that both Spectector and oo7 cannot detect the three new instances of speculative leakage discovered in this work. Furthermore, since they cannot detect the vulnerabilities in the vulnerable code in the first place, their reports on the code with defense are not meaningful, hence, reported as n/a in the table.

We note that although we have not observed the code patterns presented in Figure 2 in the wild, an attacker might use them adversarially as small gadgets to fool existing "sound" analysis tools. This might happen when the attacker has control of the code, e.g., by sneaking code into an open-source, or community-developed victim program. Such subtle cases are not detectable by existing tools, and existing defenses (e.g. SLH) might fail to mitigate them.

8.2 Security Benchmarks

Next, we evaluated SpecSafe on libgcrypt ciphers. For each cipher, we treated the encryption key as sensitive data. Since the sizes of these ciphers were relatively large, symbolic execution was not scalable to handle them directly; therefore, we had to apply taint analysis (discussed earlier) to filter functions in those programs. The patterns used for identifying interesting functions require us to specify what data is considered attacker controllable. We include plaintexts, ciphertexts, and the size of the data to process as attacker-controllable, since these inputs are commonly provided to encryption routines by users, including a malicious user.

Taint Results. The first part of Table 4 presents the results gathered by applying our taint analysis to the various ciphers in libgcrypt. For each cipher, the first column is the cipher being analyzed.

	Vulnerable	LFENCE	SLH
Spectre Examples	5.15s ⚠	41.9s 🔒	76.5s 🔒
Figure 2a	0.45s ⚠	0.06s 🔒	0.02s 🔒
Figure 2b	0.20s ⚠	0.02s 🔒	0.59s ⚠
Figure 2c	0.63s ⚠	0.61s 🔒	0.87s ⚠

Table 2. Results of SpecSafe on micro-benchmarks of both vulnerable code as well as their corresponding protected versions. Spectre examples includes all 15 from [Kocher 2018]. The numbers in the table are SpecSafe's execution times. 🔒 means no side channel is found. ⚠ means side channels are detected.

Benchmark	SpecSafe	Spectector [Guarnieri et al. 2020]	oo7 [Wang et al. 2019b]
	Vulnerable		
Spectre Examples	✓	✓	✓
Figure 2a, 2b, 2c	✓	✗	✗
	Patched with LFENCE		
Spectre Examples	✓	✓	✓
Figure 2a, 2b, 2c	✓	N/A	N/A
	Patched with SLH		
Spectre Examples	✓	✓	✗
Figure 2a, 2b, 2c	✓	N/A	N/A

Table 3. Comparison with existing tools for detecting speculative side channels. A ✓ indicates the tool correctly judges the security/insecurity of the code (i.e. marking it secure/insecure if it is secure/insecure). A ✗ indicates the tool either has a false negative (when the code is vulnerable) or a false positive (when the code is secure). N/A applies when the tool fails to detect a vulnerability in the unmitigated code and thus it does not make sense to reason about the version with a mitigation mechanism.

The next column, IR LOC, is the number of lines of IR code in the cipher, as produced by Clang. The functions column is the number of functions in the cipher, and the time column represents how long it took the taint analysis to analyze the cipher in seconds. Each of the following columns tells the number of functions that were identified by the taint analysis in the specified category.

Overall, taint analysis can take at most an hour to finish, but is often much faster. This is an improvement over applying symbolic execution to all of these functions, which would likely take many hours if it terminates at all. Even though taint analysis is rather coarse-grained, we find that it is often able to rule out about 90% of the functions in each cipher.

From a practical point of view, the taint analysis filters out benign functions quickly. Applying the symbolic execution on a subset of the program saves a significant amount of analysis time. We note that this approach is not specific to the crypto algorithms analyzed here; instead, it can also be applied to other types of applications to improve analysis time.

One source of imprecision comes from the points-to analysis used in taint analysis: it tends to collapse all of the fields, making some cases field-insensitive. We found that almost all of the sensitive branches identified by taint analysis were due to the collapsing of the encryption context and subsequent checks on benign fields of the context (e.g., **if** (ctx->prefetch_dec_fn)). In total, 10 extraneous functions identified were manually pruned; almost all of them were found in the AES cipher due to checking the encryption context for certain features. The remaining 29 interesting functions are then further analyzed by SpecSafe. We note that manual pruning is needed due to the imprecision of the taint analysis tool (i.e., it is not field-sensitive). In general, this issue can be resolved by using a more precise taint analysis tool.

Categorizing Side Channels. SpecSafe by design detects any SANI cache side channels. We present all functions violating our SANI definition under the column "SANI" in Table 4, along with the total analysis time for symbolic execution under the column Time. To classify the results into conventional ones and speculative ones, we note that SpecSafe can be easily modified to detect conventional side channels only. In particular, for cache noninterference (CNI), we modify SpecSafe to add a constraint for each branch command as follows: $ite(cond_i, pred_i = \texttt{true}, pred_i = \texttt{false})$ where $cond_i$ is the branch condition being speculated. This effectively creates a perfect oracle,

			Taint Analysis		Symbolic Execution			
Module	IR LoC	Total Func.	Time (sec.)	Vulnerable Candidates	SANI	Time (sec.)	CNI	Spec
AES	6375	49	293	4	4	58.4	4	0
DES	8619	44	618	4	4	422	4	0
arcfour	536	8	7	2	2	0.75	2	0
blowfish	3535	31	215	3	3	37.5	3	0
camellia	14436	14	4344	0	0	n/a	0	0
cast5	4458	27	455	2	2	3.03	2	0
chacha20	3500	20	88	1	1	37.1	0	1
idea	1650	16	61	3	3	3.09	3	0
md4	1272	12	22	1	0	19.1	0	0
md5	2252	12	44	1	0	19.0	0	0
salsa	1756	20	31	2	2	2.12	2	1
sha256	6086	22	424	1	0	26.7	0	0
sha512	2743	26	100	1	0	31.8	0	0
tiger	2309	17	59	1	0	39.5	0	0
twofish	8238	28	334	3	3	9.79	3	0
total	67765	346	7095	29	24	709.88	23	2

Table 4. Evaluation results for libgcrypt ciphers. The first two columns indicate the lines of LLVM IR and number of functions analyzed. The first column under taint analysis indicates how long the taint analysis ran. Vulnerable candidates indicates the number of functions flagged by the taint analysis. The CNI and Spec columns indicate the number of functions the symbolic execution found containing those kinds of vulnerabilities. The SANI column contains all functions found that violate our SANI definition, along with the total analysis time for symbolic execution.

which allows us to analyze a program that never mispredicts. With the modification, we identified the conventional side channels (column "CNI") as well as remaining ones (column "Spec") that only exhibit themselves with speculative execution. Since we count the number of functions that violate SANI, there may be functions that violate both CNI and also have a speculative vulnerability; but such a function is counted only once in the SANI column. Hence the SANI column number is not necessarily the sum of the columns of CNI and Spec.

Although we were unable to run oo7 [Wang et al. 2019b] nor Spectector [Guarnieri et al. 2020] on this data set, we believe that they likely will find all vulnerabilities in this data set since the vulnerabilities are not as intricate as the new counterexamples that we construct in Section 4. However, SpecSafe reduces false positives. For example, since SpecSafe essentially uses the same taint analysis approach of oo7 as a filter, the vulnerable candidates column should be similar to the result of oo7. We note that SpecSafe removes a few false positives from the vulnerable candidates. Moreover, as we showed in Section 8.1, both oo7 and Spectector produce false negatives and/or false positives on intricate examples in the micro-benchmarks.

Conventional Cache Side Channels. Among all conventional cache side channels, SpecSafe detects a previously *unreported*, to the best of our knowledge, potential vulnerability in the triple DES routine of libgcrypt. This routine finds weak encryption keys. The following code snippet shows the problematic code:

```
is_weak_key ( const byte *key ) {
  ...
  while(left <= right) {
    middle = (left + right) / 2;
    // this branch causes the leak
    if (!(cmp_result=memcmp(key,
            weak_keys[middle])))
      return -1;
    if ( cmp_result > 0 )
      left = middle + 1;
    else
      right = middle - 1;
  }
}
```

It is clear from the identified **if** statement that this routine will terminate early depending on the key's value, thus resulting in a different cache state. Furthermore, the index to the table *weak_keys* is dependent on the value of the key since it uses the result of the memcmp function which depends on the key.

Checking for weak keys is common in encryption algorithms such as DES, 3DES, RC4, Blowfish, IDEA, etc. NIST suggests that weak keys be avoided, at least for 3DES [Barker and Mouha 2017], since they can significantly weaken security, but we find that checking for the keys in naive manners such as the code snippet above also pose security risks by potentially leaking bits of secure keys.

Speculative Execution Results. Side channels that only show up with speculative execution are summarized in the last column of Table 4.

We detect two true positives that can potentially be used to perform a Spectre style side-channel attack. These side channels, found in *salsa* and *chacha20*, are similar. The following is a simplified code snippet highlighting the potential leakage:

```
if (n > length) // begin speculation
...
// input is attacker controllable
idx = input;
...
buf2 = buf + BLOCK_SIZE + idx;
...
addr = *buf2; // loads secret into addr
...
value = *addr; // use secret addr to load
```

In this example, we see that within the depth of speculation of the conditional branch, we have a potential out of bounds read via `addr=*buf2`; since what `buf2` points to may be controlled by the adversary, an attack can load sensitive data into `addr`. Furthermore, the sensitive data in `addr` is used as an address for a subsequent load on line `value=*addr`. As a result, a cache attack can be used to discover the sensitive data in `addr`.

We investigated if this code pattern could be used to construct a Spectre attack. To this end, we constructed a test for this snippet similar to what we described in Section 4.2. Our test shows that this code snippet is capable of leaking information via speculative execution at a rate of over 75k bits per second with over 99% accuracy.

Moreover, symbolic execution was able to verify various functions among the interesting functions identified by the taint analysis. Recall that the taint analysis uses the patterns in oo7 [Wang et al. 2019b] to identify suspicious code that could be vulnerable to Spectre attacks. Hence, the result also highlights the benefit of using precise symbolic execution, compared with pattern matching methods, such as oo7 [Wang et al. 2019b].

Take Away. To summarize our findings: 1) built on the new security definition SANI, SpecSafe is able to detect speculative cache side channels that are missed by Spectector [Guarnieri et al. 2020] and oo7 [Wang et al. 2019b], the state-of-the-art tools for detecting speculative side channels; 2) SpecSafe has shown great promise in identifying both conventional and speculative cache side channels; 3) with the help of taint analysis, SpecSafe can work on programs of roughly 3,000 source lines or about 30,000 lines of IR code in about an hour which is reasonable considering how long it would take using only symbolic execution; 4) a coarse-grained taint analysis does a great job zeroing in on problematic functions to be further refined by symbolic execution, although the symbolic execution is still, in general, more precise, especially for intricate cases.

9 LIMITATIONS AND FUTURE WORK

When SpecSafe uses an abstract model that ignores the program counter, it needs to reason about four different execution paths at each branch as two program executions with different secrets might diverge. Compared to approaches that assume the attacker-observable program counter as part of their model, SpecSafe is less scalable in terms of handling branches. While analyzing these additional paths adds more computational cost, it allows SpecSafe to be more permissive and hence, accept more secure programs under precise cache models. Additionally, evaluation results suggest that even with the additional computational overhead, SpecSafe can still analyze reasonably sized programs.

This paper focuses on speculative execution caused by conditional branch instructions that use the *Pattern History Table* (PHT) to predict the next instructions. One future work is to incorporate other avenues of speculative execution, including indirect branches using the *Branch Target Buffer* (BTB), return instructions using the *Return Stack Buffer* (RSB), and speculation that leverages Store To Load (STL) dependencies. Similar to Spectector [Guarnieri et al. 2020], with more precise modeling, the noninterference definition we propose here is sufficient to detect these leakage techniques as well. We plan on progressively adding more of these models to our tool.

The taint analysis and consequently, the symbolic execution relies on a Data Structure Analysis (DSA) tool in LLVM that can lead to imprecision. This often happens with complex data structures like structs, making it become field insensitive, leading to over tainting. Symbolic execution is limited by such imprecision since it uses the taint analysis to determine sensitive information, as well as use alias analysis to soundly handle pointers.

10 CONCLUSION

In this work, we described shortcomings in existing tools' ability to detect leakage resulting from speculative behavior in programs containing direct branches. To resolve this issue, we presented a novel non-interference definition (SANI) to capture previously unknown avenues of leakage via the data cache. We demonstrated that these new leakage patterns, both in theory and in practice, could leak data by constructing covert channels. Lastly, we produced a tool SpecSafe that uses symbolic execution to detect leakage in programs according to our SANI definition as well as verify mitigation strategies that may be applied to leakage resulting from speculative execution.

11 ACKNOWLEDGEMENT

We would like to thank the reviewers for their constructive feedback that was very helpful toward improving this work. This research was supported by NSF grants CNS-1956032, CNS-1801534, CNS-1942851, CNS-1816282, and CCF-1723571.

$$\frac{\vec{\mu}' = zero(\vec{\mu}) \qquad enabled(\vec{\mu})}{\langle m, \vec{\mu}, \texttt{fence}\rangle \rightarrow \langle m, \vec{\mu}', \texttt{skip}\rangle}\ (\textsc{Fence}) \qquad \frac{eval(e, m) = v \qquad \vec{\mu}' = decr(\vec{\mu}) \qquad enabled(\vec{\mu})}{\langle m, \vec{\mu}, x := e\rangle \xrightarrow{loc(e)\cdot x} \langle [x/v]m, \vec{\mu}', \texttt{skip}\rangle}\ (\textsc{Asgn})$$

$$\frac{\vec{\mu} \neq \emptyset \qquad enabled(\vec{\mu})}{\langle m, \vec{\mu}, \texttt{skip}\rangle \rightarrow \langle m, zero(\vec{\mu}), \texttt{skip}\rangle}\ (\textsc{Skip}) \qquad \frac{\langle m, \vec{\mu}, S_1\rangle \xrightarrow{e} \langle m', \vec{\mu}', S_1'\rangle}{\langle m, \vec{\mu}, S_1; S_2\rangle \xrightarrow{e} \langle m', \vec{\mu}', S_1'; S_2\rangle}\ (\textsc{Seq1})$$

$$\frac{}{\langle m, \vec{\mu}, \texttt{skip}; S_2\rangle \rightarrow \langle m, \vec{\mu}, S_2\rangle}\ (\textsc{Seq2})$$

$$\frac{p = O(\eta).pop() \qquad \eta \text{ is the command id} \qquad c_{next} = \begin{cases} c_1 \text{ when } p = \texttt{true} \\ c_2 \text{ when } p = \texttt{false} \end{cases} \qquad \vec{\mu}' = decr(\vec{\mu}) \cdot \langle p, m, w, \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2\rangle \qquad enabled(\vec{\mu})}{\langle m, \vec{\mu}, \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2\rangle \xrightarrow{\texttt{begin}\cdot pc(c_{next})} \langle m, \vec{\mu}', c_{next}\rangle}\ (\textsc{If})$$

$$\frac{}{\langle m, \vec{\mu}, \texttt{while } b \texttt{ do } c\rangle \rightarrow \langle m, \vec{\mu}, \texttt{if } b \texttt{ then } (c; \texttt{while } b \texttt{ do } c) \texttt{ else skip}\rangle}\ (\textsc{While})$$

$$\frac{eval(b, m) = p \qquad enabled(\vec{\mu}')}{\langle m, \langle p, m', 0, \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2\rangle \cdot \vec{\mu}', c\rangle \xrightarrow{\texttt{commit}} \langle m, \vec{\mu}', c\rangle}\ (\textsc{Commit})$$

$$\frac{eval(b, m) = v \neq p \qquad enabled(\vec{\mu}') \qquad c' = \begin{cases} c_1 \text{ when } v = \texttt{true} \\ c_2 \text{ when } v = \texttt{false} \end{cases}}{\langle m, \langle p, m', 0, \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2\rangle \cdot \vec{\mu}', c\rangle \xrightarrow{\texttt{rollback}} \langle m', \vec{\mu}', c'\rangle}\ (\textsc{Rollback})$$

Fig. 7. Speculative-Aware semantics given an oracle O and a transaction length w.

A FORMALIZING PROGRAM SEMANTICS WITH SPECULATION

Language Semantics. Recall that to model speculative execution, we introduce two parameters: a *prediction oracle*, O, as well as the speculative transaction's length w in the language semantics. Moreover, O is a partial function that takes the id of a branch command and returns a queue of predicted branch outcomes (i.e., a queue of truth values). A queue is used in the oracle so that the semantics can have different outcomes for the same branch at different execution points.

We assume that each command in the source code has a unique id (such as the line number), denoted by η. To get the next predicted outcome of the branch command with id η, we use $O(\eta).pop()$.

We define a configuration as a tuple $\langle m, \vec{\mu}, S\rangle$ that consists of a memory m, a sequence of (potentially nested) speculative states $\vec{\mu}$ as well as S, the remaining program to be executed. As standard, memory m is formalized as a map from variables/array elements to their values. A speculative state $\mu = \langle p, m, r, S\rangle$ represents a *speculative transaction* [Guarnieri et al. 2020], which consists of the predicted boolean outcome p, the memory m prior to the start of the transaction, the remaining number of instructions to be executed in the transaction r, and the commands S to be resumed in case of a rollback. Moreover, we use the following helper functions to update and check the remaining number of instructions in all transactions:

- decr($\vec{\mu}$): decrease the remaining instructions by one for each $\mu \in \vec{\mu}$.

- zero($\vec{\mu}$): set the remaining instructions to zero for each $\mu \in \vec{\mu}$.
- enabled($\vec{\mu}$): returns true if $r \neq 0$ for any $\mu \in \vec{\mu}$ or when $\vec{\mu} = \emptyset$.

Given an oracle O and a transaction length w, each evaluation rule (summarized in Figure 7) has the form of $\langle m, \vec{\mu}, S\rangle \xrightarrow{e} \langle m', \vec{\mu}', S'\rangle$ where event e tracks the memory locations being accessed and program instructions being executed; we use $pc(\eta)$ to denote the latter. Note that the given O and w, the semantics is deterministic: rules COMMIT and ROLLBACK apply when one transaction ended, while the other rules require $enabled(\vec{\mu})$; rule COMMIT requires $eval(b, m) = p$ and ROLLBACK requires $eval(b, m) \neq p$.

Given a program S, an oracle O and transaction length w, we use $[\![S]\!]^O_{\langle m,w\rangle}$ to denote the sequence of emitted events starting from the initial configuration $\langle m, \emptyset, S\rangle$ according to the semantics given O and w. For example, let S be the code in Listing 1, $O(1) = \texttt{true}$, w=1 and $m(idx) \geq m(b_{size})$, we have the following evaluation steps according to the semantics:

$$\begin{aligned}
&\langle m, \emptyset, S\rangle \\
&\xrightarrow{\texttt{begin}\cdot pc(2)} \langle m, \langle \texttt{true}, m, 1, S\rangle, \texttt{temp=A[B[idx]*512]}\rangle \\
&\xrightarrow{idx, B+idx, A+B[idx]*512} \langle m', \langle \texttt{true}, m, 0, S\rangle, \texttt{skip}\rangle \\
&\xrightarrow{\texttt{rollback}} \langle m, \emptyset, \texttt{skip}\rangle
\end{aligned}$$

Hence, $[\![S]\!]^O_{\langle m,w\rangle}$ generates an event trace $\texttt{begin}, pc(2), idx, B + idx, A + B[idx] * 512, \texttt{rollback}$ for the concrete execution.

We note that our interpretation of O and w over-approximates the ones in [Guarnieri et al. 2020], which assumes a partial function that takes as input a program, a branching history and a branch instruction, and that returns as output the predicted branch outcome and transaction length. Our formulation considers all possible outcomes of the semantics in [Guarnieri et al. 2020].

REFERENCES

Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. 2003. The EM Side-Channel(s). In *Cryptographic Hardware and Embedded Systems - CHES 2002*, Burton S. Kaliski, çetin K. Koç, and Christof Paar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 29–45.

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

Elaine Barker and Nicky Mouha. 2017. *Recommendation for the Triple Data Encryption Standard (TDEA) Block Cipher.* NIST Special Publication, 800-67, Revision 2, US Department of Commerce, National Institute of Standards and Technology, Gaithersburg, MD.

Daniel J. Bernstein. 2005. Cache-timing attacks on AES. cr.yp.to/papers.html#cachetiming.

Joseph Bonneau and Ilya Mironov. 2006. Cache-Collision Timing Attacks Against AES. In *Cryptographic Hardware and Embedded Systems - CHES 2006*, Louis Goubin and Mitsuru Matsui (Eds.). Lecture Notes in Computer Science, Vol. 4249. Springer Berlin Heidelberg, 201–215.

Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Proceedings of the 11th USENIX Conference on Offensive Technologies* (Vancouver, BC, Canada) *(WOOT'17)*. USENIX Association, Berkeley, CA, USA, 11–11. http://dl.acm.org/citation.cfm?id=3154768.3154779

Robert Brotzman. 2021. *Detecting and Mitigating Cache-Based Side-Channels.* Ph. D. Dissertation. Pennsylvania State University.

R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. Kandemir. 2019. CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation. In *2019 IEEE Symposium on Security and Privacy (S&P)*. 364–380. https://doi.org/10.1109/SP.2019.00022

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

Chandler Carruth. 2019. Speculative Load Hardening. https://llvm.org/docs/SpeculativeLoadHardening.html.

Sunjay Cauligi, Craig Disselkoen, Klaus v Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-Time Foundations for the New Spectre Era. In *Proceedings of the 41st ACM Conference on Programming Language Design and Implementation (PLDI'20), London, UK*.

K. Cheang, C. Rasmussen, S. Seshia, and P. Subramanyan. 2019. A Formal Approach to Secure Speculation. In *32nd IEEE Computer Security Foundations Symposium (CSF)*. 288–28815.

Lesly-Ann Daniel, S'ebastien Bardin, and Tamara Rezk. 2020. Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level. In *2020 IEEE Symposium on Security and Privacy (S&P)*. 1021–1038.

Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. 2017. Inside 6th-generation intel core: New microarchitecture code-named skylake. *IEEE Micro* 37, 2 (2017), 52–62.

Goran Doychev, Dominik Feld, Boris Kopf, Laurent Mauborgne, and Jan Reineke. 2013. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *Proc. the 22nd USENIX Security Symposium (USENIX Security)*. 431–446.

Goran Doychev and Boris Köpf. 2017. Rigorous analysis of software countermeasures against cache attacks. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. ACM, 406–421.

Joseph A. Goguen and Jose Meseguer. 1982. Security Policies and Security Models. In *IEEE Symp. on Security and Privacy (S&P)*. 11–20.

Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security* (Belgrade, Serbia) *(EuroSec'17)*. ACM, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/3065913.3065915

Roberto Guanciale, Musard Balliu, and Mads Dam. 2020. InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 1853–1869. https://doi.org/10.1145/3372297.3417246

Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled Detection of Speculative Information Flows. In *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*.

David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games—Bringing Access-Based Cache Attacks on AES to Practice. In *Proc. IEEE Symp. on Security and Privacy (S&P)*. 490–505.

Jann Horn. 2018. Issue 1528: speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528. Accessed: 2020-1-21.

Intel. 2018a. Bounds Check Bypass. https://software.intel.com/security-software-guidance/software-guidance/bounds-check-bypass.

Intel. 2018b. Intel Analysis of Speculative Execution Side Channels. https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf.

Vladimir Kiriansky, Ilia A. Lebedev, Saman P. Amarasinghe, Srinivas Devadas, and Joel S. Emer. 2018. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2018), 974–987.

Paul Kocher. 2018. Spectre Mitigations in Microsoft's C/C++ Compiler. https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html.

Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential Power Analysis. In *Advances in Cryptology — CRYPTO' 99*, Michael Wiener (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 388–397.

Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '96)*. Springer-Verlag, London, UK, UK, 104–113. http://dl.acm.org/citation.cfm?id=646761.706156

Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. USENIX Association, Baltimore, MD. https://www.usenix.org/conference/woot18/presentation/koruyeh

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75.

Fangfei Liu, Y. Yarom, Qian Ge, G. Heiser, and R.B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy (S&P)*,. 605–622.

J. Longo, E. De Mulder, D. Page, and M. Tunstall. 2015. SoC It to EM: ElectroMagnetic Side-Channel Attacks on a Complex System-on-Chip. In *Cryptographic Hardware and Embedded Systems – CHES 2015*, Tim Güneysu and Helena Handschuh (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 620–640.

Giorgi Maisuradze and Christian Rossow. 2018. Ret2spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2109–2122. https://doi.org/10.1145/3243734.3243761

David Molnar, Matt Piotrowski, David Schultz, and David Wagner. 2006. The program counter security model: automatic detection and removal of control-flow side channel attacks. In *Proc. 8th International Conference on Information Security and Cryptology*. 156–168.

Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. 2018. You Shall Not Bypass: Employing data dependencies to prevent Bounds Check Bypass. *ArXiv* abs/1805.08506 (2018).

Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. 2020. SpecFuzz: Bringing Spectre-type vulnerabilities to the surface. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/usenixsecurity20/presentation/oleksenko

Dag A. Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. *Topics in Cryptology–CT-RSA 2006* (Jan. 2006), 1–20.

Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In *16th ACM Conference on Computer and Communications Security (CCS)*. 199–212.

Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware guard extension: Using SGX to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 3–24.

Eran Tromer, DagArne Osvik, and Adi Shamir. 2010. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology* 23, 1 (2010), 37–71.

Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *Proceedings of the 2Nd Workshop on System Software for Trusted Execution* (Shanghai, China) *(SysTEX'17)*. ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/3152701.3152706

Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. 2019b. oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE Transactions on Software Engineering* (2019).

Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. 2019a. Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation. In *28th USENIX Security Symposium (USENIX Security 19)*. 657–674.

Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. 2017. CacheD: Identifying Cache-Based Timing Channels in Production Software. In *Proc. the 26th USENIX Security Symposium (USENIX Security)*. 235–252.

Zhenyu Wu, Zhang Xu, and Haining Wang. 2012. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. 159–173.

Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. 2017. Stacco: Differentially analyzing side-channel traces for detecting SSL/TLS vulnerabilities in secure enclaves. In *Proc. ACM Conf. on Computer and Communications Security (CCS)*. 859–874.

Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. 2011. An Exploration of L2 Cache Covert Channels in Virtualized Environments. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*. 29–40.

Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2018), 428–441.

Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*. 719–732.

Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2012. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. 305–316.