unXpec: Breaking Undo-based Safe Speculation

Mengming Li
Zhejiang University
mmli@zju.edu.cn

Chenlu Miao
Zhejiang University
clmiao@zju.edu.cn

Yilong Yang
Zhejiang University
yangyl5@zju.edu.cn

Kai Bu[⋆]
Zhejiang University
kaibu@zju.edu.cn

***Abstract*—Speculative execution attacks exploiting speculative execution to leak secrets have aroused significant concerns in both industry and academia. They mainly exploit covert or side channels over microarchitectural states left by mis-speculated and squashed instructions (i.e., transient instructions). Most such attacks target cache states. Existing cache-based defenses against speculative execution attacks fall into two categories, Invisible and Undo. Most Invisible defenses buffer execution metadata of speculative instructions and place them into the cache only if the speculatively executed instructions become determined. Motivated by the fact that mis-speculations are rare cases, Undo defenses allow speculative instructions to modify cache states. Upon a mis-speculation, they rollback cache states to the ones prior to the execution of transient instructions. However, Invisible defenses have been recently found insecure by the speculative interference attack. This calls for a deep security inspection of Undo defenses against speculative execution attacks.**

In this paper, we present unXpec as the first attack against Undo-based safe speculation. It exploits the secret-dependent timing channel exhibited through the rollback operations of Undo defenses. Specifically, the rollback process requires both invalidating cache lines brought into the cache by transient instructions and restoring evicted cache lines from the cache by transiently loaded data. This opens up a channel that encodes secret via the timing difference between when rollback involves much invalidation and restoration or not. We further leverage eviction sets to enforce more restoration operations. This yields a longer rollback time and thus a larger secret-dependent timing difference. We demonstrate the timing channel over the open-source CleanupSpec, a representative Undo solution. A single transient load can trigger a secret-dependent timing difference of 22 cycles (without eviction sets) of 32 cycles (with eviction sets), which is sufficiently exploitable for constructing a covert channel for speculative execution attacks. We run unXpec on the gem5 simulator with CleanupSpec enabled. The results show that unXpec can leak secrets at a high rate of 140 Kbps with an accuracy over 90%. Simply enforcing constant-time rollback to mitigate unXpec may induce an over 70% performance overhead.

***Index Terms*—speculative execution attack; undo speculation; covert channel attack;**

I. Introduction

Speculative execution attacks such as Spectre [21] and its variants [2], [3], [9], [20], [22], [27], [36] have aroused significant concerns in both industry and academia. They exploit the design flaws of speculative execution to leak secret information through covert or side channels over microarchitectural states. On modern out-of-order processors, speculative execution enables the CPU to execute instructions before their validity is determined toward speedup [17]. The results of speculatively executed instructions take effect only if the speculation is correct. Upon a mis-speculation, the CPU squashes the mis-speculated instruction (i.e., transient instruction) and younger instructions in the reorder buffer (ROB). This rollbacks the effects of squashed instructions on architectural states in memory and register while preserving program accuracy. However, the associative changes in microarchitectural states still remain. They reveal the footprints of transient instructions and open the door for covert or side channel attacks [21], [23]. Various attacks have been established by exploiting microarchitectural units such as caches [7], [21], [23], [27], [38], [43], ports [3], and branch predictors [10]. Most of them exploit the cache as a covert channel.

The state-of-the-art speculative interference attack [2] reveals that it is insecure by simply prohibiting speculatively executed instructions from modifying cache states. This found-to-be-insecure strategy is the core idea of Invisible defenses [1], [19], [35], [45] against speculative execution attacks. Specifically, most Invisible approaches buffer the data blocks and corresponding metadata accessed by speculatively executed instructions. Only if the speculation is correct can the buffered data be copied into the cache hierarchy and become visible to the rest of the system. Transient instructions thus leave no secret-dependent trace in the cache for exploitation. However, correct speculations account for more than 95% [34]. Complicating their operations induces a major source of slowdown. For example, the first representative Invisible defense—InvisiSpec [45]—enforces two reads per speculatively executed load and leads to a 17% slowdown. Followup Invisible defenses improve efficiency by eliminating the need of extra loads for correctly speculated instructions. Albeit becoming more efficient, Invisible defenses have been recently breached by the speculative interference attack [2] that exploits microarchitectural contention on hardware components such as MSHR to leak secrets.

The breach of Invisible defenses solicits a deep inspection of the other branch of defenses against speculative execution attacks—Undo defenses [34]. Different from Invisible defenses, the Undo approach permits transient instructions to bring data blocks to the cache. When a mis-speculation is detected, the CPU rollbacks the cache states to the ones prior to the execution of transient instructions. The rollback process requires two key operations, invalidation and restoration. Invalidation aims to invalidate any cache lines loaded by transient instructions. Restoration aims to restore the original

[⋆]Kai Bu is the corresponding author.
[*]The authors are also affiliated with Key Laboratory of Blockchain and Cyberspace Governance of Zhejiang Province.

cache lines evicted by transiently loaded data. This way, footprints of transient instructions are eliminated and thus unexploitable. Undo defenses achieve such effect by introducing extra operations to only uncommon mis-predictions (i.e., $< 5\%$ [34]). They thus promise a much higher efficiency than most Invisible defenses do. For example, the representative Undo defense—CleanupSpec [34]—slows down execution by only 5%.

In this paper, we take the first step to exploit security vulnerabilities of Undo-based safe speculation. Specifically, we identify a secret-dependent timing channel that can be exploited to leak secrets from CleanupSpec-alike Undo safe speculation solutions. The source of such a timing channel comes from the rollback operations. As aforementioned, the rollback process requires both invalidation and restoration to revert any cache state changes caused by transient instructions. The more state changes the transient instructions lead to, the more rollback operations and thus the longer rollback time it takes. We validate this intuition by collecting various time measurements over open-source CleanupSpec [33]. Two observations out of the measurements confirm the feasibility of a timing channel. One is that the rollback time upon a mis-speculation increases with the amount of cache state changes. The other is that the time for detecting a mis-speculation is relatively constant given a fixed speculation condition. These two observations demonstrate that the execution time of an Undo-based safe speculation increases with the amount of rollback operations. If we encode secrets via transient loads such that they intrigue different rollback complexities, we can build a timing channel over Undo-based safe speculation.

We present unXpec, the first speculative execution attack against Undo-based safe speculation. We delicately craft the attack algorithm to maximize the secret-dependent timing difference. Specifically, unXpec leads to no invalidation or restoration operation at all when handling a mis-speculation given secret 0. This yields the highest contrast to the time measurement given secret 1 under the same configuration. We run our unXpec attack over open-source CleanupSpec [33]. It demonstrates a secret-dependent timing difference of 22 cycles using only a single load instruction in the branch, which is sufficient for an exploitable timing channel [2], [3], [46].

We further optimize unXpec using eviction sets [41] toward a larger secret-dependent timing difference. The key idea is that we generate an eviction set of addresses and access them using transient instructions to evict original cache lines. This necessitates restoring the evicted cache lines during rollback. We thus gain an enlarged rollback time and thus a larger secret-dependent timing difference up to 32 cycles given only one in-branch load instruction. The increased secret-dependent timing difference promises a higher robustness against background noise and thus a higher attack resolution. We implement unXpec and the evaluation shows that unXpec can leak secrets from CleanupSpec-enabled systems with a 2 GHz CPU at a high rate of 140 Kbps with an accuracy over 90%.

Together with the speculative interference attack [2] against Invisible defenses, our unXpec attack complements security evaluation of existing defenses against speculative execution attacks. In summary, we make the following contributions to uncovering the security vulnerabilities of Undo-based safe speculation.

- We identify a new secret-dependent timing channel over Undo defenses against speculative execution attacks (Section IV). It arises from the fact that an Undo approach may take a different amount of time to rollback different amounts of cache state changes caused by transient instructions. If we can calibrate the scale of cache state changes through secret-dependent transient instructions, we can exploit the identified timing channel to leak secrets.
- We present unXpec as the first speculative execution attack against Undo-based safe speculation (Section V). We craft the attack algorithm to maximize the secret-dependent timing difference. Optimized by using eviction sets, unXpec can enforce more rollback operations and thus an even larger timing difference. This promises a higher robustness against background noise and thus a higher attack resolution.
- We run unXpec on the gem5 simulator and validate its attack efficacy over open-source CleanupSpec [33] (Section VI). The results show that unXpec can intrigue an average timing difference of 22 cycles and 32 cycles given only a single load in the branch without and with using eviction sets, respectively. On a 2 GHz CPU, the unXpec attacks without and with eviction sets can respectively achieve an accuracy of 86.7% and 91.6% with only a single sample per bit. This yields a sufficiently high leakage rate of 140 Kbps.
- We evaluate performance overhead of the most intuitive countermeasure against unXpec—constant-time rollback (Section VI). Experimental results demonstrate that the constant-time rollback scheme may introduce non-negligible overhead of over 70%. This calls for either a more efficient countermeasure against unXpec or a more robust Undo-based safe speculation.
- We publish the unXpec source code as well as guidelines for result reproduction at https://doi.org/10.5281/zenodo.5771649 (Artifact Appendix).

II. Background

In this section, we review the basics of speculative execution attacks and cache-based defenses.

A. Speculative Execution Attack

Speculative execution attacks exploit side channels over the transient microarchitectural states exiled by speculative execution [2], [3], [10], [11], [21], [23], [27], [32], [36], [38]–[40]. As a key source of speedup on modern out-of-order processors, speculative execution leverages spare processing cycles to execute instructions before their validity is determined [17]. Results of correct speculation can be immediately used without experiencing the execution delay. Mis-speculated instructions (also known as transient instructions) need to be squashed such

Algorithm 1 Spectre Variant 1 [21], [45]

```
Input:
 1: An uncached array used to probe the secret: $P[64 * 256]$;
 2: An arbitrary array used to access the secret: $A[n]$;
 3: An out-of-bounds index of array A: $i$;
Output: $secret$
 4: //victim code
 5: function VICTIM($index$)
 6:     if $index < n$ then
 7:         $y = P[64 * A[index]]$;
 8:     end if
 9: end function
10: //attacker code
11: function POISON()
12:     Invoke VICTIM($index$) with some in-bounds $index$es;
13: end function
14: function PROBE()
15:     Load every entry of probe array $P$;
16:     Infer the secret value by measuring the loading time;
17: end function
18: POISON(); //train if-condition toward mis-speculation
19: FLUSH(); //flush every entry of probe array $P$ and $n$
20: VICTIM($i$); //trigger victim to speculatively execute
21: PROBE(); //probe each entry of array A to infer secret
```

that they make no effective architectural state changes in the memory and registers. The corresponding microarchitectural state changes in caches of transient instructions are, however, left unattended [8], [26], [45]. This opens the door for speculative execution attacks that exploit a cache-based covert channel between an attacker process and a victim process. Specifically, the attacker manipulates the victim to issue certain transient instructions such that secret related states are loaded into caches. Through timing-based cache side-channel attacks [5], [25], [28], [47], the attacker can infer the in-cache status of these secret-related states. Then it can further infer the secret.

Next, we use Spectre [21] to showcase the workflow of a speculative execution attack.

Spectre [21], the pioneering and prevailing speculative execution attack, can leak arbitrary data from the victim to the attacker. Essentially, the attacker leverages mis-speculation to evade bounds checking such that the victim program can access out-of-bounds data. As shown in Algorithm 1 [21], [45], Spectre crafts the victim with access privilege to byte-array $A[n]$ and to byte-array P upon in-bound accesses to A (lines 5-9). To train the branch predictor such that VICTIM favors an in-bounds prediction in subsequent invocations, the attacker runs POISON to invoke VICTIM with in-bounds values of $index$ for a sufficient number of times (lines 11-13). Later on, the attacker turns to invoking VICTIM with an out-of-bounds $index\ i$ (line 20). Following previous prediction statistics, the branch predictor mis-predicts the out-of-bounds access as legal. This makes $P[64*A[i]]$ loaded into the cache; $P[64 * A[i]]$ is dependent on $A[i]$ that is not supposed to be accessed by the victim. When the bounds checking is eventually determined as illegal, the CPU discards in-ROB results rather than microarchitectural states (e.g., cache states). The attacker then runs PROBE to load every element of array P and determines whether an element is cached through measuring its loading time (lines 14-17). (Note that to remove false positives caused by elements loaded before VICTIM(i), all elements of P should be evicted from the cache using FLUSH (line 19) prior to invoking VICTIM(i) (line 20).) If the loading time of $P[j]$ indicates a cache hit, the attacker can infer the secret value of $A[i]$ as $j/64$.

B. Cache-Based Defenses

Since the root cause for most speculative execution attacks is the cache-state change caused by transient instructions, an essential defense should prohibit the attacker from inferring such changes via cache side-channel attacks [45]. This motivates two types of defenses—Invisible and Undo [34].

Invisible defenses allow no state change in the cache by transient instructions at all [1], [19], [35], [42], [45]. They usually buffer execution metadata of speculatively executed instructions and places the buffered data into the cache only if speculative instructions become determined [1], [19], [42], [45]. Albeit hiding state changes of transient instructions, the buffer introduces an additional stop before data finally reach the cache hierarchy. This is deemed as a major source of slowdown. For example, the pioneering Invisible defense—InvisiSpec [45]—slows down execution by over 17%. To get rid of the buffer, delay-on-miss [35] leverages the fact that L1 cache misses are rare cases. Upon L1 cache misses, delay-on-miss simply defers to serve access requests until the speculative prediction is determined. By applying value prediction, this can reduce the slowdown to 11%.

However, speculative interference attacks [2] recently reveal that even if an Invisible defense is adopted, they can still exploit microarchitectural contention on certain hardware components (e.g., MSHR and execution units) to leak secrets. This solicits a deep inspection of Undo robustness.

Undo defenses allow transient instructions to directly load data blocks to the cache hierarchy and rollback the so caused state changes upon mis-speculation [34]. Given that mis-speculation is an uncommon case, Undo defenses lead to a much lower overhead than most Invisible defenses that complicate the common case of correct speculation. The representative Undo defense, CleanupSpec [34], slows down execution by only 5%. Furthermore, CleanupSpec uses a random replacement policy to avoid side channels over replacement states that can be exploited by speculative interference attacks [2], [5], [43]. Figure 1 shows the timeline for CleanupSpec to secure mis-speculation. We accordingly introduce its key strategies for mitigating information leakage as follows.

We start with strategies used in the speculation window from when a speculative execution starts (T1 in Figure 1) to when mis-speculation is detected (T2 in Figure 1). Two strategies are used herein to avoid cache states from being exploited. First, CleanupSpec delays unsafe operations such as downgrading a cache coherence state from M/E to S when

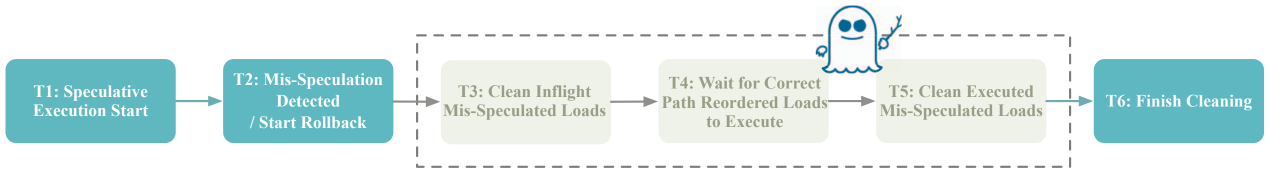
T1: Speculative Execution Start
T2: Mis-Speculation Detected / Start Rollback
T3: Clean Inflight Mis-Speculated Loads
T4: Wait for Correct Path Reordered Loads to Execute
T5: Clean Executed Mis-Speculated Loads
T6: Finish Cleaning

Fig. 1. Timeline for CleanupSpec-alike [34] Undo defenses against speculative execution attacks. We present unXpec attacks to exploit secret-dependent timing differences found in T3-T5.

prediction is unresolved. This prevents side-channel attacks that exploit cache coherence states [46]. Second, if an access request from another thread/core hits a speculatively installed line, CleanupSpec serves it with a dummy cache miss. This prevents the attacker from inferring information through a transiently installed line in the speculation window.

Once CleanupSpec detects mis-speculation, it rollbacks state changes by transient instructions using two strategies—invalidation and restoration (T3 to T5 in Figure 1). T3 and T4 conduct necessary preparation prior to state rollback. First, CleanupSpec requests MSHR to clean inflight mis-speculated loads (T3 in Figure 1). Second, to avoid recursive squash during cleanup, CleanupSpec waits for the retirement of all inflight correct-path loads, reordered due to out-of-order execution (T4 in Figure 1). Finally, CleanupSpec starts rollbacking state changes as if mis-speculated instructions had not taken effect (T5 in Figure 1). It first invalidates all the cache lines installed by transient instructions. It then restores evicted cache lines that are replaced by transiently installed lines. The addresses of transiently installed lines and and that of the evicted lines are maintained in the load queue and MSHR, respectively. Note that CleanupSpec restores evicted lines only for the L1 cache. Restoration on lower-lever caches is not adopted due to high overhead. To remedy the so caused vulnerability, CleanupSpec uses address randomization [31] on lower-level caches.

In this paper, we identify a speculative execution attack against CleanupSpec-alike Undo defenses. The key observation is that transient instructions encoded with different secret bits lead to different amounts of invalidation/restoration operations and thus different rollback timings. While CleanupSpec states that the cleanup operations may increase the stall time, it fails to uncover the associative vulnerability. First, by observing the cleanup operations on benchmarks, CleanupSpec concludes that cleanup takes only a small fraction of the squash time. It thus implicitly suggests that the attacker can hardly observe the actual cleanup time. However, we will show that the attacker can elaborately craft the attack code to eliminate the interference of inflight loads by using memory fence instructions. Second, CleanupSpec demonstrates that the restoration operations in cleanup only take a short time because they are pipelined and serviced from the L2 cache. However, our experimental results show that only the secret-dependent invalidation operations suffices to construct the timing channel and the secret-dependent restoration operations helps to enlarge the timing difference.

Next, we define the system and attacker models in Section III and measure various timings that can be exploited as side channels in Section IV. Then we accordingly craft our unXpec attack against Undo-based mis-speculation defenses in Section V.

III. System and Threat Models

In this section, we present generalized Undo-based protected cache models based on the state-of-art CleanupSpec [34]. We hereafter leverage CleanupSpec as a running example to cover the attack surface of Undo-based protected cache without loss of generality. Following attacker models in various speculative execution attacks and defenses [2], [10], [13], [34], [44], [45], we consider that the sender and receiver code run in the same thread and temporally multiplex the core with other honest programs.

A. *Undo-based Protected Cache Model*

Following CleanupSpec, the L1 cache adopts protection strategies including invalidation, restoration, partition, and random replacement. First, invalidation and restoration clean state traces by transient instructions and rollback cache states to when transient instructions have not taken place. As discussed in Section II-B, they enforce invalidating cache lines loaded by transient instructions as well as restoring cache lines replaced by transiently loaded lines. Second, the L1 cache is way partitioned (e.g., following NoMo [12]) to prevent non-speculative cache attacks (e.g., Prime+Probe [25], [28]) under simultaneous multithreading (SMT). Finally, the L1 cache adopts a random replacement policy to prevent side-channel attacks exploiting replacement states [2], [5], [43].

Protection strategies on lower-level caches include invalidation and randomized address mapping. Lower-level caches cannot afford restoration because it induces heavy storage and high complexity to buffer evictions therein and track the so caused recursive evictions throughout the entire cache hierarchy due to writebacks. Instead, they adopt randomized cache design (e.g., CEASER [31]) to mitigate security vulnerabilities.

In addition, every cache layer is further augmented with protection strategies such as delaying coherence downgrade and servicing a dummy cache miss to another thread/core that hits on a speculatively installed line (Section II-B).

B. *Attacker Model*

The attacker launching our unXpec attack (Section V) aims to extract secrets from the victim like existing covert-channel variations of speculative execution attacks. Following various speculative execution attacks and defenses [2], [10], [13], [34], [44], [45], we consider a typical attacker model as follows.

Algorithm 2 Constant Branch Resolution Time: PoC

```
Input:
 1: An array used to generate transient line installs: $P[64*n]$;
 2: A hyperparameter to control the size of array $P$: $n$;
 3: An arbitrary array used to access the secret: $A[n]$;
 4: An out-of-bounds index of array A: $i$;
 5: //sender code
 6: function SENDER($index$)
 7:     if $index < f(N)$ then
 8:         $secret = A[index]$;
 9:         $load(\&P[secret*64*1])$;
10:         $load(\&P[secret*64*2])$;
11:         ...
12:         $load(\&P[secret*64*n])$;
13:     end if
14: end function
15: //measurement code
16: function POISON()
17:     Invoke SENDER($index$) with some in-bounds $index$es;
18: end function
19: POISON(); //train if-condition toward mis-speculation
20: $load(\&P[0])$; //load $P[0]$ to the L1 cache
21: FLUSH(); //flush $P[64*1]...P[64*n]$
22: SENDER($i$); //trigger sender to speculatively execute
```

- The attacker code includes the sender and the receiver, which run in the same thread on the same physical core [2], [13], [34], [45]. It aims to leak memory contents of other security domains. Specifically, the sender is responsible for encoding the secret information through the timing channel uncovered in this paper. By analyzing the execution time of the sender, the receiver can decode the one-bit secret information in every round of attack. Like SpectreRewind [13], our attacker model can be used to implement Meltdown-like [23] attacks or leak secrets outside of a sandbox [21], [40].
- The sender and receiver share not only lower-level caches with cross-core processes but also the same-core L1 cache where cleanup operations are enabled.

IV. Timing Channel Identification

In this section, we measure the timing characteristics of various stages throughout the CleanupSpec timeline in Figure 1. We identify two essential characterstics that can be exploited to build a secret-dependent timing channel. First, it takes a relatively constant time to resolve a given branch (i.e., the time span between T1 and T2 in Figure 1). Second, the time for rollbacking cache states depends on the volume of transiently loaded cache lines (T5 in Figure 1). We further observe that such volume can be manipulated by tuning the secret value encoded in the sender code. This motivates us to build a secret-dependent timing channel over Undo-based safe speculation and accordingly design our unXpec attack in Section V and validate the attack efficacy in Section VI.

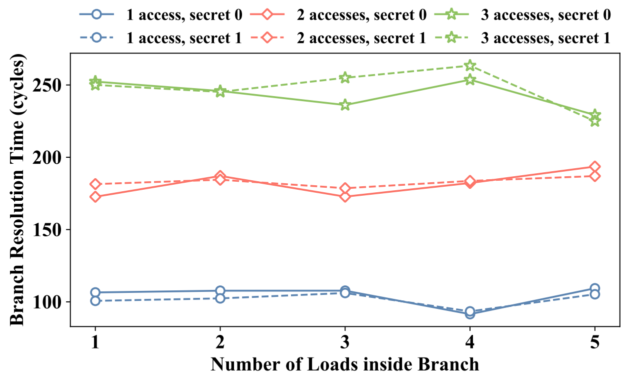

Fig. 2. The branch resolution time is relatively constant given a fixed branching statement (i.e., a fixed number of memory accesses in our experiments), regardless of the number of loads in the branch and the secret bit encoded through these loads.

A. *Constant Branch Resolution Time*

We observe that the branch resolution time under a given branching statement is relatively constant both in concept and in practice. In a speculative execution attack, the branch resolution time ranges from when a speculative execution starts to when a mis-speculation is detected (T1-T2 in Figure 1). Meanwhile, the sender speculatively executes transient instructions to encode unauthorized data through a microarchitectural covert channel. The branch resolution time should be long enough for the sender to finish all transient instructions. To this end, the attacker can induce a slow memory access by flushing bound n to the memory (Algorithm 1 - line 19). It may also use more complex branching statements. Conceptually, once the expression of a branching statement is determined (e.g., $index < n$ in Algorithm 1 - line 6), the resolution time is relatively constant and shows little relation to statements in the branch.

We run experiments on gem5 [4] to verify the constant branch resolution time. It is a cycle accurate simulator that can model an out-of-order processor and microarchitectural state effects in the cache hierarchy by speculative execution [34], [45]. We construct a program following Algorithm 2 and measure insensitivity of branch resolution time to statement complexity in the branch as well as statement complexity of branch condition. The measurements on a real processor also demonstrate the same properties (Section VI-D).

First, our measurements demonstrate that given a fixed branching statement, varying statements in the branch barely fluctuates the branch resolution time. Following Algorithm 2, we poison the branch predictor to trigger mis-speculation to the sender (line 19), load $P[0]$ into the cache, and flush the blocks $P[64*1]...P[64*n]$ residing in the cache (lines 20-21). We assume that the secret value is one bit (0 or 1) and thus the in-branch load instructions will either all hit the cache when the secret value is 0 or all result in cache misses when the secret value is 1. We vary the number of loads (lines 9-12) and the secret value to observe the branch resolution time under a fixed expression $f(N)$. As shown in Figure 2, the branch resolution time given a fixed $f(N)$ is contained within

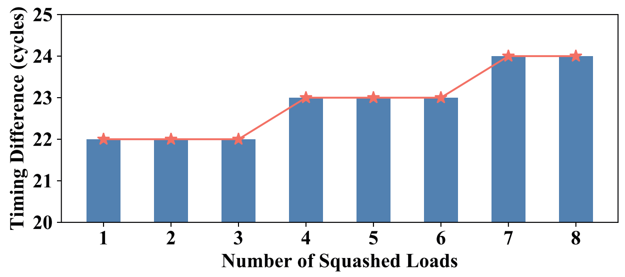

Fig. 3. The timing difference for rollbacking secret-dependent cache states is sufficient for a timing channel.

a relatively narrow band regardless of the number of in-branch loads. Furthermore, given a fixed $f(N)$ and a fixed number of in-branch loads, the branch resolution time is also insensitive to the secret value.

Second, our measurements demonstrate that the branch resolution time increases with the complexity of the branching statement. We adjust the complexity by tuning the number N of non-pipelined memory accesses required by computing expression $f(N)$ (Algorithm 2 - line 7). For example, we can assign the bound variable for bounds checking as a multi-dimensional array element and enforce multiple dependent memory accesses for getting array indices. An example configuration enforcing N=3 memory accesses is `if (index < A[A[A[2][0]][0]][0])`, where A[2][0] = 1, A[1][0] = 0, and A[0][0] = bound. As shown in Figure 2, the branch resolution time linearly increases with N. In the process of crafting attacks, we can optimize $f(N)$ such that 1) it yields a branch resolution time that is sufficiently long to cover the execution of transient instructions in the branch, and 2) it makes the branch resolution time as short as possible such that the mis-speculation window (i.e., T1-T2 in Figure 1) accounts for a limited portion of the entire timeline and therefore limits its impact on timing difference measurements.

B. Differentiable Rollback Time

We observe that secret-dependent rollback time upon mis-speculation is differentiable. In order to measure such timing difference, we craft the secret value in Algorithm 2 to vary the amount of cleanup operations on CleanupSpec. If the secret is set 0, the loads in the branch all hit the cache during speculative execution. After resolving the branch and starting to execute cleanup operations, few operations are needed because the cache states are unmodified and thus take little time. On the contrary, if the secret is set 1, the transient instructions will load some blocks from the memory to the cache. In the cleanup stage (T3-T5 in Figure 1), to rollback the changed cache states, CleanupSpec at least invalidates the transiently installed lines and restores the original lines evicted from the L1 cache if necessary. In this case, the CPU core stalls for a longer time to execute cleanup operations than when the secret is set 0.

Figure 3 reports the differentiability of rollback time with varying number of transient loads under different secret values. First, we measure the timing difference given only one transient load. When the secret is 1, it takes 22 more clock

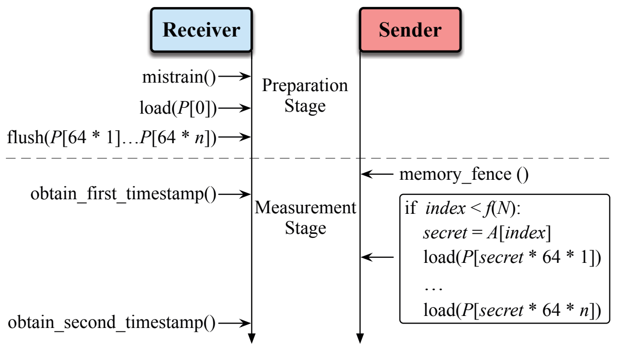

Fig. 4. unXpec attack design.

cycles for rollback than when the secret is 0. According to established attacks [3], [46], 22 cycles are sufficiently distinguishable to build a timing channel. We further measure the variation of the timing difference along with different number of transient loads. An interesting observation is that more transient loads do not necessarily yield a significant growth of timing difference [34]. This helps to control attack overhead through issuing a limited number of transient loads.

V. unXpec

In this section, we detail the design of our unXpec attack. It exploits the secret-dependent timing difference demonstrated by rollbacking transient loads, which are deemed as hard to exploit previously [34]. By further leveraging eviction sets [16], [29], [30], [37], [41], transient loads can evict more original cache lines and lead to a longer rollback time as well as a larger timing difference. We also discuss how to calibrate the number of transient loads toward improving the attack rate and resolution.

A. Attack Design

Figure 4 sketches the key framework of our unXpec attack. It consists of two stages, preparation and measurement. In the preparation stage, the receiver first poisons the branch predictor to incur subsequent mis-speculation. Furthermore, the receiver also instruments cache states via loading and flushing certain cache lines such that the measurement stage can accurately observe secret-dependent timing difference. Specifically, in the measurement stage, the receiver triggers mis-speculation in the sender to make the sender execute secret-dependent load instructions. Once the mis-speculation is detected, cleanup operations take place to rollback cache states modified by the secret-dependent load instructions. The receiver measures the sender execution time and the rollback time using instructions such as `rdtscp`. Finally, the receiver uses the time measurement to decode the secret.

Preparation Stage. Before causing the sender to enter speculative execution, in the preparation stage, the receiver brings microarchitecture into desired states. Specifically, we target two types of microarchitecture components—branch predictor and caches. First, the receiver mistrains the branch predictor by performing a series of instructions. A mistrained branch predictor leads to out-of-bounds accesses by transient instructions. Second, we instrument caches by loading $P[0]$ into the cache and flushing $P[64*1]...P[64*n]$. The essential goal for such preparation is that the sender program under secret 0 yields all-hits of $P[0]$ and the sender program under secret 1 yields all-misses of $P[64*1]...P[64*n]$. This helps generate a significant secret-dependent timing difference.

Measurement Stage. In the measurement stage, the sender is triggered to execute secret-dependent load instructions. The sender essentially encodes the secret value through the timing difference that is originated from these instructions' cleanup operations. As shown in Figure 1, the cleanup operations in T3 and T5 are interleaved with execution of inflight correct-path load instructions in T4. However, the execution time of T4 cannot be controlled by the attacker. To avoid such uncertainty from being counted into the time measurement, we at the begging of the measurement stage let the sender execute a memory fence instruction. The memory fence instruction zeros out the time of T4 from the entire cleanup timeline. Right after the memory fence instruction, we obtain the first timestamp prior to the execution of the branch statement. We obtain the other timestamp after cleanup operations as to be discussed shortly. Finally, we approximate the timestamp difference as the time measurement to decode the secret.

The sender continues to resolve the branch by computing the expression $f(N)$ and comparing it to the index. As mentioned in Section IV-A, the expression $f(N)$ is elaborately designed to make the sender sufficiently execute the transient instructions in the branch. Concurrently computing the branch statement, the CPU starts to fetch and execute the subsequent instructions leveraging out-of-order execution and speculative execution. Because the branch predictor is mistrained by the receiver in the preparation stage, the sender then uses out-of-bounds $index$ (crafted by the receiver) to execute the transient instructions in the branch. Therefore, the sender can access the target address that is specified by the value of A plus out-of-bounds $index$ and temporarily load the one-bit secret to CPU [2], [3]. Different secret values decide exact data blocks to be accessed by the subsequent transient loads.

- If secret = 0, all the load instructions in the branch will issue memory accesses to the same $P[0]$. Because $P[0]$ is loaded to the cache by the receiver in the preparation stage, the in-branch load instructions will all hit the cache. No cache state is modified by them and thus there is no need for the CPU to rollback the cache states after detecting mis-prediction.
- If secret = 1, on the contrary, those load instructions will all miss in the cache and must bring the requested data blocks from memory to the cache hierarchy. Cache states will be modified by transient instructions. After the mis-speculated branch statement is detected, CleanupSpec needs to clean up the modified cache states. For example, CleanupSpec stalls the core to invalidate the transient cache line installs in the cache hierarchy brought by the transient load instructions and restores the original L1 cache lines that are evicted. In comparison with when the secret is 0, the CPU takes a longer time to finish the cleanup operations, introducing an extra time overhead in the measurement stage.

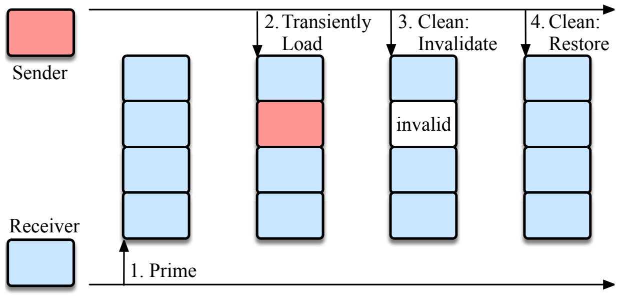

Fig. 5. unXpec using eviction sets to enlarge the secret-dependent timing difference.

After finishing the cleanup operations, the sender redirects to the correct path. The receiver will obtain the second timestamp, which nearly equals to the end of sender's cleanup operations. We estimate the time measurement by subtracting the first timestamp from the second one.

We now demonstrate that the preceding time measurement is secret dependent. Essentially, the time measurement is equivalent to the time span from T1 to T6 in Figure 1. As the memory fence instruction was executed in advance and all transient instructions in the branch have sufficient time to execute, we eliminate the impact of T3 and T4 on the entire timeline. Moreover, we have argued that the time from T1 to T2 approximates a constant and the secret-dependent cleanup operations in T5 can generate differentiable timing difference in Section IV. We conclude that only the timing difference originated from secret-dependent cleanup operations is variable in the measured time interval. By statistically analyzing the time measurement, the receiver can infer the one-bit secret information.

B. Attack Optimization

In practice, the larger the timing difference arises from different secret values, the higher resolution the attacker can achieve to infer secrets. The timing difference in our unXpec attack mainly comes from different amounts of cleanup operations for rollbacking secret-dependent transient loads. The rollback further consists of invalidation and restoration. The time for invalidation is relatively fixed according to the number of transient loads. However, the time for restoration varies with the number of L1 cache lines evicted by transiently-loaded data blocks. If the L1 cache is not sufficiently warmed up, transient loads may just bring some new data blocks into the cache without evicting any original cache line therein. In this case, the restoration contributes no time add-up to rollback and thus limits the secret-dependent timing difference. Enlarging the timing difference calls for sufficient evictions. This motivates us to leverage eviction sets [16], [29], [41] to pre-load cache lines that must be evicted by transient loads.

Specifically, we construct eviction sets [16], [29], [41] to prime the cache sets where $P[64*1]...P[64*n]$ locate in the L1 cache prior to the measurement stage. As shown in

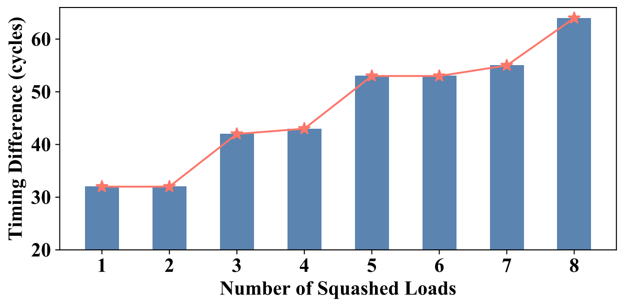

Fig. 6. The timing difference for rollbacking secret-dependent cache states can be enlarged using eviction sets.

Figure 5, the transient loads in the branch will definitely evict some original line and thus induce the L1 cache to restore the evicted lines during cleanup operations. Since $P[64 * 1] ... P[64 * n]$ correspond to transient loads when the secret value is 1, priming their cache sets introduces more restoration operations during cleanup than when the secret value is 0 and thus enlarges the secret-dependent timing difference. Although CleanupSpec employs a way-partition cache (e.g., NoMo [12]) to prevent the L1 cache from non-speculative side-channel attacks by an SMT adversary [24], it cannot prevent the adversary from constructing eviction sets in non-SMT scenarios. We can still construct eviction sets in our attack because of the non-SMT threat model we follow (Section III).

We validate the optimization effect using eviction sets and report the measurement in Figure 6. The secret-dependent timing difference has been enlarged from about 20 cycles to 32∼64 cycles. Because priming cache sets completes ahead of the measurement stage, it has no effect on the receiver's measurement except enlarging the rollback time. We can leverage the enlarged timing difference against background noise and yield a higher attack resolution.

C. Attack Parameterization

It is key to tune the number of loads in the branch toward high attack rate and accuracy. More loads in the branch may yield a longer rollback time and thus a more noticeable secret-dependent timing difference. Theoretically, we can unlimitedly increase the number of loads in the branch as long as the branch statement $f(N)$ is sufficiently complex. Since resolving the branch takes a nearly constant time given a fixed $f(N)$, it seems that the timing difference under different secret values (T1-T6 in Figure 1) can be arbitrarily enlarged. Two practical concerns, however, suggest that we should carefully choose the number of loads for our unXpec attack.

First, too many loads in the branch decrease the attack rate. This is because that more loads lead to a longer timeline for CleanupSpec (T1-T6 in Figure 1). This directly enforces a longer measurement time per round for the attacker. The associated benefit is a higher robustness against background noise and thus a higher attack resolution.

Second, too many loads may even affect the attack accuracy. More loads in the branch require a more complex branch statement. Conceptually, if the sender takes too much time to resolve the branch, it directly increases the T1-T2 time in Figure 1 and thus increases the total time observed by the receiver (i.e., T1-T6 in Figure 1). Meanwhile, if the timing difference brought by cleaning more transient cache line installs grows slowly with the increase of branch resolution time, its proportion in the entire T1-T6 time becomes smaller. This makes the attacker harder to infer the secret information and thus finally get a higher bit error probability.

As a result, we should trade off the number of load instructions in the branch against the attack rate and accuracy. As demonstrated in Section IV-B, if we do not apply any optimization to unXpec, the secret-dependent timing difference grows slowly with the number of loads in the branch. In this case, because the secret-dependent timing difference is sufficient to distinguish the secret value even when we place only one load in the branch, the attacker is more likely to employ fewer load instructions to achieve higher throughput. However, after we apply the eviction set to our attack, the secret-dependent timing difference has been greatly improved. This guides the attacker to pre-train the adaptive parameters to balance the secret-dependent timing difference and the time required for every round of attack.

TABLE I
EXPERIMENT SETUP [34].

Module	Configuration
Processor	1 core, 2 GHz, out-of-order 192-entry ROB
Private L1 I cache	32 KB, 4-way, 128-set
Private L1 D cache	32 KB, 8-way, 64-set
Shared L2 cache	2 MB, 16-way, 2048-set
Memory	50 ns RT after L2

VI. EVALUATION

Settings. In this section, we evaluate unXpec over the typical undo scheme—CleanupSpec [34]. Given that undo schemes have not applied to commercial CPUs yet, we implement unXpec and run it on gem5 in System-call Emulation (SE) mode [15]. We run the unXpec attack against the open-source CleanupSpec [33]. Our unXpec requires that the receiver share the same cache hierarchy with the sender and observe CPU stalls caused by cleanup operations by the sender. To satisfy the first requirement, we pin the receiver and sender onto the same thread [2], [10], [13], [34], [45]. To satisfy the second requirement, we configure CleanupSpec with the `Cleanup_FOR_L1L2` mode. We follow all the other configurations as identical as CleanupSpec [34]. Table I summarizes the key settings. Configuration specifics necessary for understanding will be also discussed along with the reported measurements.

Results. We validate the feasibility of unXpec by measuring secret-dependent timing differences. It demonstrates an average timing difference of 22 cycles and 32 cycles given only a single load in the branch without and with using eviction sets, respectively. Both versions of unXpec can sample up to 140,000 time measurements per second on a 2 GHz CPU. To infer 1,000 randomly generated secret bits, the unXpec

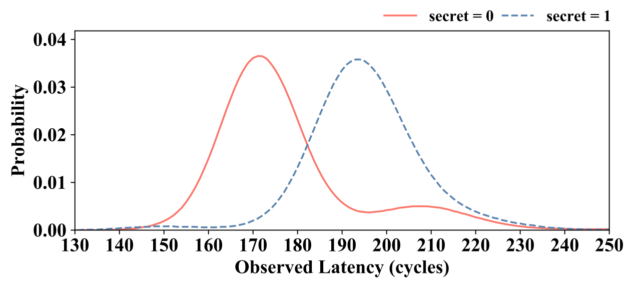

Fig. 7. Probability density function for sender execution time without using eviction sets, estimated by kernel density estimation [3]. The average secret-dependent timing difference is 22 cycles.

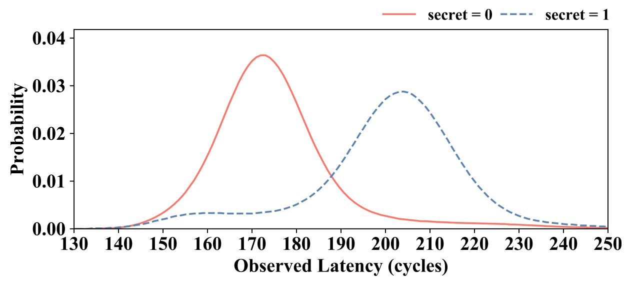

Fig. 8. Probability density function for sender execution time using eviction sets, estimated by kernel density estimation [3]. The average secret-dependent timing difference is 32 cycles.

attacks without and with eviction sets can respectively achieve an accuracy of 86.7% and 91.6% with only a single sample per bit. This yields a sufficiently high leakage rate of 140 Kbps. We validate unXpec's insensitivity to system noise on an Intel Core i7-8550U processor.

Furthermore, we investigate the impacts of the most intuitive countermeasure against unXpec—constant-time rollback—on computer performance. Experimental results demonstrate that the constant-time rollback scheme introduces non-negligible overhead, ranging from 22.4% with 25-cycle constant rollback time to 72.8% with 65-cycle constant rollback time. This solicits either a more efficient countermeasure against unXpec or a more robust safe speculation rather than Invisible and Undo solutions.

A. *Feasibility: Timing Difference*

To validate the feasibility of unXpec, we run both unXpec versions over CleanupSpec to demonstrate the secret-dependent timing difference. As discussed in Section IV, a limited complexity for the branch statement and in-branch loads suffices to differentiate rollback timings corresponding to different secret values. We thus set the branch statement $f(N)$ with only a single memory request and set the in-branch instructions with only a single load. Specifically, the related source code segment flushes N and then accesses N in the branching statement as follows: `clflush(&N); if(index < N)`, where index is a cached variable. We then collect the time measurements given different secret values. When the secret is set zero, the CPU performs no operation in the cleanup stage. When the secret is set one, the CPU conducts cleanup operations (e.g., invalidating the transient cache-line installs and restoring the evictions in the L1 cache) to rollback the cache states. We collect 1,000 measures for either case and generate the timing distribution.

Figure 7 and Figure 8 report the probability density function without and with the optimization enabled, respectively. For both versions of unXpec, the receiver can observe a noticeable secret-dependent timing difference. The secret value of one yields an averagely larger time measurement. This can be justified by the fact that when the secret is set one, the in-branch load instructions bring some new data blocks from memory to cache and thus the CPU must do some cleanup operations in the subsequent cleanup stage. Furthermore, after enabling the optimization option in our attack, the observed timing difference increases from 22 cycles to 32 cycles. This is because CleanupSpec must additionally access the lower memory hierarchy to restore the evicted original line. The preceding observations are in line with our analysis in Section IV. Based on the timing distribution in Figure 7 and Figure 8, we respectively choose the observed latency of 178 and 183 as the thresholds of time measurement to differentiate secrets.

B. *Speed: Leakage Rate*

To evaluate the leakage rate about unXpec, we measure the sample rate of time measurements. More specifically, it may take several samples of time measurements to infer a single secret bit. The scale of a single time measurement matters for the overall attack speed. Intuitively, the optimized version of unXpec takes more time per sample because performing the prime operations introduces an extra overhead. However, in a low-noise environment, we only need to prime the sets once. As long as we bring the cache states into desired states for the first time, they remain the same at the beginning of each round of subsequent attacks. This is because that even though the sender changes the cache states transiently, the sender's cleanup operations will rollback all the modified cache states before the next round of attack starts.

Both versions of unXpec demonstrate a comparative sample rate around 140,000 samples/second on a 2 GHz CPU. This promises a leakage rate of 140 Kbps as our unXpec attacks can achieve a sufficiently high resolution with only one sample per bit (Section VI-C).

C. *Effectiveness: Secret Leakage*

Since Undo-based safe speculation such as CleanupSpec has not been implemented in any commercial machine, we implement a Meltdown-alike proof-of-concept attack to demonstrate the effectiveness of both unXpec attacks. For inferring a secret bit, we first randomly generate a one-bit value in the target address. The value cannot be directly accessed by the attacker. The attacker then runs unXpec to steal the secret. Specifically, it measures the execution time of the sender and compares the measurement with the pre-defined threshold discussed in Section VI-B. The attacker guesses the secret bit as 1 if the time measurement exceeds the threshold and as 0 otherwise. We repeat the preceding process over a randomly generated 1,000-bit secret as in Figure 9.

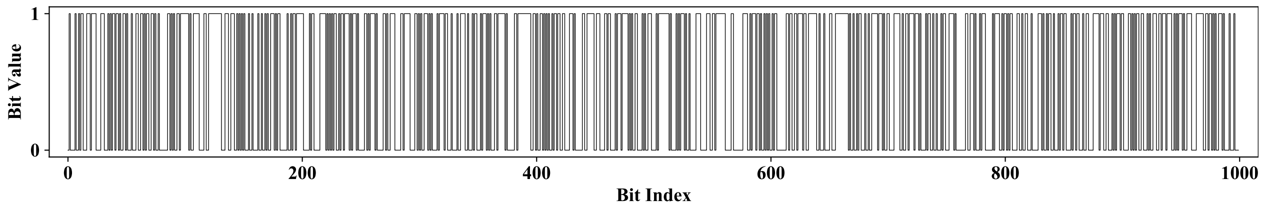

Fig. 9. Bit pattern of 1,000-bit randomly generated secrets.

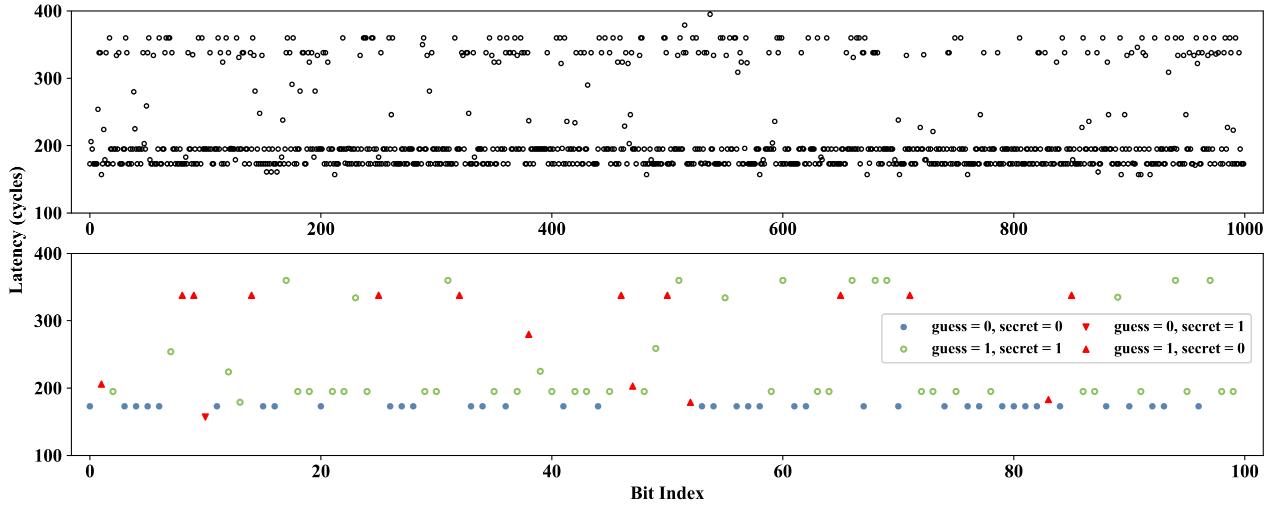

Fig. 10. The observed latency by the receiver without using eviction sets. Top: all the 1,000 measurements; Bottom: the first 100 measurements and the corresponding secret inference results in detail.

Figure 10 and Figure 11 report the observed latency and the corresponding secret guessing results for unXpec and unXpec with eviction sets, respectively. Specifically, they show the observed latency over 1,000 bits (top) and the guess results of only the first 100 bits (bottom) for ease of illustration. We start with unXpec without using eviction sets. As shown in Figure 10, most observed latency measurements locate close to the threshold while a few spread further. Such distribution complies with the statistics we have reported in Figure 7. In this case, unXpec accurately guesses 867 secret bits, resulting in an attack accuracy of 86.7% that is comparative with that of relevant attacks [2], [3]. Leveraging eviction sets, unXpec can obtain a larger secret-dependent timing difference. This makes the secret guessing process less susceptible to background noise and thus increases attack accuracy. As shown in Figure 11, unXpec using eviction sets correctly guesses 916 bits, yielding a much higher accuracy of 91.6%.

D. Robustness: Noise Insensitivity

System noise should show little effect on unXpec and the timing difference observation because of the following three reasons. First, unXpec relies on the timing difference originated from the secret-dependent cleanup operations. Since the CleanupSpec stalls the core upon cleanup operations, the secret-dependent timing difference exploited by unXpec is not affected by noise. Second, whether the secret is 0 or 1, the noise causes the same effect on the remaining execution time (e.g., branch resolution time). As shown in Figure 13, we further validate the properties of branch resolution time on the Intel Core i7-8550U processor. Albeit system noise, different secrets demonstrate approximate branch resolution timings and thus the corresponding conclusion in Section IV-A is practically validated. Combining the above two points, system noise causes trivial effect on the observed latency including the branch resolution time and the secret-dependent cleanup time. Third, following existing attacks [3], [46], the attacker can also use more samples per secret to suppress noise.

E. Mitigation: Constant-Time Rollback

Finally, we evaluate performance overhead of a potential unXpec countermeasure that enforces constant rollback time for undo schemes. Albeit CleanupSpec [34] regards the rollback process as hardly exploitable, it also considers constant-time rollback theoretically secure yet without thorough investigation. Specifically, if any mis-speculation occurs, we stall the core for a constant time to execute the cleanup operations. Even if the CPU finishes the cleanup operations ahead of schedule or no cleanup operation takes place at all when transient instructions have not modified cache states, the CPU should still be stalled until the pre-defined constant time elapses. However, such a potential countermeasure introduces non-negligible time overhead and design intricacy.

First, it is challenging to decide the value of the constant time. According to the measurements in Section IV, rollback time depends on the amount of cache changes induced by transient loads meanwhile. This factor, however, can be hard to

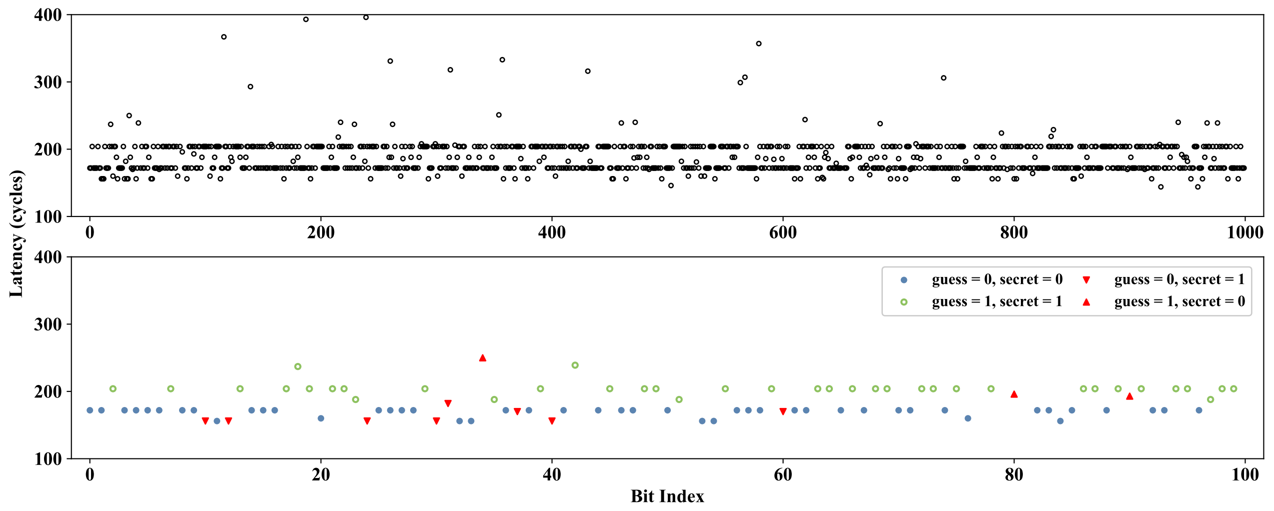

Fig. 11. The observed latency by the receiver using eviction sets. Top: all the 1,000 measurements; Bottom: the first 100 measurements and the corresponding secret inference results in detail.

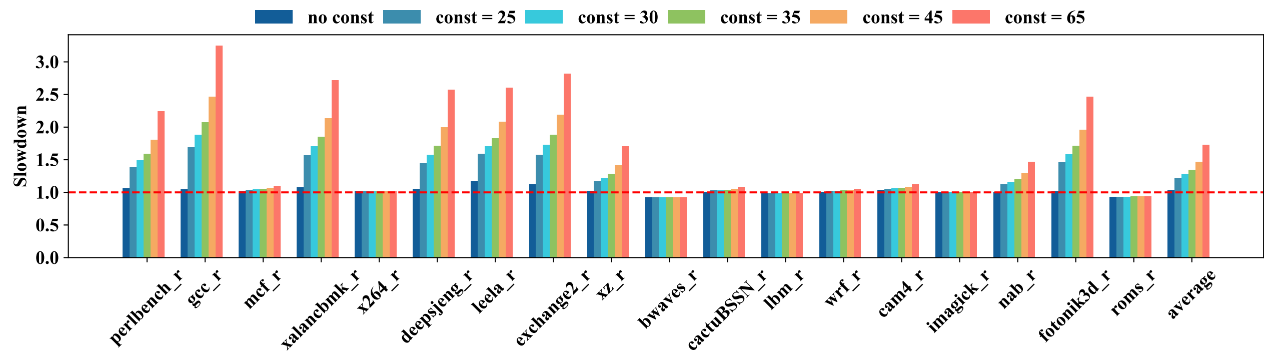

Fig. 12. Overhead of constant-time rollback with varying constant cycles using SPEC CPU 2017 benchmarks.

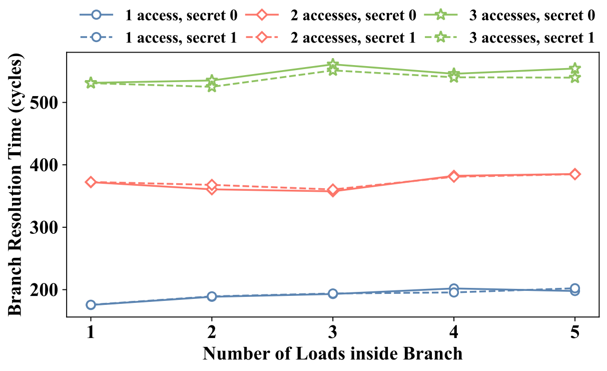

Fig. 13. The branch resolution time on an Intel i7-8550U processor.

predict. If the constant rollback time is set too large, the CPU will suffer from a heavy time overhead and thus significantly slow down performance.

Second, if the constant rollback time is set smaller than necessary, a constant-time solution may leave the door open for speculative execution attacks again. Such a solution follows two possible design strategies. One is to enforce a strict time threshold on every rollback. Given a insufficient time scale, some cache states modified by transient instructions may not be rollbacked during the cleanup stage. In this case, the rollback effect is incomplete and leaves the residual cache state changes exploitable for speculative execution attacks again. On the contrary, the other constant-time rollback strategy relaxes the time restriction for rollbacks that require more time toward no residual speculative cache state. This does not fully hide the timing difference among rollbacks and is still vulnerable to unXpec. However, in comparison with the first strategy, the relaxed one guarantees the complete effect of CleanupSpec. We thus implement the relaxed constant-time strategy to protect CleanupSpec and report the measurement results in Figure 12.

Only a small fraction (e.g., $< 5\%$) of transient load instructions change cache states and enforce cleanup operations [34]. As described above, as long as mis-speculation is detected, we should stall the core for a constant time. However, CleanupSpec [34] shows that more than 95% of transient loads hit the L1 cache and need no cleanup operations. In other words, only less than 5% of transient loads change cache states and require cleanup operations such as invalidation and restoration to rollback. Enforcing constant rollback time inevitably makes the common cases as time-consuming as the rare cases. According to the rollback-timing difference that can be measured by the attacker with or without eviction sets (Figure 3 and Figure 6), we evaluate CleaupSpec with constant rollback time ranging from 25 cycles to 65 cycles. Figure 12 reports the overhead of different levels of constant rollback time using extensive SPEC CPU 2017 benchmarks [6]. Specifically, the overhead is quantified as the execution time of constant-rollback CleanupSpec normalized by that of the unsafe baseline. The average slowdown increases with the number of cycles enforced as constant rollback time, ranging from 22.4% with 25-cycle constant rollback time to 72.8% with 65-cycle constant rollback time. Such a significant overhead necessitates either a more efficient countermeasure against unXpec or a more robust safe speculation rather than Invisible and Undo solutions.

VII. Conclusion

We have exploited secret-dependent timing differences to break undo-based safe speculation. The timing difference arises from whether or not undo operations involve much restoration (which was deemed as hardly exploitable [34]) of evicted data from the lower-level cache/memory hierarchy to the L1 cache. We construct our unXpec attack programs against the representative undo-based scheme called CleanupSpec [34] and demonstrate a 22-cycle secret-dependent timing difference. Using eviction sets to enforce more evicted cache lines and thus a longer restoration time, unXpec yields an even larger secret-dependent timing difference of 32 cycles. This promises a higher robustness against noise as well as a higher attack accuracy in terms of correct decoding of secret bits out of time measurements. We implement unXpec and the evaluation shows that unXpec can leak secrets from CleanupSpec-enabled systems at a high rate of 140 Kbps with a noise-insensitive accuracy over 90%. As a potential countermeasure, simply enforcing constant-time undo operations may induce a heavy overhead over 70%. All these findings necessitate exploring more efficient countermeasures against unXpec or more robust Undo-based safe speculation.

Our future work aims to explore a lightweight defense using dummy cleanup operations. Inspired by the *fuzzy time* technique [14], [18] for mitigating cache side channels, we can inject dummy cleanup operations to disguise the rollback time. For example, the CPU can issue dummy loads during branch resolution or inject random time delays among cleanup operations. In comparison with enforcing a constant rollback time (Section VI-E), random noise times may mitigate the unXpec attack with a lower time overhead. This is because that it no longer needs to enforce the longest waiting time for cleanup operations.

Acknowledgements

We sincerely thank HPCA 2022 Chairs and Reviewers for the review efforts and helpful feedback. We also thank HPCA 2022 AEC Chairs and Reviewers for evaluating our artifact and guiding us toward a comprehensive instruction of running the artifact and reproducing the results. We would also like to extend our gratitude to the authors of CleanupSpec for sharing the source code. We wish everyone health and safety during the pandemic.

Artifact Appendix

A. Abstract

The artifact includes unXpec's source code and documentation. In addition, we provide instructions on how to deploy unXpec on the current state-of-art undo-based safe speculation—CleanupSpec, alongside scripts to run all experiments and reproduce the reported results.

B. Artifact Check-List (Meta-Information)

- **Algorithm:** unXpec attacks to evaluate the security of Undo-based safe speculation (i.e., CleanupSpec) and a constant-time countermeasure against unXpec to evaluate protection overhead.
- **Program:** Open-source CleanupSpec as the state-of-the-art Undo-based safe speculation. (Note that the SPEC CPU 2017 benchmarks of version 1.1.0 used for performance evaluation in Figure 12 are not included because they are license protected and can be purchased at https://www.spec.org/cpu2017/.)
- **Compilation:** Python 2.7, GCC 5, G++ 5, and gem5 dependencies (e.g., protobuf and gperftools), all of which are included.
- **Run-time environment:** Ubuntu and Docker.
- **Execution:** Dozens of minutes for attack demonstration and about ten hours for countermeasure evaluation using the slowest benchmark.
- **Metrics:** Execution time in terms of the number of cycles.
- **Output:** The artifact outputs statistics for generating the reported results either in console for direct use or in files for further analysis.
- **Experiments:** Step-by-step experiment guidelines are provided in the subsequent sections as well as documented at https://doi.org/10.5281/zenodo.5771649.
- **How much disk space required (approximately)?:** 10 GB.
- **How much time is needed to prepare workflow (approximately)?:** It takes about two hours to pull two images (about 2 GB, one containing the attack artifact and the other containing the countermeasure artifact) from Docker Hub.
- **How much time is needed to complete experiments (approximately)?:** It takes about half an hour to collect statistics for attack demonstration (Section VI-A to Section VI-D) and 10+ hours for countermeasure evaluation if sufficient cores (e.g., over 20) are available for running benchmarks.
- **Publicly available?:** The artifact is publicly available at https://doi.org/10.5281/zenodo.5771649.
- **Archived (provide DOI)?:** 10.5281/zenodo.5771649.

C. Description

1) How to access: The artifact is publicly available at https://doi.org/10.5281/zenodo.5771649.

2) Hardware dependencies: The experiments results have no specific hardware dependencies. Yet the results in Section VI-D for demonstrating timing differences on a real CPU need an Intel Core i7-8550U processor to reproduce. Processors of different models may yield different time measurements while exhibiting exploitable timing differences.

3) Software dependencies:

- Ubuntu 16.04
- gem5 (Commit:39cfb85 from Nov 5, 2018)
- python2.7
- gcc-5, g++-5
- other gem5 requirements (protobuf, gperftools, etc.)
- SPEC2017-1.1.0 (Reference Input)

D. Installation

The artifact installation process requires to first install Docker via https://docs.docker.com/engine/install/ and then build two docker images, which are addressed as unxpec:01 and unxpecspec:01 in what follows.

E. Experiment Workflow

The detailed instruction for conducting experiments, collecting statistics, and reproducing the reported results is available in the file of README.md included in the artifact. Results reported in Section VI-A, Section VI-B, and Section VI-C can be reproduced in the docker image unxpec:01. Results reported in Section VI-E can be reproduced in the docker image unxpecspec:01. However, results reported in Section VI-D should be reproduced on a real processor.

For Section VI-A, Section VI-B, and Section VI-C, the reproduction process first enters the unXpec container created by unxpec:01 using the following command:

```
bash run.sh docker
```

For Section VI-E, a similar self-contained container cannot be built simply using unxpecspec:01 because the license-protected benchmarks cannot be included. An additional directory containing the licensed SPEC CPU 2017 benchmarks should be mounted to unxpecspec:01 to build the experiment environment for Section VI-E.

Next, we detail the instruction for reproducing results.

1) Section VI-A: Feasibility - Timing Difference:

- In the docker container created by unxpec:01:
  ```
  cd /home/gem5/unXpec
  ```
- To reproduce the results reported in Figure 7, run unXpec attack without eviction sets:
  ```
  bash run.sh TimingDifference
  ```
  Time measurements corresponding to secret bit 0 and secret bit 0 are logged in unXpec/TimingDifference/NoEvictionSet_Sec0.txt and unXpec/TimingDifference/NoEvictionSet_Sec1.txt, respectively.
  We extract the 1,000 sample measurements from NoEvictionSet_Sec0.txt (lines 29-1028) and assign them to an array s0. Similarly, we extract the 1,000 sample measurements from NoEvictionSet_Sec1.txt (lines 29-1028) and assign them to an array s1.
  Then we use kernel density estimation built in Matlab to estimate the probability density functions over s0 and s1, as shown in Figure 7. The Matlab code is available at unXpec/TimingDifference/kde.m.
- To reproduce the results reported in Figure 8, run unXpec attack with eviction sets:
  ```
  bash run.sh TimingDifference -e
  ```
  Time measurements corresponding to secret bit 0 are logged in unXpec/TimingDifference/EvictionSet_Sec0.txt. Time measurements corresponding to secret bit 1 are logged in unXpec/TimingDifference/EvictionSet_Sec1.txt.
  We extract the 1,000 sample measurements from EvictionSet_Sec0.txt (lines 29-1028) and assign them to an array s0. Similarly, we extract the 1,000 sample measurements from EvictionSet_Sec1.txt (lines 29-1028) and assign them to an array s1.
  Then we use kernel density estimation built in Matlab to estimate the probability density functions over s0 and s1, as shown in Figure 8. The Matlab code is available at unXpec/TimingDifference/kde.m.

2) Section VI-B: Speed - Leakage Rate:

- In the docker container created by unxpec:01:, enter the unXpec directory:
  ```
  cd /home/gem5/unXpec
  ```
- Run unXpec attack:
  ```
  bash run.sh LeakageRate
  ```
- The measured leakage rate is directly printed on the console.

3) Section VI-C: Effectiveness - Secret Leakage:

- In the docker container created by unxpec:01:, enter the unXpec directory:
  ```
  cd /home/gem5/unXpec
  ```
- To reproduce the results reported in Figure 10, run unXpec attack without eviction sets:
  ```
  bash run.sh SecretLeakage
  ```
  The test instance of 1,000-bit randomly generated secrets shown in Figure 9 is hardcoded in the attacking code. The results is logged in unXpec/SecretLeakage/NoEvictionSet_Sample1k.txt.
- To reproduce the results reported in Figure 11, run unXpec attack with eviction sets:
  ```
  bash run.sh SecretLeakage -e
  ```
  The test instance of 1,000-bit randomly generated secrets shown in Figure 9 is hardcoded in the attacking code. The result is logged in unXpec/SecretLeakage/EvictionSet_Sample1k.txt.

4) Section VI-D: Robustness - Noise Insensitivity:

- Run the measurement code on a real machine with Ubuntu OS.
  ```
  sudo bash run.sh NoiseInsensitivity
  ```
  The machine used for collecting statistics in Figure 13 uses an Intel Core i7-8550U processor.
- The results are logged in unXpec/NoiseInsensitivity/finalresult.txt.

5) Section VI-E: Mitigation - Constant-time Rollback:

Run: For Section VI-E, a self-contained container cannot be built simply using unxpecspec:01 because the license-protected benchmarks cannot be included. An additional directory containing the licensed SPEC CPU 2017 benchmarks should be copied to unxpecspec:01 to build the experiment environment for Section VI-E. Please refer to the following steps for copying and running SPEC:

First, put all benchmark files including their executable files and input files into a directory (addressed as spec_dir subsequently).

Second, copy all files in spec_dir to the docker container created by unxpecspec:01 with following commands:

`docker ps` to get the container ID (addressed as ContainerID subsequently);

```
docker cp spec_dir/. ContainerID:/home/gem5
```

to copy all files in spec_dir to the docker container.

In the docker container, enter the gem5 directory:

```
cd /home/gem5
```

Then for every benchmark reported in Figure 12, we run the following command with scheme_cleanupcache set as both `UnsafeBaseline` and `Cleanup_FOR_L1L2` modes:

```
bash run_gem5spec.sh benchmark_name
maxinst_count startinst_count
scheme_cleanupcache
```

Most benchmarks set `maxinst_count` and `startinst_count` as `2000000000` and `1000000000`, respectively. While for `mcf_r` and `imagick_r` with limited size or run-time exception, we set their `maxinst_count` and `startinst_count` as `1000000000` and `500000000`, respectively.

Results: Enter /home/gem5/test*benchmark_name* for statistics in *benchmark_name*.txt, where *benchmark_name* represents the benchmark name under test.

Extraction: For `UnsafeBaseline` mode, extract the following metadata from *benchmark_name*.txt:

- `sim_ticks`: total time to execute `maxinst_count` instructions.
- `system.cpu.fetch.startCycles`: total time to execute the first `startinst_count` instructions.

For `Cleanup_FORL1L2` mode, extract the following metadata from *benchmark_name*.txt:

- `sim_ticks`: total time to execute `maxinst_count` instructions.
- `system.cpu.fetch.startCycles`: total time to execute the first `startinst_count` instructions.
- `system.cpu.iew.lsq.thread0.extraCleanupSquashTimeCyclesXX`: extra time imposed by XX-cycle constant-time rollback on CleanupSpec (only consider the last `maxinst_count - startinst_count` instructions).

Calculation: Estimate performance overhead of the last one billion instructions reported in Figure 12 as follows. (The first one billion instructions are used to warm up the system.)

- `unsafe-time`: under `UnsafeBaseline` mode
  `sim_ticks - system.cpu.fetch.startCycles`
- `no-constant`: under `Cleanup_FORL1L2` mode without constant-time rollback
  `sim_ticks - system.cpu.fetch.startCycles`
- `XX-const`: under `Cleanup_FORL1L2` mode with XX-cycle constant-time rollback
  `sim_ticks - system.cpu.fetch.startCycles + system.cpu.iew.lsq.thread0.extraCleanupSquashTimeCyclesXX`
- **overhead**
  `no-const` or `XX-const` divided by `unsafe-time`.

References

[1] S. Ainsworth and T. M. Jones, "Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state," in *ISCA*, 2020, pp. 132–144.

[2] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. Zhao, X. Zou, T. Unterluggauer, J. Torrellas, C. Rozas, A. Morrison, F. Mckeen, F. Liu, R. Gabor, C. W. Fletcher, A. Basak, and A. Alameldeen, "Speculative interference attacks: Breaking invisible speculation schemes," in *ASPLOS*, 2021.

[3] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: exploiting speculative execution through port contention," in *CCS*, 2019, pp. 785–800.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, S. Rathijit, S. Korey, S. Muhammad, V. Nilay, D. H. Mark, and A. W. David, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[5] S. Briongos, P. Malagón, J. M. Moya, and T. Eisenbarth, "Reload+refresh: Abusing cache replacement policies to perform stealthy cache attacks," in *USENIX Security Symposium*, 2020, pp. 1967–1984.

[6] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 41–42.

[7] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, B. Jo Van, and Y. Yuval, "Fallout: Leaking data on meltdown-resistant cpus," in *CCS*, 2019.

[8] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *USENIX Security Symposium*, 2019, pp. 249–266.

[9] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution," in *EuroS&P*, 2019, pp. 142–157.

[10] M. H. I. Chowdhuryy, H. Liu, and F. Yao, "Branchspec: Information leakage attacks exploiting speculative branch instruction executions," in *ICCD*, 2020, pp. 529–536.

[11] S. Deng, B. Huang, and J. Szefer, "Leaky frontends: Micro-op cache and processor frontend vulnerabilities," *arXiv preprint arXiv:2105.12224*, 2021.

[12] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM Transactions on Architecture and Code Optimization*, vol. 8, no. 4, pp. 1–21.

[13] J. Fustos, M. Bechtel, and H. Yun, "Spectrerewind: Leaking secrets to past instructions," in *CCS Workshop - ASHES*, 2020, pp. 117–126.

[14] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, 2018.

[15] gem5, "gem5." [Online]. Available: http://www.gem5.org

[16] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer. js: A remote software-induced fault attack in javascript," in *DIMVA*, 2016, pp. 300–321.

[17] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach.* Elsevier, 2011.

[18] W.-M. Hu, "Reducing timing channels with fuzzy time," *Journal of Computer Security*, vol. 1, no. 3-4, pp. 233–254, 1992.

[19] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Safespec: Banishing the spectre of a meltdown with leakage-free speculation," in *DAC*, 2019, pp. 1–6.

[20] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," *arXiv preprint arXiv:1807.03757*, 2018.

[21] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, S. Michael, and Y. Yuval, "Spectre attacks: Exploiting speculative execution," in *S&P*, 2019, pp. 1–19.

[22] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *WOOT*, 2018.

[23] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yuval, and R. Mike Hamburg, "Meltdown: Reading kernel memory from user space," in *USENIX Security Symposium*, 2018, pp. 973–990.

[24] F. Liu and R. B. Lee, "Random fill cache architecture," in *MICRO*, 2014, pp. 203–215.
[25] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *S&P*, 2015, pp. 605–622.
[26] K. Loughlin, I. Neal, J. Ma, E. Tsai, O. Weisse, S. Narayanasamy, and B. Kasikci, "Dolma: Securing speculation with the principle of transient non-observability," in *USENIX Security Symposium*, 2021.
[27] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *CCS*, 2018, pp. 2109–2122.
[28] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. A. Boano, S. Mangard, and K. Römer, "Hello from the other side: Ssh over robust cache covert channels in the cloud." in *NDSS*, 2017.
[29] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: Practical cache attacks in javascript and their implications," in *CCS*, 2015, pp. 1406–1418.
[30] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' Track at the RSA Conference*. Springer, 2006, pp. 1–20.
[31] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *MICRO*, 2018, pp. 775–787.
[32] X. Ren, L. Moody, M. Taram, M. Jordan, D. M. Tullsen, and A. Venkat, "I see dead μops: Leaking secrets via intel/amd micro-op caches," 2021.
[33] G. Saileshwar, "cleanupspec." [Online]. Available: https://github.com/gururaj-s/cleanupspec
[34] G. Saileshwar and M. K. Qureshi, "Cleanupspec: An" undo" approach to safe speculation," in *MICRO*, 2019, pp. 73–86.
[35] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Efficient invisible speculative execution through selective delay and value prediction," in *ISCA*, 2019, pp. 723–735.
[36] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "Netspectre: Read arbitrary memory over network," in *ESORICS*, 2019, pp. 279–299.
[37] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on aes, and countermeasures," *Journal of Cryptology*, vol. 23, no. 1, pp. 37–71, 2010.
[38] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution," in *USENIX Security Symposium*, 2018, pp. 991–1008.
[39] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lippi, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, "Lvi: Hijacking transient execution through microarchitectural load value injection," in *S & P*. IEEE, 2020.
[40] J. R. S. Vicarte, P. Shome, N. Nayak, C. Trippel, A. Morrison, D. Kohlbrenner, and C. W. Fletcher, "Opening pandora's box: A systematic study of new ways microarchitecture can leak private data," 2021.
[41] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," in *S&P*, 2019, pp. 39–54.
[42] Y. Wu and X. Qian, "Reversispec: Reversible coherence protocol for defending transient attacks," *arXiv preprint arXiv:2006.16535*, 2020.
[43] W. Xiong and J. Szefer, "Leaking information through cache lru states," in *HPCA*, 2020, pp. 139–152.
[44] W. Xiong and J. Szefer, "Survey of transient execution attacks and their mitigations," *CSUR*, vol. 54, no. 3, pp. 1–36, 2021.
[45] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *MICRO*, 2018, pp. 428–441.
[46] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are coherence protocol states vulnerable to information leakage?" in *HPCA*, 2018, pp. 168–179.
[47] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *USENIX Security Symposium*, 2014, pp. 719–732.