Speculative Probing: Hacking Blind in the Spectre Era

Enes Göktaş
egoktas@stevens.edu
Stevens Institute of Technology

Kaveh Razavi
kaveh@ethz.ch
ETH Zürich

Georgios Portokalidis
gportoka@stevens.edu
Stevens Institute of Technology

Herbert Bos
herbertb@cs.vu.nl
Vrije Universiteit Amsterdam

Cristiano Giuffrida
giuffrida@cs.vu.nl
Vrije Universiteit Amsterdam

ABSTRACT

To defeat ASLR or more advanced fine-grained and leakage-resistant code randomization schemes, modern software exploits rely on information disclosure to locate gadgets inside the victim's code. In the absence of such info-leak vulnerabilities, attackers can still *hack blind* and derandomize the address space by repeatedly probing the victim's memory while observing crash side effects, but doing so is only feasible for crash-resistant programs. However, high-value targets such as the Linux kernel are not crash-resistant. Moreover, the anomalously large number of crashes is often easily detectable.

In this paper, we show that the Spectre era enables an attacker armed with a single memory corruption vulnerability to hack blind without triggering any crashes. Using speculative execution for crash suppression allows the elevation of basic memory write vulnerabilities into powerful *speculative probing* primitives that leak through microarchitectural side effects. Such primitives can repeatedly probe victim memory and break strong randomization schemes without crashes and bypass all deployed mitigations against Spectre-like attacks. The key idea behind speculative probing is to break Spectre mitigations using memory corruption and resurrect Spectre-style disclosure primitives to mount practical blind software exploits. To showcase speculative probing, we target the Linux kernel, a crash-sensitive victim that has so far been out of reach of blind attacks, mount end-to-end exploits that compromise the system with just-in-time code reuse and data-only attacks from a single memory write vulnerability, and bypass strong Spectre *and* strong randomization defenses. Our results show that it is crucial to consider synergies between different (Spectre vs. code reuse) threat models to fully comprehend the attack surface of modern systems.

CCS CONCEPTS

• **Security and privacy → Operating systems security**.

KEYWORDS

Speculative execution; Code-reuse attacks

CCS '20, November 9–13, 2020, Virtual Event, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7089-9/20/11…$15.00
https://doi.org/10.1145/3372297.3417289

ACM Reference Format:
Enes Göktaş, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. 2020. Speculative Probing: Hacking Blind in the Spectre Era. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3372297.3417289

1 INTRODUCTION

Modern systems have a large and complex attack surface determined by vulnerabilities in software (e.g., buffer overflows) and hardware (e.g., Spectre). To handle such non-trivial complexity, security researchers often partition the problem space in a number of disjoint threat models and devise mitigations to reduce the attack surface accordingly. In this paper, we show that this strategy ignores important synergies between the threat models, overestimating the effectiveness of mitigations and the resulting attack surface reduction. In particular, we focus on synergies between code-reuse [11, 74, 78, 80, 87, 91] and Spectre [43, 53, 65] threat models and present *speculative probing* primitives as part of the joint attack surface. For code reuse, our primitives show speculative execution can reduce the requirements for exploitation (to as little as a single buffer overflow) even in the face of strong randomization schemes. For Spectre, our primitives show memory corruption provides new opportunities to craft Spectre gadgets even in the face of state-of-the-art mitigations. Combined, these insights enable end-to-end exploitation using a single memory corruption vulnerability despite all advanced mitigations in both threat models.

Hacking blind. Memory corruption vulnerabilities are the cornerstone of modern software exploitation. Nevertheless, a single memory corruption vulnerability alone is insufficient to mount practical attacks against today's systems hardened with widespread ASLR implementations [77], let alone against stronger leakage-resistant variants based on fine-grained code diversity and execute-only memory [13, 22, 23, 71]. Absent additional info-leak vulnerabilities that grant attackers arbitrary memory read primitives, attackers need to resort to probing primitives to hack blind. Traditionally, such primitives are used in BROP [10] or similar attacks [7, 30, 54, 79] to repeatedly probe the victim with controlled memory accesses. A major limitation of such attacks is that they trigger repeated, detection-prone crashes. They are also limited to crash-resistant victims, ruling out high-value targets like the kernel.

Side channels and Spectre. In the Spectre era, speculative execution vulnerabilities provide the attacker additional options to craft information disclosure primitives using side channels even in the absence of additional software vulnerabilities. Nonetheless, while Spectre [43, 53, 65] and other issues [15, 16, 61, 75, 94] are difficult to mitigate completely in hardware, the industry has rolled out effective remedies for most practical attacks, rendering the remaining attack surface exceedingly hard to exploit. In this paper, we show it is possible to resurrect Spectre-style speculative control-flow hijacking primitives in a classic software exploitation scenario even on modern hardened systems. Nevertheless, directly exploiting such primitives to craft fully fledged Spectre disclosure primitives faces exactly the same challenges of regular control-flow hijacking in the presence of arbitrary randomization schemes making the target gadget location in memory unpredictable.

BlindSide. We present BlindSide, a new exploitation technique at the convergence point of software and Spectre exploitation. BlindSide uses speculative execution to turn a single memory corruption vulnerability into powerful *speculative probing* primitives. These primitives leak information by observing microarchitectural side effects rather than architectural side effects such as crashes, bypassing strong leakage-resistant randomization defenses. The key idea of using a software vulnerability instead of indirect branch poisoning [53] or injection [90] also allows attackers to bypass all the deployed mitigations against speculative execution attacks.

Moreover, since crashes and the probe execution in general are suppressed on speculative paths, speculative probing cannot be detected by existing BROP-style defenses such as anomalous crash detection [35] and booby trapping [18, 23]. This allows blind attackers to stealthily probe for gadgets by speculatively executing them. For instance, we show attackers can use this strategy to blindly locate speculative arbitrary memory read gadgets. Such gadgets are already sufficient for code-oblivious code-reuse exploits [73, 91] and data-only exploits [44]. We further show the speculative nature of such arbitrary memory read gadgets allows them to directly read code even in presence of common software-based leakage-resistant randomization schemes [71], simplifying exploitation.

We demonstrate BlindSide attacks by means of a real-world buffer overflow vulnerability in the Linux kernel, a high-value, crash-sensitive target that so far remained well out of reach of BROP-style attacks. We use our single memory corruption vulnerability in a number of end-to-end kernel exploits, which implement speculative probing, collectively bypass a variety of randomization solutions (including the recent FGKASLR [4]) as well as version entropy (e.g., the Spectre gadget we blindly probe for is present in all the kernel versions in the past ≈ 5 years), and ultimately obtain full-system compromise. One of these exploits is the first data-only software exploit running entirely in speculative execution, leaking the root password hash from memory. We also consider possible defenses. As we have not been able to eliminate memory errors despite more than thirty years of research and speculative execution is essential to the performance of today's general-purpose CPUs, we argue that the mitigation of BlindSide attacks is difficult.

The convergence of software exploitation and speculative execution attacks generalizes both. In particular, while current speculative execution defenses focus on attacks poisoning microarchitectural components such as the Branch Target Buffer (BTB) [53], the Return Stack Buffer (RSB) [65], or data buffers [90] to steer the control flow speculatively, BlindSide generalizes such control-flow manipulation to include traditional memory corruption. Similarly, it generalizes BROP-style exploitation to include microarchitectural side effects to leak information about memory contents.

Contributions. The contributions of this paper are as follows:

- We investigate how speculative execution amplifies the severity of common software vulnerabilities such as memory corruption errors by introducing *speculative probing* primitives.
- We showcase our primitives in BlindSide attacks, with end-to-end exploits that start from a simple buffer overflow, speculatively leak data to derandomize the kernel address space, and ultimately achieve leakage of sensitive data or arbitrary code execution. The source code for the exploits and demo videos are available at https://vusec.net/projects/blindside.
- As an optimization of our attacks, we present the first cross-domain Spectre attack based on the efficient FLUSH+RELOAD covert channel through the kernel's physmap.
- We evaluate BlindSide against a variety of randomization solutions and Spectre mitigations and show that they are not effective.

2 BACKGROUND

2.1 Code-Reuse Attacks

Code-reuse attacks (CRAs) exploit memory corruption vulnerabilities, e.g., out-of-bound (OOB) writes, to control critical data such as a code pointer later used by the program. At that point, control flow is hijacked and redirected to a chain of *gadgets* (i.e., existing code fragments) that implement malicious payloads [11, 78]. In a privilege escalation scenario, attackers typically control (or exploit) an unprivileged application running on the victim machine and then use CRAs (or variations [44]) against the OS kernel.

To disrupt CRAs, KASLR in modern kernels randomizes the base address where code, data, and other memory areas are loaded at boot time. With KASLR, traditional exploitation attempts (usually) lead to kernel crashes. Successful kernel exploits now require an additional info-leak vulnerability to leak the base address of code, data, and even of certain objects storing the code-reuse payload [77]. However, even a limited (e.g., single function pointer) leak can reveal the location of all the other gadgets in the code.

To mitigate info leaks, efficient fine-grained randomization (FGR) schemes [4, 34, 57] randomize the code layout by re-ordering functions, basic blocks, or even the assignment of general-purpose registers [22, 23]. In response, researchers have devised so-called JIT-ROP attacks [80], which exploit info-leak vulnerabilities, to leak code, learn its layout, and craft a code-reuse payload just-in-time.

More recent *leakage-resistant* schemes implement execute-only memory (XoM) for the code region, using software instrumentation such as selective paging [6], pointer masking [13], and range checking [71] or a variety of hardware-based isolation features on commodity architectures [55], such as Intel MPX [71], MPK [40, 70], EPT [14, 22, 23, 32], split TLB [33], or ARM's MMU/MPU built-ins [19, 59]. While these schemes prevent all reads from code memory pages, they are still vulnerable to advanced code-reuse attacks that only rely on code pointers leaked from data memory [73, 91] and data-only attacks that do not even require code pointer corruption or control-flow hijacking at all [44].

2.2 "Blind" Code-Reuse Attacks

"Blind" CRAs do not rely on info-leak bugs to divulge the location of gadgets. Instead, they exploit the target application's crash resistance to probe its address space. For example, Shacham et al. [79]

use a return-to-libc probing attack against the Apache web server to disclose the location of libc. Essentially, they repeatedly corrupt the return address of a vulnerable function, forcing the program to return to every possible address in search of a libc target, while the server recovers from crashes by spawning a new process.

Similarly, BROP [10] demonstrated blind return-oriented programming (ROP) attacks, i.e., a just-in-time CRA utilizing gadgets ending in function returns. BROP blindly probes for certain types of (ROP) gadgets instead of whole functions, by observing signals like crashes, hangs and other behavior. CROP [30, 54] demonstrates similar attacks on crash-resistant *client* programs, using arbitrary memory read/write probes. However, where such attacks only apply to crash-resistant code, we target the *crash sensitive* kernel.

2.3 Cache Attacks

Cache attacks exploit timing side channels over shared CPU caches to detect victim memory accesses and leak information. Common variants are FLUSH+RELOAD [98] (F+R) and PRIME+PROBE [69] (P+P). F+R flushes a target shared cache line, waits for the victim to access it, reloads the cache line, and measures the latency. If the reload is fast (i.e., a cache hit), then the victim must have accessed the shared cache line. P+P can operate even without shared memory between attacker and victim since it detects accesses to a (shared) cache set. In P+P, attackers first build eviction sets [82, 95] (i.e., sets of memory addresses that map to the same cache set and with at least as many elements as the cache's associativity). Accessing an eviction set replaces all the cache lines in the corresponding cache set. Briefly, in modern architectures all the cache lines in an eviction set correspond to data at the same offset of their respective memory pages. We say that these cache lines, and the corresponding eviction set (and hence their pages), have the same *color*. Under P+P, attackers access an eviction set to prime the target cache set. After a potential victim operation, the target cache set is probed using the eviction set to measure the access latency. A slow probe (i.e., a cache miss) signals the victim's activity.

2.4 Speculative Execution Attacks

Modern CPUs execute instructions ahead-of-schedule to increase performance, e.g., to hide memory-access delays. This is done by predicting the outcome of control-flow decisions, which cannot be determined yet, and *speculatively executing* instructions based on these predictions. CPUs contain multiple predictors, both for conditional and indirect (i.e., pointer-based) branch instructions. Results produced during this *speculation window* are "parked" until the branch instruction completes, i.e., the instruction is retired. If the prediction fails, the CPU discards the parked results, leaving no trace in architectural state (e.g., registers and memory). However, speculative execution does leave observable results, or side effects, in the microarchitectural state of the processor (e.g., the cache), even if the instructions operated on privileged data [61].

This observation has led to numerous attacks [53, 61, 94], coined speculative (or transient) execution attacks, where a local attacker exfiltrates data from the kernel. The attacks massage the microarchitectural state (e.g., by manipulating the state of branch predictors) to force controlled speculative execution and run specific gadgets that access sensitive data. By carefully picking the gadgets, traces of that data are left in the microarchitectural state and can be exfiltrated using a cache attack such as P+P.

For instance, in a Spectre-BCB (v1) attack, attackers may pick kernel gadgets that perform speculative out-of-bounds accesses determined by a syscall-controlled value `x`:

```
if (x < array1_size)
        y = array2[array1[x] * 4096];
```

Attackers first force the if-protected statement to be speculatively executed (e.g., by training the branch predictor with a sequence of small values for `x`). They then provide an out-of-bound `x` to speculatively read an arbitrary value in the conditional branch and use it as an index into `array2`. Finally, they use a cache attack to infer the array index and leak the value of `array1[x]`.

Similarly, in a Spectre-BTB (v2) attack, attackers poison the Branch Target Buffer (BTB) to speculatively hijack indirect calls to a controlled target (e.g., by repeatedly issuing indirect branches to a colliding user address). By carefully picking a target gadget, attackers can again use a cache attack to exfiltrate the data.

Kernel mitigations against these attacks perform ad-hoc array index masking to thwart user-controllable speculative out-of-bounds accesses [21] and either prevent user-level indirect branch poisoning with hardware support [46] or stall indirect branch speculation using Retpolines [88]. None of these (and other) mitigations affects BlindSide, which (i) exploits conditional branch mispredictions (a la Spectre-BCB) but does not rely on (masked) user-controlled array gadgets; (ii) exploits speculative control-flow hijacking (a la Spectre-BTB) but does not rely on indirect branch mispredictions—based on poisoning the BTB or other buffers [90].

3 THREAT MODEL

We assume a realistic threat model with an attacker who is able to execute code on the target machine. We further assume that the attacker has access to a software vulnerability in the higher-privileged code that allows her to overwrite code pointers. The attacker's goal is to exploit the vulnerability to escalate privileges. While this scenario is common in browsers, OS kernels, or hypervisors, in this paper we mostly focus on a modern Linux kernel with all the mitigations enabled. To defend against software vulnerabilities, the Linux kernel enforces common mitigations against control-flow hijacking attacks such as DEP, stack canaries, and (possibly fine-grained and leakage-resistant) randomization. It also enforces SMEP, SMAP, and NX-physmap to prevent ret2usr [50] and ret2dir [49] attacks. To defend against fault-based speculative execution attacks, the Linux kernel enforces Kernel Page Table Isolation (KPTI) to mitigate Meltdown [61], encodes swapped page table entries to mitigate Foreshadow [89], and flushes microarchitectural buffers to mitigate MDS [94]. To defend against Spectre, the Linux kernel restricts indirect branch speculation or instead uses retpolines to mitigate speculative hijacking of code pointers [88]. It also utilizes array-index masking to mitigate unauthorized out-of-bound memory accesses [21].

4 SPECULATIVE PROBING

Starting from the ability to corrupt a code pointer, speculative probing relies on the ability to speculatively hijack the control flow to a controlled target in the victim (e.g., the kernel). On the surface, a victim code snippet that can be exploited for this purpose looks like a hybrid Spectre-BCB/BTB snippet:

```
if (expression) {
        /* ... */
        f_ptr(...);
        /* ... */
}
```

The if conditional branch allows the attacker to control speculative execution, while the indirect call via the f_ptr function pointer enables speculative control-flow hijacking. In contrast to Spectre-BTB's speculative hijacks (caused by indirect branch misprediction), we hijack the control flow by corrupting f_ptr with a software vulnerability similar to plain code reuse. However, unlike real code reuse, we control speculative execution via the conditional branch to ensure the corrupted pointer is only dereferenced on a speculative (later-aborted) path, never in "real" execution.

Note that, since modern microarchitectures support multiple levels of speculation [66], the indirect call will also speculate on its own, but that is irrelevant for the purpose of speculative probing. When the if-induced speculative execution reaches the indirect call, the CPU starts a second-level speculative execution window due to indirect branch target prediction. But, at the end of that window, the execution goes back to first-level speculation where the "real" (but corrupted) function pointer gets executed. Defenses like Retpoline [88] can only cripple the second-level speculation here, which only makes our speculative control-flow hijack more robust by removing unnecessary speculative instructions.

In essence, to implement speculative probing, an attacker needs to deviate only slightly from a typical code-reuse workflow. After exploiting a memory corruption vulnerability to corrupt one or more function pointers, the attacker needs to pick one that is called within the speculative execution window of a conditional branch that controls its execution. Since modern processors support speculative execution windows of hundreds of instructions [53], this is often straightforward. For instance, as detailed later, over 90% of kernel indirect branches are 10 or less instructions away from a conditional branch that controls their execution. To control the prediction at a conditional branch, attackers have many different options. They can either prime the PHT part of the BTB [27] shared between user and kernel [26], tweak the input leading up to the function pointer dereference to train the dynamic branch predictor [53], or directly corrupt the data conditional using the software vulnerability to cause the CPU to take (or skip) the branch. The latter option was the simplest to use in our exploits.

In practice, the extra effort required by speculative probing compared to plain code reuse is relatively low. For instance, after locating the function pointers that we could corrupt with a given vulnerability and potential candidates for indirect call sites (as done for code reuse), it only took us a few hours to select the ideal call site for our speculative probing exploits discussed in Section 6.

5 SPECULATIVE PROBING PRIMITIVES

Starting from a speculative control-flow hijacking snippet, attackers can repeatedly hijack speculative execution to controlled targets and craft a variety of speculative probing primitives tailored to each specific exploitation scenario. For instance, in a classic KASLR code-reuse scenario we need specialized probing primitives to locate the base address of code, heap/physmap (i.e., the memory area where modern kernels map practically all physical memory in a direct mapping), and the heap object storing the code-reuse payload [49]. On the other hand, in face of fine-grained randomization, we need more general, albeit less efficient, arbitrary memory read primitives [73, 91], which in our case we implement speculatively.

We distinguish between *Stage 1* and *Stage 2* primitives, where Stage 1 denotes fundamental probing primitives that can be blindly used without any *a priori* knowledge of the code location/layout, while Stage 2 primitives use Stage 1 primitives to find gadgets targeting specific exploitation scenarios. Primitives to find executable pages (code region probing) and gadgets (gadget probing) are in the former category, while example primitives we use to find the region containing heap/physmap (data region probing), target objects inside the heap (object probing), or arbitrary memory content using a Spectre gadget (Spectre probing), are all in the latter.

All these primitives use the same underlying mechanism: they probe the address space by corrupting the chosen function pointer subsequently dereferenced during speculative execution. Afterwards, we mount a last-level cache (LLC) P+P attack (or, as detailed later, F+R when possible, as an optimization) to detect cache traces of our targets (e.g., code fragments and/or data regions) left by speculative execution of the corrupted function pointer. In the following, we discuss the Stage 1 and Stage 2 primitives using Figure 1.

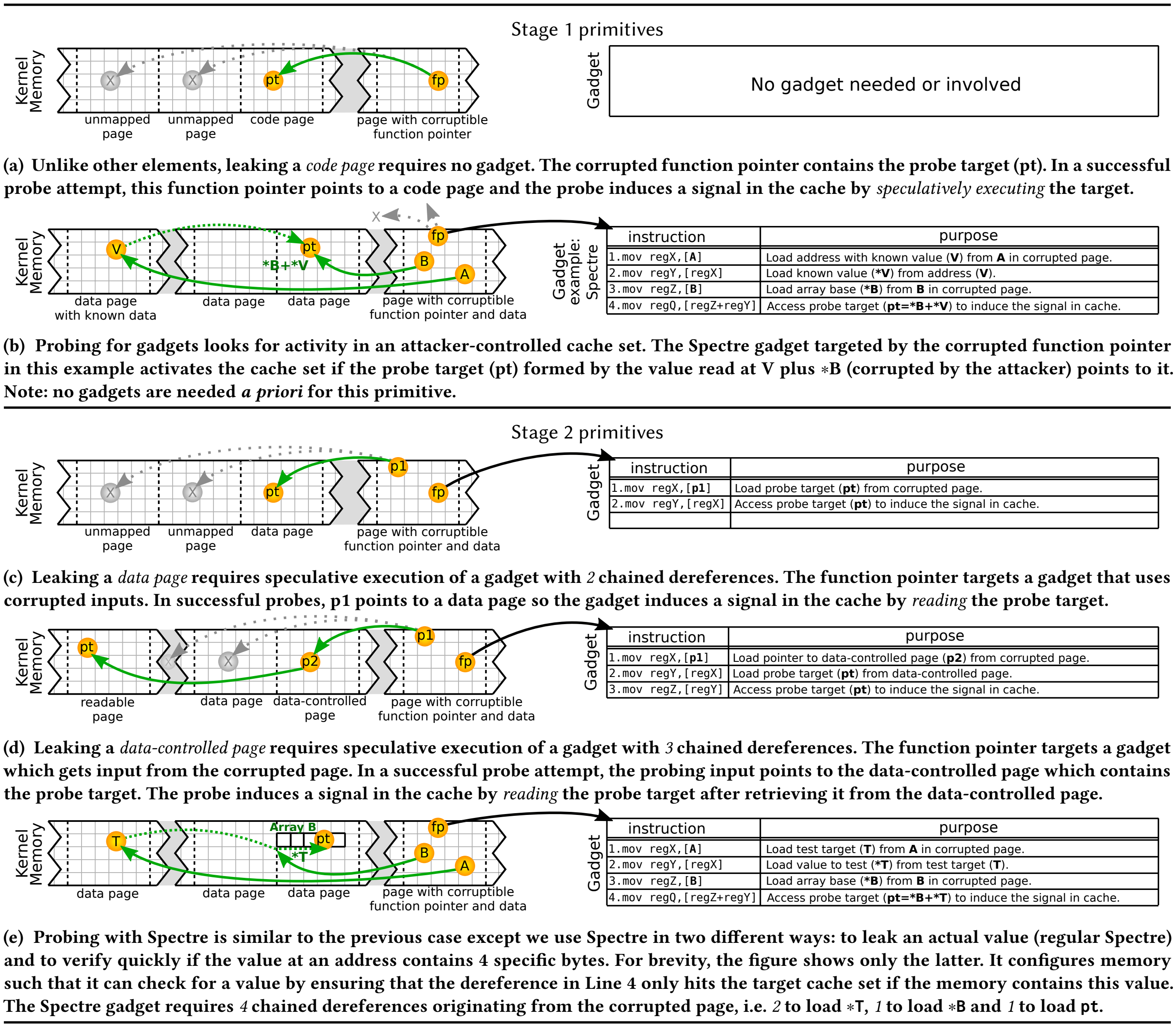
5.1 Code Region Probing

In any code-reuse attack, the first step is to identify the location of code regions in memory. In the presence of coarse-grained KASLR, finding the base of the kernel code is already sufficient to disclose the predictable location of all the necessary gadgets.

As we see in Figure 1a, probing for code consists of several steps. First, the attacker uses the software vulnerability to overwrite a victim code pointer. The next step is to train the CPU's predictor to dereference the corrupted code pointer *speculatively* next time the kernel code executes. Then the attacker primes (fills) part of the cache with an eviction set. After these preparatory steps, the attacker issues a syscall to speculatively hijack the control flow to a desired location. Even if the location is invalid or not executable, there will be no crashes, since the speculative execution will mask all exceptions. However, if the target location contains (arbitrary, even invalid) code on an executable page, the executed code speculatively fills the corresponding cache lines. By subsequently probing the matching cache sets with P+P, the attacker determines if the address is in the cache (and thus if the chosen target page is executable).

5.2 Gadget Probing

In the presence of fine-grained and possibly leakage-resistant randomization, code region probing alone is insufficient to find all the necessary gadgets. Instead, we need to blindly locate specific gadgets in the randomized code region. Gadgets of interest include traditional code-reuse fragments as well as speculative execution (e.g., Spectre) gadgets. As before, we use a speculative probing primitive, but this time we look for specific signals in the cache. While



(a) Unlike other elements, leaking a *code page* requires no gadget. The corrupted function pointer contains the probe target (pt). In a successful probe attempt, this function pointer points to a code page and the probe induces a signal in the cache by *speculatively executing* the target.

Kernel Memory
V
X
*B+*V
pt
fp
B
A
data page with known data
data page
data page
page with corruptible function pointer and data
Gadget example: Spectre

instruction	purpose
1.mov regX,[**A**]	Load address with known value (**V**) from **A** in corrupted page.
2.mov regY,[regX]	Load known value (***V**) from address (**V**).
3.mov regZ,[**B**]	Load array base (***B**) from **B** in corrupted page.
4.mov regQ,[regZ+regY]	Access probe target (**pt=*B+*V**) to induce the signal in cache.

(b) Probing for gadgets looks for activity in an attacker-controlled cache set. The Spectre gadget targeted by the corrupted function pointer in this example activates the cache set if the probe target (pt) formed by the value read at V plus ∗B (corrupted by the attacker) points to it. Note: no gadgets are needed *a priori* for this primitive.

instruction	purpose
1.mov regX,[**p1**]	Load probe target (**pt**) from corrupted page.
2.mov regY,[regX]	Access probe target (**pt**) to induce the signal in cache.

(c) Leaking a *data page* requires speculative execution of a gadget with *2* chained dereferences. The function pointer targets a gadget that uses corrupted inputs. In successful probes, p1 points to a data page so the gadget induces a signal in the cache by *reading* the probe target.

instruction	purpose
1.mov regX,[**p1**]	Load pointer to data-controlled page (**p2**) from corrupted page.
2.mov regY,[regX]	Load probe target (**pt**) from data-controlled page.
3.mov regZ,[regY]	Access probe target (**pt**) to induce the signal in cache.

(d) Leaking a *data-controlled page* requires speculative execution of a gadget with *3* chained dereferences. The function pointer targets a gadget which gets input from the corrupted page. In a successful probe attempt, the probing input points to the data-controlled page which contains the probe target. The probe induces a signal in the cache by *reading* the probe target after retrieving it from the data-controlled page.

instruction	purpose
1.mov regX,[**A**]	Load test target (**T**) from **A** in corrupted page.
2.mov regY,[regX]	Load value to test (***T**) from test target (**T**).
3.mov regZ,[**B**]	Load array base (***B**) from **B** in corrupted page.
4.mov regQ,[regZ+regY]	Access probe target (**pt=*B+*T**) to induce the signal in cache.

(e) Probing with Spectre is similar to the previous case except we use Spectre in two different ways: to leak an actual value (regular Spectre) and to verify quickly if the value at an address contains 4 specific bytes. For brevity, the figure shows only the latter. It configures memory such that it can check for a value by ensuring that the dereference in Line 4 only hits the target cache set if the memory contains this value. The Spectre gadget requires *4* chained dereferences originating from the corrupted page, i.e. *2* to load ∗T, *1* to load ∗B and *1* to load pt.

Figure 1: Probing primitives—green arrows represent successful probe attempts and gray dotted ones the unsuccessful ones.

the cache behavior of code fragments may be quite diverse, we can optimize the search by limiting ourselves to gadgets (or their neighbors [81]) that announce their presence using an easily detectable signal. For instance, we may target a gadget dereferencing a pointer that the attacker controls and check if the corresponding cache set gets activated. Gadget probing is illustrated in Figure 1b.

In principle, the attacker can observe arbitrary microarchitectural side effects due to accesses to code pages, data pages, and combinations thereof, but the current BlindSide attack targets cache behavior that attackers can observe both efficiently and reliably. In particular, we focus on gadgets for which successful execution activates *one* particular cache set that is under control of the attacker (and as noise free as possible).

As an example, suppose we want to probe for a Spectre gadget as shown in Figure 1b. In this case, the buffer overflow overwrites an object that also contains a function pointer. The values overwritten by the overflow are controlled by the attacker. Since we control the values that the target Spectre gadget consumes, we can configure those values in a way that they leave a signal in an expected cache set. We now start probing different code locations until we observe a signal that indicates a successful detection of a Spectre gadget.

Note that the only vulnerability-specific aspects here are the registers that point to the overwritten memory and the size of the buffer overflow, but BlindSide is agnostic to both: as long as it sees activity in the target cache set, it knows that it has found an appropriate gadget. For this reason, our current gadget probing implementation focuses on gadgets for which the correct behavior culminates in (and can be verified by) the activation of a single cache set selected by the attacker. However, this is not a strict requirement and more elaborate fingerprinting is possible (but unnecessary for practical exploitation, as shown by our end-to-end exploits).

For different gadgets, the detection of such behavior may take slightly different forms. For instance, suppose we are looking for a traditional (B)ROP gadget such as `pop reg; ret;` for some register. In that case, there is no direct reference to the target cache set by the gadget. To detect such gadgets, the attacker can look at the callsite for the *use* of the register upon successful completion of the gadget. After all, when code dereferences the register after the return, the appropriate cache set gets activated. In other words, it does not matter how the code activates the selected cache set, as long as one can infer the behavior of the target gadget from it.

Finally, it is helpful to discuss the usefulness of gadget probing in general *even if* it can leak a Spectre gadget already. Given a Spectre gadget, the attacker can probe the address space more directly. However, while the Spectre gadget is convenient for exploitation, as we discuss in Section 6.5, it is unable to bypass certain leakage-resistant randomization schemes. Gadget probing is not subject to these limitations, but the analysis of cache traces for each necessary gadget requires additional effort on behalf of the attacker.

5.3 Data Region Probing

While the Stage 1 primitives give attackers all that is needed to launch an exploit, solely relying on Stage 1 may not be efficient. For instance, probing the entire address space with the Spectre gadget is slow as each value that the gadget reads requires probing many cache sets (some of which may be quite noisy), even when the attackers do not care about the actual value. As an example, it is common for exploits to require the base address of a data region like heap/physmap, regardless of its content (see the exploits in §6).

For this scenario, Figure 1c shows how data region probing allows an attacker to find the kernel heap efficiently. In this case, we use a gadget that accesses memory via two chained dereferences. The gadget uses an attacker-controlled value on the corrupted page as a pointer to load another value from a target page. To verify that the target page is indeed mapped as a data page, we only need to check the cache sets at the attacker-controlled page offset. If one of these cache sets gets activated, then we know that the probe has succeeded in finding a mapped kernel data page.

5.4 Object Probing

Merely locating the base address of a data region is not always sufficient. For instance, some attacks [49] require the location of specific user objects in the physmap. Moreover, as later detailed in Section 5.6, locating user objects in the physmap is useful to build a F+R [98] covert channel as a better alternative to P+P [69].

To conveniently accommodate such exploitation techniques, object probing allows attackers to scan memory for pointer signatures: pointers to a probe target of which the attacker knows the cache set. The procedure is shown in Figure 1d. The corrupted function pointer targets a gadget that uses an attacker-controlled value as a pointer to another pointer `p2` that it subsequently dereferences. By checking the cache set corresponding to `pt`, attackers can tell if they found the address of the right object containing `p2`.

5.5 Spectre Probing

The most convenient primitive is given by a Spectre gadget that we can use to scan the *content* of memory directly. The Spectre gadget serves as a universal read primitive, as we can use it to dump the content of any memory region. For instance, we could use it to dump the full contents of the kernel code and data regions. Spectre probing could act as an alternative to object probing for locating data in physmap by leaking memory contents byte by byte (leaking mode). However, doing so is slow as each value needs explicit testing which requires probing a range of cache sets to see which one was activated. Moreover, some of the cache sets may be used a lot, leading to increased noise that slows down the attack.

For this reason, we can also efficiently use our Spectre probing in value testing mode, as illustrated in Figure 1e. In this case, the corrupted function pointer targets a Spectre gadget and the attacker configures memory such that the dereference in Line 4 hits a particular cache set if and only if the value that the Spectre gadget reads has the value that the attacker is looking for. By doing so, BlindSide greatly reduces the number of cache sets to probe during a scan. Note that the technique applies to both data and code pages. As we discuss in Section 6.5, certain mitigations, however, are immune against Spectre probing when leaking code pages. The attacker can instead use gadget probing in those circumstances.

5.6 Optimizations

Reducing Noise. During our P+P measurements for examining the state of the LLC, some cache sets always get accessed due to the code executed and data accessed by the measurement itself. These cache sets may conflict with the eviction sets that we use for checking the probe's signal. The eviction sets associated with the accessed cache sets will always result in a *slow* probe which would falsely imply that it has the signal (i.e., a false positive). To learn the cache sets that are accessed by default, we collect a footprint of the cache by performing one round, using a *void* probe target (i.e., memory address 0x0) before the actual probing starts. We avoid these cache sets when probing our target address.

Cache attacks are noisy by nature, so once we find a signal, we need to verify it is a true positive. For verification, we adjust the offset in the probe target to another cache set. If that cache set also appears to show a signal, it means the signal was a true positive and that the probe target points to the sought element in memory.

Leveraging FLUSH+RELOAD. As P+P is known to be slow and sensitive to noise, replacing it with the faster and more noise-resistant FLUSH+RELOAD [98] attack is beneficial for the probes. F+R achieves its speed up mainly by allowing a lower number of measurement repetitions per probe and having a high signal confidence on a single hit, unlike P+P which requires more hits to validate the signal. Appendix A presents a detailed comparison between P+P and F+R used with our primitives.

However, unlike P+P, F+R requires the attacker and victim to share memory. Observe that the kernel heap (or `physmap`) is implicitly "shared" between the user process and the kernel and can be used to build an efficient F+R covert channel. Specifically, the attacker can map a F+R buffer in user memory backed by 2 MB huge pages and put a signature in the beginning. To locate kernel mapping of such buffer, the attacker relies on Spectre probing to scan the physmap for the signature at 2 MB intervals. After this step, the attacker can use Spectre probing again to access the buffer via its kernel mapping, but now perform F+R (instead of P+P) using the user mapping to leak information. If huge pages are not available, the attacker can rely on side channels to detect a 2 MB user memory alignment [28, 47] or resort to spraying 4 KB pages with a unique page id attached to the signature to reduce the search space.

Our results show that F+R improves the speed of the probes on average more than 5x, which is in line with numbers reported in the literature [93]. As we shall see next, we use F+R in two of our three exploits after leaking the kernel heap and the user page within it.

6 EXPLOITATION

In this section, we present three proof-of-concept (PoC) exploits. The first exploit uses our Stage 1 code region probing primitive to bypass standard code KASLR, our Stage 2 data region probing to bypass heap KASLR, and finally our Stage 2 object probing primitive to detect the location of our ROP payload. This allows us to mount an end-to-end just-in-time code-reuse exploit and gain reliable code execution in the kernel using a single heap buffer overflow vulnerability. The second exploit first uses our Stage 1 code region probing and gadget probing to find a Stage 2 Spectre probing primitive to leak arbitrary information from the victim kernel's data region. We use this primitive to mount an *architectural* end-to-end data-only exploit using a *microarchitectural* speculative code-reuse exploit, which, as an example, leaks the root password hash. The exploit structurally bypasses fine-grained, leakage-resistant randomization and other mitigations against (architectural) code reuse such as CFI [3] which have been deployed in secure production kernels [2]. Our last exploit shows how the Spectre probing primitive can be more powerful than traditional arbitrary memory read primitives, demonstrating how it can directly read code and enable (architectural) just-in-time code reuse in the face of software-based eXecute-only-Memory (XoM) for the kernel [71].

The ultimate goal of the exploits is elevating privileges by executing a ROP payload crafted with the disclosed gadgets to disable the SMAP and SMEP protections and allow user-space code to change the process' credentials, or by compromising the root password. First, we briefly discuss the vulnerability and shared initialization of the exploits, then we go over how we used the probing primitives, and finally we discuss how we achieve privilege escalation in the final stage. We perform the attacks against Linux kernel version 4.8.0 compiled with gcc and all mitigations enabled on a machine with Intel(R) Xeon(R) CPU E3-1270 v6 @ 3.80GHz and 16 GB of RAM. We repeat all our experiments 5 times and report the median, with marginal deviations across runs. In each experiment, we set the number of probing repetitions and hits to the minimum number necessary to achieve a 100% success rate (0% error rate) in our repeated attempts on an idle system. See Appendix A for more details on the impact of repetitions on our probing primitives.

6.1 Vulnerability

For our exploits, we use a heap buffer overflow in the Linux kernel (CVE-2017-7308). This bug applies to AF_PACKET sockets with a TPACKET_V3 ring buffer. We used Konovalov's detailed write up on this vulnerability [56] to start off our exploits.

In the original exploit, once the vulnerable ring buffer is initialized, only a fixed offset beyond the buffer can be overwritten. In our exploits, we create two such vulnerable objects. The first object serves to corrupt (adjust) the fixed write offset stored in the second. This results in a non-linear out-of-bound write through the second object with a range of up to 64 KB (due to the *offset* being of type `unsigned short`). For details about how the out-of-bound write is triggered, we point the interested readers to Konovalov's write-up.

Note that BlindSide can work with any vulnerability that provides a write primitive similar to the one used here. Examples include CVE-2017-1000112, CVE-2017-7294, and CVE-2018-5332.

6.2 Speculative Probing Initialization

For speculative probing, we exploit a conditional branch and an indirect branch combination in the code related to sockets. We place a socket object adjacent to the out-of-bound write primitive and corrupt its function pointer consumed by the indirect branch for probing. We trigger the execution of the conditional and indirect branch combination using a `sendto` system call.

To ensure that the conditional branch is taken towards the indirect branch by default, we prepare a non-corrupted socket object for the purpose of training the execution of the conditional branch towards the indirect branch. To trigger speculation, we flip the direction of the conditional branch by simply corrupting the conditional data using the out-of-bound write vulnerability. To ensure that speculation succeeds in reaching our target indirect branch, we spawn a thread on a separate core to constantly evict the conditional data from the cache and maximize the speculation window.

6.3 Exploit 1: Breaking Coarse-grained KASLR

In our first exploit, we focus on applying BlindSide to the stock Linux kernel with default mitigations including KASLR.

Locating kernel image. To discover the base of the kernel image (i.e., code and adjacent data), we perform code region probing on memory range `0xffffffff80000000` - `0xffffffffc0000000` (1 GB) with a step size of 8 MB. The kernel image size is a little over 8 MB. Once we get a hit, we lower the step size to 2 MB and restart probing from the last unmapped page. Note that the kernel image is mapped with huge pages and thus aligned to 2 MB. Once we know the base of the kernel image, we know the location of all gadgets.

Results. While searching for an executable page, we measured a probing speed of 95.4 pages per second with 14 repetitions per cache set. On average, it takes around 0.7s to find the kernel image base (i.e., on average located in the middle of the possible range).

Locating the kernel heap. To build the ROP payload, we need to leak its location in memory in order to use payload pointers inside the payload. We first use data region probing to locate the kernel heap and then use object probing starting from the base of the heap. We use the following gadget in both probes:

```
0x146a3: mov rax, qword ptr [rbx + 0x158]
0x146aa: mov rax, qword ptr [rax + 0x138]
0x146b1: mov rax, qword ptr [rax + 0x78]
```

Listing 1: Gadget in `uncore_pmu_event_start` and at kernel image offset 0x146a3.

For data region probing, we use the first two instructions and, for object probing, we use all three instructions. The rbx register points to the socket object which we corrupt for speculative probing. We probe for the heap base in memory range `0xffff880000000000` - `0xffffa40000000000` (i.e., a 28 TB memory range which we found empirically). Ideally we would use a 16 GB step size, but we noticed an unmapped gap of 1GB in the heap. To avoid such gaps, we instead use a step size of 8GB (and 1GB on the slow path).

Results. While searching for a data page, we measured a probing speed of 36.4 pages per second with 36 repetitions per cache set. On average, it takes around 49.2s to find the heap base (i.e., on average located in the middle of the possible range).

Locating the ROP payload. Once we find the heap base, we use object probing to find the ROP payload's location. Essentially, we search for the location where we have the out-of-bound write capabilities as we write the ROP payload at that location. We start the probe at the discovered heap base and use a step size of `0x8000` bytes as the vulnerable buffer used for the out-of-bound write is aligned to `0x8000`. Once we observe a signal through our object probing primitive, it means that we have disclosed the location of the target ROP payload.

Results. While searching for the target location, we measured a probing speed of 3,910.8 pages per second on average with 43 repetitions per cache set. On average, it takes around 67.0s to find our target object if it is located in the middle of the heap.

6.4 Exploit 2: Speculative Data-only Attacks

In our second exploit, we assume state-of-the-art mitigations against code reuse and speculative execution to be enabled. In this exploitation scenario, starting after the code region probing step detailed earlier, we use our gadget probing primitive to find a Stage 2 Spectre probing primitive.

Locating a Spectre gadget. We pick the following out-of-band Spectre gadget to be probed using our gadget probing:

```
0x4f8990: // function prologue
   ...
0x4f89a4: mov  r13, rdi
0x4f89a7: push r12
0x4f89a9: push rbx
0x4f89aa: mov  r12, qword ptr [rdi + 0x2f8]
0x4f89b1: mov  rbx, qword ptr [r12]
0x4f89b5: cmp  r12, r14
0x4f89b8: je   0xffffffff88cf8a0f
0x4f89ba: cmp  byte ptr [r13 + 0x3b0], 0
0x4f89c2: mov  esi, dword ptr [r12 + 0x28]
0x4f89c7: je   0xffffffff88cf89f7
0x4f89c9: mov  rdx, qword ptr [r13 + 0x380]
0x4f89d0: mov  eax, esi
0x4f89d2: mov  rax, qword ptr [rdx + rax*8]
```

Listing 2: Gadget in `vp_del_vqs` and at kernel image offset 0x4f89a4. The rdi register points to the packet socket object.

While probing for this gadget, we arrange the memory at the corrupted function pointer such that, when our probe targets this gadget, the instruction at 0x4f89d2 leaves a signal in an expected cache set. Data to be leaked (i.e., `rax`) is added to an array pointer (i.e., `rdx`) and then the resulting pointer is dereferenced. Note that the data pointer (i.e., `r12`) and the array pointer are both loaded from the non-linear out-of-bound write region using the `rdi` register. This means we only need to provide the gadget with valid pointers to dereference in order to get a signal in our target cache set. Next we look at two important aspects of our Spectre gadget before discussing two optimizations to speed up the execution of the exploit using this gadget.

Bypassing mitigations. Note that our target gadget is resistant to both Spectre (since it is out-of-band and not protected by array index masking) and randomization mitigations. In particular, since this gadget does not feature function calls or branching code, it is resistant to function-level and basic block-level randomization by construction. We also experimentally confirmed our gadget is resilient to FGKASLR [4]—a recent fine-grained function-level randomization scheme proposed by Intel and currently being considered for the mainline Linux kernel.

Moreover, all the gadget's required inputs are derived from the `rdi` register which cannot be randomized with register-level randomization since it is an argument (not a general-purpose) register [23]. Hence, our Spectre gadget has no internal entropy and we can probe for it even with strong fine-grained and leakage-resistant randomization. Furthermore, since this piece of code is not expected to process user-provided input, it is not guarded against speculative execution attacks using e.g., `lfence` or array index masking.

The gadget's longevity. Notably, we found our target Spectre gadget is available from Linux kernel v3.19 until v5.8 (i.e., the most recent version at the the time of writing), surviving 31 major Linux kernel releases across over 5 years. This shows an attacker armed with a write vulnerability can perform BlindSide attacks on a wide range of recent production Linux kernel versions even when blind to the particular kernel version.

Optimization: single cache set. Since we have only leaked the kernel image location so far, we can only provide pointers to the kernel image and not the heap. We use pointers to enum constants to be used as data pointers and a pointer to the kernel image as the array pointer. By using a code page as the array pointer, we are able to distinguish the color of the page through code region probing. Discovering the color of the array allows us to check for a signal in only one cache set out of the many that map to different colors.

Optimization: function alignment. Because the gadget still gives a signal when executed from the function entry point, we used a step size of 16 bytes (i.e., function entry point alignment).

Results. While searching for the Spectre gadget, we measured a probing speed using P+P of 3,650.4 code locations per second with 44 repetitions per cache set. On average, it takes around 76.7s to find the gadget (i.e., on average located in the middle of the code).

Enabling FLUSH+RELOAD. After leaking the kernel heap base similar to Exploit 1, we probe for a mapped user page with a signature in the physmap with a step size of 2 MB to enable F+R. We measured a probing speed of 3,658.0 pages per second with 44 repetitions per cache set. On average, it takes around 1.1s to find the target user page (i.e., on average located located in the middle of the physmap).

Leaking the root password hash with Spectre probing. assuming strong mitigations against architectural code-reuse attacks, we show how one can still leak sensitive information using Spectre probing. As an example, we aim to leak the root password hash in a data-only attack.

We force the system to load the contents of the /etc/shadow file into the page cache by performing an unsuccessful authentication using sudo, similar in spirit to prior hardware-based attacks [72, 94]. The memory page that stores the contents of /etc/shadow file starts with the root:$ prefix. We use Spectre probing to leak the first 4 bytes of each 4 KB page and in case of match with 'root', we verify the hit by also checking the 4 bytes 'ot:$' at page offset 2. Upon a match, we continue and leak the root password hash.

Results. While searching for the root:\$ snippet, we measured a probing speed using F+R with Spectre probing of 19,520.5 pages per second with 8 repetitions per cache line, looking for 1 hit in the target cache line. On average, it takes around 107.4s to find the snippet assuming that it is located in the middle of the kernel heap.

Cracking the root password hash. Assuming a default SHA-512 root password hash on Linux, a 60 node GPU cluster can brute-force an eight character alphanumeric password in roughly one hour [31]. On Amazon EC2 [1], this would cost less than $ 32.

6.5 Exploit 3: Breaking Software-based XoM

Our gadget probing primitive can leak gadgets regardless of the deployed randomization technique. The target gadgets, however, need to leave an observable trace in the LLC. Furthermore, analyzing the suitability of each gadget for gadget probing can be burdensome: as an example, our ROP chain requires eight gadgets for successful exploitation. In our last exploit, we show that our Spectre probing primitive provides a powerful arbitrary memory read primitive that can even *speculatively read* code and bypass mitigations.

We simply aim our Spectre probing to the kernel image location to leak the code contents. To our surprise, this bypasses software-based XoM techniques for the kernel [71] by simply reading code blocks that are protected by code randomization. In particular, this simple strategy trivially bypasses the software-based range checks (skipped in nested speculative execution) proposed in [71] even when they are enhanced by hardware support (i.e., Intel MPX, whose bounds checks are also deferred in speculative execution). Other software-based implementations such as pointer masking [55] can also be bypassed with the right gadgets (i.e., by skipping over the mask operation), but we decided against complicating our exploit since pointer masking is anyway difficult to support in the kernel's non-linear address space [71].

Our investigation also shows that execute-only memory defenses that rely on hardware-enforced permission checks such as EPT [14, 32] are protected against Spectre probing. This is due to the fact that speculative execution does not load data from the cache lines that are marked as execute-only by EPT.

Dealing with aliasing. While leaking the entire kernel code, we encountered multiple issues at certain memory addresses due to address aliasing handling in modern CPUs. An example was an aliasing issue caused by a stack store instruction at the beginning of the Spectre gadget. When the given load address to leak from happened to 4k-alias the address of the earlier stack store instruction, a stall introduced by the store-to-load forwarding logic [67, 83] disrupted the signal. To address this issue, an option is to chain together multiple speculative gadgets [8] and perform stack pivoting before executing the Spectre gadget. We confirmed this strategy eliminates the issue, but also requires blindly probing for another gadget. To lift this requirement, we opted for a simpler approach, namely having the PoC switch to the legacy int 0x80 syscall interface to misalign the kernel stack (compared to the regular syscall interface) when needed. Another example was an aliasing issue caused by a lock-prefixed load instruction in the vulnerable code path disrupting the signal when leaking from the same page offset. To address this issue, we relied on multiple vulnerable objects with different addresses for the lock-prefixed load instruction. By applying these and other aliasing remedies, we were able to leak all but 4 of the 8,961,112 kernel code bytes (due to residual aliasing issues). To recover the missing 4 bytes, rather than further complicating the exploit, one can simply perform disassembly and mount a straightforward code inference attack [81].

Results. After probing for a mapped user page to enable F+R similar to Exploit 2, we dumped the entire kernel code. We measured a leakage speed using F+R with Spectre probing of 2,645.7 bytes per second with 7 repetitions per cache line. This resulted in leaking the entire kernel code in around 56 minutes.

6.6 Exploit Finalization

We finalize the exploits by escalating privileges to root. For Exploit 2, we can simply use the cracked root password. For Exploit 1 and Exploit 3, we trigger the control-flow hijack in regular (non-speculative) execution, diverting to a ROP chain with 8 gadgets disclosed from the code region. The ROP chain disables SMAP/SMEP and finally diverts execution to user memory a la ret2usr [50]. Executing directly in user space releases the attacker from the complexities of a ROP attack. The user-space code essentially updates the credentials of the controlled process to root as follows:

```
commit_creds(prepare_kernel_cred(0));
```

Listing 3: Code snippet updating process credentials to root.

To build the ROP chain, we use the disclosed ROP payload location as a way to move a value from one register to another since we miss a convenient gadget that does this specifically for the rax-rdi transfer. Essentially, we need to move an updated control register value from rax into rdi, which we then move to the CR4 control register to disable SMAP/SMEP. We achieve the transfer by writing the value in rax back into the ROP payload and then popping it again into rdi. During the just-in-time ROP payload preparation, we use the payload's disclosed location to prepare a pointer in the payload that points to the ROP payload location where a `pop rdi; ret' gadget pops from.

After privilege escalation, the user-space code restores the kernel stack pointer and returns to the hijacked indirect branch to continue normal execution instead of instantly context switching to the user-space using an iret instruction. Resuming normal execution from the hijacked indirect branch ensures that locked resources are released.

7 DETAILED ANALYSIS

We have so far evaluated the throughput of our probing primitives and the time to reliably complete the corresponding exploitation steps. In this section, we present additional experiments to show (i) we can effectively exploit kernel indirect branches to implement our speculative control-flow hijacking building block for blind probing (Stage 1) and (ii) exploit disclosed kernel code to implement

```
1 if(flush_fp) clflush(obj->fp);
2 clflush(obj->fp_enabled);
3 mfence();
4 if (obj->fp_enabled)
5   obj->fp(obj, offset); // => obj->array[(offset+FID)*512]
```

Listing 4: Code snippet in our kernel module. The comment illustrates the body of the targeted function. FID is a hardcoded function id, distinct for each function.

usable gadgets for more informed probing (Stage 2). We refer the interested reader to Appendix A for a detailed analysis on the impact of the number of repetitions on the success rate of our speculative probing primitives.

For our gadget analysis, we used the Capstone (v4.0.1) disassembler and statically analyzed the vulnerable Linux kernel version 4.8.0 used for our proof-of-concept exploits. To find potentially exploitable indirect branches on the same kernel version, we used the IdaPro (v7.2) interactive disassembler. We preferred IdaPro over Capstone for this analysis as we performed backward analysis from the indirect branches which required the cross-reference information added by IdaPro.

To verify that the identified indirect branches, and speculative probing in general, are not hindered by state-of-the-art mitigations against speculative execution attacks, we tested a recent (non-vulnerable) Linux kernel version 5.3.0-40-generic with all the mitigations (e.g., Retpoline) enabled on an Intel i7-8565U CPU with the microcode update for the IBPB, IBRS and STIBP mitigations.

7.1 Mitigation Resistance

We evaluate speculative probing's ability to bypass mitigations that explicitly seek to prevent speculative control-flow hijacking: Retpoline, IBPB, IBRS and STIBP. For this purpose, we create a kernel module with an indirect branch guarded by a conditional branch, both controlled by mock heap objects (see Listing 4).

For each test, we create two objects of the same type, each pointing to a different function through its `fp` pointer. When called, these functions leave a unique and easily measurable fingerprint in the cache. In the first object, `fp_enabled` is set to 1 to train the branch predictor towards calling `fp` on line 5. In the second it is set to 0, so that the indirect branch is only reached speculatively—which is facilitated by the `clflush` and `mfence` operations. We use the first (training) object five times, followed by one run with the second.

For the experiment, we perform 1,000 iterations per configuration, where each iteration consists of 10,000 tests as described above. After each test, we probe the cache for hits that reveal which function, if any, was speculatively executed. In each configuration, we apply mitigations individually, with and without flushing the object's function pointer—to verify that the mitigations work correctly by nudging speculative execution towards the training function. Finally, we include a test with two threads, where each thread continuously uses one of the objects. This aims to test whether IBRS and STIBP prevent indirect branch-target poisoning across logical CPU cores. Table 1 shows our results.

As expected, the results show that the mitigations prevent speculative execution of the training function, with (close to) 0% success rates for all mitigations. However, the CPU did speculatively execute the indirect branch and its target function in many cases, reaching (close to) 100% success rates across all mitigations.

Defense	Flush FP	Target Function Success (Avg. Hits)	Training Function Success (Avg. Hits)
		Single Thread Executions:	
None	No	100.0% (9999.93)	43.3% (0.88)
	Yes	100.0% (208.37)	100.0% (9999.96)
Retpoline	No	100.0% (9990.92)	0.0% (0.00)
	Yes	100.0% (164.71)	0.0% (0.00)
IBPB	No	100.0% (9999.62)	0.0% (0.00)
	Yes	100.0% (292.32)	0.0% (0.00)
		Two Co-located Thread Executions:	
None	No	99.8% (21.91)	62.3% (0.76)
	Yes	31.6% (0.88)	100.0% (21.06)
IBRS	No	99.3% (35.26)	0.2% (0.00)
	Yes	18.9% (19.28)	0.1% (0.00)
STIBP	No	99.7% (38.09)	0.0% (0.00)
	Yes	19.1% (13.03)	0.0% (0.00)

Table 1: BlindSide's speculative probing vs. mitigations. *Success* rate indicates the percentage of iterations in which the function pointed to by FP executed speculatively. *Avg. Hits* indicates the average of total hits in all iterations.

In addition to the Intel Whiskey Lake CPU in our evaluation, we confirmed similar results on Intel Xeon E3-1505M v5, Xeon E3-1270 v6 and Core i9-9900K CPUs, based on the Skylake, Kaby Lake and Coffee Lake microarchitectures, respectively, as well as on AMD Ryzen 7 2700X and Ryzen 7 3700X CPUs, which are based on the Zen+ and Zen2 microarchitectures. Overall, our results confirm speculative probing is effective on a modern Linux system on different microarchitectures, hardened with the latest mitigations.

7.2 Availability of Indirect Branches

For indirect branches to be exploitable by speculative probing, we need them to be relatively close to the nearest conditional branch that controls their execution. Furthermore, the closer the indirect branch is to the conditional branch, the more cycles from the speculation window are available for the instructions executed speculatively at the target of the indirect branch.

To study the prevalence of exploitable branches, we employed static analysis with a conservative definition of control-dependent indirect branches. In particular, we say that an indirect branch is control-dependent on a conditional branch if one conditional branch target dominates the indirect branch while the other target has no path to the indirect branch. For simplicity, our analysis caps the maximum number of instructions to 50, and while our analysis is interprocedural and may include multiple calls in a call stack (e.g., the conditional branch may be in the caller of the function that contains the indirect branch), we exclude additional call-return pairs between the conditional branch and the indirect branch.

Figure 2 depicts the results of gathering the shortest distance, in number of instructions, between each indirect branch in the kernel and the closest preceding conditional branch on which it depends. Even with our conservative analysis, we found that 7,929 (more than 50% of the total 15,762) indirect branches are control-dependent on a nearby conditional branch. The vast majority of the indirect branches are even very close to a conditional branch. For instance, over 90% are 10 or fewer instructions away from the closest preceding conditional branch on which they depend and around 75% are as close as 5 or fewer instructions away. These branches

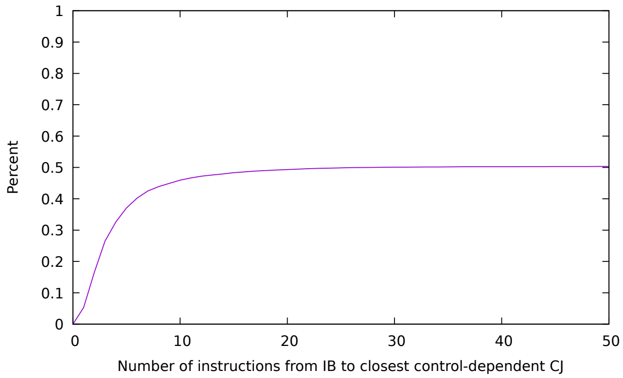


Figure 2: Distance CDF. X-axis: #instructions between indirect branch (IB) and closest preceding conditional jump (CJ) on which it is dependent. Y-axis: % of indirect branches with less than given distance.

account for over 45% and over 37% of the *total* number of indirect branches, respectively. Since a speculation window has the potential to execute hundreds of CPU cycles, this result confirms that a large number of indirect branches can indeed be executed within a speculation window while leaving ample room for speculative gadget execution.

7.3 Gadgets with Dereferences

For our proof-of-concept exploits, we were only interested in memory dereferencing gadgets where the registers RBX and RDI are dereferenced first for attacker-specified values. However, other exploits may rely on different registers. By means of an interprocedural analysis (of paths without call-return pairs), we collected all gadgets of up to 25 instructions that offered up to 4 chained memory dereferences originating from any general-purpose register, using the gadget templates described in earlier sections. Table 2 presents our results.

Since we searched for gadgets at any offset in the kernel image, gadgets starting at different offsets could end up at the same (last) memory dereferencing instruction in the chain of dereferences. For counting purposes, we considered such gadgets as a single gadget.

As shown in the table, there are numerous gadgets with memory dereferences in the kernel codebase, with significantly (4.6 to 10.5 times) more gadgets with 2 chained dereference than with 3, as expected. Furthermore, we observe that the distribution of gadget frequency per general-purpose register is in line with the System V AMD64 calling convention [62] used on Intel x86-64.

Gadgets with memory dereferences via callee-saved registers (i.e., RBX, RBP, and R12–R15) are highly prevalent. As these registers preserve their values while executing in the function, they are used for persistent computations. For example, it is common to move values from function argument registers (i.e., RDI, RSI, RDX, RCX, R8, and R9) to callee-saved registers in the function's prologue and compute on such registers. The scratch register RAX is highly used in computations, which explains the large number of available gadgets. Finally, the first memory dereference of many of the gadgets with source register RBP happens on the local variable area of the function's stack frame. Although numbers for Spectre gadgets are low, a single fitting gadget is sufficient. Also, if a vulnerability already pre-loads some of the necessary Spectre gadget input, we can relax the template of the Spectre gadget so that many more will be available. Overall, attackers can choose gadgets with a wide range of register-originating memory accesses, across both registers and memory areas.

Source	# Dereferences			Source	# Dereferences		
Register	2	3	4 (Spectre)	Register	2	3	4 (Spectre)
RAX	3086	540	1	R8	96	14	0
RBX	4385	640	8	R9	75	11	0
RCX	317	35	0	R10	85	8	0
RDX	682	114	1	R11	36	5	0
RSI	667	125	0	R12	2070	344	1
RDI	3842	844	15	R13	1278	182	1
RBP	3774	506	14	R14	1166	161	6
RSP	482	85	1	R15	1114	149	0

Table 2: Number of gadgets with up to 4 chained dereferences, originating from general-purpose registers.

8 MITIGATIONS

Preventing probes. BlindSide's probes rely on the ability to control a memory error vulnerability and speculative execution. To hinder the former, there are a variety of memory safety solutions documented in literature, but they are expensive and have found limited applicability in practice. Moreover, an attacker may also opt for other vectors to corrupt code pointers in speculative execution, such as speculative memory corruption [52] or CPU bugs like LVI [90]. To hinder the latter, one could build on existing Spectre mitigations and treat indirect branches as potentially dangerous. For instance, building on Spectre-BCB mitigations, we would add fence instructions behind all the conditional branches that are shortly followed by indirect branch instructions. Unfortunately, our analysis shows these gadgets are pervasive and this strategy would severely limit the number of conditional branches that can benefit from speculation (and its performance gains).

Detecting probes. Unlike BROP-style probes, there is no software-supported mechanism to detect BlindSide's probes; hardware support is needed. An option is for future Performance Monitoring Units (PMUs) to interrupt software execution after detecting an excessive number of "crashes" (i.e., exceptions) that occur during speculative execution. However, compared to regular execution, speculative execution is much more prone to accidental exceptions and even control-flow hijacks (due to relatively frequent mispredictions), hence a speculative anomaly detector may be more prone to false positives. For the same reason, hardware-supported (speculative) booby trapping [18, 23] seems difficult to come by.

Hindering probes. BlindSide's probes rely on being able to observe microarchitectural side effects through a covert channel. As a result, we could hinder the probes by drawing from solutions that break covert channels. However, this is particularly challenging in the case of speculative probing, since an attacker may use arbitrary 1-bit covert channels to detect specific (even unaligned) gadgets, objects, etc. Moreover, the probes run in the context of the victim program, so partitioning microarchitectural resources by security domain is not helpful. Hardware-enforced side-effect-free speculative execution would stop speculative execution attacks [51, 99], but none of the proposals have yet found practical applicability.

9 RELATED WORK

Here we complement the related work already discussed in §2, focusing on probing attacks, other software-based derandomization attacks, and microarchitectural attacks for software exploitation.

Probing attacks. Recent probing attacks focus on breaking information hiding-based defenses that use randomization as a building block. Missing the pointer [25] uses arbitrary memory read/write probes to scan the address space for low-entropy hidden regions. Thread spraying [35] shows similar probing attacks are possible against even high-entropy thread-local hidden regions when attackers can spawn many threads. Allocation oracles [68] exploit memory overcommit behavior to craft huge allocation probes and locate even max-entropy hidden regions with few or no crashes.

Defenses against prior probing attacks fall into two main classes. A first class protects valuable targets (e.g., hidden regions) with booby traps in code [18, 23] or data [68] regions to catch probing attempts and immediately flag detection. A second class employs explicit detection of anomalous probe-like events (e.g., crashes, huge allocations, etc.). An option is to simply raise an alert upon detection of a large number of anomalous events [35, 79]. More sophisticated techniques instead trigger just-in-time re-randomization [63], authentication [35], or hot patching [7]. In contrast to all existing attacks, BlindSide relies on speculative probing primitives to stealthily leak through microarchitectural side effects from crash-sensitive targets and bypass all such defenses.

Other derandomization attacks. We already discussed a class of *leakage-resistant* schemes [13, 14, 19, 22, 23, 32, 33, 40, 55, 59, 70, 71] based on execute-only memory for code in §2.1. These schemes are still vulnerable to generative attacks in scripting environments such as JavaScript [64] and data-driven disclosure attacks [73, 91] in the presence of information disclosure primitives. However, without such primitives, the attack surface for common systems software is believed to be limited. PIROP [36] shows position-independent code-reuse attacks are still possible with at least massaging primitives, but only against basic ASLR. In contrast, BlindSide can operate in absence of information disclosure primitives and blindly craft such primitives despite fine-grained, leakage-resistant randomization.

Other schemes periodically re-randomize the address space to invalidate any leaked information [9, 18, 34, 97], but an attacker can still mount just-in-time attacks between randomization intervals [80] and frequent intervals can be costly for commodity kernels [71]. Other schemes suggest garbling code right after it is read to immediately invalidate any leaked code knowledge [84, 96], but an attacker can still indirectly infer the code layout [81].

Microarchitectural attacks. While early microarchitectural attacks such as classic cache side-channel attacks [69, 98] or even more recent attacks [5, 24, 37, 38, 67, 93] primarily focus on breaking crypto implementations, there is a large body of work on microarchitectural attacks to support software exploitation. Such attacks typically use side-channel disclosure to mimic limited memory read primitives [12, 26, 39] and fault attacks like Rowhammer to mimic limited memory write primitives [12, 20, 28, 29, 42, 72, 76, 85, 86, 92].

Most attacks use side channels to break basic ASLR, for instance by leaking information from MMU-induced cache accesses [39], branch predictors [26], and store-to-load forwarding [16]. Some attacks focus specifically on kernel-level ASLR (or KASLR), derandomizing the kernel address space using TLBs [45, 58], way predictors [60], cache prefetchers [41], hardware transactional memory [48], or speculation [17, 66].

Nonetheless, all these attacks cannot break more fine-grained randomization schemes. This was only believed possible by combining side-channel attacks with speculative execution vulnerabilities able to leak arbitrary values [15, 16, 43, 53, 61, 65, 75, 94], but such vulnerabilities are target of pervasive mitigation efforts on commodity platforms. In contrast, BlindSide bypasses all the state-of-the-art mitigations against speculative execution attacks, while bypassing even fine-grained leakage-resistant randomization.

10 CONCLUSION

Code-reuse attacks and defenses have been extensively studied in the past decade. As the community now devotes much attention to new classes of attacks such as those concerned with speculative execution vulnerabilities, the common assumption is that the well-understood code-reuse attack surface is “stable”. In this paper, we revisited this assumption and uncovered complex interactions between traditional code-reuse and the emerging speculative execution threat models—allowing us to generalize both. We presented BlindSide, a new exploitation technique that leverages an under-explored property of speculative execution (i.e., crash/execution suppression) to craft *speculative probing* primitives and lower the bar for software exploitation. We showed our primitives can be used to mount powerful, stealthy BROP-style attacks against the kernel with a single memory corruption vulnerability, without crashes and bypassing strong Spectre/randomization-based mitigations.

ACKNOWLEDGMENTS

We would like to thank Andrea Bittau (1983-2017) for inspiring us to work on “Speculative” BROP. We would also like to thank the anonymous reviewers for their valuable feedback. This work was supported by the European Union’s Horizon 2020 research and innovation programme under grant agreements No. 786669 (ReAct), No. 825377 (UNICORE) and No. 690972 (PROTASIS), by Intel Corporation through the Side Channel Vulnerability ISRA, by the Netherlands Organisation for Scientific Research through grants NWO 639.021.753 VENI "PantaRhei", and NWO 016.Veni.192.262, and by the Office of Naval Research (ONR) under awards N00014-16-1-2261 and N00014-17-1-2788. This paper reflects only the authors’ view. The funding agencies are not responsible for any use that may be made of the information it contains.

REFERENCES

[1] 2020. Amazon EC2 G4 Instances. https://aws.amazon.com/ec2/instance-types/g4/

[2] 2020. Frequently Asked Questions About RAP. https://grsecurity.net/rap_faq

[3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *CCS*.

[4] Kristen Carlson Accardi. 2020. Function Granular KASLR. https://lwn.net/Articles/826539/

[5] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port contention for fun and profit. In *IEEE S&P*.

[6] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. 2014. You can run but you can't read: Preventing disclosure exploits in executable code. In *CCS*.

[7] Koustubha Bhat, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. 2019. ProbeGuard: Mitigating Probing Attacks Through Reactive Program Transformations. In *ASPLOS*.

[8] Atri Bhattacharyya, Andrés Sánchez, Esmaeil M Koruyeh, Nael Abu-Ghazaleh, Chengyu Song, and Mathias Payer. 2020. SpecROP: Speculative Exploitation of ROP Chains. (2020).

[9] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely rerandomization for mitigating memory disclosures. In *CCS*.

[10] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking blind. In *IEEE S&P*.

[11] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-oriented Programming: A New Class of Code-reuse Attack. In *ASIACCS*.

[12] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *S&P*.

[13] Kjell Braden, Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Stephen Crane, Michael Franz, and Per Larsen. 2016. Leakage-Resilient Layout Randomization for Mobile Devices. In *NDSS*.

[14] Scott Brookes, Robert Denz, Martin Osterloh, and Stephen Taylor. 2016. ExOShim: Preventing Memory Disclosure Using Execute-Only Kernel Code. In *ICCWS*.

[15] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security*.

[16] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, et al. 2019. Fallout: Leaking data on meltdown-resistant cpus. In *CCS*.

[17] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. 2016. KASLR: Break It, Fix It, Repeat. In *ASIACCS*.

[18] Xi Chen, Herbert Bos, and Cristiano Giuffrida. 2017. CodeArmor: Virtualizing the Code Space to Counter Disclosure Attacks. In *EuroS&P*.

[19] Yaohui Chen, Dongli Zhang, Ruowen Wang, Rui Qiao, Ahmed M Azab, Long Lu, Hayawardh Vijayakumar, and Wenbo Shen. 2017. NORAX: Enabling execute-only memory for COTS binaries on AArch64. In *IEEE S&P*.

[20] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. 2019. Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks. In *S&P*.

[21] Jonathan Corbet. 2018. Meltdown and Spectre mitigations: a February update. https://lwn.net/Articles/746551/

[22] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical code randomization resilient to memory disclosure. In *IEEE S&P*.

[23] Stephen J Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. 2015. It's a TRaP: Table randomization and protection against function-reuse attacks. In *CCS*.

[24] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. 2017. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *USENIX Security*.

[25] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. 2015. Missing the point(er): On the effectiveness of code pointer integrity. In *IEEE S&P*.

[26] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *MICRO*.

[27] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. [n.d.]. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *ASPLOS'18*.

[28] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *S&P*.

[29] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. TRRespass: Exploiting the Many Sides of Target Row Refresh. In *S&P*.

[30] Robert Gawlik, Benjamin Kollenda, Philipp Koppe, Behrad Garmany, and Thorsten Holz. 2016. Enabling Client-Side Crash-Resistance to Overcome Diversification and Information Hiding. In *NDSS*.

[31] C. Ge, L. Xu, W. Qiu, Z. Huang, J. Guo, G. Liu, and Z. Gong. [n.d.]. Optimized Password Recovery for SHA-512 on GPUs. In *CSE'17*.

[32] Jason Gionta, William Enck, and Per Larsen. 2016. Preventing kernel code-reuse attacks through disclosure resistant code diversification. In *CNS*.

[33] Jason Gionta, William Enck, and Peng Ning. 2015. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *CODASPY*.

[34] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. 2012. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *USENIX Security*.

[35] Enes Goktas, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. 2016. Undermining Information Hiding (And What to do About it). In *USENIX Security*.

[36] Enes Goktas, Benjamin Kollenda, Philipp Koppe, Erik Bosman, Georgios Portokalidis, Thorsten Holz, Herbert Bos, and Cristiano Giuffrida. 2018. Position-independent Code Reuse: On the Effectiveness of ASLR in the Absence of Information Disclosure. In *EuroS&P*.

[37] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. 2020. ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures. In *NDSS*.

[38] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security*.

[39] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS*.

[40] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L Scott. 2019. IskiOS: Lightweight defense against kernel-level code-reuse attacks. *arXiv preprint arXiv:1903.04654* (2019).

[41] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *CCS*.

[42] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer.js: A remote software-induced fault attack in Javascript. In *DIMVA*.

[43] Jann Horn. 2018. Spectre Attacks: Exploiting Speculative Execution. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html

[44] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-oriented programming: On the expressiveness of non-control data attacks. In *IEEE S&P*.

[45] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *IEEE S&P*.

[46] Intel. 2018. Speculative Execution Side Channel Mitigations. https://software.intel.com/security-software-guidance/api-app/sites/default/files/336996-Speculative-Execution-Side-Channel-Mitigations.pdf

[47] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. 2019. {SPOILER}: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In *USENIX Security*.

[48] Yeongjin Jang, Sangho Lee, and Taesoo Kim. 2016. Breaking kernel address space layout randomization with Intel TSX. In *CCS*.

[49] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. 2014. ret2dir: Rethinking kernel isolation. In *USENIX Security*.

[50] Vasileios P Kemerlis, Georgios Portokalidis, and Angelos D Keromytis. 2012. kGuard: lightweight kernel protection against return-to-user attacks. In *USENIX Security*.

[51] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. [n.d.]. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation *(DAC'19)*.

[52] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757* (2018).

[53] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE S&P*.

[54] Benjamin Kollenda, Enes Goktas, Tim Blazytko, Philipp Koppe, Robert Gawlik, Radhesh Krishnan Konoth, Cristiano Giuffrida, Herbert Bos, and Thorsten Holz. 2017. Towards Automated Discovery of Crash-Resistant Primitives in Binaries. In *DSN*.

[55] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *EuroSys*.

[56] Andrey Konovalov. 2017. Exploiting the Linux kernel via packet sockets. https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html

[57] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P Kemerlis, and Michalis Polychronakis. 2018. Compiler-assisted code randomization. In *IEEE S&P*.

[58] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. TagBleed: Breaking KASLR on the Isolated Kernel Address Space Using Tagged TLB. In *EuroS&P*.

[59] Donghyun Kwon, Jangseop Shin, Giyeol Kim, Byoungyoung Lee, Yeongpil Cho, and Yunheung Paek. 2019. uXOM: Efficient eXecute-Only Memory on {ARM} Cortex-M. In *USENIX Security*.

[60] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. 2019. Take A Way: Exploring the Security Implications of AMD's Cache Way Predictors. (2019).

[61] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security*.

[62] Hongjiu Lu, Michael Matz, Milind Girkar, Jan Hubiaka, Andreas Jaeger, and Mark Mitchell. 2018. System V Application Binary Interface AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models) Version 1.0. https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf

[63] Kangjie Lu, Wenke Lee, Stefan Nürnberger, and Michael Backes. 2016. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization. In *NDSS*.

[64] Giorgi Maisuradze, Michael Backes, and Christian Rossow. 2016. What cannot be read, cannot be leveraged? revisiting assumptions of JIT-ROP defenses. In *USENIX Security*.

[65] Giorgi Maisuradze and Christian Rossow. 2018. Ret2Spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*.

[66] Giorgi Maisuradze and Christian Rossow. 2018. Speculose: Analyzing the security implications of speculative execution in CPUs. *arXiv preprint arXiv:1801.04084* (2018).

[67] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. 2018. MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations in SGX. In *CT-RSA*.

[68] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. 2016. Poking Holes in Information Hiding. In *USENIX Security*.

[69] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA*.

[70] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel {MPK}). In *USENIX ATC*.

[71] Marios Pomonis, Theofilos Petsios, Angelos D Keromytis, Michalis Polychronakis, and Vasileios P Kemerlis. 2017. kRˆ X: Comprehensive kernel protection against just-in-time code reuse. In *EuroSys*.

[72] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. 2016. Flip Feng Shui: Hammering a Needle in the Software Stack. In *USENIX Security*.

[73] Robert Rudd, Richard Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Stephen Crane, Christopher Liebchen, Per Larsen, Lucas Davi, Michael Franz, et al. 2017. Address Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity.. In *NDSS*.

[74] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *IEEE S&P*.

[75] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*.

[76] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat* (2015).

[77] Fermin J Serna. 2012. The info leak era on software exploitation. *Black Hat USA* (2012).

[78] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *CCS*.

[79] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the effectiveness of address-space randomization. In *CCS*.

[80] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE S&P*.

[81] Kevin Z Snow, Roman Rogowski, Jan Werner, Hyungjoon Koo, Fabian Monrose, and Michalis Polychronakis. 2016. Return to the zombie gadgets: Undermining destructive code reads via code inference attacks. In *IEEE S&P*.

[82] Wei Song and Peng Liu. 2019. Dynamically Finding Minimal Eviction Sets Can Be Quicker Than You Think for Side-Channel Attacks against the LLC. In *RAID*.

[83] Dean Sullivan, Orlando Arias, Travis Meade, and Yier Jin. 2018. Microarchitectural Minefields: 4K-Aliasing Covert Channel and Multi-Tenant Detection in IaaS Clouds. In *CCS*.

[84] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. 2015. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *CCS*.

[85] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Defeating Software Mitigations against Rowhammer: A Surgical Precision Hammer. In *RAID*.

[86] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Throwhammer: Rowhammer Attacks over the Network and Defenses. In *USENIX ATC*.

[87] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. 2011. On the Expressiveness of Return-into-libc Attacks. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID)*.

[88] Paul Turner. 2018. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886

[89] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. [n.d.]. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *SEC'18*.

[90] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *S&P'20*.

[91] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrida. 2017. The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later. In *CCS*.

[92] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *CCS*.

[93] Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In *USENIX Security*.

[94] Stephan van Schaik, Alyssa Milburn, Sebastian Osterlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-flight Data Load. In *S&P*.

[95] Pepe Vila, Boris Köpf, and José Francisco Morales. 2019. Theory and Practice of Finding Eviction Sets. In *IEEE S&P*.

[96] Jan Werner, George Baltas, Rob Dallara, Nathan Otterness, Kevin Z Snow, Fabian Monrose, and Michalis Polychronakis. 2016. No-execute-after-read: Preventing code disclosure in commodity software. In *ASIACCS*.

[97] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and deployable continuous code re-randomization. In *OSDI*.

[98] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*.

[99] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. [n.d.]. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *MICRO'19*.

A APPENDIX A - IMPACT OF REPETITIONS

This appendix details the impact of repetitions on the success rate of individual probes of the primitives used in our exploits. In our experiments, we arrange memory such that the probes are expected to give a signal. We report experimental results (Figures 3-7) on the setup detailed earlier and measured over 20 runs.

Note that for the noise-sensitive PRIME+PROBE (P+P), we require a certain number of hits on the target page to assert with a high certainty that we have a signal. We express this amount as a *threshold* in percentage indicating the required minimum number of hits out of the total number of measurement repetitions. We compute the threshold for each primitive that uses P+P by taking the minimum number of hits in 100 measurements over 20 runs and reduce this number by 10% to cover potential outliers. We then use the calculated threshold to determine whether we obtain a signal over the given number of measurement repetitions. For example, a success rate of 90% means that *for the given number of repetitions per run, the number of hits exceeded the calculated threshold in 90% of the 20 runs.* For P+P-based probing, we picked the lowest number of repetitions with a 100% success rate (highlighted with a dot in Figures 3-6).

For the more noise-resistant FLUSH+RELOAD (F+R), we found that having a single hit at the expected cache line is sufficient to assert that we have a signal (i.e., for gadget probing and Spectre probing in testing mode). This is because the verification step is sufficient to weed out false hits caused by the prefetcher—our Spectre gadget loads consecutive cache lines for consecutive F+R buffer offsets. As such, for calibration, we picked the maximum of repetitions (8) required to produce the first hit across 20 runs ($N = 1$ in Figure 7).

However, when we do not know which cache line will produce a signal (i.e., for Spectre probing in leaking mode), it is preferable to aim for more hits. We found 2 hits to be sufficient to avoid interference from the prefetcher in practice for our gadget. As such, for calibration, we initially picked the maximum of repetitions (9) required to produce the first 2 hits across 20 runs ($N = 2$ in Figure 7). As an optimization, we lowered this value to 7 repetitions without reducing the (100%) success rate, since the redundancy offered by our gadget in leaking mode allowed us to efficiently detect and recover from occasionally erroneous leaked byte values.

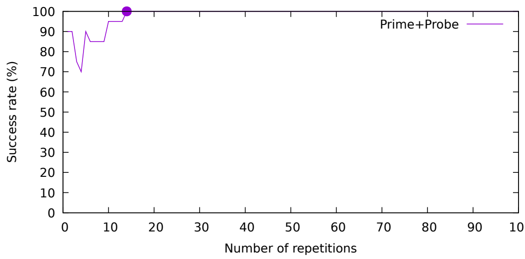


Figure 3: Success rate vs. number of repetitions to sample the target cache signal with P+P for our code region probing primitive (calculated threshold: 78.3%).

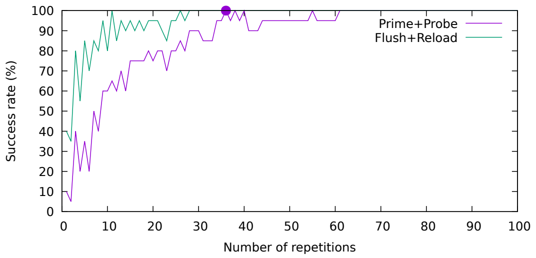


Figure 4: Success rate vs. number of repetitions to sample the target cache signal with P+P and F+R for our data region probing primitive (calculated thresholds: 52.2%).

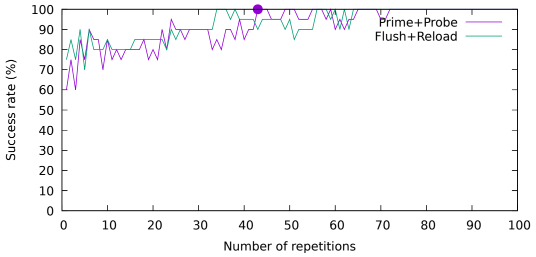


Figure 5: Success rate vs. number of repetitions to sample the target cache signal with P+P and F+R for our object probing primitive (calculated thresholds: ≈49%).

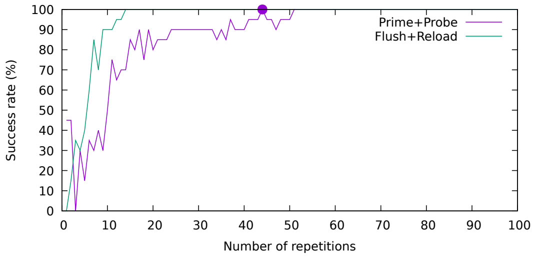


Figure 6: Success rate vs. number of repetitions to sample the target cache signal with P+P and F+R for our gadget probing and Spectre probing (testing mode) primitives (calculated thresholds: 45.0% and 27.9%, respectively).

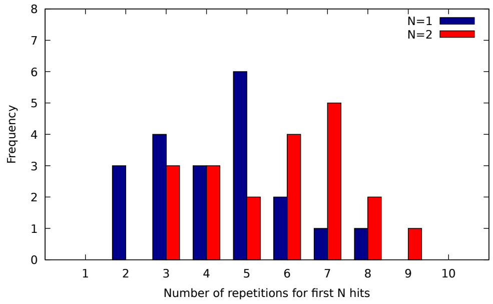


Figure 7: Frequency of the number of repetitions at which the first or second hit was seen in the user page using the Spectre gadget with F+R. When using Spectre probing in testing (leaking) mode we consult the histogram with N=1 (N=2).