sciendo

TATRA MOUNTAINS
Mathematical Publications
DOI: 10.2478/tmmp-2019-0010
Tatra Mt. Math. Publ. **73** (2019), 131–144

SELF-SPECTRE, WRITE-EXECUTE AND THE HIDDEN STATE

Gregory Morse

Eötvös Loránd Tudomány Egyetem University, Budapest, HUNGARY

ABSTRACT. The recent Meltdown and Spectre vulnerabilities have highlighted a very present and real threat in the on-chip memory cache units which can ultimately provide a hidden state, albeit only readable via memory timing instructions [Kocher, P.—Genkin, D.— Gruss, D.— Haas, W.—Hamburg, M.—Lipp, M.––Mangard, S.—Prescher, T.—Schwarz, M.—Yarom, Y.: *Spectre attacks: Exploiting speculative execution,* CoRR, abs/1801.01203, 2018]. Yet the exploits, although having some complexity and slowness, are demonstrably reliable on nearly all processors produced for the last two decades.

Moving out from looking at this strictly as a means of reading protected memory, as the large microprocessor companies move to close this security vulnerability, an interesting question arises. Could the inherent design of the processor give the ability to hide arbitrary calculations in this speculative and parallel side channel? Without even using protected memory and exploiting the vulnerability, as has been the focus, there could very well be a whole class of techniques which exploit the side-channel. It could be done in a way which would be largely unpreventable behavior as the technology would start to become self-defeating or require a more complicated and expensive on-chip cache memory system to properly post-speculatively clean itself. And the ability to train the branch predictor to incorrectly speculatively behave is almost certain given hardware limitations, and thus provides exactly this pathway.

A novel approach looks at just how much computation can be done speculatively with a result store via indirect reads and available through the memory cache. A multi-threaded approach can allow a multi-stage computation pipeline where each computation is passed to a read-out thread and then to the next computation thread [Swanson, S.—McDowell, L. K.—Swift, M. M.—Eggers, S. J.––Levy H. M.: *An evaluation of speculative instruction execution on simultaneous multithreaded processors,* ACM Trans. Comput. Syst. **21** (2003), 314–340].

2010 Mathematics Subject Classification: 62K05.
Keywords: x86, x86-64, Spectre, Meltdown, assembly language, self-modifying code, white-box cryptography, side-channel vulnerabilities, CPU cache, speculative execution, predictive branching.

Through channels like this, an application can surreptitiously make arbitrary calculations, or even leak data without any standard tracing tools being capable of monitoring the subtle changes. Like a variation of the famous physics Heisenberg uncertainty principle, even a tool capable of reading the cache states would not only be incredibly inefficient, but thereby tamper with and modify the state. Tools like in-circuit emulators, or specially designed cache emulators would be needed to unmask the speculative reads, and it is further difficult to visualize with a linear time-line.

Specifically, the AES and RSA algorithms will be studied with respect to these ideas, looking at success rates for various calculation batches with speculative execution, while having a summary view to see the rather severe performance penalties for using such methods.

Either approaches could provide for strong white-box cryptography when considering a binary, non-source code form. In terms of white-box methods, both could be significantly challenging to locate or deduce the inner workings of the code. Further, both methods can easily surreptitiously leak or hide data within shared memory in a seemingly innocuous manner.

1. Introduction

Study of various computational modes which execute with various layers of indirection is one of the primary obfuscation methods used in modern software. Two new techniques in this regard are proposed and their difficulty in implementation and quantitative performance are evaluated. Part of the motivation behind the techniques is based on findings from studying white box cryptography which has placed a demand on finding mathematically or practically sound obfuscation methods. The sophistication of tracing tools and the capabilities of those using them has made white box schemes flimsy and highly vulnerable to key extraction.

The first is motivated by recent processor vulnerability disclosures. Speculative execution has become a trending topic in 2018 due to researchers publicly disclosing large modern processor bugs which have existed for nearly two decades. These vulnerabilities known as Spectre and Meltdown show not only the dangers and pitfalls of speculative execution but also the utility and potential capabilities that perhaps were previously ignored.

The second builds upon a model previously introduced for what was denoted as pure and infinitely self-modifying code [4]. Using a less limiting definition which allows any code which does not read data but always merely writes memory or executes control flow transfer instructions, an alternative and perhaps more practical Turing-complete model is established.

2. Background

This paper will very closely follow the x86 and x86-64 processor architectures, though there is no reason that the techniques and tactics discussed cannot be seen in generality and adapted to other process architectures where applicable. The models are thus presented in a general manner, while the inner-workings, details and explanations of some design decisions can be derived from the specific platforms. The process used and the type of low-level thinking is thus important to keep in mind if trying to translate this work across architectures.

The Meltdown vulnerability, which is also known as Spectre-V3: Rogue Data Cache Load (RDCL), primarily effects Intel processors. It exploits the way out-of-order execution works on the processor. A memory space of 256 local 512-byte pages is flushed from the cache representing every possible value of a byte. To read a single byte of protected memory, it executes a pair of instructions which reads the byte of protected memory followed by an indirect read based on this previously read byte as an indexing value into the prepared flushed pages. This protected read will ultimately not execute as it only speculatively executed for out-of-order processing. It will silently fail the privilege check and be skipped or conceivably fail with an interrupt exception. Yet the cache of one of the 256 local pages will have been made dirty which has opened up a straight-forward timing side channel to determine the value.

Spectre is a term referring to the whole class of speculative execution vulnerabilities and the fact that the problem will be troubling the chip-makers for some years to come. Beyond out-of-order execution, an important strategy called branch-prediction is implemented which can significantly speed execution by correctly speculatively executing a branch in code which the processors complicated internal state shows will be more likely to execute. The vulnerability thus lies in mis-prediction down a branch. The vulnerability comes in 3 categories including the Meltdown one previously mentioned. The remaining ones of the initial vulnerability disclosure are Spectre-V1: Bounds Check Bypass (BCB), Spectre-V2: Branch Target Injection (BTI). There are also 2 more newly discovered ones which are Variant 3a: Rogue System Register Read (RSRE) and Variant 4: Speculative Store Bypass (SSB). The focus in this paper is only on Spectre-V1 which is one the hardware vendors are largely choosing not to try to fix, but rather to all software implementations to mitigate this vulnerability usually through compiler strategies.

The details of how the original C source code [5] for the vulnerability posted on Github as a proof of concept (PoC) is functioning are necessary to continue further with making a model based on it. When generalizing Spectre for all processor models in the x86 family over recent generations, up to 3 processor intrinsic instructions need to be used. It should also be noted that malware

scanners are currently flagging the code pattern of the PoC due to the potential malicious capability.

These processor instructions all revolve around flushing pages from the cache and measuring precise timing information. The modern `CLFLUSH` instruction will flush cache memory, marking it to be fetched from memory on the next read. On older processors this would be implemented by the long Streaming Single Instruction, Multiple Data (SIMD) Extensions (SSE) instruction `_mm_stream_ps` which flushes the cache 4 bytes at a time. For precision timing, `RDTSCP`, if available, provides the current processor ticks when serialized. If the serialized timing instruction is not available, then using a memory fence instruction `MFENCE` with the older `RDTSC` processor tick retrieval instruction will avoid instruction reordering for correct measurement. If `MFENCE` is not present on the processor either, then `RDTSC` can be used alone as these older processors fortunately do not appear to reorder timing measurement instructions.

There are two more notable parts of the implementation to understand. Namely the branch predictor must be mis-trained and fooled so that speculative execution can be controlled as it is seen in Fig. 1, and lastly that when reading out cache timings, stride prediction where the processor starts fetching memory pages in advance based on in-order reading patterns must also be prevented. Instead of reading forward in a linear order, the following simple strategy is used: `mix_i = ((i * 167) + 13) & 255;`.

```
for (volatile int z = 0; z < 100; z++) {}
x = ((j % 6) - 1) & ~0xFFFF;
x = (x | (x >> 16));
x = training_x ^ (x & (malicious_x ^ training_x));
```

FIGURE 1. Mis-training the Branch Predictor with C code.

3. Self-Spectre

A so-called "Self-Spectre" implementation thus begins with determination of the required properties for the actual speculative function where the precise code will execute. This code must be maximally efficient in net clock cycles required to execute on a CPU at least on average. This is an unusual methodology as generally code optimization does not focus on squeezing performance into a precise code location. By minimizing both the number of assembler instructions, and minimizing the clock cycles of each instruction, a near optimum can be achieved. Further this code would need to be pre-compiled or built in-memory and emitted on the fly before its invocation. Compiler optimizations are sophisticated and well tested enough to rely on for generation of such optimal code, or at least

a skeletal basis which can be converted both manually or at runtime by code into a final code slice. Further inside this critical section, the maximal performance is needed, while outside the region, more complex and slow code can run without harming the technique.

By custom assembling the execution code in-memory, unnecessary reads in the critical code can be eliminated by replacing for example memory reads with the memory actually read at those addresses as immediate values. This is why the level of optimization is quite unique when compared to the normal perspective applied in industry. Of course, some obvious factors based on processor architecture are observed such as: 64-bit operations will perform faster on a 64-bit processor, 32-bit operations are often faster than 64-bit operations, and 64-bit addressing is typically faster than 32-bit addressing.

A key question that must be determined is: How deep is the speculative branching pipeline? This is a difficult to answer question with any simple formulas, as the modern processors have become complex state machines whose behavior is so influenced by parallel and prior execution factors, not to mention hardware algorithms hidden behind trade secrets and so forth. Parallelism and task switching are particularly unpredictable. The types of cache on the processor are reasonable well-known including the last-level (often L2 or in newer processors where L2 has become unique per core, L3) cache, a form of cache which is shared between cores on a multi-core processor. This also adds a factor of potential interference in the cache. Instruction clock cycles can be used to determine the expected maximal execution time, but it is not capable of giving the minimal or average time. Empirical analysis is the easiest way to determine the average time, while very technical and detailed knowledge of processor internals are the only way to find the minimal. It is however known and assumed that later models of processors and later processors in a given model, are getting the same or further execution depth. The measurement of speculative branching depth, is the net time taken from the time the speculative processing starts until the actual branch is reached.

The actual function to use for hidden computation which is speculatively executed resides in the PoC C code (in file spectre.c). It has a function that is treated in the original security vulnerability context as a victim function and named thereby. The memory cache pages are cleared, where each page represents one possible byte value that will be determined by which cache page was loaded. The branch predictor is trained to make a false detection. This compiled code has 3 assembly instructions for the jump and comparison portion of the code, followed by either 7 or 11 instructions for the cache read, depending if on a 32 bit or 64 bit architecture. Fig. 2 shows this function which the original PoC code uses for exploitation. In our new context, it will be used for secret computation as opposed to an exploit that reads protected memory.

```
char * secret =
  "The Magic Words are Squeamish Ossifrage.";
uint8_t temp = 0; /* Used so compiler
  won't optimize out victim_function() */
void victim_function(size_t x) {
  if (x < array1_size) {
    temp &= array2[array1[x] * 512];
  }
}
```

FIGURE 2. The chance to hide the state.

Whatever operations will be used, it is at least a weak requirement that the computation is divisible into roughly equal chunks of work. And the smallest such chunk of work should be sufficiently one-way. For a practical application of the technique, a couple of common cryptographic one-way operations were chosen. Namely the Advanced Encryption Standard (AES) in Electronic Codebook (ECB) mode and Rivest-Shamir-Adleman (RSA) encryption primitives. AES with an unknown key can be implemented solely using table lookups and logical `xor` operations. More specifically AES-128 has 10 rounds where the first and last round are slightly incongruent. The first round only used logical `xor` operations, while the last is just a single byte table lookup and recombination. RSA on the other hand is merely modular exponentiation. Realistically RSA must use big integer libraries to achieve good encryption strengths starting least at RSA-1024 or RSA-2048. However for a PoC, 32-bit RSA would of course only require a 32-bit multiplication with a 64-bit output. This is quite natural to implement in efficient assembly code. It also requires a 64-bit divisor divided by a 32-bit dividend with a 32-bit remainder. Although the quotient cannot overflow 32-bits of output, the remainder potentially can. These math primitives mention comes from the fact that modular exponentiation is simply some rounds of modular multiplication. In a naive and inefficient implementation, it requires E modular multiplications where E is the exponent of modular exponentiation. Contrast this to an efficient implementation which requires $\log_2 E$ + `count_bits(E)` modular multiplications.

To further assist with AES are processor intrinsics, which x86 fully supports from both Intel and AMD. Sébastien Riou (3rd place for Strawberries in 2017 WhiBox contest) has provided code for this [6] in an AES brute force toolkit with a reference AES-128 implementation. The key data is in a slightly different form than typical reference C implementations. A snippet of this code doing the encryption is shown in Fig. 3.

The collected results for the depth of modern processors showed a remarkably deep speculative pipeline. Statistics are provided for both AES-128 with 10 rounds and RSA-32 in Tables 1 and 2 respectively. Sample collected output is shown in Fig. 4 highlighting where the boundary of speculative execution

```
#include <wmmintrin.h>
const u32 *rk = key->rd_key;
__m128i m = _mm_loadu_si128((__m128i*)in);
m = _mm_xor_si128(m, *((__m128i*)&rk[0]));
m = _mm_aesenc_si128(m, *((__m128i*)&rk[4]));
m = _mm_aesenc_si128(m, *((__m128i*)&rk[8]));
m = _mm_aesenc_si128(m, *((__m128i*)&rk[12]));
m = _mm_aesenc_si128(m, *((__m128i*)&rk[16]));
m = _mm_aesenc_si128(m, *((__m128i*)&rk[20]));
m = _mm_aesenc_si128(m, *((__m128i*)&rk[24]));
m = _mm_aesenc_si128(m, *((__m128i*)&rk[28]));
m = _mm_aesenc_si128(m, *((__m128i*)&rk[32]));
m = _mm_aesenc_si128(m, *((__m128i*)&rk[36]));
m = _mm_aesenclast_si128(m, *((__m128i*)&rk[40]));
_mm_storeu_si128((__m128i*)out, m);
```

FIGURE 3. AES-128 encryption via processor instrinsics.

left incomplete results. Since the interpretation of these results still requires processor knowledge, these are drawn for the reader. First note that modular multiplication for 6 rounds always fails on 32-bit compiled C or assembly, 93+5*44 or 70 instructions respectively, while 64-bit compiled C and assembly require 47 or 46 instructions respectively. Multiplication and division are expensive and require many more clock cycles which explains why less instructions can execute in RSA than with AES. As AES will use table lookups and logical `xor` operations, about three times as many instructions can potentially execute more than with the expensive arithmetic instructions. Since the quotient of the division instruction is not needed when computing remainder, hand written assembly can use two division calls to avoid overflow which is much faster

TABLE 1. AES-128 Speculative (10 rounds).

	Intrinsics	C Ref.	1 round Instrinsics	1 round C Ref.
32-bit Correct	0%	0%	100%	0%
32-bit Confident	-	-	100%	-
32-bit instrs.:	201	1030	145	313
64-bit Correct	100%	0%	100%	0%
64-bit Confident	100%	-	100%	-
64-bit instrs.:	133	737	115	195

TABLE 2. RSA Speculative (number of modular multiplications `(a * (uint64_t)b) % c`).

	2 round C	4 .asm	4 round C	5 .asm	5 round C
32-bit Correct	100%	100%	0%	100%	0%
32-bit Confident	100%	100%	-	100%	-
32-bit instrs.:	53+2*44	54	73+4*44	62	83+5*44
64-bit Correct	100%	100%	100%	0%	0%
64-bit Confident	100%	100%	100%	-	-
64-bit instrs.:	31	38	39	42	43

than using library instructions, hence why 32-bit C compiled code is very limited with multiplication and division involving 64-bit operands. This is largely a compiler issue as the type system in C has historically had input and output of arithmetic instructions as the same bit size despite that the assembly language instructions never have followed such an inefficient paradigm. 64-bit code can sometimes execute slower due to instruction size increase and larger arithmetic units. But ultimately 64-bit code typically executes faster because far less instructions are needed. Due to out-of-order execution, and the complex nature of modern processors, it is impractical to even calculate precise clock cycles by hand, though precise information is available through the processor based on the actual clock counter. Basically empirical analysis such as those tabulated, are required to determine the depth of specific operations before finalizing a solution.

```
Correct AES Intrinsic = 0xF487F82611599E2562D2555AF60E9642 . . . Time: 0.003426 seconds
Unclear: Incorrect 0xF487F82611599E2562D2555AFFB9FFB3 score=770 821 813 811 718 510
814 747 228 149 70 5 0 1 0 1 (second best: 0xCA663B8380FFFF8181FFFFBEFE80FE79 score=1 1
2 1 1 0 0 2 1 0 0 1 0 1 0 1) Time: 0.500205 seconds
Unclear: Incorrect 0xF487F82611599E2562D2555AF6F6F1FF score=757 806 795 799 687 479
798 715 184 113 76 2 1 1 1 0 (second best: 0xFF81BEF1BE5ABEF1BEFFF6F62EB7BEFE
score=0 1 2 1 1 1 1 1 2 0 1 1 1 1 1 0) Time: 0.504171 seconds
Success: Incorrect 0xF487F82611599E2562D2555AFFFFFFFF score=787 832 826 822 726 553
816 749 236 195 82 1 0 0 0 0 (second best: 0xF1F679C1F6FFFFB3F6EAFFFFFEFEFEFE score=1
1 1 1 1 0 0 1 1 1 0 0 0 0 0 0) Time: 0.491277 seconds
Unclear: Incorrect 0xF487F82611599E2562D2555AF6BEFFBE score=743 793 787 780 659 507
776 707 259 186 85 2 1 1 0 2 (second best: 0x33FFF6F1A36466FF2E28FFED3966FEB3 score=2 0
1 1 1 1 1 0 2 1 0 1 1 1 0 1) Time: 0.500223 seconds
Unclear: Incorrect 0xF487F82611599E2562D2555AEBF6B922 score=701 741 729 722 613 486
716 647 238 162 81 2 1 1 1 1 (second best: 0x41BECACBF9FFFF867BF6BE0186BE66FF score=1
1 1 1 1 0 0 1 1 1 1 1 1 1 1 0) Time: 0.528884 seconds
. . .
Total Successes: 0 Confidence Success: 0 Time: 5.117587
```

FIGURE 4. Depth Statistical Results – Sample Output: 32-bit AES using intrinsics with 12 bytes correct.

4. Write-Execute

The next proposed model termed based on its memory access properties is Write-Execute. This is somewhat of an assembly language programming challenge due to the awkward constraint of not using direct memory read instructions which is the norm. The three basic memory access flags have been read, write and execute. In some contexts, "execute" is also sometimes considered to be a form of read, as implicitly instructions must be read to be executed. But from the strict point of view of the processor, reading is explicitly done by the code, and the implicit memory fetch read for instruction execution is only requiring the execute flag. So there can be no reads from memory besides the read required to fetch a current instruction. This is exactly what the technique refers to. From the point of view of a tracing tool, it will see a linear fetch-write sequence but no patterns unless the execution loops. Just as had occurred with pure, infinitely self-modifying code, a novel machine code pattern emerges.

The model is further inspired from the traditional von Neumann view of computer architecture where data and code are effectively synonymous. However, in practical modern computers, the traditional view is that all memory is data, some of which just so happens to be code. So an alternative view would be that all memory is code, some of which just so happens to be data flowing through it. This alternative view continues expanding knowledge for obfuscation, efficiency and complexity problem issues. The processor view, it should be noted, is merely to fetch, decode and execute the contents of the instruction pointer.

The model requires some primitives that will require table lookups or in the case of binary operations, double table lookups. These operations include firstly unary bitwise left shifts by a constant amount between 1 and 7 from 8-bits to 16-bits for relative jumps. The unary `not` primitive is required, and negation via `neg` is optional. Pure 8-bit binary operations required are `and`, `or` and `xor`. Finally 8-bit binary operations with an extra single shift or carry bit needed are bitwise shift left, bitwise shift right, addition and subtraction. By building up from these primitives, next higher level primitives can be constructed including: copying data and conditional jump via unconditional jumps. Finally higher bit sizes can be constructed including 32-bit addition and subtraction, 32-bit multiplication with 64-bit output and division with a 64-bit divisor, and 32-bit dividend, quotient and remainder. This effectively allows simulation of 32-bit processor instructions which only operate with read addressing memory modes.

The practical details are also very assembly language involved and specific. First of all, no registers are used in the instructions, and only jumps and data movement instructions are used. The stack could be made as an exception to this by allowing the `call`, `push` and `pop` instructions thus avoiding a rather tedious

emulation of the stack through a no-operation reserved code area. 32-bit inequalities, comparisons with a conditional jump, modular exponentiation, and even the table lookups and boolean operations for AES-128 are easily implemented at this point. Because 64-bit addressing mode uses RIP-relative instruction addressing, a simple macro introduced here as `ONLY64` will evaluate to its value on 64-bit machines and simple to 0 on 32-bit platforms. 32-bit absolute addressing is generally disadvantageous and slower in this context, causing more complicated code for double table lookups specifically and that operation being the worst performance difference area between the two addressing modes.

To appreciate the implementation of the model, some code snippets will be introduced to demonstrate the specific assembler techniques. The first technique of using jump tables is seen in Fig. 5. A macro is used to define a table of every possible byte value of the unary logical negation primitive, and shift left by a constant from 8-bits to 16-bits is used to scale the jump.

Copying memory is demonstrated via the implementation of the `xor` binary instruction. Since there is a double table lookup, one of the arguments must not only specify a jump to modify, but modify a location to where the other

```
DEFINENOT macro Index, Lbl
        &Lbl&_&Index&: mov byte ptr [Lbl+6], not Index
        jmp [Lbl] ;; preprocessor wrongly assumes 10 bytes for jump
                  ;; unfortunately not 2 or 5 bytes
IF Index GT 0f6h ;; preprocessor cannot look at distance to future labels
        db 7 dup (090h)
ELSE
        db 4 dup (090h)
ENDIF
endm

WriteExecNOT PROC
notarg:: mov byte ptr [shlarg4+2], 0
mov dword ptr [shlres4+1+2], notargshl + 1 - shlret4
mov dword ptr [shlret4+1], notargshl - shlret4 - 5
jmp WriteExecSHL4

notargshl: jmp notres_0+08000h ;; E9 00 80 00 00
idx_value = 0
WHILE idx_value LT 100h
        DEFINENOT %idx_value, notres
        idx_value = idx_value + 1
ENDM
notres:: mov byte ptr [notres+6], 0
notret:: jmp $+80000000h ;; force long jump
WriteExecNOT ENDP
```

FIGURE 5. Example of `not` instruction.

argument must modify its jump. This is due to there being 256 different second table lookups, and that jump must be put at the right indexed place. Otherwise the code is relatively straight forward compared to a unary operation. Fig. 6 gives the relevant assembly code to understand what is involved in code copying which implements a clever and simple looping structure which is safely re-executable.

```
mov dword ptr [xorargshl1 + 1+2], xorj+2-ONLY64(xorargshl1-1-2-4-2)
mov byte ptr [xorloop + 1], xorhead-xorloop-2
jmp xorargshl1
xorhead: mov byte ptr [xorloop+1], 0 ;exit loop
xorargshl1: mov word ptr [xorj+2], 8000h
ifdef X64
mov dword ptr [xorargshl1 + 1+2], xorargshl2+2+2-xorargshl1-1-2-4-2
else
mov dword ptr [xorargshl1 + 1+2], xorargshl2jmp+1
endif
xorloop: jmp xorhead

ifndef X64
xorargshl2jmp: jmp xorargshl2begin+08000h ;; E9 00 80 00 00
$\ldots$
xorargshl2: mov word ptr [xorarg1_0+1], 8000h
xorj: jmp xorarg1_0+0800000h ;; E9 00 00 80 00
```

FIGURE 6. Example snippet from the `xor` binary operation which copies the first argument to modify its jump, and modify the location where the second argument should modify its jump.

The last important operation is that of an explicit conditional jump, which given the loop structure in the copy operation can already be deduced as being possible. This is in fact even simpler than constructing a loop and simply using the constant bit shifting operation and an unconditional jump whose displacement is overwritten precisely. Fig. 7 demonstrates this last construct from the context of the middle of a chain of adders where the carry bit may or may not be transferred to the next adder.

A full compiler can be written and Turing-completeness could be readily proven from the model as already presented. As for its white-box cryptographic implications, then to undo Write-Execute, this would at least require a custom coded tool. This tool would need to identify math primitives including boolean operations, bit shifts, addition, subtraction, multiplication, division and modulo operations. It would need to unwind the data copying and conditional jump constructs. Then it would need to determine data which was encoded as effectively no-operation instructions. At this point it could construct a semantic equivalent with simple assembly instructions. As for tracing tools, they would see execution paths and memory writes. This model would certainly not be strong enough for white-box cryptography with well-known algorithms like AES-128.

```
carry&Idx&: mov byte ptr [shlarg+2], 0
mov dword ptr [shlres+1+2], carry&Idx&retval + 1 - ONLY64(shlret)
mov dword ptr [shlret+1], carry&Idx&shlret - shlret - 5
jmp WriteExecSHL1

carry&Idx&retval: jmp nocry&Idx&+8000h ;; E9 00 80 00 00
nocry&Idx&: jmp add&Sz&arg1_&NxtIdx&copy ;; EB xx
hascarry&Idx&:
```

FIGURE 7. Example snippet from 32-bit `add` to conditionally perform the carry increment.

5. Performance

Simulation results for AES-128 are presented in Table 3 while those for 32-bit RSA are in Table 4. The grayed out results which took over 500 seconds for

TABLE 3. AES-128 (1000 operations in succession).

	Intrinsics	C Ref.	Spec. I.	S.I.1	Spec. C. R.	Write-Exec
32-bit	0.000047	0.00096s	569s	2.06s	558s	1.35s
64-bit	0.000051s	0.00027s	2.35s	1.38s	549s	1.22s

TABLE 4. 32-bit RSA/modular exponentiation (100 operations in succession).

	Fast C	Write-Execute	Simple C	Speculative
32-bit	.000067s	55.33s	0.000036s	0.40s (5 .asm)
64-bit	.000057s	48.80s	0.000019s	0.43s (4 .asm)

AES-128 represent the worst case speed when failure occurs for speculative execution and no clear cache pages can be determined as having the correct value after some number of bytes. The AES-128 intrinsics were compared to C reference (C Ref.) implementations. Then speculative intrinsics (Spec. I.), speculative intrinsics with only a single round of AES-128 (S. I.1), and the speculative C reference implementations (Spec. C. R.) were compared. Sometimes a single round of AES could perform with intrinsics, while all 10 rounds could not, as it is seen in 32-bit mode on the processor.

Note that Write-Execute implements fast modular exponentiation and hence its C implementation is termed "Fast C". Yet the simple naive C implementation used for the speculative execution to make dividing it into chunks easier, shows a better performance with "Simple C". This is explained by the fact that the Simple C and speculative execution used small fixed exponents while the other model allowed random 32-bit exponents.

6. Conclusion

For the first hidden state model, unlike with the Spectre and Meltdown security vulnerabilities, there is no software mitigation here as it is by hardware design, and at the discretion of the software designer to use or not use mitigations. Compiler authors have released versions of libraries, and compiler options to enable such mitigations with the cost of performance. One way this can be detected by an adversary is through looking for low-level memory flushing and precision clock reading assembly instructions. Hardware mitigation is theoretically possible but at a great cost for such a marginal issue easily mitigated by software. The state machine complexity would increase dramatically and this has not even been discussed as a possibility for the time being given the excessive complexity to mitigate something that can be merely documented and left to software designers. In order to do this, all speculative state changes would have to be unwound, and it is beyond the scope of any known security vulnerability. Further research is however needed in techniques involving mis-training of the branch predictor for advanced code hiding. Furthermore, leaking data surreptitiously across non-memory protected boundaries is also a new potentially dangerous security risk for future study.

Write-Execute has provided a fully Turing-complete proof of concept for a new type of pure self-modifying code. Although it is slow, it would provide a good obfuscation method for custom code where such performance penalty can be tolerated.

The two techniques provide new perspective and insight into ways future protection schemes can be built. Although quite different, they both provide software designers further paradigms based on the hardware that effectively obfuscate and hide or alter computation, including exploiting the complex state machine the processor is built on top of, or using assembly language in very limited ways to achieve generality. Future work in this area could hopefully even have an effect on processor architecture itself as making it more easy for those legitimately protecting commercial software, while simultaneously making it more difficult for malicious software to be written would positively effect the bottom line of the whole IT software industry.

REFERENCES

[1] KOCHER, P.—GENKIN, D.—GRUSS, D.—HAAS, W.—HAMBURG, M.—LIPP, M.–
–MANGARD, S.—PRESCHER, T.—SCHWARZ, M.—YAROM, Y.: *Spectre attacks: Exploiting speculative execution,* CoRR, abs/1801.01203, 2018.

[2] SWANSON, S.—MCDOWELL, L. K.—SWIFT, M. M.—EGGERS, S. J.—LEVY H. M.: *An evaluation of speculative instruction execution on simultaneous multithreaded processors,* ACM Trans. Comput. Syst. **21** (2003), 314–340.

[3] NORDIN, P.—BANZHAF, W.: *Evolving turing-complete programs for a register machine with self-modifying code.* In: Proc. of the 6th Internat. Conf. on Genetic Algorithms, Pittsburgh, PA, USA, 1995, Morgan Kaufmann Publ., San Francisco, CA, 1995, pp. 318–327.

[4] MORSE, G.: *Pure infinitely self-modifying code is realizable and turing-complete,* Internat. J. Electron. Telecomm. **64** (2018), 123–129.

[5] AUGUST, E.: *Spectre attacks: exploiting speculative execution,* `https://github.com/crozone/SpectrePoC`. [Accessed: 2018.03.30].

[6] *Using Intel AES-NI and c++ threads to search an AES128 key*, `https://github.com/sebastien-riou/aes-brute-force`. [Accessed: 2018.06.01].

Received August 23, 2018

Faculty of Informatics
Eötvös Loránd Tudományegyetem/University (ELTE)
Pázmány Péter sétány 1/C
H–1117-Budapest
HUNGARY

E-mail: gregory.morse@live.com