CONJURING: Leaking Control Flow via Speculative Fetch Attacks

Ali Hajiabadi
National University of Singapore

Trevor E. Carlson
National University of Singapore

ABSTRACT

In this work, we propose a new attack called CONJURING that exploits one of the main features of CPUs' frontend: speculative fetch of instructions. We show that the Pattern History Table (PHT) in modern CPUs are a great channel to learn and leak control flow of victim applications. Unlike prior work, CONJURING does not require that one primes the PHT or interferes with the victim execution enabling a realistic and unprivileged attacker to leak control flow information. By improving the branch predictors, our attack becomes even more serious and practical. We demonstrate the feasibility of our attack on different existing Intel, AMD, and Apple CPUs.

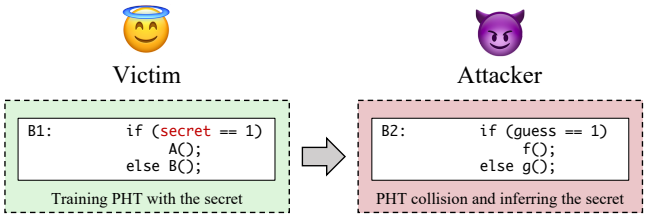

Figure 1: CONJURING attack overview.

1 INTRODUCTION

Modern CPUs deploy various optimizations to improve performance. One of the main feature of these processors is speculative fetch and execution upon branch instructions. Introduction of Spectre [20] in 2018 demonstrated the vulnerabilities arising from speculation. However, almost all speculation-based attacks focus on speculative execution of instructions and its side effects on the CPU's backend. As a consequence, all proposed defenses mitigate the misspeculated leaks in the backend and let the frontend to operate normally. In this work, we demonstrate that even speculatively fetching instructions without executing them can lead to data leaks that were not intended by the normal and correct execution of the programs. We refer to such attacks as *speculative fetch attacks*.

BranchScope [11] is one of the few speculative fetch attacks that uses the Pattern History Table (PHT) as a channel to learn the secret bits of a victim. To achieve this, (1) the attacker primes one of the PHT entries into a known state (e.g., strongly taken), (2) the victim executes a secret dependent branch (colliding with the primed PHT entry), and (3) the attacker extracts the secret bit by probing the same PHT entry to investigate changes to its state. In addition, BranchScope requires OS control (i.e., a privileged attacker) to interrupt the victim exactly at each iteration of the secret dependent branch to extract each bit of the secret.

In this work, we propose the CONJURING attack that leaks the victim control flow *without needing to prime the PHT and interfere with the victim execution* which enables an unprivileged attacker to accurately leak secret information. Figure 1 shows an overview of our attack. We evaluate different branch predictors and show that existing branch predictors are capable of remembering a significant portion of the victim's control flow decisions. Hence, we exploit the fact that the victim itself can train the branch predictor to remember the secret control flow pattern and later the attacker creates PHT collisions to infer the victim's decisions on a specific secret dependent branch. As a case study, we review RSA implementations found in various existing crypto libraries [24, 30] and show that TAGE-SC-L-64KB and Tournament branch predictors can remember more than 60% of the private exponent of realistic RSA keys which is sufficient for full key recovery by exploiting the redundancies in RSA public keys storage [15]. By improving branch predictors, they can lead to even more serious attacks without any cryptanalysis efforts.

Moreover, we present our Proof-of-Concept (PoC) attack to further demonstrate the feasibility of CONJURING on existing Intel, AMD, and Apple CPUs. Our results confirm that modern processors can show meaningful (i.e., secret dependent) differences upon fetching branches, leading to victim's control flow extraction.

The main contributions of this work are as follows:

- Proposing a practical variant of speculative fetch attacks, called CONJURING, demonstrating vulnerabilities for unprivileged attackers without requiring an additional PHT priming step;
- Analysis of different branch predictors and their vulnerabilities against speculative fetch attacks;
- Demonstrating the feasibility of CONJURING with PoC attacks targeting existing CPUs;
- Investigating different mitigation strategies to effectively block speculative fetch leaks; we propose efficient and comprehensive solutions for future CPUs.

The rest of the paper is organized as follows: In Section 2, we provide the necessary background on speculative fetch attacks. In Section 3, we introduce the CONJURING attack, and investigate the leakage of existing branch predictors against CONJURING. We further demonstrate the feasibility of CONJURING in Section 4 through our PoC attacks on Intel, AMD, and Apple CPUs. In Section 5, we discuss existing defenses against CONJURING and propose more efficient and comprehensive solutions for future CPUs. Finally, we discuss the Related Work in Section 6 and conclude in Section 7.

Responsible Disclosure. We have responsibly disclosed our attacks to the affected CPU vendors and received the approval to distribute our findings.

2 BACKGROUND

Speculative fetch attacks exploit the fact that the branch predictor remembers the decisions originally made by the victim. BranchScope [11] is one of the early versions of speculative fetch attacks. While BranchScope does not acknowledge the main source of the attack, it is able to extract victim's control flow by leaking data from the frontend. BranchScope exploits the victim gadgets containing secret dependent branches, and follows three steps:

```
1 void RSA_decode (int c, uint8 *d, int n){
2     int R0 = 1, R1 = c, i;
3     for (i = sizeof(d)-1; i >= 0; i++){
4         if (d[i] == 1){ R0 = R0 * R1 % n; R1 = R1 * R1 % n;}
5         else { R1 = R0 * R1 % n; R0 = R0 * R0 % n;}
6     }
7 }
```

Listing 1: RSA modular exponentiation using Montgomery Ladder Powering [18]; d is secret and control flow decisions on line 4 are secret dependent.

Step 1 *Priming the PHT.* The attacker initializes one of the Pattern History Table (PHT) entries to a known state (e.g., strongly taken).

Step 2 *Running the victim.* The attacker initiates the victim execution which its secret dependent branch collides with the same primed PHT entry.

Step 3 *Probing the PHT.* In the last step, the attacker probes the same PHT entry and determines the victim's branch outcome (i.e., the secret) by timing the probe or monitoring specific performance counters like the number of branch mispredictions.

BranchScope is an example of a speculative fetch attack since the leakage occurs at the processor frontend through secret dependent predictions made by the PHT. While many systems are still vulnerable to BranchScope, the original attack is limited since it needs to control the OS to extract secret keys with more than one bit by interrupting the victim gadget at each secret dependent branch execution. In addition, BranchScope requires that the attacker primes and uses simple 1-level predictors of the PHT to have a successful and high-resolution attack. However, in this work, we show that **state-of-the-art branch predictors can remember decision patterns of conditional branches with high accuracy**, which enables an *unprivileged* attacker to reconstruct the secret data with a high resolution without interrupting the victim execution.

3 CONJURING ATTACK

We present Conjuring, a new variant of speculative fetch attacks that, unlike previous work [11], allows an *unprivileged* attacker to successfully launch an attack, *without requiring the OS to precisely control the victim execution* and *without requiring the attacker to prime the branch predictor*. We exploit the fact that state-of-the-art branch predictors can remember branch patterns and therefore can leak the confidential information that has already been learned. Our attack performs only two steps without interfering with the victim's execution:

Step 1 **Training the PHT with the secret (*victim*).** The victim runs normally and trains the PHT with its secret information and branch decisions;

Step 2 **Measuring the execution time of the attack gadget (*attacker*).** After victim execution, the attacker needs to (a) create a PHT collision with the victim branch [5, 19] and (b) take a guess about the secret. The attacker evaluates the correctness of the guess to extract the secret.

Figure 2 shows an example of victim and attack gadgets, along with a timeline of the events during the second step of the attack.

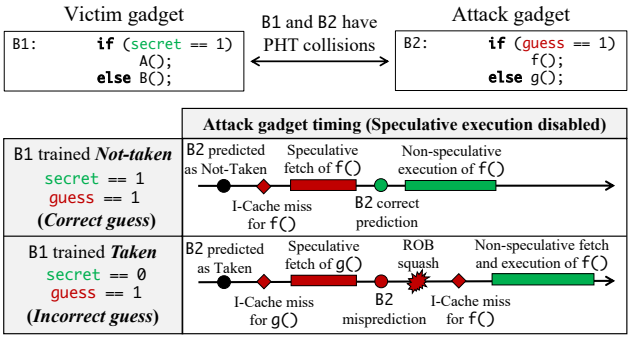

Figure 2: Events and timing of correct prediction (correct guess) and misprediction (incorrect guess) in Conjuring attack. We assume that before executing the attack gadget all instructions and data of the attacker are flushed from the caches. Note that, Not-Taken means executing the fall-through path (A() and f() in this example).

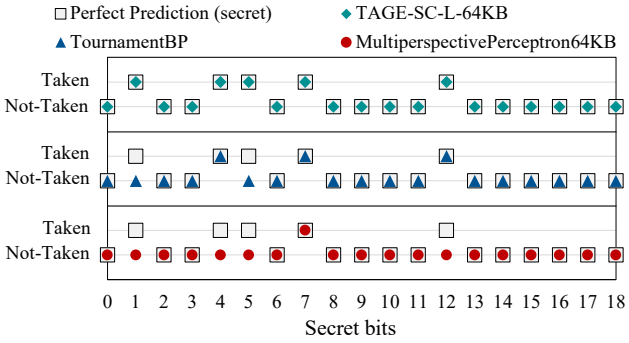

Figure 3: Investigation of different branch predictors in gem5 and their prediction for RSA private key after 1,000 rounds of decryption. The results show that TAGE-SC-L-64KB remembers the entire key. Note, that a Taken decision means that the secret bit is 0 based on the RSA code in Listing 1. The secret is $(1011001011110111111)_2$.

If the attacker guesses correctly, the instructions are fetched from the correct path and are not squashed later. However, in case of an incorrect guess, the incorrectly fetched instructions are squashed and the processor starts fetching and executing instructions from the correct path (speculative execution is disabled in Figure 2). As shown in the figure, the attacker can evaluate the correctness of the guess by measuring the execution time of the attack gadget.

This attack exploits the victim itself to prime and train the PHT state with the secret. Our investigations using gem5 simulation [2] with different branch predictors (Figure 3) show that TAGE-SC-L-64KB [6] is able to learn all 19 bits of the private exponent in a Montgomery Ladder RSA [18] (Listing 1) after 1,000 rounds of decryption. Older branch predictors fail to remember all the bits. For example, MultiperspectivePerceptron64KB [16, 17] provides incorrect predictions for 4 of the secret bits. Furthermore, Table 1 shows the accuracy of branch predictors in extracting recommended RSA key sizes [33] after 10,000 rounds of training. Prior work demonstrates that only recovering 60% of the private RSA key is sufficient

Table 1: Percentage of full RSA key recovery with different key sizes in different branch predictors.

RSA Key Size	Key Recovery Percentage		
	Multiperspective-Perceptron64KB	TournamentBP	TAGE-SC-L-64KB
512	50.39%	89.26%	74.61%
768	45.57%	86.20%	69.27%
1024	52.25%	83.20%	63.77%
1536	51.89%	79.95%	61.59%
2048	49.37%	75.24%	59.33%

for full key recovery by exploiting redundancies in the storage of RSA public keys [12, 15]. The results show that TournamentBP and TAGE-SC-L-64KB already show an acceptable accuracy for full key recovery. Deploying increasingly better branch predictors make speculative fetch attacks even more serious than was previously realized. In Section 4, we demonstrate the feasibility of our attack on real existing CPUs.

3.1 Threat Model

The attack assumptions are: (1) the leaking channel (e.g., the PHT) must be shared between the attacker and the victim. This means that the attacker and the victim run on the same physical core, (2) the attacker can trigger or is able to watch for and validate the victim code execution. (3) Our attack works for different privilege levels, including an unprivileged attacker.

3.2 Attack Surface

Any program with confidential control flow is vulnerable to speculative fetch attacks. Many existing cryptographic implementations have secret dependent branches, like the RSA modular exponentiation in LibreSSL v3.3.6 [30][1] and MbedTLS v3.5.0 [24]. Moreover, many real-world applications have branches that can leak confidential information (e.g., users' Bluetooth connections [3] and battery properties [1] in the Linux kernel).

4 PROOF-OF-CONCEPT ATTACKS

To demonstrate the feasibility of the attack on real existing CPUs, we run our Proof-of-Concept (PoC) attack on three different CPUs: (1) Intel, (2) AMD, and (3) Apple. In our PoC, shown in Listing 2, we use the same function as the victim gadget and the attack gadget (see `func()`). This choice ensures a perfect PHT branch collision. Prior work have demonstrated different techniques to create PHT collisions [5, 8, 19, 40]. In `main` function, we assume the secret is 1 and train the PHT with this secret by running the victim 1,000 times. Afterwards, we call the same gadget with our guess for the secret. We try 0 and 1 as our guesses multiple times to test if we can observe timing differences between correct guesses (i.e., `GUESS` defined as 1) and incorrect guesses (i.e., `GUESS` defined as 0).

Experimental Setup. Table 2 shows the system configurations that we used for our PoC attack. For the Intel and AMD CPUs, we use the native `_rdtsc()` function as `current_cycle_count()` in Listing 2. However, Apple CPUs do not have such a native function

[1]We have tested and reported this attack surface to Apple as part of our disclosure (the default OpenSSL library installed on macOS Ventura 13.0).

```
#define GUESS 1

void func(int input){
    if (input){
        asm volatile(
        ".rept 10000"
        "nop\n"
        ".endr"
        );
    }
    else{
        asm volatile(
        ".rept 10000"
        "nop\n"
        ".endr"
        );
    }
}

void main() {
    for(int i = 0; i < 1000; i++)
        func(1); //victim call (input is the secret, i.e., 1)
    int cycles = current_cycle_count();
    func(GUESS); // attacker call (input is the guess;
                 // different guesses in differet trials)
    printf("%d\n", current_cycle_count()-cycles);
}
```

Listing 2: CONJURING Proof-of-Concept.

and we implemented `current_cycle_count()` by incrementing a shared variable with the `OSAtomicIncrement64()` function from the `libkern/OSAtomic.h`. The attacker starts a second `pthread` that loops and continuously increments this shared variable. The main thread can estimate the passage of time by reading the value from the shared variable.

Figure 4 shows the results for different CPUs. We illustrate the execution time of the attack gadget for 20 trials for both cases that the attacker has a correct guess (correct prediction from the PHT) and incorrect guess (branch misprediction). For the Intel CPU (see Figure 4(a)), the execution time of the attack gadget, in the case of an incorrect guess is on average 2.29× longer compared to the execution time when the attacker has a correct guess. In addition, we observe that the timing difference between an incorrect guess and a correct guess is at least 1657 cycles (denoted as `MISPRED_THREAHOLD`) in 20 trials of the attack.

Figure 4(b) shows the execution time of the attack gadget for 20 trials for the AMD CPU. This figure indicates system noise for the first and the seventh trials which are potentially a result of ICache misses of the instructions. Figure 4(c) shows a zoomed-in version of results in order to precisely observe the timing difference of correct prediction vs. branch misprediction. As seen in Figure 4(c), the execution of the attack gadget has more fluctuations in case of the misprediction which can be a result of ROB squashes and fetching new instructions for each trial. We observe that the minimum execution time of mispredictions is always higher than the execution of the correct predictions. We show the `MISPRED_THREAHOLD` in Figure 4(c) which is 24 cycles. On average, the execution time of incorrect guesses is 1.18× longer compared to when the attacker has a correct guess.

Figure 4(d) shows the attack trials for the Apple CPU. Similar to the AMD CPU, we observe system noise in three trials. To better observe the timing differences, Figure 4(e) illustrates a zoomed-in

Table 2: Experimental setup for CONJURING PoC attack on three different CPUs.

	Intel	AMD	Apple
Model Name	Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz	AMD EPYC 7532 32-Core Processor	MacBook Pro (Apple M1 Pro chip)
CPU Cores	4	32	10 (8 performance and 2 efficiency)
CPU MHz	849.410	1499.479	2064 - 3220
Cache Size	8192KB	512KB	L1: 128KB (perf.) 64KB (eff.), L2: 4MB
Operating System	Ubuntu 18.04	Ubuntu 20.04	macOS Ventura 13.0
Compiler	gcc (Ubuntu 7.5.0-3ubuntu1 18.04) 7.5.0, -O0	GCC 9.4.0, -O0	Apple clang version 14.0.0, -O0

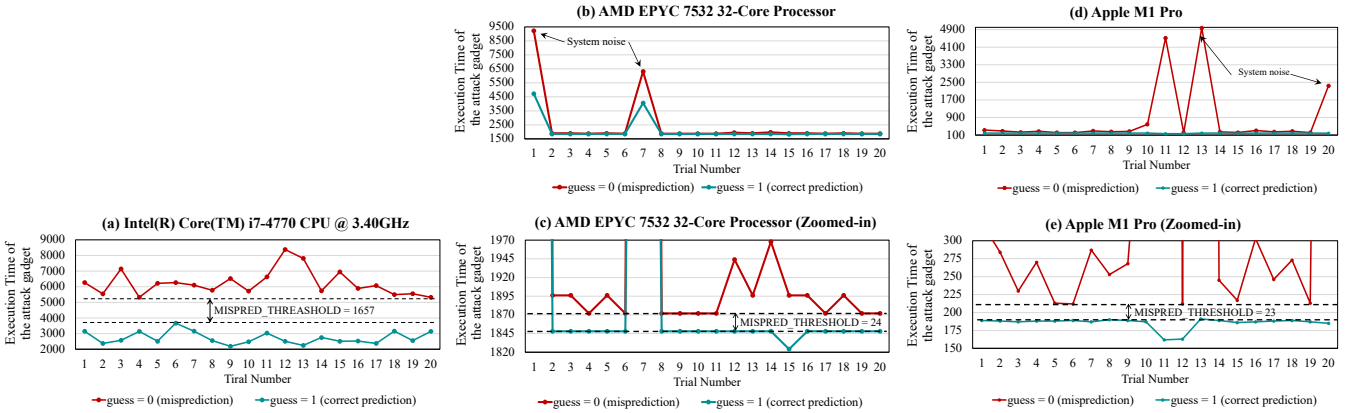

Figure 4: PoC attacks results: (a) Intel CPU that the `MISPRED_THREADHOLD` is 1657 cycles; (b,c) AMD CPU which (c) is the zoomed-in version of (b). `MISPRED_THREADHOLD` is 24 cycles for AMD; (d,e) Apple CPU which (e) is the zoomed-in version of (d). `MISPRED_THREADHOLD` is 23 cycles for Apple.

Table 3: Summary of CONJURING PoC attacks. Average ratio indicates the significance of average latency for incorrect guesses compared to correct guesses.

CPU	Average Timing for Correct Guess	Average Timing for Incorrect Guess	Average Ratio	Min. Difference (`MISPRED_THREADHOLD`)
Intel	2726.65	6233.10	2.29×	1657
AMD	2101.20	2478.00	1.18×	24
Apple	185.45	823.95	4.44×	23

version of the results. The results confirm that incorrect guesses always experience higher latency with more fluctuations (as a result of squashing misspeculated instructions and fetching instruction from the correct path). The `MISPRED_THREAHOLD` is 23 cycles for the Apple CPU and incorrect guesses show 4.44× higher latency on average compared to correct guesses.

Table 3 shows a summary of the PoC attacks and their statistics. Our results demonstrate that Intel significantly shows higher visibility for the CONJURING attack. This can be attributed to more complicated frontends and different paths for instruction delivery in Intel x86 CPUs, demonstrated in prior works [7, 9, 44].

5 MITIGATING SPECULATIVE FETCH ATTACKS

In this section, we investigate existing mitigations for CONJURING and propose a new comprehensive and efficient defense for future CPUs. Our mitigation uses taint tracking to mark secret dependent control flow instructions (i.e., vulnerable to CONJURING leakage and speculative fetch attacks) and inform the CPU to avoid speculative operations at the frontend to prevent unintended encoding of the secrets in frontend structures.

Data-oblivious programming. Comprehensive software-only defenses adopt strict policies like data-oblivious programming [23, 25, 29, 32] (e.g., removing secret dependent branches). Adapting to such guidelines is challenging for real-world and control flow intensive programs. Even many cryptographic modules still use secret dependent branches (see the examples in Section 3.2) because of (1) the engineering efforts to port the legacy codes to data-oblivious principles and (2) significant performance overheads [41]. Instead, these applications secure non-speculative leaks of the program with lightweight spot defenses based on the adversaries' capabilities[2]; however, they are still vulnerable to CONJURING.

While some crypto libraries provide data-oblivious (also known as constant-time) implementations of the algorithms, the unsafe gadgets still exist in the libraries and it is the users' responsibility to choose safe implementations. For example, Listing 3 shows the usage of modular exponentiation for RSA decryption in LibreSSL v3.3.6 [30]. In line 3, the user needs to specify to enable the constant-time implementation, otherwise the default implementation is vulnerable to CONJURING. In addition, existence of unsafe attack surfaces brings many opportunities for attackers to

[2]For example, a minor compiler update mitigates the fetch width misalignments exploited by Frontal attacks [28].

```
BIGNUM *d = BN_new();
BN_hex2bn(&d, "0x5db...a10");//private exponent
BN_set_flags(d, BN_FLG_CONSTTIME);//for constant-time decryption
BN_mod_exp_mont(decrypted_message,encrypted_message,
                d,n,ctx,mont_ctx);
```

Listing 3: Usage of RSA modular exponentiation in LibreSSL v3.3.6 [30].

exploit, especially with emerging tools using Large Language Models (LLMs) to assist programming. For example, Schuster et al. [31] demonstrate poisoning attacks to force code-autocompletion features of IDEs to suggest unsafe flags and versions of crypto modules.

Secure branch prediction designs. A line of research aims to design secure Branch Prediction Units (BPUs) to prevent the attacks originating from branch prediction (e.g., some variants of Spectre [20] and speculative fetch attacks [11]). These defenses implement flushing [10, 22, 34], partitioning [35, 40], randomizing [13, 21, 43, 45], or a combination of these techniques [46]. Most of these implementations incur high performance and area overheads and offer limited protection. Moreover, they share two main limitations: (1) they fully trust the OS to apply appropriate mitigations at context switches; i.e., not covering malicious OS (like Intel SGX), and hyperthreading in most cases. (2) These defenses secure specific structures in the BPU since their goal is to prevent a known set of attacks. However, speculative fetch attacks, including Conjuring, can potentially exploit any structure in the frontend containing branch prediction information. For example, BunnyHop [44] demonstrates that the instruction prefetcher in Intel processors preserve branch prediction information. Securing all individual components in processors' frontend that store branch prediction information, explicitly or implicitly, seems challenging with unknown efficiency impacts, given that many processors have complicated frontends with undisclosed structures and behaviors [7, 9].

5.1 Mitigating Conjuring in Future CPUs

As we explained earlier in this section, it is unlikely to migrate all crypto implementations to *efficient* data-oblivious implementations in near future and completely remove *all* unsafe attack surfaces. To efficiently and comprehensively block speculative fetch attacks through all potential leaking channels in the frontend and cover strong adversaries (including malicious OSes), we propose a selective fetch restriction strategy, called as *selFetch-Restriction*, based on taint tracking. State-of-the-art taint tracking tools can detect all instructions of crypto modules that are tainted by sensitive information (i.e., secrets). Hence, a hardware/software co-operative solution can mark the tainted (secret dependent) control flow instructions and inform the hardware about the sensitivity of the instructions. In case of fetching a tainted control flow instruction, the hardware will restrict accessing frontend structures (like BPU and prefetchers) and waits until the control flow condition is resolved before redirecting the fetch. Since frontend stalls only occur for secret dependent branches in a fine-grained way, performance overheads are minimal.

Experimental Results. Figure 5 shows normalized execution time of three designs: (2) *Insecure Baseline* which is the baseline OoO processor with no protection, (2) *selFetch-Restriction* which is

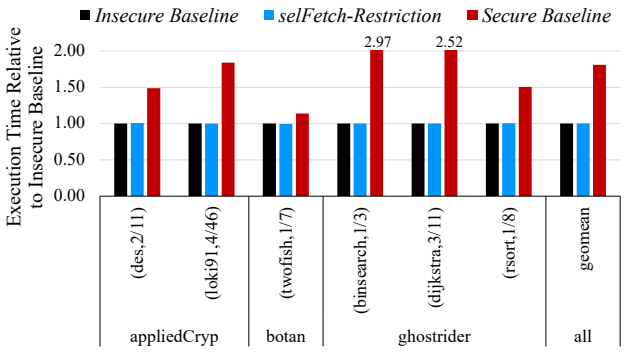

Figure 5: Execution of *selFetch-Restriction* and *Secure Baseline* mitigation strategies normalized to *Insecure Baseline*. The x-axis shows the number of tainted (i.e., secret dependent) and total branches for each application (*tainted/total*).

our proposed mitigation, and (3) a *Secure Baseline* that completely disables the BPU to comprehensively block speculative fetch attacks (the same protection scope as *selFetch-Restriction*). We evaluate a set of realistic crypto modules and standard microbenchmarks for constant-time enforcements [4, 23, 29, 38]. We integrate a taint tracking mechanism into the LLVM compiler to mark secret dependent branches [4]. Note, that we only consider the applications with tainted branches, and each application in Figure 5 is denoted with the number of tainted branches and total branches (*tainted*/*total*). In addition, we implement *selFetch-Restriction* mechanism in the gem5 simulator [2] with a Golden-Cove like microarchitecture configuration [27] (e.g., ROB size of 512 entries) with TAGE-SC-L-64KB branch predictor.

The results in Figure 5 show almost zero performance overhead for *selFetch-Restriction* (0.03% overhead over the *Insecure Baseline*). However, the *Secure Baseline* incurs significant overhead of 80.98% which can go up to 2.97× compared to the *Insecure Baseline*.

6 RELATED WORK

Speculative attacks on the frontend. As we discussed in Section 2, BranchScope [11] is a prior example of speculative fetch attacks. The main difference of Conjuring and BranchScope is that Conjuring does not require (1) priming the PHT prior victim execution and it uses the victim itself to train and encode the secrets, and (2) it does not require OS control privileges to interfere with the victim execution, which enables a realistic and practical attack.

Phantom [37] is another attack that exploits the branch mispredictions detected at the decode stage. This attack shows that Intel and AMD CPUs can resteer the fetch direction at the frontend and decode stage, earlier than classic Spectre attacks [20] that exploit the backend resteers. However, unlike Conjuring, Phantom attacks still need to execute memory loads in some of their exploits and also targets the same attack surface of Spectre attacks. Moreover, Phantom relies on another channel, like Flush+Reload [39] via caches, to transmit secrets. However, in Conjuring attack, the frontend itself serves as a transmission channel.

In addition, some attacks use the PHT as a transmission channel like BranchSpectre [8]. BranchSpectre targets Spectre gadgets and primitives to leak arbitrary data, and moreover, it requires a nested

branch inside mispredicted branch in order to encode the data inside the PHT. In other words, BranchSpectre still exploits the backend speculations, similar to Spectre, and only uses the PHT as a transmission channel. Weisse et al. [36] launch the same attack while using the BTB as a transmission channel. All these attacks will be blocked with mitigations for speculative execution attacks [14, 26, 36, 42]. However, Conjuring and speculative fetch attacks are still effective even if Spectre defenses are deployed.

Non-speculative attacks on the frontend. The Branch Shadowing attack [22] uses the BTB to leak the target address of a secret dependent branch. Frontal Attacks [28] exploit different fetch width alignments of secret dependent branches to distinguish the secrets. Frontend vulnerabilities revealed by Leaky Frontends [9] also exploit the fact the instructions take different paths in the frontend and exploit this difference for side channel and covert channel attacks. All these frontend based attacks use the frontend structures as a channel to leak information, however, they do not exploit speculative operations in the frontend, and unlike Conjuring, rely on the sequential (i.e., non-speculative) leaks of the programs.

7 CONCLUSION

In this work, we propose Conjuring as a new variant of speculative fetch attacks; Conjuring can realistically leak control flow information of the victim without requiring OS control privileges and priming the PHT. Conjuring leverages the victim itself to train the PHT and encode its sensitive control flow decisions. Later, we exploit the frontend misspeculations to distinguish correct and incorrect guesses and successfully extract the victim's control flow. We demonstrate that better branch predictors are more vulnerable to Conjuring since they are capable of remembering longer histories of branch decisions. We show that existing branch predictors can remember an acceptable percentage of real-world RSA private keys, leading to full key extraction. In addition, we demonstrate the feasibility of Conjuring on three different real CPUs: Intel, AMD, and Apple. Finally, we propose an efficient defense to block the Conjuring leaks for the vulnerable attack surfaces.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback. In addition, we thank Archit Agarwal, Andreas Diavastos, Lingfeng Pei, Arash Pashrashid, Shweta Shinde, and Prateek Saxena for their valuable comments and discussions that helped us to improve the work.

REFERENCES

[1] Battery properties management 2021. https://git.kernel.org/pub/scm/bluetooth/bluetooth-next.git/tree/drivers/hid/hid-input.c#n247.
[2] Nathan Binkert, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* (2011).
[3] Bluetooth connection activity management 2021. https://git.kernel.org/pub/scm/bluetooth/bluetooth-next.git/tree/drivers/hid/hid-input.c#n383.
[4] Pietro Borrello, et al. 2021. Constantine: Automatic side-channel resistance using efficient control and data flow linearization. In *CCS 2021*.
[5] Claudio Canella, et al. 2019. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security 2019*.
[6] CBP-5 Kit 2016. In *Proceedings 5th Championship on Branch Prediction*.
[7] Yun Chen, et al. 2024. GadgetSpinner: A new transient execution primitive using the Loop Stream Detector. In *HPCA 2024*.
[8] Md Hafizul Islam Chowdhuryy et al. 2021. Leaking secrets through modern branch predictor in the speculative world. *IEEE Trans. Comput.* (2021).
[9] Shuwen Deng, et al. 2022. Leaky Frontends: Security vulnerabilities in processor frontends. In *HPCA 2022*.
[10] Dmitry Evtyushkin, et al. 2016. Understanding and mitigating covert channels through branch predictors. *ACM TACO* (2016).
[11] Dmitry Evtyushkin, et al. 2018. BranchScope: A new side-channel attack on directional branch predictor. In *ASPLOS 2018*.
[12] Ben Gras, et al. 2018. Translation Leak-aside Buffer: Defeating cache side-channel protections with TLB Attacks. In *USENIX Security 2018*.
[13] Brian Grayson, et al. 2020. Evolution of the Samsung Exynos CPU microarchitecture. In *ISCA 2020*.
[14] Ali Hajiabadi, et al. 2024. Levioso: Efficient compiler-informed secure speculation. In *DAC 2024*.
[15] Nadia Heninger et al. 2009. Reconstructing RSA private keys from random key bits. In *Crypto 2009*.
[16] Daniel A Jiménez. 2003. Fast path-based neural branch prediction. In *MICRO 2003*.
[17] Daniel A Jiménez et al. 2001. Dynamic branch prediction with perceptrons. In *HPCA 2001*.
[18] Marc Joye et al. 2002. The Montgomery powering ladder. In *CHES 2002*.
[19] Ofek Kirzner et al. 2021. An analysis of speculative type confusion vulnerabilities in the wild. In *USENIX Security 2021*.
[20] Paul Kocher, et al. 2019. Spectre Attacks: Exploiting speculative execution. In *S&P 2019*.
[21] Jaekyu Lee, et al. 2020. Securing branch predictors with two-level encryption. *ACM TACO* (2020).
[22] Sangho Lee, et al. 2017. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security 2017*.
[23] Chang Liu, et al. 2015. GhostRider: A hardware-software system for memory trace oblivious computation. In *ASPLOS 2015*.
[24] Arm Mbed. 2019. MbedTLS: An open source, portable, easy to use, readable and flexible SSL library. *ARM Holdings plc* (2019).
[25] David Molnar, et al. 2006. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *ICISC 2005*.
[26] Arash Pashrashid, et al. 2023. HidFix: Efficient mitigation of cache-based Spectre attacks through hidden rollbacks. In *ICCAD 2023*.
[27] Popping the hood on Golden Cove 2021. https://chipsandcheese.com/2021/12/02/popping-the-hood-on-golden-cove/.
[28] Ivan Puddu, et al. 2021. Frontal Attack: Leaking control-flow in SGX via the CPU frontend. In *USENIX Security 2021*.
[29] Ashay Rane, et al. 2015. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security 2015*.
[30] RSA LibreSSL 3.3.6 2023. https://github.com/PowerShell/LibreSSL/blob/edfc7eb366aa242e8966332e6d98a16cd4264d6c/crypto/bn/bn_exp.c#L481.
[31] Roei Schuster, et al. 2021. You autocomplete me: Poisoning vulnerabilities in neural code completion. In *USENIX Security 2021*.
[32] Martin Schwarzl, et al. 2021. Specfuscator: Evaluating branch removal as a Spectre mitigation. In *FC 2021*.
[33] Nigel Smart, et al. 2012. ECRYPT II yearly report on algorithms and keysizes (2011–2012). *European network of excellence in cryptology (ECRYPT II)* (2012).
[34] Mohammadkazem Taram, et al. 2019. Context-sensitive fencing: Securing speculative execution via microcode customization. In *ASPLOS 2019*.
[35] Ilias Vougioukas, et al. 2019. BRB: Mitigating branch predictor side-channels. In *HPCA 2019*.
[36] Ofir Weisse, et al. 2019. NDA: Preventing speculative execution attacks at their source. In *MICRO 2019*.
[37] Johannes Wikner, et al. 2023. Phantom: Exploiting decoder-detectable mispredictions. In *MICRO 2023*.
[38] Meng Wu, et al. 2018. Eliminating timing side-channel leaks using program repair. In *ISSTA 2018*.
[39] Yuval Yarom et al. 2014. Flush+Reload: A high resolution, low noise, L3 cache Side-Channel attack. In *USENIX Security 2014*.
[40] Hosein Yavarzadeh, et al. 2023. Half&Half: Demystifying Intel's directional branch predictors for fast, secure partitioned execution. In *S&P 2023*.
[41] Jiyong Yu, et al. 2019. Data oblivious ISA extensions for side channel-resistant and high performance computing. In *NDSS 2019*.
[42] Jiyong Yu, et al. 2019. Speculative Taint Tracking (STT): A comprehensive protection for speculatively accessed data. In *MICRO 2019*.
[43] Tao Zhang, et al. 2022. STBPU: A reasonably secure branch prediction unit. In *DSN 2022*.
[44] Zhiyuan Zhang, et al. 2023. BunnyHop: Exploiting the instruction prefetcher. In *USENIX Security 2023*.
[45] Lutan Zhao, et al. 2021. A lightweight isolation mechanism for secure branch predictors. In *DAC 2021*.
[46] Lutan Zhao, et al. 2022. HyBP: Hybrid isolation-randomization secure branch predictor. In *HPCA 2022*.