All Your PC Are Belong to Us: Exploiting Non-control-Transfer Instruction BTB Updates for Dynamic PC Extraction

Jiyong Yu
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
jiyongy2@illinois.edu

Trent Jaeger
Pennsylvania State University
University Park, Pennsylvania, USA
trj1@psu.edu

Christopher W. Fletcher
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
cwfletch@illinois.edu

ABSTRACT

Leaking a program's instruction address (PC) pattern, completely and precisely, has long been a sought-after capability for micro-architectural side-channel attackers. Case in point, such a primitive would be sufficient to construct powerful control-flow leakage attacks (inferring program secrets impacting control flow) that defeat existing control-flow leakage mitigations, or even reverse-engineer private binaries through PC-trace granular fingerprinting. However, current side-channel attack techniques only capture PCs at a coarse granularity or for only specific instruction types.

In this paper, we propose the first micro-architectural side-channel attack that is capable of *directly* observing the exact PCs of arbitrary victim dynamic instructions—i.e., even the PCs of non-control-transfer instructions and even if the program code is private. Our attack exploits several previously overlooked characteristics in modern Intel Branch Target Buffers (BTBs). The core observation is perhaps counter-intuitive: despite being a structure related to control-flow prediction, the BTB incurs observable state changes after the execution of potentially *any* instruction, not just control-transfer instructions.

Through reverse-engineering and analyzing said BTB vulnerabilities, we design and implement an attack framework named NightVision. We demonstrate how NightVision is capable of efficiently and accurately identifying a subset, or the entirety, of a victim program's dynamic PC trace (depending on the attacker's capabilities). We show how NightVision enables a new control-flow attack that bypasses prior defenses. Additionally, we show that when combined with code fingerprinting techniques, NightVision enables reverse-engineering of private programs.

CCS CONCEPTS

• **Computer systems organization → Architectures**; • **Security and privacy → Hardware attacks and countermeasures**.

KEYWORDS

Side-channel attack, hardware security, Branch Target Buffer, code privacy, function fingerprinting, Intel SGX

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISCA '23, June 17–21, 2023, Orlando, FL, USA.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0095-8/23/06…$15.00
https://doi.org/10.1145/3579371.3589100

ACM Reference Format:
Jiyong Yu, Trent Jaeger, and Christopher W. Fletcher. 2023. All Your PC Are Belong to Us: Exploiting Non-control-Transfer Instruction BTB Updates for Dynamic PC Extraction. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3579371.3589100

1 INTRODUCTION

Micro-architectural side-channel attacks have emerged as a critical security threat. By co-locating to the same processor, these attacks enable attackers to deduce program characteristics (e.g., control-flow decisions, the set of addresses touched in data memory) by observing the micro-architectural effects stemming from a victim program's execution (e.g., through the cache, branch predictors and more [4, 5, 7, 18, 26, 44, 58, 65]).

One such program characteristic of fundamental interest to side-channel attackers is the victim's *dynamic PC trace*. That is, the sequence of PCs corresponding to the victim's dynamic instructions over the course of its execution. If attackers are able to *directly* learn the victim program's PC trace, they would be able to beat state-of-the-art defenses for current attacks, and even effectuate new types of attacks. For example, consider control-flow leakage attacks whereby an attacker tries to learn secret-dependent branch decisions in otherwise public programs (e.g., RSA) [18, 26, 39, 42, 46, 55]. There has been significant work to mitigate these attacks, e.g., through branch balancing [42, 46] and control-flow randomization [25], while keeping the secret-dependent control flow intact for performance reasons (i.e., not converting to data-oblivious code [11, 67]). Yet, all of these defenses immediately fail if the attacker is able to directly learn the victim program's PC trace. Going beyond control-flow leakage attacks, knowing the victim program's PC trace enables new attacks on *private programs*. For example, extracting a significant portion of the victim's dynamic PC trace enables code fingerprinting on otherwise private code.

Yet, no existing side channel can precisely and directly extract any given instruction's PC, or for that matter the program's PC trace. For example, controlled-channel attacks and their variants [23, 56, 60, 64] learn the PC trace at a page or cache-level granularity. This is too coarse to be useful against basic defenses, which impose simple restrictions to confine control flow within the minimal observation granularity (e.g., a single cache line) [50]. Several other attacks are capable of extracting PCs of particular instruction types (e.g., loads/stores [14, 19], jumps [17, 39]) or for specific code patterns (e.g., Frontal [46]). But this is not sufficiently general to infer arbitrary secret-dependent control flow, let alone recover the entire PC trace for attacks such as code fingerprinting.

This Work. We demonstrate the first micro-architectural side-channel attack capable of leaking the byte-granular PC of any victim dynamic instruction. Our attack is enabled by new findings related to how Branch Target Buffers (BTBs) are implemented in modern Intel processors. In a nutshell: It's well known that *control-transfer instructions* update the BTB, as a function of their PC and predicted target. Indeed, this behavior underpins current attacks [17, 35, 39, 69]. Our key observation is that modern BTBs, i.e., implemented with pipelined superscalar cores in mind, *are also* updated based on the execution of *non-control-transfer instructions*. We show how this enables the BTB to leak the exact PCs of even non-control-transfer instructions, e.g., of the instructions in the shadow of a branch or even the instructions in straight-line code.

The above might seem counter-intuitive. The BTB only maps branch/jump PCs to predicted target PCs, which is seemingly irrelevant to non-control-transfer instructions. However, in modern pipelined processors, the BTB is accessed without knowledge of the fetched instruction except for its PC. Additionally, modern BTB tag checks usually leave out the highest-order bits to save on area [39, 69]. Hence, it is possible for a BTB tag check to 'succeed' but to provide a prediction corresponding to a different instruction, even when the instruction corresponds to a non-control-transfer instruction.

Such *false hits* overwhelmingly result in pipeline squashes. To counteract this potential performance loss, we find that modern Intel processors *deallocate* the involved BTB entry upon detecting a false hit—as soon as instruction decoding finishes and even if the instruction causing the false hit doesn't retire. Since BTB entries correspond to branches, a BTB hit on a PC that decodes to a non-control-transfer instruction is necessarily a false hit. Thus, *non-control-transfer instructions that alias with existing entries in the BTB create observable BTB state changes.*

We find that the above effects are further amplified by how the BTB is implemented to handle superscalar fetch. Since modern superscalar processors fetch instructions in bundles, BTB lookups in fetch have range query-like semantics. That is, the BTB compares the fetch PC with all BTB entries whose PCs are greater than *but nearby* the fetch PC. Effectively, this means that BTB deallocations can be an indicator of not only whether a fetched instruction matches a certain PC value, but also whether it falls within a BTB entry-defined address range.

Based on the above, we propose an attack framework called NightVision. The core of NightVision is a novel BTB Prime+Probe-like primitive which captures updates made by a co-located victim's execution, using attacker-allocated BTB entries. Using the false hit-induced deallocation effect, we show how NightVision can determine the PCs of (in the best case) individual victim dynamic instructions. By further exploiting BTB range query semantics, we show how the attacker can efficiently binary search through larger address ranges to recover said PCs.

We demonstrate how to use NightVision to attack victim programs in user-/supervisor-level attacker settings, where we construct control-flow leakage attacks for leaking secret data influencing program control flow. Our attacks circumvent defenses that mitigate prior control-flow leakage attacks [17, 25, 39, 42, 46]. When equipped with supervisor-level capabilities, such as managing system resources like virtual memory and interrupts, we showcase how NightVision can deduce the exact PC of every victim dynamic instruction. With the extracted PC trace, we show how NightVision can be used to reverse-engineer private programs with function fingerprinting techniques, in settings where TEEs (e.g., we evaluate on SGX) would otherwise provide code confidentiality protection.

We evaluate NightVision using two existing cryptographic functions with secret-dependent control flow: big number comparison in Intel's IPP Cryptographic library [31] and the Greatest Common Divisor in MbedTLS [2], both evaluated by a recent work [46]. We show that several prior software defense mechanisms, like branch balancing [42], basic block alignment [46] and control-flow randomization [25], and hardware mitigations (IBRS/IBPB) are ineffective at mitigating our attack. We further demonstrate how NightVision with supervisor privileges manages to identify both of the above functions from a corpus of 175K other functions by applying its function fingerprinting on the recovered dynamic PC traces, when code privacy is enforced by Intel SGX.

To summarize, we make the following contributions:

- We show how modern BTB design enables learning the PCs of arbitrary victim dynamic instructions.
- We implement an attack framework, named NightVision, and demonstrate how NightVision can observe select victim instructions' PCs, or the victim's entire dynamic PC trace, depending on the attacker's capability.
- We show how variants of NightVision can be used to leak secret data influencing program control flows even with prior software/hardware mitigations enabled, and reverse-engineer private binaries using function fingerprinting.

We responsibly disclosed to Intel who acknowledged our findings.

2 BTB MECHANISM

2.1 Existing BTB Reverse-Engineering Takeaways

Branch prediction is an essential performance optimization in modern processors to reduce the delay caused by control-transfer instructions. The core of branch prediction is a Branch Target Buffer (BTB), which is a lookup table mapping branch/jump addresses (PCs) to their target PCs [24]. Since the BTB is shared by all software running on the same core, prior work has attempted to reverse-engineer the BTB and use it to mount side-channel attacks [17, 35, 39, 69]. Next, we highlight the BTB mechanism disclosed by prior work.

In a nutshell, the BTB mechanism comprises two key actions: *access* and *update*. At the instruction fetch stage, the BTB is *accessed* to select an entry for predicting the target of the fetched branch/jump, which becomes the next instruction to fetch. If no BTB entry is selected or the selected BTB entry contains an incorrect target, the BTB is later *updated* with the correct target of the branch/jump, e.g., by *allocating* a new BTB entry or *replacing* an existing entry.

The BTB's organization on modern processors resembles a set-associative cache. Every BTB access uses tag, set index, and offset fields computed from the current instruction pointer. The set index field determines the BTB set, and the tag and the offset are used for selecting the matched BTB entry in the set. The offset field is usually

5 bits [33, 69], meaning branches within the same 32-byte-aligned block are mapped to the same set. Unlike the processor cache which uses all higher-order address bits as the tag, BTB tags tend to be truncated to reduce the BTB size[1], since the BTB is a predictor and needs not to guarantee correctness. This causes branches with the same lower-order bits to *collide* on the same BTB entry: the first branch's execution allocates a BTB entry, which is later used to predict for a different instruction with the same lower-order address bits.

Previous BTB side-channel attacks focus on BTB collisions between an attacker branch and a victim branch in two different ways. First, by executing after the victim branch, an attacker branch may access the BTB entry allocated by the victim branch. This enables the attacker to learn, through timing channels, the victim branch decision [17, 18]. Second, the attacker can execute branches first so as to corrupt the BTB and influence the prediction of the victim branch that executes after, thereby facilitating transient execution attacks (e.g., Spectre V2) [12, 35, 69]. While attacks such as Spectre V2 are mitigated with recent hardware defenses (e.g., Intel IBRS/IBPB) by preventing the collisions between indirect jumps, as we will elaborate in §4.1, collisions remain achievable in general and these lead to more fundamental attacks like NightVision.

2.2 Overview of Unexplored BTB Behaviors

Existing BTB side-channel attacks focus on the BTB behavior in response to only branches. This view of the BTB, however, oversimplifies the BTB mechanism and misses critical details about BTBs in modern processors. In this section, we conduct experiments to reveal two previously unexplored BTB behaviors.

Firstly and counter-intuitively, we learn that not only control-transfer instructions, but also non-control-transfer instructions can update the BTB. Since modern processors are deeply pipelined, instructions are unrecognized until they are decoded, which happens several cycles after instruction fetch. When the instruction pointer points to a non-control-transfer instruction, with no information about the instruction except its PC, the BTB must be accessed as usual. However, because the BTB lookup disregards higher-order PC bits, a *false hit* may occur, making a control transfer for the non-control-transfer instruction, eventually causing a pipeline squash after the instruction is decoded. This is where a non-control-transfer instruction can affect the BTB state—we find that the chosen BTB entry gets *deallocated* to prevent misprediction on the next encounter of the same non-control-transfer instruction. We demonstrate this behavior in §2.3.

Our second investigation examines the impact of superscalar pipelines on BTB accesses. Modern Intel CPUs fetch several consecutive instructions within a 32-byte aligned fetch block every cycle [33, 34, 69]. This bundle of fetched instructions, known as a *prediction window* or *PW* [34, 36], consists of either non-control-transfer instructions with a trailing taken branch/jump, or purely non-control-transfer instructions that end at the 32-byte boundary. Since the instructions within the bundle are not known until the decode stage, the BTB access logic must operate at PW granularity rather than instruction granularity. Specifically, the BTB access logic must first predict the location of the next branch/jump after the current PC (end of the current PW), and then the target of that branch/jump (beginning of the next PW). To achieve this, as we will show in §2.4, the BTB access hits an entry if that BTB entry has the same tag and the set index, and *the same or larger offset* compared to the current PC offset. When multiple BTB hits are present, the one with the smallest offset, yet no smaller than the current PC offset is selected.

[1]The actual number of lower-order PC bits used for BTB lookup depends on the processor generation. Based on our reverse engineering, BTBs in Intel SkyLake/KabyLake/CoffeeLake/CascadeLake CPUs ignore address bits 33 and above, while IceLake BTB ignores address bits 34 and above.

```
1  F1: jmp L1; // address range: [F1, F1+1]
2      ...
3  L1: ret;
4      ...
5  < 4/8 GB padding >
6      ...
7  F2: nop; nop; ... nop; // address range: [F2, L2-1]
8  L2: ret;
9
10 /* Experiment1 */
11 for (i = 1 ... 1000) {
12     flushBTB();
13     F1();  // allocate a BTB entry
14     F2();  // may update the allocated BTB entry
15     F1();  // observe the BTB prediction outcome
16 }
```

Figure 1: BTB experiment in §2.3 for showing how non-control-transfer instructions update (deallocate) BTB entries.

2.3 BTB Updates with Non-branches

We use the code in Figure 1 to observe the BTB update made by non-control-transfer instructions. In each loop iteration, we first flush the BTB (line 12) using the BTB cleanup routine from [18]. The code then calls `F1` to allocate a BTB entry representing the jump in `F1` (line 1). The returns on line 3 and 8 are used for returning to the caller and are not the focus of the experiment. `F2` contains a series of `nop`s that may affect the allocated BTB entry state. In the second call to `F1` on line 15, we measure the prediction outcome of `jmp L1` to identify whether a BTB entry update occurs.

Given that the BTB uses the lower 32 (or 33, based on the CPU version) address bits for indexing, we place `F1` and `F2` in different 4 GB (or 8 GB) regions in memory, allowing instructions belonging to `F1` and `F2` to possibly create BTB collisions, i.e. some `nop` in `F2` may have identical lower 32 address bits as `jmp L1`. For simplicity, we only specify the lower-order bits for all addresses and ignore the higher-order bits. To ensure the BTB entry allocated by `jmp L1` is only affected by `nop`s in `F2`, when exploring different `L1` and `L2` values in this experiment, we always maintain `F1` ≤ `L2-2` (note that `jmp L1` is 2-bytes long) hence `jmp L1` can only collide with `nop`s.

Experimental Methodology. We perform the experiment on a series of Intel CPU architectures, including SkyLake (Xeon 8124), KabyLake (Core 7700), CoffeeLake (Core 9700/9900), CascadeLake (Xeon 8252/8259), and IceLake (Xeon 8375). To capture BTB updates, we use Last Branch Record (LBR)[2], a feature available in all modern mainstream Intel processors that logs runtime information of all retired control-transfer instructions, including the branch PC, the

[2]Here, LBR could be replaced with a traditional timestamp counter (`rdtsc`)-based measurement or Intel Process Trace (PT). We opted to use LBR since it is orders-of-magnitude less noisy, as pointed out by [39].

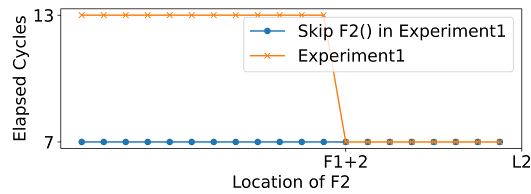

Figure 2: The averaged elapsed cycles between the retire of `jmp L1` (line 1 in Figure 1) and the subsequent return (line 3) as we change the value of F2. The orange line is the reported cycle count by Experiment 1 in Figure 1, whereas the blue line is the cycle count if we remove the call to F2 on line 14. The gap between the two lines when F2 < F1 +2 indicates that when `jmp L1` collides with the nops in F2, its BTB entry is updated (deallocated) thus leading to a misprediction.

predicted direction (only valid for conditional branches), and the elapsed cycles between the retire of the last recorded branch to the retire of the current branch. To measure the prediction outcome of `jmp L1` during the second call of F1 on line 15, we retrieve the elapsed cycles reported for the subsequent return (line 3), which represents the duration between retiring `jmp L1` and the subsequent `ret`. When `jmp L1` is predicted correctly, `jmp L1` and the following `ret` should fetch/execute/commit back-to-back, therefore this duration is expected to be smaller compared to when `jmp L1` is mispredicted.

Result and Takeaway. The yellow line in Figure 2 shows the elapsed cycles between the retire of `jmp L1` and the subsequent `ret` when varying F2, the starting address of the nops. When F2 starts at an address prior to F1+2, i.e. when `jmp L1` collides with a nop in F2, we see a larger cycle count compared to when `jmp L1` does not collide with any nop. For reference, we plot the elapsed cycles in Figure 2 when removing the call to F2 on line 14 in Figure 1 as the blue line. The difference between the two lines illustrates how the execution of F2 influences the BTB entry allocated by `jmp L1`. The same pattern remains when varying F1 and L2, or replacing nops with other non-branches such as adds. Also, the observation is consistent across all tested Intel CPUs. Thus we conclude that:

> **Takeaway 1:** The BTB update (deallocation) can occur when only non-control-transfer instructions collide with the branch recorded by the BTB entry.

2.4 BTB Accesses with PWs

We use Figure 3 to study the BTB access logic of modern superscalar processors. Similar to the previous experiment, we start each test by flushing the BTB (line 14). The code executes two different jumps (`jmp L1` on line 2 and `jmp L2` on line 8) by calling J1 and F2 back-to-back on line 15 and 16. Importantly, we fix the address range of `jmp L1` at [0x1e, 0x1f] and limit the starting address of `jmp L2`, F2, to an arbitrary value within [0, 0x1c] (notice both direct jumps are 2-bytes long, and they share the same set index bits and tag bits, but not offset bits and bits higher than bit 33), so the two jumps do not collide but allocate entries within the same BTB set. Then, on line 17 we jump to a series of nops before `jmp L1`, and

```
1  F1: nop; ... nop; // F1 ∈ [0, 0x1e]; # of nops = 0x1e - F1
2  J1: jmp L1; // address range [0x1e, 0x1f]
3      ...
4  L1: ret;
5      ...
6  < 4 / 8 GB padding >
7      ...
8  F2: jmp L2; // address range [F2, F2+1] (F2 ∈ [0, 0x1c])
9      ...
10 L2: ret;
11
12 /* Experiment2 */
13 for (i = 1 ... 1000) {
14     flushBTB();
15     J1();  // allocate a BTB entry
16     F2();  // allocate another BTB entry
17     F1();  // observe the BTB prediction outcome
18 }
```

Figure 3: The BTB experiment used in §2.4 for studying the BTB access behavior.

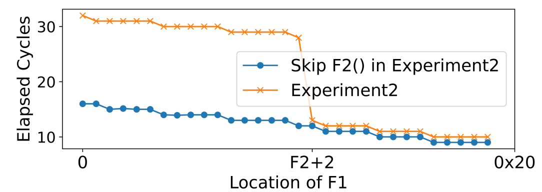

Figure 4: The average elapsed cycles between the retire of the call to F1 (line 17 in Figure 3) and the subsequent return after `jmp L1` (line 4) as we change F1 between [0, 0x1e]. The orange line is the reported cycle count by Experiment 2 in Figure 3, whereas the blue is the reported cycle count when we remove the call to F2 (line 16). The gap between the two lines when F1 < F2 +2 indicates that the BTB entry allocated by `jmp L2` is used when fetching nops at F1, leading to misprediction.

execute a longer PW code (from the first nop at F1 to `jmp L1`). In this experiment, we observe *any misprediction when processing this PW* as we vary F1 and F2 without violating the above constraints.

Experimental Methodology. We test this experiment on the same Intel machines as the previous experiment, and use LBR to measure the prediction outcome. Different from the previous experiment which measures the prediction of a single jump, this experiment requires inferring the prediction decision during the execution of the entire PW (line 1 to 2). Therefore, we use LBR to extract the total elapsed cycles between the call to F1 (line 17) and the return after `jmp L1` (line 4). This cycle count reflects the prediction decision made for any preceding nops as well as `jmp L1`.

Result and Takeaway. As before, we plot two lines in Figure 4. The yellow line plots the elapsed cycles between the retire of line 17 and line 4 as we change the value of F1. The blue line reports the same measurement with the call to F2 (on line 16) skipped, reflecting the execution of the prediction window from F1 to `jmp L1` without any misprediction. The blue line gradually decreases as F1 increases due to fewer executed instructions (nops). By comparing the two lines, we observe that when the PW base address F1 has an offset

larger than that of `jmp L2`, the execution of the PW behaves as if the call to `F2` does not exist. This implies that calling `F2` has no impact on the BTB entry allocated for `jmp L1` during the call to `J1`. In contrast, when the PW starts at an address with an offset equal to or less than `jmp L2`, misprediction occurs, leading to a constant increase in the elapsed cycle, as shown in Figure 4. Given the above observation that the execution of `jmp L2` should not affect the BTB entry allocated for `jmp L1`, the misprediction can only be attributed to the use of the BTB entry allocated for `jmp L2`. Specifically, in this case, when predicting the next fetch target with the current instruction pointer pointing towards `F1`, the BTB entry representing `jmp L2`, rather than the entry representing `jmp L1`, is selected. Similar to the previous experiment, this observation is also consistent among the tested machines. From this result, we can deduce the following takeaway:

Takeaway 2: Due to superscalar fetch, a BTB hit requires an identical tag/set index. The offset of the selected entry must be equal to/greater than the current PC offset. If multiple BTB hits are present, the one with the smallest offset is selected.

Lastly, we note that the above two takeaways are consistent with an earlier Intel Patent [52] that specifies the implementation details of a set-associative BTB structure (Takeaways 1 and 2 are mentioned in columns 20 and 12, respectively).

2.5 Differences from Prior BTB Reverse-Engineering Efforts

Our reverse-engineering approach differs from prior BTB attacks [12, 17, 35, 39, 69] in two respects. First, while prior efforts limit the scope to branches, our insight is that superscalar processors fetch PWs every cycle, therefore the BTB access must exhibit range semantics instead of simply PC matches. Second, while prior efforts mostly focus on how one branch allocates BTB entry which affects the future branch predictions, we additionally investigate how allocated BTB entries can be deallocated, displaying the role of non-control-transfer instructions in the BTB mechanism.

3 ATTACK MODELS AND NIGHTVISION OVERVIEW

In this work, we consider two common attacker models adopted by existing side-channel research:

- **User-level**: the attacker controls one or several user-space processes, which can co-locate with the victim process on the same CPU core via context switches (SMT is not mandatory), hence sharing the same BTB.
- **Supervisor-level**: the attacker has full control over the OS kernel, and can arbitrarily interrupt the victim program's (e.g., an SGX enclave's) execution at a fine temporal granularity, i.e., per instruction, and monitor/manipulate system resources, such as page tables.

We now overview the NightVision attack. NightVision extracts the dynamic PCs visited by the victim program. For either of the above attacker models, the attacker can extract the PCs of both control-transfer and non-control-transfer instructions at byte granularity. That said, depending on the attacker model, the attacker may be able to extract the full trace (i.e., every PC belonging to every dynamic instruction) or only a subset of it (i.e., depending on the temporal granularity at which the victim program is context switched).

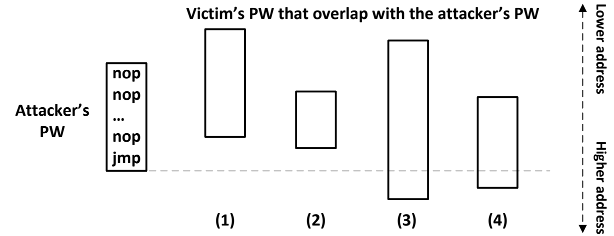

Figure 5: Different ways that the victim's PW overlaps with the attacker's PW. In those cases, the second execution of the attacker's PW in NV-Core incurs observable mispredictions.

The core of NightVision is a BTB Prime+Probe style primitive that is built on top of Takeaways 1 and 2 from §2. This primitive, dubbed NightVision-Core or NV-Core in short, determines if a fragment (i.e., the instructions executed during one context switch) of the victim program's execution contains instruction bytes overlapping with a specified virtual address range. We construct two NightVision variants from NV-Core, depending on the attacker model, namely NightVision-User (or NV-U) for the user-level attacker and NightVision-Supervisor (or NV-S) for the supervisor-level attacker. Both variants repeatedly apply NV-Core to every fragment of the victim's execution. As described above, NV-U is (practically) capable of recovering a subset of elements in the PC trace, whereas NV-S can achieve much finer-grain measurement, ideally recovering the PC of every victim dynamic instruction. §4 explains the design details of NV-Core and both variants.

We later demonstrate two attack applications using the two NightVision variants. §5 will explain how NV-U can serve as a control-flow leakage attack that aims at leaking secret data influencing program control-flow, even when the program is protected against prior control-flow leakage attacks. §6 further highlights how NV-S enables binary fingerprinting to reverse-engineer private enclave binaries, using full PC-trace extraction.

4 NIGHTVISION ATTACK DESIGN

4.1 NightVision-Core (NV-Core)

Inspired by the two takeaways from §2, we first introduce NV-Core, which is a BTB Prime+Probe style primitive to determine if instructions executed by the victim overlap with an attacker-specified prediction window. The attacker starts by creating a PW code snippet, with a sequence of `nop`s followed by a direct jump. Based on the definition of PW, the address range of this code snippet is restricted to within a 32-byte aligned block. The attacker first executes the PW code to allocate a BTB entry, then allows the victim program to run for a certain period of time, and finally executes the same PW code again and measures the predictions during this second execution of the PW code, similar to the technique in §2.4.

Our key insight is that, as the victim's execution also fetches PWs, when the attacker PW address range overlaps with a victim PW's address range, i.e., some instruction bytes from the attacker

```
1   // p is a fragment of victim's execution
2   bool NV-Core(PW, p):
3     Prime BTB with PW
4     execute p
5     match = Probe BTB with PW
6     return match
7
8   // optimized NV-Core, by priming/probing multiple PWs
9   bool[] NV-Core(PWs[], p)
10
11  // P is the victim program
12  bool[][] NV-U(PWs[], P):
13    match = []
14    while (P is not finished) {
15      p = next fragment of P to execute
16      match.append(NV-Core(PWs, p))
17    }
18    return match
19
20  bool[][] NV-S(PWs[], P):
21    match = []
22    while (P is not finished) {
23      i = next instruction of P to execute
24      match.append(NV-Core(PWs, i))
25    }
26    return match
```

Figure 6: The basic workflow of NV-Core, NV-U, and NV-S. NV-U measures a time slice of the victim's execution, whereas NV-S leverages supervisor privilege to measure every dynamic instruction. Both NV-U and NV-S can benefit from optimized NV-Core to monitor multiple PWs at a time.

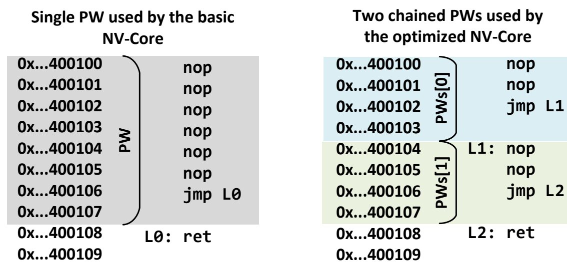

Figure 7: Comparison between a single PW and two chained PWs.

PW share the same lower-order (lowest 32 or 33) address bits as some instruction bytes from the victim PW, the second execution of the attacker's PW will incur an observable misprediction.

To illustrate this point, Figure 5 shows all four scenarios when the attacker PW and the victim PW overlap. In (1) and (2), the victim PW ends at an address aligned with the middle of the attacker's PW range. In these cases, the victim PW must end with a taken branch/jump otherwise the PW will extend to (and be truncated at) the 32-byte boundary. Since the victim PW includes a branch/jump before the attacker's jump, the execution of the attacker's PW must suffer from a misprediction caused by the victim's branch/jump, in the same ways as we showed in §2.4. In the other two cases (3) and (4), the attacker PW ends at an address aligned with the middle of the victim PW range. Since the overlapping part of the victim PW is non-control-transfer instructions, fetching those instructions necessarily deallocates the BTB entry allocated at the prime step, resembling the experiment in §2.3.

We can optimize NV-Core by priming and probing multiple contiguous, non-overlapping PW ranges at once. Figure 7 compares the basic NV-Core monitoring one PW with an optimized NV-Core which monitors two PW ranges. In the optimized one, both the prime and the probe execute the chained PW snippets, and the prediction outcomes of all direct jumps are measured during the probe. This optimization not only allows NV-Core to simultaneously monitor multiple address ranges, but also expands the overall coverage.

Intel's Recent BTB Mitigations. Intel recently introduced Indirect Branch Restricted Speculation (IBRS) [28] and Indirect Branch Predictor Barrier (IBPB) [27] to mitigate Spectre V2 [35]. Although conventional wisdom might suggest that these schemes simply flush the BTB (which, if true, naturally defeats NV-Core), we tested NV-Core with both schemes enabled and still observe an update made by the victim's execution to the BTB entry state established by the attacker-controlled PW code. Our finding reveals that IBRS and IBPB only change state for part of the BTB, i.e., entries corresponding to indirect branches, rather than flushing the entire BTB. This is in line with the official security claims of IBRS and IBPB, which state that both schemes only apply to indirect branches [27, 28], and consistent with its goal to mitigate Spectre attacks: as direct jump targets are resolved early in decode, they cannot result in a large enough speculation window to enable Spectre attacks.

4.2 NightVision-User (NV-U)

As shown in Figure 6, NV-U invokes NV-Core for each victim execution "fragment", i.e., each time the victim is context switched onto/off of the core hosting the attacker-controlled BTB. This enables the attacker to learn dynamic PC information at scheduling-epoch granularity.

Since the BTB's size is limited, each context switch should run as few instructions as possible to minimize the chance that attacker BTB entries are evicted. NV-U leverages existing user-space preemptive scheduling attacks [3, 8, 18, 20–22, 49, 53, 59] to drastically reduce this victim time slice duration to on-order hundreds of cycles. In a nutshell, these preemptive scheduling attacks exploit the process scheduling mechanism adopted by modern Operating Systems (like Linux) by mounting a denial-of-service attack with hundreds of attacker-spawned child processes[3]. The attacker can (roughly) control the victim time slice by carefully controlling when the attacker processes unblock and yield to each other. We show in §5 how NV-U leaks victim control flow for programs hardened to withstand prior control-flow side-channel attacks.

4.3 NightVision-Supervisor (NV-S)

Same as NV-U, NV-S continuously invokes NV-Core across the entire victim's execution. Other than relying on the coarse-grained user-level scheduling technique, as a supervisor attacker, NV-S can leverage interrupts or signals (such as [42] and [13]) to *single-step* the victim's execution. In other words, every fragment is exactly one victim dynamic instruction, as indicated by Figure 6. This allows

[3]The implementation of the preemptive scheduling attack is orthogonal to our work and has been demonstrated by prior work (e.g., [22]). It is a common ingredient in side-channel attacks (including NightVision) for acquiring fine-grained side-channel measurements. We discuss its use and potential impact to NightVision in §6.3 and §8.1.

NV-S to determine if each victim dynamic instruction resides within specified PW(s) range(s). Although this information trivially enables the control-flow attack we describe in §5, it uniquely enables a new attack on private code fingerprinting that we will describe in §6.

5 USE CASE 1: CONTROL-FLOW LEAKAGE ATTACKS

5.1 Motivation and Threat Model

We assume a user-level attacker described in §3 who wishes to extract the victim's secret data. The secret must affect the victim program's control flow, e.g., as a branch predicate. The victim code is assumed to be *public*, but can use software mitigations such as branch balancing or control-flow randomization (CFR) [25]—see below—to block existing control-flow attacks. The victim could also use SGX to provide an extra layer of data protection.

The Control-flow Leakage Arms Race. Prior control-flow leakage attacks [17, 18, 23, 26, 39, 42, 46, 56, 60, 64] share the same attacker goal and have similar assumptions. However, the side channels that they exploit can be mitigated by incremental defenses that block various tell-tales of control-transfer instructions. For example, existing attacks on the BTB and the directional predictor [18, 26, 39] infer conditional branch outcomes by observing branch predictions. Correspondingly, CFR [25] mitigates those attacks by replacing observable secret branches with randomized jumps allocated at runtime, and indirect jumps are not exploitable with Intel's mitigation in place (§4.1). Several attacks [42, 46, 55] in turn leverage observed features in the code executed in the shadow of the branch, e.g., instruction count [42], type [55] or alignment [46], to leak control-flow decisions. Yet these attacks fail against branch balancing-style defenses, e.g., padding both sides of the branch to the same instruction count, type, or alignment [46].

All of the above defenses fail against NightVision, due to its ability to extract victim PCs directly. For example: CFR fails because it tries to protect the branch decision, but NightVision does not rely on the branch decision. Balancing fails because it tries to make the execution of both sides of the branch look the same, but NightVision directly extracts the PCs of instructions shadowing the branch.

5.2 Control-flow Leakage Attack Procedure Using NV-U

For a control-flow leakage attack, with the knowledge of the victim's program, the attacker first selects one or several consecutive victim instructions that control-depend on the secret, i.e. instructions that execute if and only if the branch swings in a particular direction. The attacker then creates a PW code snippet, and ensures the PW range is within the virtual address range of the chosen victim instructions. Finally, the attacker employs NV-U along with the victim's execution to determine whether and roughly when (within which time slices) the victim executes the chosen instructions during its execution, and deduces the secret value based on that information.

NightVision achieves a byte-granularity observation since the PW can start at any byte address. Given the shortest PW snippet contains a 2-byte direct jump, any instruction longer than one byte

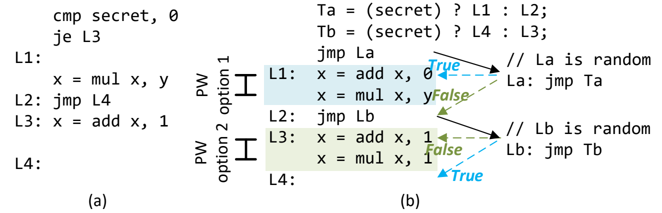

Figure 8: (a) The original leaky code (b) The hardened code with branch balancing and control-flow randomization [25]. NV-U defeats both defenses by observing which side of the branch (L1 or L3) is executed with a PW that is either a sub-interval of the address range of L1 (PW option 1) or L3 (PW option 2).

(almost all x86 instructions except `ret` and `nop`[4]) or any number of consecutive instructions can be measured with a PW. For example, one may use NV-U to observe whether a basic block executes, by making the PW a sub-range of the address range of that basic block.

Figure 8 demonstrates the control-flow leakage attack when the victim program is using branch balancing and CFR to protect the secret branch condition. Given the secret-dependent control flow, i.e., either the then side (L1) or the else side (L3) will be executed, NV-U can create a PW code snippet, and ensure that the PW range is either within the address range of L1 or L3. By observing whether L1 or L3 is executed, NV-U can deduce the value of the secret.

Note that the attacker often needs to measure the decisions of multiple dynamic instances of the same branch inside a loop. As mentioned in §4.2, NightVision relies on a preemptive scheduling attack, a technique applied by many existing attacks for achieving fine-grain temporal resolution side-channel measurements (e.g., per loop iteration) [14, 20, 21, 45, 57, 59]. This technique does have a limitation: it does not provide synchronization between the attacker and victim, since it does not ensure the preemption of the victim exactly once per loop iteration. To help overcome this limitation, we note that NV-U provides additional opportunities to deduce the victim's execution progress. For example, in Figure 8, if the attacker monitors both L1 and L3 with both PW options 1 and 2, it is possible to detect excessive preemptions occurring within one loop iteration—NV-U performed in those excessive preemptions will show that neither L1 nor L3 is executed.

6 USE CASE 2: FINGERPRINTING PRIVATE CODE

6.1 Intel SGX

Intel SGX [6, 41] is a hardware-based trusted execution environment solution that is widely deployed by major cloud vendors today. It protects the integrity and confidentiality of (a part of) a user-level application from a powerful attacker who controls the entire software stack (including the OS, hypervisor, and BIOS), with a new CPU mode called *enclave*. Enclaves leverage hardware-managed memory isolation to guarantee that the enclave state can only be accessed by its owner enclave. However, SGX enclaves still rely on an untrusted OS for managing resources, such as page tables

[4] And some rarely used instructions, such as halt (`hlt`), complement carry flag (`cmc`).

and interrupts, enabling a privileged attacker to interrupt enclave execution, as shown by controlled-channel attacks [64], microarchitectural replay attacks [51], SGXStep [54], etc. Aside from the typical data protection, Intel SGX also provides code confidentiality via Protected Code Loader (SGX PCL) [30], or similar mechanisms [9, 37, 47, 66], by keeping the enclave binary encrypted until it is loaded into the enclave. SGX also serves as a design paradigm for future TEE solutions, such as Intel TDX [32].

6.2 Threat Model and Motivation

We assume an untrusted, privileged attacker whose goal is to obtain the victim's code. Given this strong attacker model, we make a realistic assumption that the victim relies on a TEE mechanism, in this work Intel SGX. SGX provides data confidentiality such that the victim's program state is inaccessible to the attacker, and processor features such as LBR, Intel Processor Trace, and performance counters are disabled in enclave mode. SGX also provides code confidentiality (using PCL [30] or similar mechanisms [9, 37, 47, 66]), therefore the attacker has no knowledge of the victim enclave code.

Different from §5 which leaks data in a public program setting, this section demonstrates how NV-S enables reverse-engineering of private programs. Existing side-channel attacks mostly consider programs as public and available for offline analysis. However, an attacker without knowledge of the code, for instance, cannot locate Spectre gadgets [12], deduce that certain types of side-channel vulnerabilities exist [16, 61], nor determine how to monitor the side-channel [63]. Although this may imply that keeping the program private could be a holistic defense strategy by breaking an essential requirement for side-channel attacks, this "security by obscurity" fails with NV-S, which extracts every element in the dynamic PC trace with high accuracy. Although instruction addresses cannot directly reflect the binary content, a sufficiently long sequence of PCs may contain enough entropy to identify a specific function. Using this insight, we design a function fingerprinting approach that identifies functions of interest from the extracted PC trace. NightVision thus complements existing side-channel attacks by satisfying their assumption about victim programs being public, even when the victim enclave program is unreadable+unwritable and only executable.

6.3 Dynamic PC Trace Extraction Using NV-S

NV-S infers the complete byte-granularity control-flow information of an enclave program's execution, in the form of a sequence of PCs of every retired dynamic enclave instruction. No existing control-flow leakage attack achieves this goal: they either only recover coarse spatial-granularity control-flow information [56, 60, 64] or only leak the addresses of specific instruction types/patterns [14, 19, 39, 46].

Figure 9 illustrates the entire attack flow. In the following, we focus on how to deduce a victim instructions' page offset bits, since the victim instruction virtual page numbers can be leaked through complementary work on controlled-channel attacks [56, 60, 64] (line 2-4 in Figure 9) that induce page faults to learn page-level information. The attack repeatedly invokes NV-S (line 1), and every invocation of NV-S continuously applies the optimized NV-Core to measure every dynamic instruction against multiple PW ranges

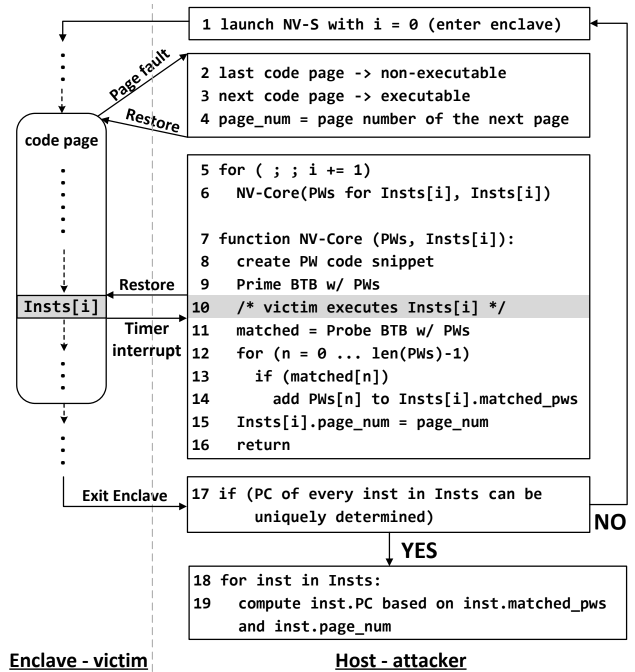

Figure 9: How NV-S infers the address of every dynamic enclave instruction. Definitions of symbols: `Insts`: the sequence of dynamic enclave instructions; `Insts[i].pc`: the PC of `Insts[i]`; `Insts[i].matched_pws`: set of PWs which collide with `Insts[i]`; `Insts[i].page_num`: page number of the code page containing `Insts[i]`.

(line 5-6). To execute exactly one enclave instruction during each call of NV-Core, we use SGXStep [54] to single-step the enclave, by choosing a proper timer interrupt interval such that the enclave execution is interrupted precisely after each dynamic instruction is retired. Once an instruction retires, the timer interrupt delivers the control to NV-Core inside the interrupt handler. By probing the BTB with the PW code snippet (line 11), NV-Core determines if the instruction overlaps with the tested PW ranges (line 12-14), and then launches NV-Core for the next instruction by choosing and creating the PW code snippet for the next instruction and starting the prime step before resuming the enclave execution (line 8-9). How NV-S chooses the PWs is described in the next paragraph. When the enclave execution finishes, for each dynamic instruction, NightVision collects the PWs that overlap with the instruction. If for any dynamic instruction, its matched PWs so far are insufficient for determining its PC, NightVision performs another round of NV-S (line 17).

PW Traversal. We now explain how to choose a proper set of PW ranges for NV-Core to measure each instruction during every NV-S call. We leverage the range semantics of PWs to perform a binary search through progressively-smaller PWs for each victim instruction. Searching for the instruction PC is divided into multiple passes. Each pass splits the matched PW range in the previous pass, and measures which sub-PW range contains the base address of the instruction. As shown in Figure 10, the measurement of every

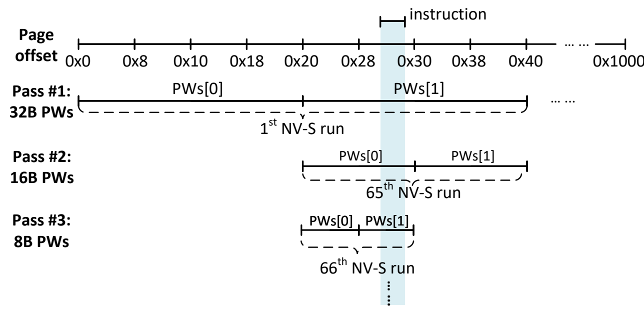

Figure 10: The PW traversal approach is explained in §6.3. NightVision first generates 128 mutually-disjoint 32-byte PWs. The matched PW is further converted to smaller PWs recursively. We assume N = 2, therefore each NV-Core tests the current instruction with two PWs.

dynamic instruction starts by dividing the 4 KB page size range into 128 mutually disjoint 32-byte PW ranges. Suppose every call to NV-Core tests N PW ranges. The first pass takes 128/N enclave executions (NV-S calls), after which NightVision determines inside which 32-byte PW each dynamic enclave instruction starts. Next, this 32-byte PW is split into N sub-PWs, and with one more NV-S call, NightVision can determine which sub-PW each dynamic instruction starts. This process is repeated until it determines the base address of the target instruction.

Impact of Speculative Execution. The enclave single-stepping retires exactly one instruction per interrupt. However, succeeding instructions may speculatively execute, and update the BTB state before the current instruction retires. So instead of measuring only the address range of the single-stepped instruction, NightVision may measure the address ranges of all executed instructions, including the succeeding speculatively-executed ones. When speculation does not involve control transfer instructions, the base of the (extended) measured range still corresponds to the PC of the target instruction. However, when speculation involves control transfers, NV-S may produce multiple candidate PCs: one being the PC of the target instruction, and the others being the target PCs of the succeeding control-transfer instructions which execute speculatively. The correct one can be deduced after the next single step, when NV-S again generates a set of candidate PCs for the next instruction, including the false PCs corresponding to the control-transfer targets. The actual PCs belonging to the two single-stepped instructions are captured by comparing the two PC sets and ruling out the repeated candidates.

6.4 Function Fingerprinting for Private Code

Although NightVision only collects PC traces, which have no information about the actual instruction bytes, PC traces of sufficient length can be used to fingerprint known PC sequences corresponding to functions in existing code bases. This technique is called function fingerprinting [43] and is employed by NightVision for deducing whether the enclave execution contains functions from a known binary file (these functions are called *reference functions*). In general, it is impractical to assume the attacker owns a set of

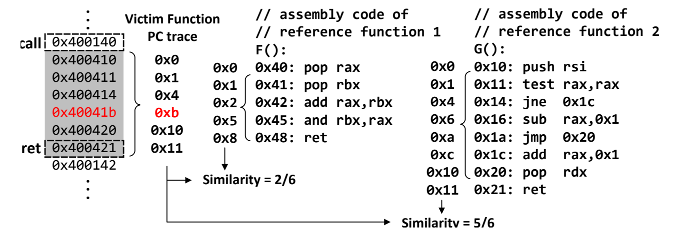

Figure 11: NightVision function fingerprinting computes the similarity between the victim function and two reference functions F and G. Red PC value represents an incorrect measurement.

binaries that will always include enclave code. However, we deem this reasonable for at least cryptographic code since most enclave programs use several popular off-the-shelf, open-source cryptographic libraries, several of which contain vulnerable functions that have been exploited by existing side-channel attacks. With the function fingerprinting, such use of vulnerable functions can be identified in private programs.

The attack proceeds in two steps, as shown in Figure 11. First, we collect the victim PC trace through NV-S and pre-process the trace. Second, we compare the *similarity* of victim functions included in the PC trace to reference functions.

Step 1: Victim PC trace preprocessing. NightVision uses the complete attack flow described in §6.3 to obtain a victim PC trace. As illustrated by Figure 11, it partitions the whole PC trace at function call boundaries into per-function traces. Each function-level trace is normalized to be position-independent by subtracting the PC of the function start from all PCs in the trace, thus each function-level trace starts with zero. Every function-level trace represents an invocation of an unknown *victim function*, and will then be considered separately during function similarity matching (Step 2).

Function-level traces are sliced by locating `call`/`ret` pairs in the original trace, using the following approach to identify PCs corresponding to `call`s and `ret`s. First, we capture jumps between PCs that are greater than 16 bytes, which indicates a control-transfer instruction. Second, unlike other control-transfer instructions, `call`/`ret` also access data memory. Thus, we additionally monitor whether a suspected `call`/`ret` accesses a data page (through a controlled-channel attack [64]). Note for this work, we assume functions are only entered/exited via `call`s/`ret`s.

Step 2: PC trace similarity test. Next, we compare the function-level PC traces collected during Step 1 that represent unknown victim functions with *sets* of PCs corresponding to reference functions. Importantly, the PC traces collected during Step 1 are *dynamic* instruction sequences. To avoid having to build a large (potentially exponential) number of dynamic PC traces for reference functions, we simply analyze the PCs corresponding to static instructions within each reference function. Testing similarity between victim and reference functions then proceeds as follows, as illustrated by Figure 11.

(1) (One time, offline) For each reference function `f*`, collect the static PCs in that function (relative to the entry PC, to maintain position independence) into a set called `S*`.
(2) For each victim function-level PC trace `t` (from Step 1), convert `t` to a set called `S` and compute similarity = $(|S \cap S*|)/|S|$) for each reference PC set `S*`.

The percentage of PCs that survive the intersection indicates the similarity between the victim function and the reference function. We note that this heuristic makes use of x86's variable-length instruction encoding in an essential way: due to the nature of variable-length instruction encoding, the instruction length is directly influenced by the instruction semantics. For example, x86 uses different byte lengths for different opcodes and addressing modes. Additionally, variable instruction lengths add extra entropy to the PC trace, which serves as the fingerprint in our case. Also, notice that the similarity based on set intersection sacrifices information such as the ordering of PCs for simplicity. §8.3 discusses a more sophisticated approach that considers instruction ordering.

7 EVALUATION

7.1 Experimental Methodology

We perform the evaluation on Intel 9700 and 9900(K) CPUs, all running Ubuntu 18.04 kernel version 4.15.0, with Hyper-threading disabled. All tested cryptographic programs are compiled with gcc 7.5.0 (more compiler versions are used when evaluating function fingerprinting in §7.3). When evaluating control-flow leakage attack in §7.2, the attacker and the victim run separated userspace processes. For the fingerprinting experiment in §7.3, the victim programs are written with Intel SGX SDK [15] and run inside the enclave.

7.2 Evaluating Control-flow Leakage Attack (Use Case 1)

We first evaluate whether NightVision can leak secret data in common cryptographic code through vulnerable functions with secret-dependent control flow. Specifically, we focus on leaking the secret key during the RSA key generation procedure in mbedTLS [2] version 3.0 by inferring the secret-dependent control-flow behavior in Great Common Divisor (`GCD`) function. `GCD` contains a loop, in which a perfectly-balanced branch repeatedly evaluates the secret key values. Recovering the secret key requires determining the direction of the balanced branch at each loop iteration. The recent Frontal attack [46] has already exploited this vulnerable function. On the other hand, Frontal can be mitigated by aligning two sides of the branch to the same base address modulo the instruction fetch window size (16-byte in Intel CPUs) using a simple compiler flag `-falign-jumps=16`. This flag is applied in our experiment.

We implement a proof-of-concept control-flow leakage attack by simulating the preemptive scheduling attack, following the same methodology as prior work [14, 18, 20, 21, 45, 59]. Specifically, we make the victim call `sched_yield()` system call after the branch body to yield to the attacker process. The attacker process executes the NV-U routine to deduce the secret control-flow decision of the current victim loop iteration, followed by a `sched_yield()` to transfer the control back to the victim for the next loop iteration.

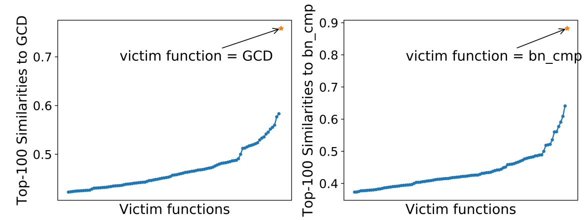

Figure 12: The top-100 highest similarity of the measured victim functions with respect to `GCD` (left) and `bn_cmp` (right). When matching against `GCD` as the reference function, the highest similarity is 75.8% when the victim function is also `GCD`. The highest similarity to `bn_cmp` is 88.2% (when the victim function is `bn_cmp`).

We notice that the per-victim-loop-iteration time is over 300 cycles, which is greater than the minimal time slice achievable by the preemptive scheduling attack [8, 22, 49]. Therefore, although our proof-of-concept attack simulates the attack preemption with `sched_yield()`, per-loop-iteration measurement is possible for `GCD` in practice.

We apply the strategy described in §5.2 to infer the direction of the balanced branch. Because the instructions on the *then* path occupy address range [`0x5940`, `0x597c`], and the instructions on the *else* path occupy address range [`0x5980`, `0x59bc`], we simply apply NV-U oracle with a PW range [`0x5980`, `0x598f`] that only overlaps with the first several instructions on the *else* path. When the attacker observes a misprediction in the probe step of NV-U, its BTB entry is updated (deallocated), meaning that the else path should be taken. We repeat the attack on 100 different victim process executions. Each run calls the RSA key generation function for generating a new key, and on average loops over the vulnerable branch 30 times in `GCD`. NightVision achieves 99.3% accuracy in measuring the direction of the vulnerable branch.

We additionally use NightVision to infer the secret predicate of a similarly balanced branch in the big number comparison (`bn_cmp`) function in Intel's IPP-Crypto [31] v2020. Frontal also evaluates this function. Similar to `GCD` above, Frontal cannot succeed when the basic block alignment flag is enabled. NightVision again is able to achieve 100% accuracy in inferring the branch direction across 100 different runs.

7.3 Evaluating Function Fingerprinting (Use Case 2)

We now evaluate the effectiveness of NightVision's function fingerprinting in the private program setting. Since the goal of function fingerprinting is to reveal the use of vulnerable functions in the private victim binary, here we show that it is possible to use the proposed function fingerprinting mechanism to detect the use of `GCD` and `bn_cmp` evaluated in §7.2.

An effective fingerprinting should only match an unknown victim function to a reference function when the victim function is indeed the reference function. To validate that NightVision fulfills this requirement, we generate victim PC traces for `GCD` and `bn_cmp` as well as other 175,168 additional functions from many

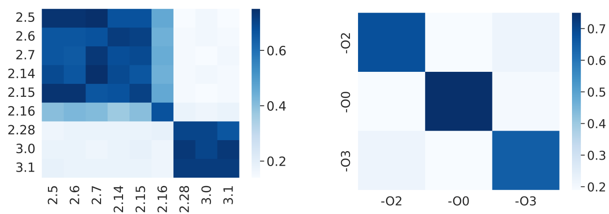

Figure 13: (Left) The similarities of GCD in eight different MbedTLS versions (2.5-3.1) to GCD in the same/different versions computed by NightVision's fingerprinting mechanism. (Right) The similarity of GCD compiled with three different optimization levels to GCD with the same/different optimization levels.

open-source SGX projects listed in [1]. Then we compute the similarity of all tested victim functions to the two reference functions GCD and bn_cmp. Figure 12 demonstrates that for both GCD and bn_cmp, we can observe the highest similarity when the victim function is indeed GCD or bn_cmp, whereas all other functions that are not the reference function exhibit lower similarity. This shows that NightVision can identify the two vulnerable functions in a large group of unknown, executed victim functions.

We notice that the similarities of GCD and bn_cmp with themselves are not even 100%. We compare PC traces, with their originating assembly code, and notice that nearly all incorrectly measured instructions correspond to *macro-fusion* structures [29, 62]. Since macro-fusion combines adjacent instructions (usually arithmetic-branch or load-arithmetic-branch) into a single, executable macro-op, one single step actually executes and retires all instructions in a macro-fusion structure. Therefore, NightVision only manages to measure the leading instruction in a macro-op structure. The impact of macro-fusion on enclave single-stepping attacks has also been observed and studied by prior work, e.g., CopyCat [42].

NightVision's fingerprinting only handles functions in binary form, meaning that the version of the library to which the function belongs, and the compiler configuration, can both influence fingerprinting results. To evaluate how robust our fingerprinting is to these effects, we compile GCD by tuning three different parameters:

(1) MbedTLS library version: 8 different versions as shown in Figure 13 (left)
(2) GCC version: 7.5, 8.4, 9.4, 10.3
(3) Compiler optimization flag: -O0, -O2, -O3

We then run each compiled GCD with NightVision and compute the similarity of each specific GCD with all compiled versions of GCD. We draw the following conclusions. First, the impact of the library versions on fingerprinting depends on whether the source code is changed across the library versions. We found that, for example, the GCD function source code does not change across MbedTLS versions 2.5-2.15, but version 2.16 has a different implementation than previous versions. This is reflected by Figure 13 (left): the similarity amongst versions 2.5-2.15 is much higher than the similarity between a version before 2.16 and one after 2.16. Second, the compiler version alone usually does not affect the function binary. Third, as demonstrated by Figure 13 (right), compiler flags can significantly impact the fingerprinting result. The similarity between the victim function and the target function may not be high enough when the victim function and the target function are compiled with largely-different compilation options. In summary, to successfully identify the use of a specific target function in an unknown binary, the attacker should compile the function from different library versions, and with different compilation options.

8 DISCUSSION

8.1 Limitations

Affected CPUs. In this work, we examined a limited set of Intel desktop/server CPUs listed in §2.3. However, any CPU that facilitates BTB updates by non-control instructions is potentially vulnerable to the same attack. Since deep processor pipelining and superscalar design are the underlying causes factors motivating this behavior, NightVision can also likely affect other existing high-performance CPUs. We leave the analysis of CPUs from other vendors as future work.

Limitations of Control-flow Leakage. As explained in §5.2, NV-U relies on the preemptive scheduling attack to achieve fine-grained measurement of the victim's execution, similar to many existing side-channel attacks [3, 8, 14, 18, 20–22, 45, 49, 53, 59]. This technique is infeasible in restricted environments such as browsers, which limit the attacker's capability of arbitrarily spawning threads. Additionally, the preemptive scheduling attack does not provide perfect synchronization between the NV-U attacker and the victim, thus the attacker needs complementary techniques (such as the example in §5.2) to deduce the victim's execution progress.

Limitations of Binary Fingerprinting. Our fingerprinting attack is based on several assumptions mentioned in §6. First, the attack is useful primarily in the context of variable-length ISAs (like x86), as the variety in the instruction length amplifies the entropy in the PC sequence, resulting in fewer false positives/negatives in fingerprinting. Second, the fingerprinted function itself must have a sufficiently long PC trace to produce enough entropy to differentiate it from incorrect candidates. Third, the attacker must possess knowledge of the target function in assembly form, which may require non-trivial effort from the attacker to prepare different possible assembly forms for the target function, similar to §7.3.

8.2 Mitigations

Data-Oblivious Programming. The only reliable software mitigation for NightVision is to ensure the program control flow never depends on secret information, using data-oblivious programming. Achieving data-obliviousness for normal programs requires engineering effort to port existing applications [11, 40, 48, 68] and suffers from significant performance overhead [67].

We have seen existing cryptographic libraries gradually adopt data-oblivious programming for eliminating secret-dependent control-flows [31] and design efficient data-oblivious implementations for critical cryptographic operations [10]. That said, non-data-oblivious cryptographic libraries used by legacy binaries are still susceptible to NightVision. In addition, NightVision's function fingerprinting capability is unaffected by data-oblivious programming.

BTB Hardening. NightVision can be mitigated by constantly flushing BTB state [39], or enforcing strict isolation between security domains [38, 70]. However, neither approach has been adopted by current processors, due to the performance cost and implementation complexity. The only effort in this direction is IBRS [28] and IBPB [27] which only prevent indirect branches belonging to different security domains from influencing each other, as stated in §4.1. Future processor generations could extend those defense proposals to block attacks like NightVision.

8.3 Improving Function Fingerprinting

NightVision's fingerprinting technique constructs function fingerprints by compressing sequences of dynamic PCs into sets of static PCs. This simplifies the fingerprint-matching problem to performing set intersections, but sacrifices information about the original PC sequences such as the instruction order (e.g., loops). An alternative fingerprinting mechanism could directly use the dynamic PC trace as the function fingerprint. In this case, matching the PC sequence to reference functions becomes a more-complicated pattern-matching problem: the PC sequence must obey the control flow of the original function, modulo the measurement error. We note that this process is similar to genomic (DNA) sequence matching, which pattern-matches a DNA sequence against several sample DNA sequences, and at the same time, circumvents the interference from mutated genes. We leave applying related approaches to improve NightVision's fingerprinting as future work.

9 RELATED WORK

Existing Side Channels Extracting PC-related Information. Prior side-channel attacks leak partial information about victim's dynamic PCs. First, controlled-channel attacks manipulate page permissions and attributes (access/dirty bits) to observe the page-granularity PC trace [56, 60, 64]. Succeeding attacks leverage the instruction cache to obtain cache-granularity PC information [23]. These attacks potentially leak information of all dynamic PCs, but at a coarse spatial granularity. Correspondingly, several mitigations have been proposed to confine secret-dependent control flow, e.g., inside a single page or cache line [23, 50]. The more recent Frontal attack [46] exploits instruction decoding timing for deducing information about basic block alignment, i.e., code address offsets relative to 16 B blocks. However, the timing channel discovered by the Frontal attack is not sufficiently precise for determining byte-granularity PC information (use case 2 in our case); rather, it uses differential analysis to differentiate branch directions (use case 1 in our case, which can be mitigated as described in §7.2). Additionally, Frontal requires memory writes to be present inside the basic block.

Several other attacks achieve more fine-grained PC measurements for specific PC types. For instance, prior attacks targeting the BTB [17, 39] leverage collisions between control-transfer instructions to deduce if a branch/jump at a specific PC executes at runtime. Although these side-channel approaches have proved to successfully achieve different attack goals (e.g., control-flow leakage attacks, breaking ASLR, crafting Spectre attacks), control-flow randomization [25] converts conditional branches into randomized indirect jumps to thwart attacks targeting conditional branches, and recent hardware mitigations such as IBRS further provide protection for indirect jumps. Note that a prior attack BranchShadowing [39] is similar to NV-S in that both leverage SGX single-stepping and BTB Prime+Probe. However, NV-S's BTB Prime+Probe mechanism is inherently different from BranchShadowing (§2.5), hence NV-S can infer the PC of every dynamic enclave instructions while BranchShadowing is limited to behaviors of branches with known PCs. Other hardware structures, e.g., the hardware prefetcher [14], TLB [19, 60], reflect the PC information of memory loads and stores. NightVision is the first side-channel capable of leaking the precise byte-granular PC for all instructions types, and in the ideal case, for every single victim dynamic instruction.

Other Control-flow Leakage Attacks. Many control-flow leakage attacks also do not rely on extracting dynamic PCs directly. Attacks which exploit directional branch predictors [18, 26] are well-known for leaking secret branch conditions. Similarly, such attacks can be mitigated with software defenses protecting conditional branches [25]. Recent attacks, such as CopyCat [42] and Nemesis [55], assume a privileged attacker who can single-step the victim's execution, which is the same as the supervisor-level attacker model in this work. They show how to acquire secret-dependent control flow by counting instructions or deducing the executed instructions' types. However, such observations (instruction counting/types) are insufficient for control-flow leakage attacks if the victim deploys defenses such as branch balancing, and contain much less entropy for binary fingerprinting when compared to NightVision.

10 CONCLUSION

NightVision is the first micro-architectural side-channel attack that extracts dynamic, byte-granular PCs from the victim program's execution—in the best case, for every victim instruction. NightVision reveals victim dynamic PCs *directly*, thereby bypassing prior defenses that attempt to block channels leaking partial/indirect information about the PC sequence (such as the length of subsequent basic blocks). NightVision also showcases its ability to identify unknown programs, challenging the notion of "security through obscurity" while complementing existing side-channel attacks which by default require public code.

ACKNOWLEDGMENTS

This work was funded by NSF under grants CNS-1942888, CNS-1954521, and CNS-1801534. We would like to thank the anonymous shepherd and reviewers for their insightful comments during the review process, which helped to significantly strengthen the paper.

REFERENCES

[1] 2022. Awesome SGX Open Source Projects. https://github.com/Maxul/Awesome-SGX-Open-Source.

[2] 2022. Mbed-TLS: An open source, portable, easy to use, readable and flexible SSL library. https://github.com/ARMmbed/mbedtls.

[3] Onur Aciiçmez. 2007. Yet another microarchitectural attack: exploiting I-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture*. 11–18.

[4] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. Predicting secret keys via branch prediction. In *Proceedings of the 7th Cryptographers' track at the RSA conference on Topics in Cryptology*. 225–242.

[5] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port contention for fun and profit. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 870–887.

[6] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, Vol. 13. ACM New York, NY, USA, 7.

[7] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2015. On Subnormal Floating Point and Abnormal Timing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. 623–639.

[8] C Ashokkumar, Ravi Prakash Giri, and Bernard Menezes. 2016. Highly efficient algorithms for AES key retrieval in cache access attacks. In *2016 IEEE European symposium on security and privacy (EuroS&P)*. IEEE, 261–275.

[9] Erick Bauman, Huibo Wang, Mingwei Zhang, and Zhiqiang Lin. 2018. Sgxelide: enabling enclave code secrecy via self-modification. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 75–86.

[10] Daniel J Bernstein and Bo-Yin Yang. 2019. Fast constant-time gcd computation and modular inversion. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2019), 340–398.

[11] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. 2017. Fact: A flexible, constant-time programming language. In *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 69–76.

[12] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2019. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 142–157.

[13] Yuan Chen, Jiaqi Li, Guorui Xu, Yajin Zhou, Zhi Wang, Cong Wang, and Kui Ren. 2022. SGXLock: Towards Efficiently Establishing Mutual Distrust Between Host Application and Enclave for SGX. In *31st USENIX Security Symposium (USENIX Security'22)*. 4129–4146.

[14] Yun Chen, Lingfeng Pei, and Trevor E Carlson. 2023. AfterImage: Leaking Control Flow Data and Tracking Load Operations via the Hardware Prefetcher. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 16–32.

[15] Intel Corporation. 2016. Intel® Software Guard Extensions SDK for Linux* OS (Developer Reference). (2016).

[16] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2015. Cacheaudit: A tool for the static analysis of cache side channels. *ACM Transactions on information and system security (TISSEC)* 18, 1 (2015), 1–32.

[17] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13.

[18] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. (2018), 693–707.

[19] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *27th USENIX Security Symposium (USENIX Security'18)*. 955–972.

[20] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. 2016. Flush, gauss, and reload–a cache attack on the BLISS lattice-based signature scheme. In *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 323–345.

[21] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. 2016. Cache storage channels: Alias-driven attacks and verified countermeasures. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 38–55.

[22] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache games–bringing access-based cache attacks on AES to practice. In *2011 IEEE Symposium on Security and Privacy*. IEEE, 490–505.

[23] Marcus Hähnel, Weidong Cui, and Marcus Peinado. 2017. High-resolution side channels for untrusted operating systems. In *2017 USENIX Annual Technical Conference (USENIX ATC'17)*. 299–312.

[24] John L Hennessy and David A Patterson. 2011. Advanced Techniques for Instruction Delivery and Speculation. In *Computer architecture: a quantitative approach*. Elsevier, Chapter 3.9, 203–206.

[25] Shohreh Hosseinzadeh, Hans Liljestrand, Ville Leppänen, and Andrew Paverd. 2018. Mitigating branch-shadowing attacks on Intel SGX using control flow randomization. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*. 42–47.

[26] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. 2020. Bluethunder: A 2-level directional predictor based side-channel attack against sgx. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2020), 321–347.

[27] Intel. 2018. Indirect Branch Predictor Barrier. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-predictor-barrier.html.

[28] Intel. 2018. Indirect Branch Restricted Speculation. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html.

[29] Intel. 2018. Intel 64 and IA-32 Architectures Optimization Reference Manual. https://intel.ly/2UbLwk2.

[30] Intel. 2018. Intel® Software Guard Extensions (Intel® SGX) Protected Code Loader (PCL) for Linux. (2018).

[31] Intel. 2020. Intel IPP Crypto Library (2020). https://github.com/intel/ipp-crypto/tree/ipp-crypto_2020.

[32] Intel. 2022. Intel Trust Domain Extensions. https://software.intel.com/content/dam/develop/external/us/en/documents/tdxwhitepaper-v4.pdf.

[33] Yasuo Ishii, Jaekyu Lee, Krishnendra Nathella, and Dam Sunwoo. 2021. Re-establishing fetch-directed instruction prefetching: An industry perspective. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 172–182.

[34] Joonsung Kim, Hamin Jang, Hunjun Lee, Seungho Lee, and Jangwoo Kim. 2021. UC-Check: Characterizing Micro-operation Caches in x86 Processors and Implications in Security and Performance. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 550–564.

[35] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–19.

[36] Jagadish B Kotra and John Kalamatianos. 2020. Improving the Utilization of Micro-operation Caches in x86 Processors. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 160–172.

[37] Titouan Lazard, Johannes Götzfried, Tilo Müller, Gianni Santinelli, and Vincent Lefebvre. 2018. TEEshift: Protecting code confidentiality by selectively shifting functions into TEEs. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*. 14–19.

[38] Jaekyu Lee, Yasuo Ishii, and Dam Sunwoo. 2020. Securing branch predictors with two-level encryption. *ACM Transactions on Architecture and Code Optimization (TACO)* 17, 3 (2020), 1–25.

[39] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security'17)*. 557–574.

[40] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. Oblivm: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 359–376.

[41] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative instructions and software model for isolated execution. (2013), 1–1.

[42] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. 2020. CopyCat: Controlled Instruction-Level Attacks on Enclaves. In *29th USENIX Security Symposium (USENIX Security'20)*. 469–486.

[43] Lina Nouh, Ashkan Rahimian, Djedjiga Mouheb, Mourad Debbabi, and Aiman Hanna. 2017. BinSign: Fingerprinting binary functions to support automated analysis of code executables. In *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 341–355.

[44] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Cryptographers' track at the RSA conference*. Springer, 1–20.

[45] Riccardo Paccagnella, Licheng Luo, and Christopher W Fletcher. 2021. Lord of the Ring (s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical. In *30th USENIX Security Symposium (USENIX Security'21)*. 645–662.

[46] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Čapkun. 2021. Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend. In *30th USENIX Security Symposium (USENIX Security'21)*. 663–680.

[47] Ivan Puddu, Moritz Schneider, Daniele Lain, Stefano Boschetto, and Srdjan Čapkun. 2022. On (the Lack of) Code Confidentiality in Trusted Execution Environments. *arXiv preprint arXiv:2212.07899* (2022).

[48] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium (USENIX Security'15)*. 431–446.

[49] Bholanath Roy, Ravi Prakash Giri, C Ashokkumar, and Bernard Menezes. 2015. Design and implementation of an espionage network for cache-based side channel attacks on AES. In *2015 12th International Joint Conference on e-Business and Telecommunications (ICETE)*, Vol. 4. IEEE, 441–447.

[50] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. 2016. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. 317–328.

[51] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W Fletcher. 2019. Microscope: Enabling microarchitectural replay attacks. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 318–331.

[52] A. Smith and B. Johnson. 1998. Method and apparatus for implementing a set associative branch target buffer.

[53] Dan Tsafrir, Yoav Etsion, and Dror G Feitelson. 2007. Secretly Monopolizing the CPU Without Superuser Privileges.. In *USENIX Security Symposium*. 239–256.

[54] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd*

Workshop on System Software for Trusted Execution. 1–6.

[55] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 178–195.

[56] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *26th USENIX Security Symposium (USENIX Security'17)*. 1041–1056.

[57] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2021. CacheOut: Leaking data on Intel CPUs via cache evictions. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 339–354.

[58] Jose Rodrigo Sanchez Vicarte, Pradyumna Shome, Nandeeka Nayak, Caroline Trippel, Adam Morrison, David Kohlbrenner, and Christopher W Fletcher. 2021. Opening pandora's box: A systematic study of new ways microarchitecture can leak private data. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 347–360.

[59] Daimeng Wang, Zhiyun Qian, Nael Abu-Ghazaleh, and Srikanth V Krishnamurthy. 2019. Papp: Prefetcher-aware prime and probe side-channel attack. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–6.

[60] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. 2017. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2421–2434.

[61] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. 2018. Microwalk: A framework for finding side channels in binaries. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 161–173.

[62] WikiChip. 2020. Macro-Operation Fusion (MOP Fusion). https://en.wikichip.org/wiki/macro-operation_fusion..

[63] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. 2017. Stacco: Differentially analyzing side-channel traces for detecting SSL/TLS vulnerabilities in secure enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 859–874.

[64] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 640–656.

[65] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security'14)*. 719–732.

[66] Jiyong Yu, Xinyang Ge, Trent Jaeger, Christopher W Fletcher, and Weidong Cui. 2022. Pagoda: Towards Binary Code Privacy Protection with SGX-based Execute-Only Memory. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 133–144.

[67] Jiyong Yu, Lucas Hsiung, Mohamad El'Hajj, and Christopher W Fletcher. 2019. Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing. (2019).

[68] Samee Zahur and David Evans. 2015. Obliv-C: A Language for Extensible Data-Oblivious Computation. *IACR Cryptol. ePrint Arch.* 2015 (2015), 1153.

[69] Tao Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. 2020. Exploring branch predictors for constructing transient execution trojans. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. 667–682.

[70] Lutan Zhao, Peinan Li, Rui Hou, Michael C Huang, Jiazhen Li, Lixin Zhang, Xuehai Qian, and Dan Meng. 2021. A lightweight isolation mechanism for secure branch predictors. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1267–1272.