ExSpectre: Hiding Malware in Speculative Execution

Jack Wampler
University of Colorado Boulder
jack.wampler@colorado.edu

Ian Martiny
University of Colorado Boulder
ian.martiny@colorado.edu

Eric Wustrow
University of Colorado Boulder
ewust@colorado.edu

***Abstract*—Recently, the Spectre and Meltdown attacks revealed serious vulnerabilities in modern CPU designs, allowing an attacker to exfiltrate data from sensitive programs. These vulnerabilities take advantage of speculative execution to coerce a processor to perform computation that would otherwise not occur, leaking the resulting information via side channels to an attacker.**

In this paper, we extend these ideas in a different direction, and leverage speculative execution in order to *hide malware* from both static and dynamic analysis. Using this technique, critical portions of a malicious program's computation can be shielded from view, such that even a debugger following an instruction-level trace of the program cannot tell how its results were computed.

We introduce *ExSpectre*, which compiles arbitrary malicious code into a seemingly-benign payload binary. When a separate trigger program runs on the same machine, it mistrains the CPU's branch predictor, causing the payload program to *speculatively* execute its malicious payload, which communicates speculative results back to the rest of the payload program to change its real-world behavior.

We study the extent and types of execution that can be performed speculatively, and demonstrate several computations that can be performed covertly. In particular, within speculative execution we are able to decrypt memory using AES-NI instructions at over 11 kbps. Building on this, we decrypt and interpret a custom virtual machine language to perform arbitrary computation and system calls in the real world. We demonstrate this with a proof-of-concept dial back shell, which takes only a few milliseconds to execute after the trigger is issued. We also show how our corresponding trigger program can be a pre-existing benign application already running on the system, and demonstrate this concept with OpenSSL driven remotely by the attacker as a trigger program.

ExSpectre demonstrates a new kind of malware that evades existing reverse engineering and binary analysis techniques. Because its true functionality is contained in seemingly unreachable dead code, and its control flow driven externally by potentially any other program running at the same time, ExSpectre poses a novel threat to state-of-the-art malware analysis techniques.

I. Introduction

Modern CPU designs use speculative execution to maintain high instruction throughput, with the goal of improving performance. In speculative execution, CPUs execute likely future instructions while they wait for other slower instructions to complete. When the CPU's guess of future instructions is correct, the benefit is faster execution performance. When its guess is wrong, the CPU simply discards the speculated results and continues executing along the true path.

Previously, it was assumed that speculative execution results remain invisible if discarded, as careful CPU design maintains strict separation between speculative results and updates to architectural state. However, recent research has revealed side channels that violate this separation, and researchers have demonstrated ways to exfiltrate results from speculative computation. Most notably, the Spectre vulnerability allows attackers to leak information from purposefully mis-speculated branches in a victim process [28]. The Meltdown vulnerability uses speculative results of an unauthorized memory read to sidestep page faults and leak protected memory from the kernel [33]. Both of these vulnerabilities focus on extracting secret data from a process or operating system. Recent follow-up work has revealed other Spectre "variants", including speculative buffer overflows, speculative store bypass, and using alternative side channels besides the cache [27], [36]. In addition, several attacks have leveraged Spectre to attack Intel's SGX [10], [40], [8], and perform remote leakage attacks [49].

In this paper, we explore another attack enabled by speculative execution: ExSpectre, which *hides computation* within the "speculative world". Taking advantage of the CPU's speculation to secretly perform computation, we can produce binaries that thwart existing reverse engineering techniques. Because the speculative parts of a program never "truly" execute, we can hide program functionality in the unreachable dead code in a program. Even a full instruction trace, captured by a hardware debugger or software emulator, will be unable to capture the logic performed speculatively. This technique could lead to sophisticated malware that hides its behavior from both static and dynamic analysis.

Existing malware use several techniques to evade detection and make it difficult for analysts to determine payload behavior of reported malware. For example, binary *packers* or *crypters* encode an executable payload as data that must be "unpacked" at runtime, making it difficult to tell statically what a program will do [21]. Malware may also use *triggers* that only run the payload when certain conditions are present, preventing it from executing when it is inside an analysis sandbox or debugger [3], [47].

However, with some effort, these existing malware techniques can be defeated. Analysts can use dynamic execution to unpack malware and reveal its behavior [3], and can use symbolic execution or code coverage fuzzers to determine the

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23409
www.ndss-symposium.org

inputs or triggers that will reveal malicious behavior [38], [48], [56], [12].

ExSpectre provides a new technique to malware authors, allowing them to hide program functionality in code that appears to not execute at runtime by leveraging Spectre as a feature [20]. This technique defeats existing static and dynamic analysis, making it especially difficult for malware analysts to determine what a binary will do.

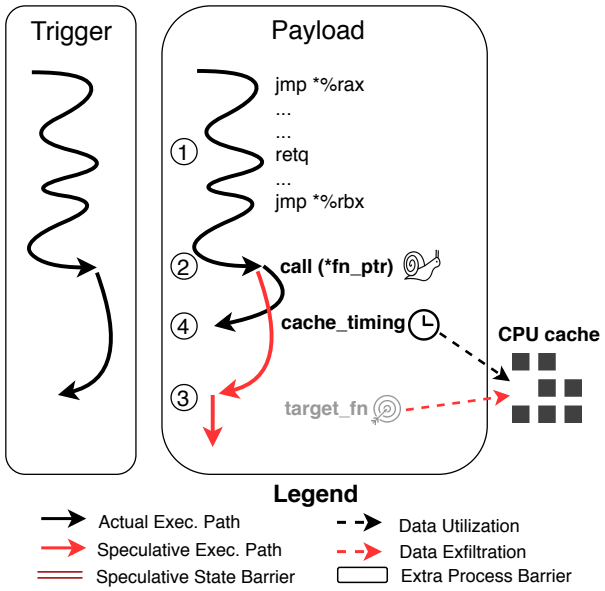

Fig. 1. **ExSpectre** — Both the trigger and payload binaries perform the same initial series of indirect jumps (Step 1), with the goal of having the trigger program (mis)train the branch predictor. In the payload program, `fn_ptr` has been set to point to the `cache_timing` function but is flushed from cache. Following the pattern in the trigger program, the branch predictor mispredicts the jump (Step 2) and instead speculatively jumps to `target_fn` (red line). `target_fn` briefly executes speculatively (Step 3), until the `fn_ptr` is resolved and the process redirects computation to the (correct) `cache_timing` function (Step 4). This function then measures information computed in the speculative `target_fn` by measuring a covert cache side channel.

At a high level, ExSpectre consists of two parts: a payload program, and a trigger. The trigger can take the form of a special input (as in typical malware), or an unrelated program running on the same system. When run without the trigger, the payload program executes a series of benign operations, and measures a cache-based side channel[1]. Once the trigger activates—either by the attacker providing specially crafted input, or the trigger program running—it causes the CPU to briefly *speculatively* execute from a new target location inside the payload program.

This target location can be in a region that is neither read nor executed normally by the payload program, making this logic effectively dead code to any static or dynamic reachability analysis. After the CPU discovers the mis-speculation at the target location, it will discard the results and continue executing from the correct destination. However, this still gives the payload program a limited speculative window where it can perform arbitrary computation, and can communicate results back to the "real world" via a side channel. Figure 1 shows the variant of ExSpectre that uses a trigger program to mis-train the CPU's indirect branch predictor, causing the payload program to briefly execute a hidden target function speculatively.

It is also possible for a trigger program to be a *benign* program already on the victim's computer. We show this using the OpenSSL library as a benign trigger program in Section V-A, activating a malicious payload program when an adversary repeatedly connects to the infected OpenSSL server using a TLS connection with a specific cipher suite.

We also show it is possible to obviate the trigger program entirely, and instead use *trigger inputs*, which are data inputs the attacker provides directly to the payload, causing the CPU to speculatively execute at the attacker's chosen address. Unlike traditional malware input triggers, these inputs cannot be inferred from the payload binary using static analysis or symbolic execution, as the logic these triggers activate happen speculatively in the CPU, which existing analysis tools do not model. We describe this technique in more detail in Section V-B.

Simulating or modelling the speculative execution path is a difficult task for a program analyst hoping to reverse engineer an ExSpectre binary. First, the analyst must reverse engineer and accurately model the closed-source proprietary components of the target CPU, including the branch predictor, cache hierarchy, out-of-order execution, and hyperthreading, as well as taking into account the operating system's process scheduling algorithm. In contrast, the ExSpectre author only has to use a partial model of these components and produce binaries that take advantage of them, while the analyst's model must be complete to capture all potential ExSpectre variants. Second, the analyst must run all potential trigger programs or inputs through the simulator, including benign programs with real world inputs. Both of these contribute to a time-consuming and expensive endeavor for would-be analysts, giving the attacker a significant advantage.

In order to study the potential of ExSpectre, we implement several example payload programs and trigger variants, and evaluate their performance. We find that a payload program's speculative window is mainly limited by the CPU's reorder buffer, which allows us to execute up to 200 instructions speculatively on modern Intel CPUs. While brief, we show how to perform execution in short steps, communicating intermediate results back to the "real world" part of the payload program. Using this technique, we demonstrate implementing a universal Turing machine (demonstrating arbitrary computation), a custom instruction set architecture that fits within the constraints of speculative execution, and show the ability to perform AES decryption using AES-NI instructions.

Using these building blocks, we demonstrate the practicality of hiding arbitrary computation by implementing a reverse shell in our speculative instruction set, with instructions decrypted in the speculative world. We show that this simple payload is able to perform several system calls in a reasonable time, ultimately launching a dial-back TCP shell in just over 2 milliseconds after the trigger is present.

[1]We note that other side channels could be used in place of a cache

II. Background

Modern CPU designs employ a wide range of tricks in order to maximize performance. In this section, we provide preliminary background as they are relevant to our system, as well as a brief summary of the Spectre vulnerability.

A. Out-of-Order Execution

Many CPUs attempt to keep the pipeline full by executing instructions *out of order*, with the CPU allowing future instructions to be worked on and executed while it waits for slower or stalled instructions to complete. To maintain correctness and the original (Von Neumann) ordering, instructions are tracked in a *reorder buffer* (ROB), which keeps the order of instructions as they are worked on out of order. Instructions are *retired* from the ROB when they are completed and there are no previous instructions that have yet to retire. Upon retiring, an instruction's results are committed to the architectural state of the CPU. Thus, the ROB ensures that the program (or debugger) view of the CPU state always updates in program execution order, despite out of order execution.

B. Speculative Execution

CPUs also attempt to keep their pipeline full by predicting the path of execution. For example, a program may contain a branch that depends on a result from a prior slow instruction. Rather than wait for the result, the CPU can *speculatively execute* instructions down one of the paths of a branch, storing the results of the speculative instructions in the ROB. If the guess of the branch target turns out to be correct, the CPU can quickly retire all the instructions it has speculatively executed while waiting. If the guess is incorrect, the CPU must discard the (incorrectly) speculated instructions from the ROB, and continue executing from the correct branch target.

C. Branch Prediction

When a CPU mispredicts a branch, the speculative execution results are discarded, costing the CPU several cycles as the pipeline is flushed. To minimize this, CPUs employ *branch predictors* that attempt to guess the path of execution. Branch predictors maintain a short history of previous branch targets for a particular branch (e.g. whether a certain branch is frequently taken or not taken), and use this to inform the CPU's guess for speculative execution.

There are two kinds of branches a CPU handles: *direct* and *indirect*. A *direct* branch may either jump to a provided address or continue executing straight through depending on the state of the CPU (e.g. condition registers). While there are only two statically-known targets for a direct branch, the CPU may not know if the branch is taken or not until preceding instructions retire. An *indirect* branch is always taken, but its address is determined by the value of a register or memory address. Direct branches are typically used for control flow such as `if` or `for`/`while` statements, while indirect branches are used for function pointers, class methods, or case statements.

D. Spectre

In early 2018, researchers revealed the Spectre vulnerability, which allows an attacker to leak information from a victim program [28], [23], [34]. Spectre uses the fact that speculative execution can influence system state via side channel. In Spectre, an attacker mistrains the branch predictor of a CPU running a victim program by providing inputs to it. Once mistrained, the attacker then sends a new input that will cause a different in-order execution path. However, because the CPU's branch predictor has been mistrained, it will still speculatively execute the previous path.

Consider the following code snippet from the Spectre paper [28]:

```
if (x < array1_size)
    y = array2[array1[x] * 256];
```

The `if` statement correctly protects an out-of-bounds reads from `array1`. But if the branch predictor makes an incorrect guess on the branch's direction and speculatively executes inside the `if` statement, it may cause a read beyond the boundary of `array1`. The result of this will then (speculatively) be multiplied by 256 and used as an index into `array2`. Although the CPU will not commit the speculative update to `y`, it will still issue a memory read to `array2[array1[x]*256]`, which will be cached. Importantly, even after the CPU realizes the branch misprediction, it does not rollback the state of the cache, as this does not directly influence program correctness. However, the set of cached values is observable to the program via a side-channel: by timing reads to `array2[i]`, the fastest read will reveal the speculative value of `array1[x]*256`, for any value of `x`. An attacker that is able to perform such a side-channel inference on the cache can learn the speculative result of an out-of-bounds read from `array1`.

Spectre can also be applied to indirect branches. Branch predictors use the history of previous branches to predict the destination of an indirect jump when the destination is not yet known. For direct branches, only one of two destinations (taken or not taken) are possible to speculatively execute. But for indirect branches, a mistrained branch predictor can potentially be coerced into speculatively executing from *any* target instruction in the binary.

We take advantage of the behavior of indirect branch prediction to hide the location of our speculative computation.

III. Architecture

ExSpectre malware is comprised of two independent pieces: a payload program, and a trigger. The payload, and some form of the trigger, must be installed on the victim's computer (e.g. via trojan, remote exploit, or phishing). A running payload performs innocuous operations while waiting for the trigger to become present.

One form the trigger can take is another local program that interacts with the payload via the indirect branch predictor. In this case, both programs must run on the same physical CPU. We note that this constraint is not a significant burden, as programs can either use `taskset`, or, if not available, run multiple instances or wait for the OS scheduler to execute both programs on the same core.

At a high level, the trigger program performs a series of indirect jumps in a loop, training the branch predictor to this pattern. Meanwhile, the payload program performs a subset of

this jump pattern, then forces the CPU to speculate by stalling the resolution of an indirect branch via a slow memory read. The CPU will (mistakenly) predict the jump to follow the pattern performed by the trigger program, and speculatively execute that destination in the payload program.

The trigger can also take the form of a special input to the payload program, rather than a separate program. In this case, the payload program parses input data and performs innocuous operations with it. Once the trigger input is provided, it causes the program to *speculatively* overflow a buffer, despite correct bounds checks in the program. The speculative buffer overflow (described in Section V-B) causes the program to speculatively execute at an address chosen by the trigger input and controlled by the attacker.

A. Threat Model

We assume a scenario where an adversary wishes to hide or obscure the behavior of a malicious program (malware) from an analyst attempting to reverse engineer it. We note this is distinct from the goals of evading malware detection, where malware escapes classification by an anti-virus or other automated tool. While we believe ExSpectre could also be used to make automated detection more difficult, our main focus is on reverse engineer resistance, useful for evading manual classification concerned with malware behavior. For anti-virus evasion, we refer the reader to several existing techniques that are sufficient to defeat existing anti-virus systems [24], [39], [45], [52], [55].

We assume the adversary is able to install binaries on the target machine (e.g. via a trojan or remote exploitation), and the analyst is attempting to determine what the malware will do using traditional debugging tools. We assume the analyst may be aware that speculative execution is used to obfuscate behavior, but does not have special-purpose hardware that allows introspection of the CPU's speculative state. This assumption is realistic, as modern processors do not allow developers or other users to directly interact with proprietary CPU optimizations and features.

We further assume that the malware has a specific trigger that the analyst is not privy to, and the adversary can influence. In our examples, this trigger is often behavior exhibited by some other (potentially benign) process running on the same system as the malware. As the adversary is able to control when such a trigger is deployed (potentially remotely), the analyst will not be able to observe or force this trigger to happen at will. We emphasize that while this may also be true for existing trigger-based malware, analysts can often reverse engineer the trigger out of the malware, for example by observing control flow within the malware and using adaptive fuzzers [61], [51] to generate inputs that explore other execution paths of the binary. In contrast, ExSpectre malware's trigger influences behavior of the payload program speculatively, making it effectively invisible to the analyst. As with typical malware, the analyst may attempt to reverse engineer the trigger to reveal the malware's behavior, but we will show (in Section IV-B1) how this type of analysis can be defeated.

B. Indirect jumps

In this subsection, we will describe the trigger program variant, and defer discussion on how input data can be used as a speculative trigger to Section V-B.

In ExSpectre, we cause the CPU to mis-speculate the destination of an indirect branch in the payload program, causing it to speculatively execute instructions that are never truly executed. We term the destination where speculation begins the *speculative entry point*. ExSpectre uses indirect jumps to allow speculative execution from *any* instruction in the payload process' address space. Because it can jump to any instruction, the malware analyst has a difficult task in determining where a payload program's speculative entry point is.

In fact, the location of this entry point is not determined by the payload program, but rather the corresponding trigger program. This means that with only the payload program, an analyst does not posses enough information to find the speculative entry point.

Indirect branch predictors allow the CPU to predict the destination address of a branch based solely off its source address and a brief history of previous branch sources and destinations. While the inner-working details of modern CPU branch predictors are proprietary, it is possible to reverse engineer parts of their behavior, which we do for ExSpectre.

We observe that Intel CPUs consider three types of x86_64 indirect branches: `retq`, `callq *%rax`, and `jmpq *%rax`[2]. We created a simple trigger program that performs a series of indirect branches using `jmpq *%rax` instructions. Between each jump, we incremented `%rax` accordingly to continue on to the next jump. After these jumps, we load a function pointer into `%rax` and do a final indirect branch using `callq *%rax`. In our trigger, we perform these jumps repeatedly in a loop.

In our payload program, we first perform the same indirect jumps. We ensure the source and destination addresses of these jumps is the same as in the trigger program by manually defining their containing function at a fixed address inside a linker script. We also do the final indirect call to a function pointer, but with two differences. First, the destination in the function pointer is a different address, and second, the memory location of the function pointer itself is uncached. This forces the CPU to predict the destination of the final indirect call while it waits for the function pointer to load from memory. Due to the similar history of branches with the trigger program, the CPU will (incorrectly) predict the destination to be the same as the one in the trigger program, which determines the speculative entry point for the payload. Even though the in-order execution of payload program never executes or even reads from this address, the CPU will briefly execute instructions there speculatively.

In Table I we analyze the number of necessary training indirect jumps various processors require to consistently ($> 95\%$) have the payload program enter the speculative world at the chosen speculative entry point in the trigger program. We found that 28 indirect jumps was sufficient for our trigger

[2]other general purpose registers besides `%rax` can be used as well

Processor	Released	Micro-arch.	Nested Spec.	Indirect jumps	μ-ops (nop)
Intel Xeon CPU E3-1276 v3	2014	Haswell		26	178
Intel Core i5-7200U	2016	Skylake	✓	26	220
Intel Xeon CPU E3-1270 v6	2017	Skylake	✓	28	220
AMD EPYC 7251	2017	Ryzen		4	178

TABLE I. **PROCESSOR FEATURES**— WE ANALYZED THE CAPABILITY OF EXSPECTRE ON THREE INTEL PROCESSORS AND ONE AMD. BOTH SKYLAKE PROCESSORS WERE CAPABLE OF NESTED SPECULATION (SECTION III-C5). INDIRECT JUMPS IS THE NUMBER OF COMMON TRAINING INDIRECT JUMPS NEEDED IN THE TRIGGER PROGRAM TO RELIABLY ($> 95\%$ OF THE TIME) COERCE THE PAYLOAD PROGRAM TO FOLLOW THE PATTERN AND JUMP TO THE SPECULATIVE ENTRY POINT SPECIFIED IN THE TRIGGER PROGRAM. μ-OPS IS THE UPPER BOUND OF μ-OPS THAT CAN BE PERFORMED SPECULATIVELY.

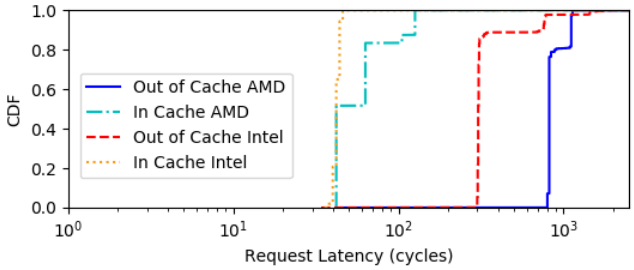

Fig. 2. **Cache latency** — Cumulative distribution function of the cache hit and miss latency for an Intel Xeon-1270 and AMD Epyc 7251. If a cache miss is used to force CPU speculation, the CPU must wait at least 300-800 cycles before the speculated branch can be resolved. However, we find the CPU is occasionally limited to far fewer instructions speculatively, suggesting another limit is at play.

program on each of the test processors to reliably ensure the speculative execution began at the correct speculative entry point.

Eventually, the de-reference of the uncached function pointer in the payload program will be resolved, and the CPU will recognize it has incorrectly predicted the destination of its `callq` instruction. The results from the speculative entry point instructions will be discarded, and the CPU will continue executing from the correct destination. However, the speculative code can change what is loaded into the cache based on its computation, allowing it to covertly communicate its results to the "real world" program.

C. Limits of Speculative Execution

We performed several experiments to determine how much computation can be performed speculatively, as well as what components are responsible for the limit. We report results from our experiments on an Intel Xeon-1270 (Sandy Bridge), though we note we found similar results across other Intel processor generations, including an i5-7200U (Kaby Lake), an i5-4300U (mobile Haswell), an i5-4590 (desktop Haswell), as well as an AMD EPYC 7251.

1) Cache Miss Duration: When executing instructions speculatively we rely on a memory load of a function pointer from uncached memory. Thus, one limit on our computation comes in the form of the time it takes for the memory read to return with a result (and for the CPU to determine the result was mis-predicted). We measured the number of cycles a cache miss takes to return by artificially evicting an item from cache and timing reads from its address. Figure 2 shows the CDF of cycles taken. In the typical case, an evicted item takes approximately 300 clock cycles to load from the Level 3 cache (L3), which would allow a limit of roughly 300 speculative instructions (depending on specific cycles per instruction (CPI)) to be executed during that time. We note that when an item is not in L3, it takes considerably longer to load, in theory allowing for thousands of speculative instructions in a significant fraction of runs.

2) Reorder Buffer Capacity: We also measured the capacity of the reorder buffer (ROB) using a method outlined by [58]. We measure the maximum number of cycles taken to perform two uncached memory reads, and vary the number of filler instructions between them. If the number of filler instructions is small, both memory reads will fit inside the ROB, and it can issue their memory reads in parallel. However, if the filler instructions fill the ROB, the second memory read will have to wait for the first to return before it can be issued, causing a noticeable step increase in the cycle count. Figure 3 shows this step occurs at approximately 220 instructions for our processor, suggesting a hard upper bound regardless of how long the cache miss takes to resolve.

3) Speculative Instruction Capacity: To verify the upper limit of speculative instructions, we instrumented our trigger and payload programs to test a simple gadget of variable-length before it communicated a signal to the real world via a cache side channel. If the cache side channel revealed no signal in the real world, then we know the speculative execution did not make it to the signal instructions before the mis-speculated branch was resolved.

We also tested whether instruction complexity or data dependencies impact the number of instructions that can be completed. We find that data dependencies and instruction complexity both have an impact on the number of instructions that can be executed. Instruction complexity is determined by the number of μ-ops that the instruction uses, which appears to be what is tracked in the ROB. For instance, on our Skylake architecture, the 64-bit `idiv` instruction takes 57 μ-ops, and we can execute up to 3 of them in the speculative world. Meanwhile, we can execute up to 18 32-bit `idiv` instructions, which each take 10 μ-ops [17]. This suggest we can execute on the order of 175 μ-ops before the speculative world expires.

Most notably instructions that use the extended x86 registers are still valid within the speculative context. Specifically, Intel's hardware accelerated AES-NI encryption and decryption instructions, which each use 128-bit registers. As shown in Figure 3, speculative environments can complete a significant number of AES rounds—over 100 rounds in our experiments, more than enough to decrypt a full block using simple AES modes (e.g. AES-CTR). We investigate the use of AES instructions in the speculative environment further in Sections IV-B.

We find that when executing speculatively, the number of instructions completed has a soft limit and a hard limit. The duration in cycles applies a soft limit, as shown in Figure 3 with the `idiv (32-bit)`, `mul`, and `aesdec` instructions. As we attempt to execute more instructions speculatively, we see a steep drop in in the fraction of trials that are able to

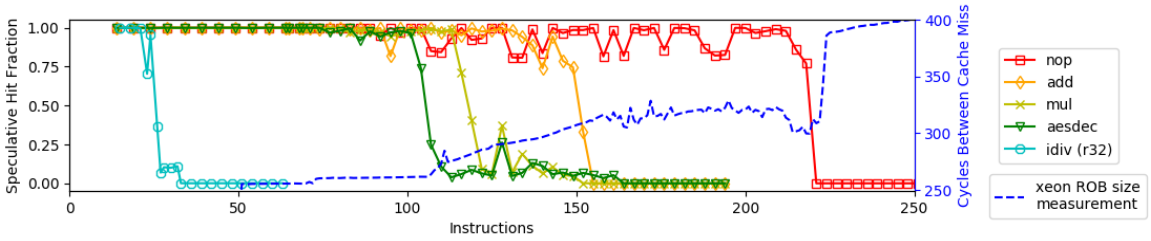

Fig. 3. **Speculative limits** — We placed a memory read after an increasing number of (speculatively executed) instructions and measured the fraction of times the loaded value was subsequently in cache. This tells us the upper bound of instructions we can reliably execute speculatively. We identify two limitations on the speculative lifetime: cache miss latency resolving the speculative branch, and the CPU's reorder buffer (ROB) size. We observe that different instructions have varying speculative limits: for example, a 32-bit `idiv` can complete only 18 instructions (not including 14 single μ-op instructions for the signal gadget), as each instruction inserts 10 μ-ops into the ROB, while cheaper instructions that use fewer μ-ops can execute more instructions.

signal via the cache-side channel. However, this speculative hit fraction does not drop to zero until the later hard limit, imposed by the number of CPU micro-operations (μ-ops) composing those instructions. Figure 3 demonstrates that the number of μ-ops of the instructions is the major limiting factor that define an upper bound of approximately 150 instructions[3].

4) Hyperthreading: When running our tests, we assign the payload and trigger program to the same core using `taskset`. We note in the absence of `taskset`, we can run multiple instances of trigger programs to occupy all cores, eventually having the payload program and trigger program become co-resident.

We also explore using hyperthreading, where the CPU presents two virtual cores for each physical core, allowing the OS to schedule programs to each simultaneously. In effect, this can cause the interleaving of instructions between two programs to be much finer-grained: at the instruction level rather than changing only at the OS-controlled context switch. We find that this has two effects on speculative programs. First, the finer-grained interleaving allows for a higher hit rate from the cache, suggesting that each indirect jump pattern is more likely to result in speculatively executing from the intended position. Second, because the physical CPU is being shared, it effectively halves the number of instructions that can be run in the speculative context. Figure 4 shows the instructions that can be run when running trigger and payload on a single core vs. a pair of hyperthreaded cores.

We note that Single Thread Indirect Branch Predictors (STIBP) have been implemented in most environments to prevent cross thread branch predictor interference. While this does remove the ExSpectre trigger's influence in scenarios where a process runs on an isolated cpu, in most modern environments tasks are scheduled to all available processors. This means that the trigger and payload will be coresident eventually, allowing progress to continue.

5) Nested Speculation: We explore the ability for the CPU to "double speculate", where a second stalled indirect jump while the CPU is already speculating causes it to predict the target and speculate a second time. For instance, suppose a payload program truly jumps to target A, but the CPU is mistrained by a trigger program that jumps to B, thus causing the payload program to speculatively execute at B. At B, suppose there is a second indirect jump, perhaps using the same register as the first jump (which has still not resolved). If the trigger program jumps to C, the payload program may speculate a second time and continue speculative execution at C. Figure 6 demonstrates Nested Speculation in action.

We find that not all Intel CPUs support nested speculation. For example, it appears Haswell chips do not speculate while already speculating, but nonetheless support non-nested ExSpectre. Both Sandy Bridge (which preceded Haswell) and Kaby Lake (which followed Haswell) support nested speculation. We find that when a CPU does support nested speculation, there appears to be no limit to nested depth besides the speculative instruction limit. We use a 16-deep nested speculation in Section VI-B to protect speculative decryption keys from reverse engineering.

[3]While `nop` is able to execute up to the full 220 ROB capacity, instructions that do useful work (and/or use multiple μ-ops) cannot reach this limit. In addition, data dependency and execution unit availability add further complications to modelling the exact number of instructions that can be executed speculatively.

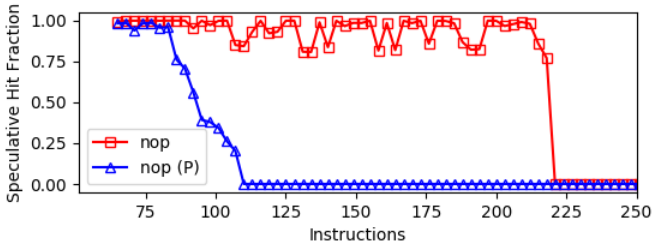

Fig. 4. **Hyperthreading** — We measured the impact hyperthreading has on speculative execution. Trigger and payload programs running on the same logical core require a context switch to alternate processes, but allows each to have full utilization of the ROB and execution units when they run. Running the programs on parity hyperthreads (denoted by (P)) allows them to run simultaneously without context switching, but we observe this configuration effectively halves the amount that each program can speculatively execute, suggesting that hyperthreads share parts of the ROB or execution units.

D. Speculative Primitive

We summarize our findings into a *speculative primitive*, which allows our payload program to speculatively (and covertly) perform on the order of 100 arbitrary instructions while an accompanying trigger program is running, and communicate a short (e.g. single byte) result to the real world via a cache side channel. These speculative instructions are

able to read from any cpu state accessible to the process in the real world including memory and registers, but they cannot perform updates or writes directly. To read memory the *speculative primitive* makes use of the ability to bring things into cache. If a load for an uncached memory location is initiated speculatively it will not finish within the speculative window (meaning no value can be exfiltrated to the real world). However, the memory read is not canceled and the value will be available from cache when the processor accesses it speculatively again. To update memory, the speculative instructions must communicate to the real world. We use a cache side channel to do so, but other side channels compatible with Spectre could also be used [27].

We note a performance tradeoff between the size of communication (e.g. 4 bits vs 8 bits) and the time it takes the real world to recover the result from the side channel. Using Flush+Reload [60] as our cache side channel, recovering the result requires accessing all elements in an array exponential in the size of the result (e.g. 2^8 array reads to recover an 8-bit result). Therefore, there is a performance advantage for keeping the size of the result small, and communicating out small pieces of information that are aggregated by the real world over multiple speculative executions. Meanwhile, smaller channels introduce more overhead in recovering information. We investigate this tradeoff in Section VI-B, and find that 8 bits is near optimal in practice.

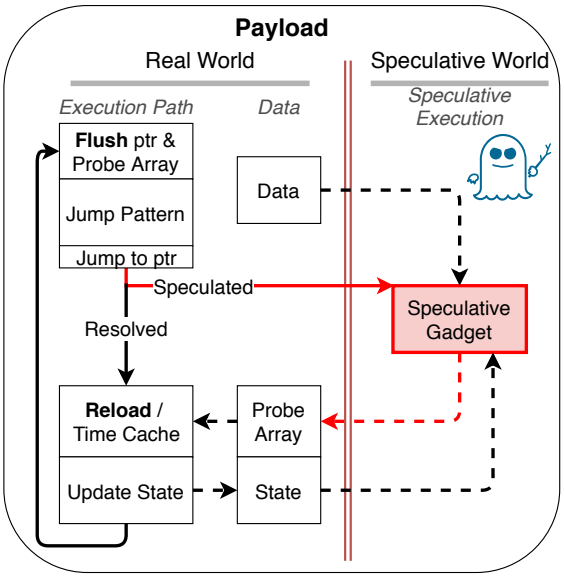

Fig. 5. **ExSpectre model** — General model of speculative computation within the payload process when triggered. The *Speculative Gadget* has read-only access to all memory within the process, but can only return updates/results via a cache side channel (by accessing the `probe_array`). The process can subsequently *Reload* from the cache side channel to learn the speculatively-computed result, and update the state of the *Real World* process.

IV. Application Payloads

While the amount of computation done in a single speculative execution is small, we demonstrate several applications that can take advantage of multiple speculative runs to carry out computation.

As a first step, we observe that the speculative primitive can be used to trivially implement a finite state machine: logic can be done in the speculative world, while updates to the state are communicated to the real world where they are stored. On the next run of the speculative instructions, the state is read from the real world state (along with any inputs), state transitions are computed and communicated back. In this mode, the state is maintained by the real world, while updates are controlled by code executed speculatively.

We further observe we are not limited to finite state machines, but can support any model of computation where *updates* to any state are finite (i.e. can fit within the bandwidth constraints of the speculative primitive). This encompases Turing machines [53] as well as certain random access machines, which we investigate next. Figure 5 demonstrates the execution flow of a sample ExSpectre malware.

A. Turing Machine

To demonstrate that arbitrary computation can be performed cooperatively between the speculative world and real world, we implement a Turing machine, and configure it to run a 5-state Busy Beaver function [53], [9], [22]. This configuration allows us to run a large number of steps with very minimal logic.

Updates to this Turing machine are computed speculatively, while the real world keeps track of the state and full tape of the machine. Thus, the logic of the machine is entirely contained in the speculative world, while the state may be externally visible (e.g. to a dynamic debugger). We note that the machine only operates when the trigger executes, making it difficult for an analyst with only access to the Turing machine to determine exactly what the machine will do from its initial state.

However, this toy example is meant only as an illustrative example of arbitrary computation, not as a robust means of obfuscation. Indeed, even the initial state of a Turing machine alone may reveal a significant amount of information. Furthermore, the analyst may attempt to locate potential speculative entry points, even without the help of the trigger program. We describe ways to address both of these next.

B. Unpacking and Decryption

While a Turing machine demonstrates that arbitrary speculative computation is possible, hiding malware this way has several drawbacks. First, Turing machines are a poor choice for practical computation, as they are inefficient and have no direct way to interface with the rest of the system (e.g. via system calls). Second, as mentioned, they leave a great deal of information available to the analyst, including the initial tape state, and a potentially small (enumerable) number of possible speculative entry points.

We explore a more practical application of using ExSpectre to perform *decryption* speculatively. To hide keys from the analyst, the key and decryption code only occur in the speculative world, while the initial payload program contains only the ciphertext. While partial plaintext will be available in the real world during execution, we emphasize that this only occurs when the trigger runs. Before this, the state of the program reveals only the ciphertext that will be decrypted. While the speculative entry point enumeration attack could be

used to reveal the keys used to decrypt this ciphertext, we describe a way to derive the decryption key entirely from the trigger program. Thus, an analyst that only has access to the payload program will be unable to learn the key or decrypt the embedded ciphertext.

We also note that even when the trigger runs, decryption does not occur outside the speculative context, meaning that any traditional traps or debugging breakpoints placed on decryption instructions or routines will not occur, even as they are used speculatively. These instructions could even be obfuscated themselves by placing them in other misaligned instructions, and choosing a speculative entry point that jumps to the middle of other instructions.

We note that 200 instructions is too short for most software-implemented cryptography. However, modern Intel CPUs provide hardware support for AES, which we find only takes a handful of μ-ops to perform the instructions needed in AES decryption. We discuss details of our speculative AES decryption in Section VI-B.

1) Obfuscating keys with nested speculation: As mentioned, even with encryption, an analyst that can locate the speculative entry point and discover the decryption key. For instance, the analyst could locate the speculative entry point by searching for AES-NI instructions in the payload program, ultimately discovering the keys it derives and uses.

We can overcome this by having the trigger program communicate the decryption key to the payload program via the branch predictor. While prior work has used the branch predictor to exfiltrate keys from other sensitive processes [2], we inject a key into the speculating payload program from the external trigger program. To do this, we use multiple speculative entry points, each that derives a unique decryption key before calling a common decryption routine. Since the exact speculative entry point is determined by the trigger program, an analyst cannot trivially discover the decryption key directly from the payload program.

Still, an analyst could enumerate all potential entry points, testing each one until they find one that correctly decrypts the ciphertext. In a 1 MB binary, there are (at most) only 1 million possible entry points, providing just 20 bits of security, trivial for an analyst to brute force. An analyst simply needs to test each of the 2^{20} entry potential entry points to discover the correct key.

To increase security, we instead use nested speculation to *chain* entry points together. Rather than derive the key from a single entry point, we have each potential entry point perform another indirect jump that the CPU cannot immediately resolve, forcing it to speculate while already executing speculatively. In other words, in the speculative world, we make an indirect jump that depends on a cache-evicted variable, prompting the CPU to double-speculate. The predicted target of that jump will also be driven by the trigger program's (mis)training of the indirect branch predictor. On CPUs that support double (or arbitrarily nested) speculation, we can repeat this process, with each new subsequent entry point determined by the trigger program. At each entry point, we shift in additional bits to a register as the AES key. Without the trigger program, an analyst cannot determine the path the payload program will take speculatively.

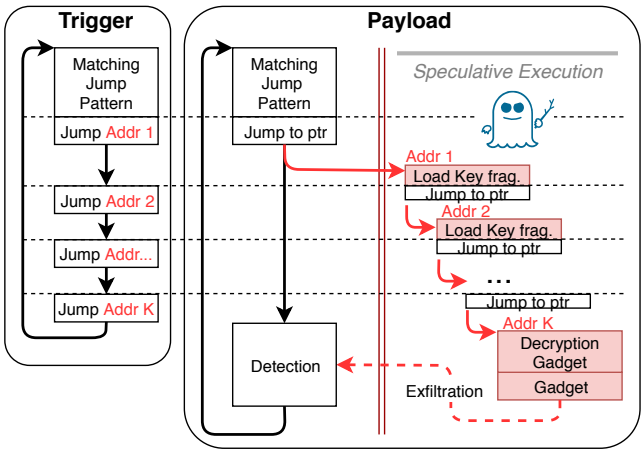

Fig. 6. **Nested Speculation** — Some CPUs support nested speculation, allowing the branch predictor to speculate a branch while already executing speculatively. We use this to obfuscate **key derivation**. The trigger program executes a sequence of indirect jumps, which the payload program will follow speculatively. Each jump target in the payload program will add a small number of bits to a speculatively-computed key. Without knowing the exact pattern of jump targets (specified only in the trigger program), the analyst will be unable to determine the key when a sufficient speculative depth/number of targets is used. In our implementation, we used a speculative depth of 16 with 2^8 targets to derive a 128-bit key. While the *decryption gadget* may be easy for an analyst to find, without the key, the encrypted data remains inaccesible.

As an illustrating example, imagine a trigger program makes 30 training jumps, followed by 10 additional indirect jumps, and the payload program performs the same 30 training jumps before a stall. At this point, the CPU will predict the payload program will also perform the next 10 jumps, speculatively following the pattern of the trigger program.

If each nested speculative jump has the potential to land in 4096 (2^{12} possible locations, each entry point can shift in 12-bits to the key, for a total of 120-bits over the 10 jumps before calling the common decryption routine. A key constructed in this way would be infeasible for an analyst to brute force, as the payload program yields no information about which of the potential 2^{120} keys will be derived.

We describe our implementation of nested speculative execution in Section VI-B, where we speculatively derive a 128-bit AES key. Figure 6 demonstrates this in practice, the trigger program running a series of training jumps followed by indirect jumps influences the payload program to follow the same path.

C. Emulation

To combine our encryption and arbitrary computation in an efficient way, we implemented an emulator that gets its instructions from the speculative decryption described previously. In the payload program, the emulated instructions are initially encrypted under a key that will be delivered by the trigger, as described previously. Once the trigger executes, it will cause instructions to be decrypted and run by the emulator.

Traditional reverse engineering methods will reveal only that emulation is being done, while the program being emulated remains encrypted. Even when the trigger is running, only the parts of the code that execute would be revealed to

a careful analyst observing the CPU's committed state, while the remainder of the emulated program would remain hidden.

We design a custom emulator and instruction set—SPASM (Speculative Assembly)—that accommodates the constraints of our speculative primitive. SPASM is a 6-bit Instruction set, where all instructions (including operand, registers, and arguments) fit within 6-bits. This allows each step of the speculative world to emit a single SPASM instruction to the real world for emulation by a light-weight SPASM emulator. Using SPASM, developers can write programs, assemble and encrypt them into a payload program. When the associated trigger program runs, the payload will decrypt SPASM instructions in the speculative world, and execute them one at a time.

While the custom emulator that we developed gives higher level abstraction to an author, it still requires programs to be written in a custom assembly language. We note that the ExSpectre model is not intrinsically linked to the SPASM emulator. A wrapper could be implemented around other existing emulators to construct instructions incrementally through the fixed-width channel (e.g. using 4 8-bit reads to reveal a single 32-bit ARM instruction), allowing for encrypted payloads to be written in higher-level languages. We also note that this provides flexibility to the authors, allowing them to completely redefine instructions or use a different instruction set altogether to hamper detection.

V. Triggers

So far, we have described using a custom program as a trigger, which performs a pattern of indirect jumps to mistrain the indirect branch predictor, leading the payload program to its speculative entry point. In this section, we describe alternative triggers, including using benign programs already on the system, and recent Spectre variants.

A. Benign Program Triggers

Custom trigger programs that are installed with malicious payload programs may be easy for an analyst to pair up and analyze. As an alternative to trying to hide the trigger program from the analyst, ExSpectre can use *benign* programs already installed on the system as a trigger.

For example, if a benign application makes a series of indirect jumps—thus training the indirect branch predictor—an ExSpectre payload can make similar indirect jumps leading up to its speculative entry point. The payload's speculative entry point will be determined by the benign application, but may be even more difficult for an analyst to discover, as now the ExSpectre trigger could be *any* application running concurrently on the system.

a) OpenSSL: We experimented using the OpenSSL library as a potential benign trigger application, as its source code has a gratuitous use of function pointers which compile to indirect jumps. In addition, it has many complicated code paths that can be easily selected by remote clients through their choice of cipher suite. This allows a remote attacker to trigger ExSpectre malware on a server running a (benign) TLS stack supported by OpenSSL, simply by making a large number of TLS connections with a specific cipher suite. We describe our implementation using OpenSSL as a trigger in Section VI-D.

In addition, an adversary could use a benign application (like OpenSSL) to *communicate* information covertly to the malicious payload program. For example, with OpenSSL, the attacker could have a pair of uncommon cipher suites, where using one results in communicating a 1-bit, while use of the other communicates a 0-bit to the payload. To receive data, the payload would have to do indirect jump patterns corresponding to OpenSSL code for processing both cipher suites, with the corresponding speculative entry points shifting in the appropriate bit. Thus, an adversary can communicate remotely (over a network) to the payload program indirectly via a benign application.

We observe that communication could also go in the other direction: from the payload program back to the remote adversary, also via the benign application intermediary. The payload program could influence the performance of the benign application, and the adversary could time responses from the benign application to receive covert information from the malicious payload [49]. This would allow the malicious payload to operate *entirely speculatively*, without assistance from the real world for keeping state.

B. Speculative Buffer Overflow

In addition to using separate trigger programs, ExSpectre can also use *trigger inputs* to a payload program to initiate its malicious behavior. While existing fuzzers and symbolic execution tools can discover traditional input triggers, we can leverage other Spectre variants to obfuscate our triggers.

To do this, we use Speculative Buffer Overflows (SBO) [27] to redirect control flow to a speculative entry point specified by user input. We design a payload program that takes arbitrary user input and performs appropriate bounds checks to ensure no traditional control flow violations could be exploited. However, using the Spectre 1.1 variant, control flow can still be violated speculatively, allowing the adversary to force a speculative entry point based entirely off the input provided. This allows the trigger to be an input, potentially even provided over a network if the payload program accepts network data. Traditional symbolic execution and code coverage fuzzers will be unable to discover this trigger input, as they do not model the speculative state of the CPU.

To create an SBO-triggered payload program, we make the following code pattern, as seen from Kiriansky and Waldspurger's Spectre 1.1 description [27]:

```
if (y < lenc)
    c[y] = z;
```

With user controlled `y` and `z` and sporadically uncached `lenc`, an attacker can *speculatively* overflow array `c` to overwrite a return address (or function pointer, etc) and redirect control flow. Note that the bounds check on `y` will ensure that this program will not actually allow a buffer overflow to occur, but the attacker can nonetheless use this to influence a speculative entry point based on their choice of `y` and `z`.

We make use of this pattern in a willing payload such that user input can intentionally mistrain the branch predictor by repeatedly sending valid (in-bounds) values of `y` before sending a value that would overflow the bounds of `c`. The

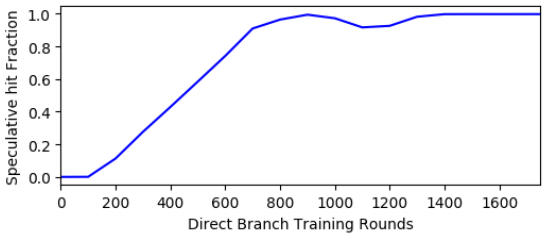

Fig. 7. **Speculative buffer overflow warm-up** — The direct branch predictor must be trained to expect that a branch will go a specific way before speculative buffer overflows can be used. We varied the number of times a branch was trained to be taken and observed the fraction of times we achieved a speculative buffer overflow execution immediately after (measured by observing if a speculatively-loaded value was present in cache avgeraged over 20,000 trials). We find that a branch must be trained in a direction hundreds of times before it can be reliably used in a speculative buffer overflow.

speculative entry point is also chosen by the trigger in this scenario as `z` contains the address of the speculative entry point, allowing the attacker to create a ROP-style speculative execution path through the payload.

We implemented an experiment to determine the number of times a branch needs to be "trained" before it can be used as a speculative buffer overflow. Figure 7 shows that several hundred benign inputs are needed to reliably be able to observe speculative buffer overflow behavior.

C. Speculative Store Bypass

Speculative Store Bypass (SSB) (Spectre variant 4) can similarly be used to construct an internal (input-based) trigger using the CPU's speculative load-store forwarding [37]. In a speculative store bypass, the CPU incorrectly speculates that a store will not alias with a future load, and uses a stale (wrong) value for the result of the speculative load.

To redirect control flow, a payload program could use a function pointer or indirect branch target register as the destination for a speculative store bypass, causing the CPU to use a stale value to speculatively determine where it would go. The stale value could be controlled by a previous unrelated input, allowing an adversary to specify the speculative entry point in a carefully crafted data input. While the program never executes at this stale address in reality, the CPU will briefly speculatively execute there, enabling ExSpectre payloads. Like the speculative buffer overflow, this trigger also allows ROP style chains to execute a series of speculative gadgets.

VI. Implementation and Evaluation

In this section, we discuss implementation details for our payload and trigger programs.

A. Turing Machine

We designed our Turing machine implementation to work with our custom trigger program, with 28 indirect jumps mimicked by the Turing payload program. We implemented a 2-symbol 5-state Busy Beaver Turing machine logic at the speculative entry point (in 42 x86-64 instructions), returning the state update, symbol to write, and tape move direction in a single byte via a cache side channel.

We observed in our implementation that it is important that all values used in the speculative world—as well as the code itself—be cached. If these are evicted, the speculative code may fail to run, or the CPU may *speculate* on the value of the uncached item, which may be incorrect. While this does not impact the correctness of normal programs whose incorrect speculations will be resolved, our speculative code reports results back to the real world before this resolution. In our Turing example, we observed this as incorrect state transitions.

This error is particularly devious, as it is not an error of bit flips or noise, but rather the processor speculating what the speculative gadget will read from memory. Thus, error correcting codes on the reported result do not improve the situation.

Instead, we repeat the execution several times and look for the modal value over all iterations. We measured the error rate of our implementation as a function of how many redundant iterations of the same step, and found that 10 redundant iterations resulted in 1 error every million Turing steps, with the error rate dropping exponentially as iterations increase. We choose 11 iterations as a conservative bound (error rate measured to be 0), and computed 1 million Turing steps at a rate of 1351 steps per second.

B. AES Decryption

The speculative world is able to take advantage of the AES-NI instructions to decrypt messages. However, the speculative upper-limit of about 175 μ-ops is not enough to allow us to compute the key expansion, even using the `aeskeygenassist` instructions. To avoid this, we can either preload the expanded key schedule into the program (instead of the key), or use a cheaper (non-standard) key expansion algorithm. For the former, we note that an analyst could observe the structure of a normal key schedule, but we can avoid this by simply selecting 11 random round keys. We note that this should not weaken the security of AES, as we can ensure the round keys are not linearly related.

We wrote our AES decryption payload in 35 x86_64 instructions and 2 lines of C (which compiles to an additional 31 x86_64 instructions). The payload implements AES-CTR mode decryption, reading a global index and returning the decrypted byte at that location in the ciphertext via the cache side channel. In this model, the speculative function decrypts a full 16-byte AES block each iteration, but only returns the bits specified by the index.

We demonstrate the speed that information can be decrypted via the speculative world, and we vary the channel width of the side channel from 1 to 12 bits to measure its performance. At low channel width, reading from the cache side channel requires timing reads from only 2 locations, while at 12-bits, the side channel requires reading 2^{12} locations. On the other hand, there is a fixed overhead per speculative iteration that favors increased channel width to maximize bandwidth. As shown in Figure 8, 8 bits is the optimal side channel width, allowing us to decrypt over 5,000 bits per second (625

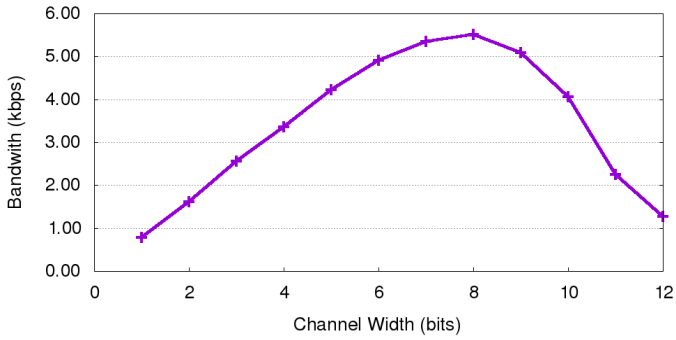

Fig. 8. **Speculative Bandwidth** — Using our speculative primitive, 1KB of data can be decrypted and exfiltrated at a speed of 5.38 Kbps from the speculative world with 20 redundant iterations per round (to ensure correctness). Increased channel width exfiltrates more data per round, but takes longer to measure the cache side channel. Optimal throughput is achieved with an 8-bit channel.

Bytes/sec). We note an improvement over this rate by loading multiple values into the probe array during speculation. Instead of a single probe array of 1024 entries communicating 10-bits, we can split the array into four sections of 256 entries each, and signal four times (one per section) during speculation. This provides a 32-bit channel overall, while still only having to probe 1024 entries. This method is capped by the limited μ-operation budget, however our implementation using these parameters (four sections of 256 entries) is able to decrypt over 11 Kbps (1,425 Bytes/sec).

We also implemented our nested speculation technique for obfuscating keys, making 256 speculative landing spots that each shift 8 unique bits into the 128-bit register `%xmm0`, and then performing an indirect jump. We then had a custom trigger program perform 16 indirect jumps (after the initial 28) that corresponded with 16 randomly-chosen landing spots in the payload program, training the branch predictor. When the payload program reaches the first speculative jump, it follows the same pattern speculatively, eventually filling `%xmm0` with the corresponding $16 * 8$ bits. We then used the `aesenc` instruction to expand these 128-bits to a full key schedule, and performed decryption as described previously. Thus, without the trigger program, an analyst has no information about what key is used to decrypt the ciphertext in the payload.

C. Emulator

We have implemented our custom instruction set architecture—SPASM—as a model using two pseudo-registers, and 6-bit instruction length which allows for a relatively direct programming model in which structured values can be entered into memory locations before making a systemcall.

In this model of computation there are effectively no instruction arguments, as we must return an entire instruction from the speculative world inside the limited-width cache side channel. Although other small instruction sets exist, they either allow variable instruction lengths, are too long even in reduced form, or did not have significant support to make them favorable for developers.

We used 6 bits in the construction of this instruction set as our goal is to limit the length of each opcode as much as possible. Note that this is different from the goal in maximizing bandwidth, as our goal now is to maximize instruction throughput. Given our short instructions, loading values into registers requires shifting in 4-bits at a time. SPASM has two registers that act as a pointer and working register, that can be used to perform jumps, arbitrary memory reads and writes, and basic arithmetic. We also have a `syscall` instruction that makes a real system call to the underlying operating system with parameters loaded from the SPASM state, allowing us to interact with the real world.

In SPASM we have implemented multiple example programs that we encrypted and loaded into a ExSpectre payload, which decrypts and emulates SPASM instructions only when the corresponding trigger program is running. We have implemented a *HelloWorld* program that prints to `stdout`, and a *FizzBuzz* program that demonstrates control flow and arithmetic operations while printing to stdout. Finally, we implemented a *ReverseShell* program that opens and connects a TCP socket to an attacker-chosen location before executing a local shell and allowing the remote adversary to issue shell commands on the victim machine. Figure 9 details the high-level flow of a SPASM payload.

Our *ReverseShell* program consists of 355 SPASM instructions, and makes six system calls to open a socket, connect to it, duplicate I/O file descriptors, and perform an `execve` system call to open a shell. In our tests using 5 iterations per decrypted instruction, the *ReverseShell* program takes just over 2ms to launch a reverse shell once triggered.

D. OpenSSL Trigger

To demonstrate a benign trigger application, we implemented an ExSpectre payload that would trigger when running concurrently with OpenSSL. We disable ASLR for simplification, but note that branch predictors can also be used to

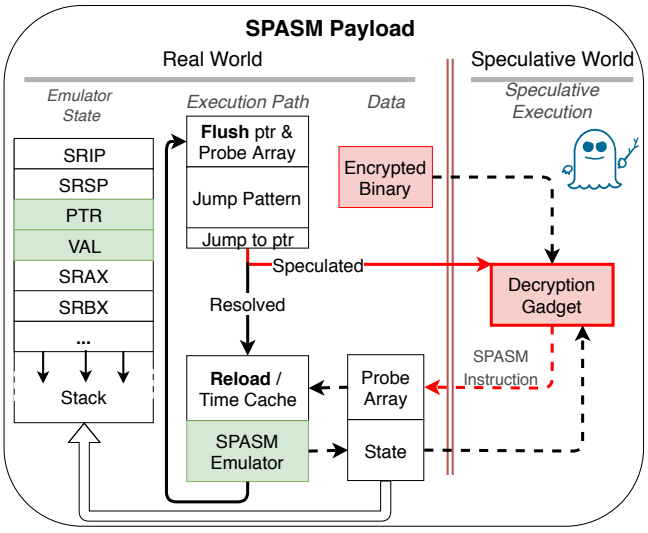

Fig. 9. **SPASM model** — Our SPASM emulator speculatively decrypts instructions, and emulates them in the real world. The *Speculative Computation* decrypts the encrypted SPASM binary using AES, returning the result through the side channel to allow the *Real World* to update the emulated state and make system calls on behalf of the speculative world.

determine ASLR offsets of co-resident applications, and our attack adjusted accordingly [14].

We used `gdb` to run an instance of an OpenSSL server (version 1.0.1f), and printed out every instruction executed and its address after a breakpoint on the `SSL_new` function. We then made a TLS connection to the server, which produced over 13 million instructions, including over 359,000 direct jumps and 28,000 indirect jumps. We then searched for the longest repeated set of more than 28 indirect jumps that ends with a unique jump (i.e. source and destination do not occur in the previous 28+ indirect jumps).

We discovered a candidate that corresponds to code in OpenSSL's `nistp256.c` that contained 31 indirect jumps repeated 254 times each handshake. This code is used during the TLS key exchange as the server computes the ECDHE shared secret. We made a list of 31 source-destination address pairs for these indirect jumps, and constructed a `jump`/`ret` chain to mimic the same jump pattern in our payload program. Our payload program mimics the first 30 indirect jump source/destination pairs, with a final jump going to a cache timing function in our payload program. However, due to the prior pattern, this last jump is frequently mis-speculated (about 3.5% of the time), and instead goes to the destination corresponding to the 31st jump in OpenSSL, which serves as our speculative entry point.

We ran experiments on an Intel Haswell i5-4590 CPU, with OpenSSL and our payload program pinned to the same core using `taskset`. We induced the jump pattern in OpenSSL by running Apache benchmark against it to generate thousands of TLS connections using the ECDHE key exchange with the secp256r1 curve (ECDHE-RSA-AES256-GCM-SHA384). When running Apache benchmark locally, our payload program reliably executes (speculatively) at the intended speculative entry point about 3.5% of the time. When apache benchmark runs on a remote machine, this rate drops to approximately 2.0%. Nonetheless, these are both sufficient to perform computation, as our payload can simply increase the amount of iterations needed to extract meaningful results from the speculative world.

We verified that our payload program did not execute at the speculative entry point when we ran other programs that simply consumed CPU on the same core. In addition, when we used Apache benchmark to create thousands of connections with a different cipher suite (DHE-RSA-AES128-GCM-SHA256), we similarly saw no speculation at the entry point. This could allow an adversary to use an obscure or uncommon cipher suite to trigger a malicious ExSpectre payload program on a remote server.

VII. Discussion

A. Defenses

We now address possible defenses to detecting and reverse engineering malware that uses ExSpectre.

1) Implemented Mitigations: Multiple patches and microcode updates have been developed to mitigate Spectre vulnerabilities, however, none of these entirely prevent ExSpectre malware from working, as they are generally not designed to protect programs that willingly use Spectre against themselves.

a) Indirect Branch Predictor Barrier: IBPB is used when transitioning to a new address space, allowing a program to ensure that earlier code's behavior does not effect its branch prediction. IBPB requires CPU and operating system support. However, we observe on Linux that processes running under the same user group do not receive IBPB protection, enabling ExSpectre when the trigger and payload run under the same group. Furthermore, IBPB does not prevent the speculative buffer overflow variant of ExSpectre described in Section V-B.

b) Single Thread Indirect Branch Predictors: STIBP prevents sibling hyperthreads from interacting via the indirect branch predictor. However, this does not prevent co-resident processes from cooperating when they run on the same logical core.

c) Indirect Branch Restricted Speculation: IBRS prevents code in less privileged prediction modes from influencing indirect branch prediction in higher privileges (e.g. the kernel). This does not prevent speculative execution in a willing payload program in a less privileged speculation mode.

d) Retpoline: is a software mitigation that replaces indirect jumps with a special call/overflow/return sequence, controlling where the CPU will speculate the indirect branch to a contained (and benign) section [54]. However, this defense is opt-in which ExSpectre binaries could simply choose to not use, or alternatively use the unaffected speculative buffer overflow variant.

2) Malware Detection: While not a primary goal of ExSpectre, we consider the ability of ExSpectre malware to hide from detection.

When using the cache side channel variant of ExSpectre, the payload program must at least occasionally watch this side channel, offering a potential method for detecting ExSpectre malware. Analysts could search for telltale signs of cache inference behavior, such as the use of `clflush` instructions or reading cycle timings. At the cost of performance, ExSpectre could choose to use a more subtle cache side channel that does not require this, such as `Prime+Probe`, or by exploiting race conditions between multiple threads to allow the speculative world to influence the behavior of the real world.

ExSpectre could also use another side channel method that avoids the cache to exfiltrate information from the speculative gadgets, such as the branch predictor itself [16], memory bandwidth, power utilization, or contention over other shared resources [27]. While cache channels tend to have the highest throughput, they are not the only resource that must be monitored to detect or prevent these types of attacks.

a) Anti-Virus Detectors: We verified that modern Anti-Virus technologies were unable to detect and flag ExSpectre malware. We used ClamAV, BitDefender and rkhunter, which mainly rely on signature and string based detection. BitDefender does feature support for unpacking or extracting malware, though appears to simply try unpacking using several known packers and encoding formats [1]. Thus, it is not surprising that these tools cannot detect ExSpectre.

b) Bare Metal: Modern malware often uses hardware minutia to identify and fingerprint execution environments in order to detect when it is under debugging or inspection [32],

[3], [43]. To prevent such identification, analyzers often employ "bare metal" execution [25], running the malware on dedicated hardware that allows introspection and observation of the system without interfering with its normal operation. This prevents malware from using so-called "red pill" checks to observe that it is under test (and hide its malicious behavior) [26]. However, to the best of our knowledge, no publicly available bare-metal environments allow introspection on the speculative state of the CPU, making it difficult to analyze ExSpectre malware. However, such environments could be useful for observing the behavior of ExSpectre malware in the presence of its trigger if available, as modifications in the real world could be easily tracked.

Symbolic execution has also been used to find environmental red pill checks [48]. However, such analysis would be ineffective against ExSpectre, as symbolic execution does not reason about speculative paths and how they might influence a program.

3) Reverse-engineering triggers: For program-based triggers, an analyst could attempt to find the trigger program by examining the execution path of the payload program, and locating a common indirect jump pattern between payload and potential triggers. Since both programs must share a common indirect jump pattern to interact via the indirect branch predictor, there must be some overlap which is unlikely to occur randomly between two programs.

We note that while the analyst may learn the execution path (and thus true indirect jump pattern) of the payload program, they may not be able to capture every potential execution path in all potential triggers. For example, in the OpenSSL trigger, the analyst may not have captured all potential indirect jump patterns, as doing so would require exhaustively connecting to OpenSSL with different cipher suites, extensions, and failed handshakes. However, the analyst can still make a list of indirect jump locations in a suspected trigger program, comparing these to the jumps taken by the payload. If there is significant overlap, the analyst could spend time to discover what inputs to the trigger program produce similar indirect jump patterns, thus discovering the trigger.

ExSpectre malware could attempt to thwart this analysis by using decoy indirect jumps that do not correspond with the trigger, but potentially correspond with other (non-trigger) binaries. In addition, this analysis method is ineffective at inspecting the speculative buffer overflow variant described in Section V-B, as it does not use a separate trigger program.

Alternatively an analyst may attempt to identify sections of the program or dead code that will be used to access the probe array and thereby find the speculative gadgets. However, identifying sections of memory that will access the probe array is equivalent to the "Must Alias" or "Points-To Problem" which has been proven undecidable without significant restrictions [46], [30].

B. Future Work

ExSpectre demonstrates a general model for hiding execution in the speculative world and examines the implications and limitations on modern processors. Given the wide-spread nature of the Spectre vulnerability and the ubiquity of side-channels, we believe that this work can be directly extended to other architectures, such as ARM, and other processors making use of speculative branch prediction.

1) Multiple Triggers: To create further difficulties for an analyst, or to further target the execution environment, it is possible to have the payload program to combine multiple triggers. Instead of requiring only a single trigger program, the payload could require multiple trigger programs to be running simultaneously, or in a particular order. Alternatively, the payload could combine trigger programs with input triggers, forcing an analyst to understand multiple variants simultaneously.

This could allow fine-grained targeting of malware. For instance, the attacker could distribute trigger programs through different channels to target different sets of victims, and have the ultimate payload only operate at the intersection of these groups. As an example, one trigger program could be distributed to a particular country (e.g. Iran), and another to a particular device globally (centrifuge controllers), resulting in the malicious payload (Stuxnet) only being revealed and executed on the intersection of these two groups.

2) Virtual Machines: Virtual environments could also be host to ExSpectre malware and triggers. For instance, malware on one EC2 instance could potentially be triggered by a trigger program on another seemingly unrelated instance. We have found that the hypercall context switch from guest to host on VirtualBox is lightweight enough that a trigger program running in a guest can activate a payload program running in the host on the same CPU core. However, we have so far been unable to go in the opposite direction, and similarly have yet to achieve guest-to-guest interference. More work is needed to determine if such barriers are possible to overcome, and if stronger isolation is needed in the virtual machine context.

VIII. Related Work

A. Weird Machines

ExSpectre shares many properties with *weird machines*—a machine which takes advantage of bugs or unexpected idiosyncracies in existing systems to perform arbitrary computation [6], [7]. In particular ExSpectre showcases the ability to use CPU speculation to compute.

Recently there has been a trend of features in modern processors such as multiple threads sharing system resources and optimizations done across the process isolation boundary which lead to opportunities for "weird machines" [11]. In particular, previous examples of "weird machines" include traditional vulnerabilities such as buffer overflows, format string exploits and return oriented programming [41], [19], [50]. Weird machines have also been built using operating system page faults, enabling the computation of arithmetic and logic operations without the use of traditional instructions [4]. ExSpectre extends research in "weird machines", and takes advantage of speculative execution to execute instructions that otherwise appear to be dead code.

B. Covert Channels

Spectre builds upon prior work on cache side channels, and similarly uses them to leak information from processes [44], [62], [42]. In ExSpectre, we use the branch predictor as a

covert channel [29] between the trigger program and malware payload, allowing the malware's (speculative) execution path to be influenced by the trigger.

Previous work has examined how to share information over covert channels, such as across virtualized environments on cloud systems [59], using L1 and L2 cache to share information [44], measuring temperature to create a thermal covert channel [35], [5], and taking advantage of processor architecture to leak information [57]. This includes using the branch predictor itself as a covert channel [15], [13], which ExSpectre similarly uses.

However, we note that the covert channel used in ExSpectre need not involve two cooperative programs, and we demonstrate using the benign OpenSSL as a non-colluding program involved in utilizing this covert channel.

C. Speculative Execution

ExSpectre builds on Spectre [28] and Meltdown [33] which leverage speculative execution to leak sensitive information from vulnerable processes. Follow up work has identified several new Spectre variants, including speculative buffer overflows and speculative store bypass [27], [37], and has investigated additional ways to leak information using branch predictors as a side channel [16]. Researchers have also leveraged Spectre and speculative execution more generally to demonstrate web-based vulnerabilities [18], [49] as well as to leak control flow, keys, and other information from the hardware isolation provided by Intel SGX [40], [10], [8], [31]. Spectre has additionally been proposed as a way to thwart taint tracking by using speculative execution to copy data between buffers [20]. ExSpectre likewise takes advantage of speculative execution, but with the goal of hiding arbitrary computation from reverse engineering, rather than extracting secrets from vulnerable programs. ExSpectre also benefits from new Spectre variants: as we showed, speculative buffer overflows ("Spectre 1.1") can be used as an alternative trigger for malware.

IX. Conclusion

We have presented ExSpectre, a model for hiding computation in speculative execution that is fundamentally different than existing methods of code obfuscation. Through a series of experiments we have classified the capabilities and limitations of this speculative primitive and demonstrated various example applications. We have demonstrated the potential of using speculative execution in several applications, including a Turing machine, SPASM emulator, remotely-triggered payloads, and AES decryption. We have also examined Intel's responses to Spectre and Meltdown and noted how their defenses affect ExSpectre and how further variants of Spectre can be adapted to ExSpectre. Current analysis techniques for reverse engineering are insufficient to reason about the behavior of these programs.

Ultimately, silicon and microarchitecture patches will be needed to secure CPUs against this kind of malware. Until then, attackers may iterate and find new variants of ExSpectre-like malware. In the meantime, new detection techniques and software-level mitigations are desperately needed.

Acknowledgements

We would like to thank Aimee Coughlin, Daniel Genkin, and Yuval Yarom for their initial discussions on the ExSpectre idea, and we additionally thank our reviewers and paper shepherd, Ahmad-Reza Sadeghi for helpful feedback on our work.

References

[1] "Bitdefender antivirus technology," 2018. [Online]. Available: https://www.bitdefender.com/files/Main/file/BitDefender_Antivirus_Technology.pdf

[2] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *Cryptographers Track at the RSA Conference*. Springer, 2007, pp. 225–242.

[3] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna, "Efficient detection of split personalities in malware." in *NDSS*, 2010.

[4] J. Bangert, S. Bratus, R. Shapiro, and S. W. Smith, "The page-fault weird machine: Lessons in instruction-less computation." in *WOOT*, 2013.

[5] D. B. Bartolini, P. Miedl, and L. Thiele, "On the capacity of thermal covert channels in multicores," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 24.

[6] S. Bratus, "What hacker research taught me," In Rss, 2009, accessed: 2018-05-01. [Online]. Available: http://www.cs.dartmouth.edu/sergey/hc/rss-hacker-research.pdf

[7] S. Bratus, M. Locasto, M. Patterson, L. Sassaman, and A. Shubina, "Exploit programming: From buffer overflows to weird machines and theory of computation," {*USENIX; login:*}, 2011.

[8] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[9] G. J. Chaitin, "Computing the busy beaver function," in *Open Problems in Communication and Computation*. Springer, 1987, pp. 108–112.

[10] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SGXPECTRE attacks: Leaking enclave secrets via speculative execution," *arXiv preprint arXiv:1802.09085*, 2018.

[11] S. M. D'Antoine, "Exploiting processor side channels to enable cross vm malicious code execution," Ph.D. dissertation, Rensselaer Polytechnic Institute, 2015.

[12] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM computing surveys (CSUR)*, vol. 44, no. 2, p. 6, 2012.

[13] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Covert channels through branch predictors: a feasibility study," in *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2015, p. 5.

[14] ——, "Jump over ASLR: Attacking branch predictors to bypass ASLR," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[15] ——, "Understanding and mitigating covert channels through branch predictors," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 1, p. 10, 2016.

[16] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, D. Ponomarev *et al.*, "Branchscope: A new side-channel attack on directional branch predictor," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 693–707.

[17] A. Fog, "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, AMD and VIA CPUs," *Copenhagen University College of Engineering*, vol. 93, p. 110, 2011.

[18] D. Genkin, L. Pachmanov, E. Tromer, and Y. Yarom, "Drive-by key-extraction cache attacks from portable code," 2018.

[19] gera and riq, "Advances in format string exploiting." Phrack Magazine , 59(7), July 2001., 2001. [Online]. Available: http://www.phrack.org/archives/59/p59-0x07.txt

[20] S. Guelton, "Spectre is not a Bug, it is a Feature," https://blog.quarkslab.com/spectre-is-not-a-bug-it-is-a-feature.html, 2018.

[21] F. Guo, P. Ferrie, and T.-C. Chiueh, "A study of the packer problem and its solutions," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2008, pp. 98–115.

[22] R. Herken, "The universal Turing machine: A half-century survey," 1992.

[23] J. Horn, "Project Zero: Reading privileged memory with a side-channel," https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html, 2018, accessed: 2018-05-01.

[24] S. Jana and V. Shmatikov, "Abusing file processing in malware detectors for fun and profit," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 80–94.

[25] D. Kirat, G. Vigna, and C. Kruegel, "Barebox: efficient malware analysis on bare-metal," in *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 2011, pp. 403–412.

[26] ——, "Barecloud: Bare-metal analysis-based evasive malware detection." in *USENIX Security Symposium*, 2014, pp. 287–301.

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

[28] 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, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[29] B. W. Lampson, "A note on the confinement problem," *Communications of the ACM*, vol. 16, no. 10, pp. 613–615, 1973.

[30] W. Landi, "Undecidability of static analysis," *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 1, no. 4, pp. 323–337, 1992.

[31] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *26th USENIX Security Symposium, USENIX Security*, 2017, pp. 16–18.

[32] M. Lindorfer, C. Kolbitsch, and P. M. Comparetti, "Detecting environment-sensitive malware," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2011, pp. 338–357.

[33] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[34] G. Maisuradze and C. Rossow, "Speculose: Analyzing the security implications of speculative execution in CPUs," *arXiv preprint arXiv:1801.04084*, 2018.

[35] R. J. Masti, D. Rai, A. Ranganathan, C. Müller, L. Thiele, and S. Capkun, "Thermal covert channels on multi-core platforms." in *USENIX Security Symposium*, 2015, pp. 865–880.

[36] M. Miller, "Analysis and mitigation of speculative store bypass (CVE-2018-3639)," May 2018. [Online]. Available: http://www.guru3d.com/news-story/intel-has-to-delays-patches-for-new-spectre-ng-vulnerabilities.html

[37] ——, "Analysis and mitigation of speculative store bypass (cve-2018-3639)," 2018. [Online]. Available: https://blogs.technet.microsoft.com/srd/2018/05/21/analysis-and-mitigation-of-speculative-store-bypass-cve-2018-3639/

[38] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *Security and Privacy, 2007. SP'07. IEEE Symposium on*. IEEE, 2007, pp. 231–245.

[39] J. Oberheide, M. Bailey, and F. Jahanian, "Polypack: an automated online packing service for optimal antivirus evasion," in *Proceedings of the 3rd USENIX conference on Offensive technologies*. USENIX Association, 2009, pp. 9–9.

[40] D. O'Keeffe, D. Muthukumaran, P.-L. Aublin, F. Kelbert, C. Priebe, J. Lind, H. Zhu, and P. Pietzuch, "Spectre attack against SGX enclave," 2018, accessed: 2018-05-01. [Online]. Available: https://github.com/lsds/spectre-attack-sgx

[41] A. One, "Smashing the stack for fun and profit." Phrack Magazine, 1996. [Online]. Available: phrack.org/issues/49/14.html#article

[42] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *Cryptographers Track at the RSA Conference*. Springer, 2006, pp. 1–20.

[43] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi, "A fistful of red-pills: How to automatically generate procedures to detect CPU emulators," in *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, vol. 41, 2009, p. 86.

[44] C. Percival, "Cache missing for fun and profit," 2005.

[45] G. Poulios, C. Ntantogian, and C. Xenakis, "Ropinjector: Using return oriented programming for polymorphism and antivirus evasion," *Black-hat USA*, 2015.

[46] G. Ramalingam, "The undecidability of aliasing," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1467–1471, 1994.

[47] J. Rutkowska, "redpill... or how to detect VMM using (almost) one CPU instruction," 2004. [Online]. Available: https://web.archive.org/web/20110726182809/http://invisiblethings.org/papers/redpill.html

[48] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Security and privacy (SP), 2010 IEEE symposium on*. IEEE, 2010, pp. 317–331.

[49] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, "NetSpectre: Read arbitrary memory over network," 2018.

[50] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 552–561.

[51] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, 2016, pp. 1–16.

[52] A. Swinnen and A. Mesbahi, "One packer to rule them all: Empirical identification, comparison and circumvention of current antivirus detection techniques," *BlackHat USA*, 2014.

[53] A. M. Turing, "On computable numbers, with an application to the Entscheidungsproblem," *Proceedings of the London mathematical society*, vol. 2, no. 1, pp. 230–265, 1937.

[54] P. Turner, "Retpoline: a software construct for preventing branch-target-injection," 2018. [Online]. Available: https://support.google.com/faqs/answer/7625886

[55] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers," in *2015 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2015, pp. 659–673.

[56] F. Wang and Y. Shoshitaishvili, "angr - the next generation of binary analysis," in *Cybersecurity Development (SecDev), 2017 IEEE*. IEEE, 2017, pp. 8–9.

[57] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE, 2006, pp. 473–482.

[58] H. Wong, "Measuring reorder buffer capacity," May 2013. [Online]. Available: http://blog.stuffedcow.net/2013/05/measuring-rob-capacity/

[59] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: High-speed covert channel attacks in the cloud." in *USENIX Security symposium*, 2012, pp. 159–173.

[60] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, l3 cache side-channel attack," in *USENIX Security Symposium*, 2014, pp. 719–732.

[61] M. Zalewski, "American fuzzy lop," 2015.

[62] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 305–316.