Whitepaper

Cache Speculation Side-channels

Author: Richard Grisenthwaite

Date: January 2018
Version 1.1

Introduction

This whitepaper looks at the susceptibility of Arm implementations following recent research findings from security researchers at Google on new potential cache timing side-channels exploiting processor speculation. This paper also outlines possible mitigations that can be employed for software designed to run on existing Arm processors.

Overview of speculation-based cache timing side-channels

Cache timing side-channels are a well understood concept in the area of security research. As such, this whitepaper will provide a simple conceptual overview rather than an in-depth explanation.

The basic principle behind cache timing side-channels is that the pattern of allocations into the cache, and, in particular, which cache sets have been used for the allocation, can be determined by measuring the time taken to access entries that were previously in the cache, or by measuring the time to access the entries that have been allocated. This then can be used to determine which addresses have been allocated into the cache.

The novelty of speculation-based cache timing side-channels is their use of speculative memory reads. Speculative memory reads are typical of advanced micro-processors and part of the overall functionality which enables very high performance. By performing speculative memory reads to cacheable locations beyond an architecturally unresolved branch (or other change in program flow), and, further, the result of those reads can themselves be used to form the addresses of further speculative memory reads. These speculative reads cause allocations of entries into the cache whose addresses are indicative of the values of the first speculative read. This becomes an exploitable side-channel if untrusted code is able to control the speculation in such a way it causes a first speculative read of location which would not otherwise be accessible at that untrusted code. But the effects of the second speculative allocation within the caches can be measured by that untrusted code.

At this time, three variant mechanisms have been identified. Each potentially using the speculation of a processor to influence which cache entries have been allocated in a way to extract some information which would not otherwise be accessible to software.

This paper examines the nature of these three mechanisms, their state of knowledge and potential mitigations for the mechanisms in Arm software.

Variant 1: bypassing software checking of untrusted values

Overview of the Mechanism

For any form of supervisory software, it is common for untrusted software to pass a data value to be used as an offset into an array or similar structure that will be accessed by the trusted software. For example, an application (untrusted) may ask for information about an open file, based on the file descriptor ID. Of course, the supervisory software will check that the offset is within a suitable range before its use, so the software for such a paradigm could be written in the form:

```
1   struct array {
2      unsigned long length;
3      unsigned char data[];
4    };
5    struct array *arr = ...;
6    unsigned long untrusted_offset_from_user = ...;
7    if (untrusted_offset_from_user < arr->length) {
8      unsigned char value;
9      value =arr->data[untrusted_offset_from_user];
10      ...
11    }
```

In a modern micro-processor, the processor implementation commonly might perform the data access (implied by line 9 in the code above) speculatively to establish value before executing the branch that is associated with the `untrusted_offset_from_user` range check (implied by line 7). A processor running this code at a supervisory level (such as an OS Kernel or Hypervisor) can speculatively load from anywhere in Normal memory accessible to that supervisory level, determined by an out-of-range value for the `untrusted_offset_from_user` passed by the untrusted software. This is not a problem architecturally as, if the speculation is incorrect, then the value loaded will be discarded by the hardware.

However, advanced processors can use the values that have been speculatively loaded for further speculation. It is this further speculation that is exploited by the speculation-based cache timing side-channels. For example, the previous example might be extended to be of the following form:

```
1 struct array {
2   unsigned long length;
3   unsigned char data[];
4 };
5 struct array *arr1 = ...; /* small array */
6 struct array *arr2 = ...; /*array of size 0x400 */
7 unsigned long untrusted_offset_from_user = ...;
8   if (untrusted_offset_from_user < arr1->length) {
9     unsigned char value;
10    value =arr1->data[untrusted_offset_from_user];
11    unsigned long index2 =((value&1)*0x100)+0x200;
12    if (index2 < arr2->length) {
13      unsigned char value2 = arr2->data[index2];
14    }
15 }
```

In this example, `value`, which is loaded from memory using an address calculated from `arr1->data` combined with the `untrusted_offset_from user` (line 10), is then used as the basis of a further memory access (line13). Therefore, the speculative load of value2 comes from an address that is derived from the data speculatively loaded for `value`.

If the speculative load of `value2` by the processor cause an allocation into the cache, then part of the address of that load can be inferred using standard cache timing side-channels. Since that address depends on data in `value`, then part of the data of value can be inferred using the side-channel. By applying this approach to different bits of `value`, (in a number of speculative executions) the entirety of the data of value can be determined.

As shown earlier, the untrusted software can, by providing out-of-range quantities for `untrusted_offset_from_user`, access anywhere accessible to the supervisory software, and as such, this approach can be used by untrusted software to recover the value of any memory accessible by the supervisory software.

Modern processors have multiple different types of caching, including instruction caches, data caches and branch prediction cache. Where the allocation of entries in these caches is determined by the value of any part of some data that has been loaded based on untrusted input, then in principle this side channel could be stimulated.

As a generalization of this mechanism, it should be appreciated that the underlying hardware techniques mean that code past a branch might be speculatively executed, and so any sequence accessing memory after a branch may be executed speculatively. In such speculation, where one value speculatively loaded is then used to construct an address for a second load or indirect branch that can also be performed speculatively, that second load or indirect branch can leave an indication of the value loaded by the first speculative load in a way that could be read using a timing analysis of the cache by code that would otherwise not be able to read that value. This generalization implies that many code sequences commonly generated will leak information into the pattern of cache allocations that could be read by other, less privileged software. The most severe form of this issue is that described earlier in this section, where the less privileged software is able to select what values are leaked in this way.

Practicality of this side-channel

This side-channel has been demonstrated on several processors using code that is run in kernel space using the eBPF bytecode interpreter or JIT engine contained in the Linux kernel. The code run in this way holds a routine to perform the necessary shifting and dereferencing of the speculatively loaded data. The use of this mechanism has avoided the need to search for suitable routines in kernel space that can be directly exploited.

Note: it should be appreciated that this is one example way of exploiting the speculation. Analysis of code has shown that there are a small number of places where the value loaded using an untrusted offset is itself used to form an address to the extent that meaningful amounts of information can be retrieved using this mechanism.

It is very common that processors will speculate past an unresolved branch, and as such this is likely to be observed on cached Arm processors which perform execution out of order. For Arm processors that perform their execution in-order, there is insufficient speculative execution to allow this approach to be used to cause the necessary allocations into the cache. A definitive list of which Arm-designed processors are potentially susceptible to this issue can be found at **www.arm.com/security-update**.

Software Mitigations

The practical software mitigation for the scenario where the value being leaked is determined by less privileged software is to ensure that the address derived from the secret (that is the address that will be used to load `value2` in the example in page 2) is only indicative of the secret (the data in `value`) when the access that derived the secret was one that would be executed non-speculatively.

This can be achieved on most Arm implementations by using a conditional selection or conditional move instruction based on the condition that is used to determine the outcome of the branch (that is, in the example on page 2, to sanitize the `untrusted_offset_from_user`). In the implementations where this does not work, a new barrier, [defined below] can be used (this instruction is a NOP on implementations where the conditional select/conditional move can be used). The combination of both a conditional select/conditional move and the new barrier are therefore sufficient to address this problem on ALL Arm implementations. The details of the new barrier are described later in this section.

It is generally unusual for sequences that allow exploitation of this side-channel to exist in privileged code. However, the compilation of byte-code supplied by a lower level of privilege is an avenue to inject such sequences into privileged software. It is particularly important that just-in-time compilers that compile such byte-code use these mechanisms as part of their compiled sequences. Arm also recommends that the provision of code injection mechanisms of this type (for example eBPF) is disabled in systems where that is practical.

Note: For Android systems, the bpf() syscall is not available, and the only BPF available to user space, seccomp-bpf, is believed to insufficient to be able to trigger this issue.

Another area that could be subject to this issue is where there are software-enforced privilege boundaries within a single exception level, as may occur with JavaScript interpreters or Java runtimes. For example, in an interpreter, a key element of the software enforcement of privilege involves the sort of sanitization of untrusted values seen in this example, so potentially giving examples of this mechanism. Similarly, the sequences generated by a run-time compilation of Java byte-code may need to incorporate the work-around in their generated sequences.

Where it is impractical to insert this barrier, an alternative approach of inserting the combination of an DSB SYS and an ISB can be inserted to prevent speculation, but this is likely to have a much greater performance effect than using the conditional select/conditional move and CSDB barrier, and so should only be used where the conditional select/conditional move and CSDB cannot be inserted due to challenges with code generation.

Details of the CSDB barrier

The new barrier is called CSDB, and has the following encodings:

```
A64:
 1101_0101_0000_0011_0010_0010_100_11111
A32:
 1110_0011_0010_0000_1111_0000_0001_0100
T32:
 1111_0011_1010_1111_1000_0000_0001_0100
```

The semantics of the barrier are:

AArch64:

CSDB is the new conditional speculation barrier.

Until the barrier completes:

1) For any load, store, data or instruction preload, RW2, appearing in program order after the barrier, that has an address dependency on the result of a conditional select instruction, where:
   i. the conditional select instruction has a register data dependency on a load R1, that has been executed speculatively, for one of its input registers, and
   ii. the conditional select instruction does not have a register dependency on R1 for its other input register, and
   iii. the condition for the conditional select instruction is such that the input that is not dependent on R1 is selected if R1 is not architecturally executed,

   then the speculative execution of RW2 does not influence the allocations of entries in a cache in a way that can be used to determine any part of the value of the speculatively loaded data value from R1 by an evaluation of which entries in the cache have been allocated or evicted.

2) For any indirect branch (B2), appearing in program order after the barrier, whose target address has a register dependency on the result of a conditional select instruction. where:
   i. the conditional select instruction has a register data dependency on a load R1, that has been executed speculatively, for one of its input registers, and
   ii. the conditional select instruction does not have a register dependency on R1 for its other input register, and
   iii. the condition for the conditional select instruction is such that the input that is not dependent on R1 is selected if R1 is not architecturally executed

   then the speculative execution of B2 does not influence the allocations of entries in a cache in a way that can be used to determine any part of the value of the speculatively loaded data value from R1 by an evaluation of which entries in the cache have been allocated or evicted.

The barrier cannot be executed speculatively, but otherwise can complete once it is known not to be speculative.

AArch32:

CSDB is the new conditional speculation barrier.

Until the barrier completes:

1. For any load, store, data or instruction preload, RW2 , appearing in program order after the barrier, that has an address dependency on the result of a conditional move instruction, where
   i. the conditional move instruction does not have a register dependency on a load R1, that has been executed speculatively, for its input register, and
   ii. the condition for the conditional move instruction is such that the condition passes if R1 is not architecturally executed

   then the speculative execution of RW2 does not influence the allocations of entries in a cache in a way that can be used to determine any part of the value of the speculatively loaded data value from R1 by an evaluation of which entries in the cache have been allocated or evicted.

2. For any indirect branch (B2), appearing in program order after the barrier, whose target address has a register dependency on the result of a conditional move instruction where:
   i. the conditional move instruction does not have a register dependency on a load R1, that has been executed speculatively, for its input register, and
   ii. the condition for the conditional move instruction is such that the input that is not dependent on R1 is selected if R1 is not architecturally executed

   then the speculative execution of B2 does not influence the allocations of entries in a cache in a way that can be used to determine any part of the value of the speculatively loaded data value from R1 by an evaluation of which entries in the cache have been allocated or evicted.

The barrier cannot be executed speculatively, but otherwise can complete once it is known not to be speculative.

Use of the Barrier

These examples show how we expect the barrier to be used in the assembly code executed on the processor.

Taking the example shown previously:

```
 struct array {
  unsigned long length;
  unsigned char data[];
};
struct array *arr1 = ...; /* small array */
struct array *arr2 = ...; /* array of size 0x400 */
unsigned long untrusted_offset_from_user = ...;
  if (untrusted_offset_from_user < arr1->length) {
    unsigned char value;
    value = arr1->data[untrusted_offset_from_user];
    unsigned long index2 = ((value&1)*0x100)+0x200;
    if (index2 < arr2->length) {
      unsigned char value2 = arr2->data[index2];
    }
}
```

This example would typically be compiled into assembly of the following (simplified) form in AArch64:

```
  LDR X1, [X2] ; X2 is a pointer to arr1->length
  CMP X0, X1 ; X0 holds untrusted_offset_from_user
  BGE out_of_range
  LDRB W4, [X5,X0] ; X5 holds arr1->data base
  AND X4, X4, #1
  LSL X4, X4, #8
  ADD X4, X4, #0x200
  CMP X4, X6 ; X6 holds arr2->length
  BGE out_of_range
  LDRB X7, [X8, X4] ; X8 holds arr2->data base
out_of_range
```

The side-channel can be mitigated in this case by changing this code to be:

```
  LDR X1, [X2] ; X2 is a pointer to arr1->length
  CMP X0, X1 ; X0 holds untrusted_offset_from_user
  BGE out_of_range
  LDRB W4, [X5,X0] ; X5 holds arr1->data base
  CSEL X4, XZR, X4, GE
  CSDB ; this is the new barrier
  AND X4, X4, #1
  LSL X4, X4, #8
  ADD X4, X4, #0x200
  CMP X4, X6  ; X6 holds arr2->length
  BGE out_of_range
  LDRB X7, [X8, X4] ; X8 holds arr2->data base
out_of_range
```

For AArch32, the equivalent code is as follows:

Original code:

```
  LDR R1, [R2] ; R2 is a pointer to arr1->length
  CMP R0, R1 ; R0 holds untrusted_offset_from_user
  BGE out_of_range
  LDRB R4, [R5,R0] ; R5 holds arr1->data base
  AND R4, R4, #1
  LSL R4, R4, #8
  ADD R4, R4, #0x200
  CMP R4, R6  ; R6 holds arr2->length
  BGE out_of_range
  LDRB R7, [R8, R4]; R8 holds arr2->data base
out_of_range
```

Code with the mitigation added:

```
  LDR R1, [R2] ; R2 is a pointer to arr1->length
  CMP R0, R1 ; R0 holds untrusted_offset_from_user
  BGE out_of_range
  LDRB R4, [R5,R0] ; R5 holds arr1->data base
  MOVGE R4, #0
  CSDB
  AND R4, R4, #1
  LSL R4, R4, #8
  ADD R4, R4, #0x200
  CMP R4, R6  ; R6 holds arr2->length
  BGE out_of_range
  LDRB R7, [R8, R4]; R8 holds arr2->data base
out_of_range
```

In order to prevent this side-channel from being created in data caches, instruction caches or branch prediction caches, this mitigation approach should be used when:

- A data address is determined from a value read from an untrusted offset
- An indirect branch destination is determined from a value read from an untrusted offset
- A branch decision is determined from a value read from an untrusted offset

When applied to a particular code sequence involving the use of an untrusted value, this mitigation will prevent that code sequence from being able to be used to exploit this side-channel to access any data.

For some, but not all, Arm implementations, mapping particularly important secrets, such as Cryptographic keys, in Device memory will prevent their being allocated into a cache. Mapping such data in this way, where it is feasible under the operating system, could be used as an additional safeguard for those implementations, albeit at significantly increased performance cost.

Note: Arm has investigated using Device memory under Linux in this way for bulk memory, and does not believe that this additional safeguard is practical to deploy.

Tooling to help with the Software Mitigation

Details of tooling to support the software mitigation can be found at **www.arm.com/security-update.**

Variant 2: forcing privileged speculation by training branch predictors

Overview of the Method

All modern processors, including those from Arm, have a variety of different mechanisms for branch prediction that cause the processor to speculatively change the instruction stream in response to predictions of the directions of future branches. The forms of such branch predictors are not described by the architecture, and implementations can employ a variety of different mechanisms to speculate the changes of instruction stream.

In order to give high-performance execution, these predictors are designed to use the history of previous branches to speculate the change of instruction stream. The resulting speculation can take considerable time to be resolved. This delay in resolution can result in the processor performing speculative memory accesses, and so cause allocation into the caches.

In some implementations, including many of those from Arm, the history of previous branches used to drive the speculation is not filtered by the exception level that the processor was in. Therefore, it is possible for the code running at one exception level to train the branch predictors in a manner that causes other exception levels (or other contexts) to perform speculative memory accesses. This can then be used to stimulate the speculation-based cache timing side-channel by having a lower exception level train the branch predictors to influence the speculative instruction stream of a higher exception level, or in a different context, to read data otherwise inaccessible at the lower exception level, and additionally to allocate items speculatively into the caches based on that data. Code running at the lower exception level can then examine the impact of cache allocations, so exploiting the cache timing side-channel.

As advanced implementations can typically have multiple outstanding speculative changes of address stream caused by branch prediction, in principle it is possible to string together a number of different pieces of privileged code – in effect to create a string of *speculation gadgets*. These gadgets are strung together using the trained predictions of the branch predictor – to construct sequences to read arbitrary data and use this data to form the addresses to allocate into the caches.

Practicality of this side-channel

This side-channel has been demonstrated on some processors by the training of indirect branches to allow the extraction of data from a KVM based hypervisor (though the same approach could also be used for an operating system kernel, or in principle for a different application or virtual machine). This demonstration used the branch predictor to speculatively enter the back-end of the eBPF bytecode interpreter contained in the host Linux kernel to run some user-space held byte-codes to perform speculatively the necessary shifting of the speculatively loaded data from the address space of the hypervisor. It should be noted that this speculative use of the eBPF bytecode interpreter does not rely on the bytecode interpreter being enabled, but can be used simply if the bytecode interpreter is present in the hypervisor (or operating system kernel) image and is marked as executable.

The use of this mechanism has avoided the need to string together speculation gadgets as part of an initial proof of concept, but it is possible that the more general approach of using speculation gadgets may be developed over time.

Most current out-of-order Arm processors have branch predictors of the form that allow training from one exception level to influence the execution at other exception levels or in other contexts. The exact mechanism of the branch predictor training varies between different implementations, and so significant reverse engineering of the branch prediction algorithms would be necessary to achieve this training. Current cached Arm processors which perform their execution in order do not exhibit sufficient speculative execution for this approach to be used to extract useful information.

A definitive list of which Arm-designed processors are potentially susceptible to this issue can be found at **www.arm.com/security-update**.

Software Mitigations

For Arm implementations, there is no generic mitigation available that applies for all Arm processors. However, many Arm processors have implementation specific controls that can be used to either disable branch prediction or to provide mechanisms to invalidate the branch predictor. Where an ability to invalidate the branch predictor exists, it should be used on a context switch. It should also be used on an exception entry from an exception level where it is judged that code might be used to attack a higher level. Invalidations on exception entry will likely have a non-trivial performance impact.

Similarly, where an implementation has a capability to disable branch prediction, then this should be invalidated for exception levels that are judged to be particularly vulnerable to attack.

Variant 3: using speculative reads of inaccessible data

Overview of the Mechanism

In some, but not all, Arm implementations, a processor that speculatively performs a read from an area of memory with a permission fault (or additionally in AArch32, a domain fault) will actually access the associated location, and return a speculative register value that can be used as an address in subsequent speculative load instructions. If the speculation is not correct, then the results of the speculation will be discarded, so there is no architectural revelation of the data accessed at the permission faulting location. However, on some implementations, the data returned from the speculative load can be used to perform further speculation. It is this further speculation that is exploited by the speculation-based cache timing side-channels.

For example, in AArch64, a piece of EL0 code could be constructed with the form:

```
1  LDR X1, [X2] ; arranged to miss in the cache
2  CBZ X1, over ; This will be taken but
3               ; is predicted not taken
4  LDR X3, [X4] ; X4 points to some EL1 memory
5  LSL X3, X3, #imm
6  AND X3, X3, #0xFC0
7  LDR X5, [X6,X3] ; X6 is an EL0 base address
8 over
```

where:

- EL1 memory is something mapped as Kernel-only in the page table
- EL0 base address is the address of a User accessible array in memory used for the subsequent timing readings on the cache

The perturbation of the cache by the `LDR X5, [X6,X3]` (line 7) can be subsequently measured by the EL0 code for different values of the shift amount `imm` (line 5). This gives a mechanism to establish the value of the EL1 data at the address pointed to by `X4`, so leaking data that should not be accessible to EL0 code.

The equivalent situation can be used for AArch32, for PL0 code attempting to access PL1 memory:

```
  LDR R1, [R2] ; arranged to miss in the cache
  CMP R1, #0
  BEQ over ; This will be taken but
             ; is predicted not taken
  LDR R3, [R4] ; R4 points to some PL1 memory
  LSL R3, R3, #imm
  AND R3, R3, #0xFC0
  LDR R5, [R6,R3] ; R6 is an PL0 base address
over
```

Practicality of this side-channel

For some implementations where a speculative load to a permission faulting (or in AArch32 domain faulting) memory location returns data that can be used for further speculation, this side-channel has been demonstrated to allow the leakage of EL1-only accessible memory to EL0 software. This then means that malicious EL0 applications could be written to exploit this side-channel.

A definitive list of which Arm-designed processors are potentially susceptible to this issue can be found at **www.arm.com/security-update**.

It is believed that at least some Arm processors designed by Arm and its architecture partners are susceptible to this side-channel, and so Arm recommends that the software mitigations described in this whitepaper are deployed where protection against malicious applications is required.

Software Mitigations

For Arm software, the best mitigation for the memory leakage with this mechanism is to ensure that when running at EL0, there are minimal mappings pointing to Kernel-only data mapped in page tables, or present in the TLB. This is done in preference to the common technique of having the Kernel-only data mapped in the translation tables but with EL1-only access permissions.

A patch to the Linux kernel for AArch64 is available from Arm to perform this mitigation, using two different ASID values for each application to prevent any TLB maintenance entailed when switching between user and privileged execution.

For information on the latest Linux Kernel patches related to this issue, please go to **www.arm.com/security-update**.

For support with this mitigation on other operating systems, please contact **support@arm.com**.

Subvariant 3a - using speculative reads of inaccessible data

Overview of the Mechanism

In much the same way as with the main Variant 3, in a small number of Arm implementations, a processor that speculatively performs a read of a system register that is not accessible at the current exception level, will actually access the associated system register (provided that it is a register that can be read without side-effects). This access will return a speculative register value that can be used in subsequent speculative load instructions. If the speculation is not correct, then the results of the speculation will be discarded, so there is no architectural revelation of the data from the inaccessible system register. However, on such implementations, the data returned from the inaccessible system register can be used to perform further speculation. It is this further speculation that is exploited by the speculation-based cache timing side-channels.

For example, in AArch64, a piece of EL0 code could be constructed with the form:

```
1  LDR X1, [X2] ; arranged to miss in the cache
2  CBZ X1, over ; This will be taken
3  MRS X3, TTBR0_EL1;
4  LSL X3, X3, #imm
5  AND X3, X3, #0xFC0
6  LDR X5, [X6,X3] ; X6 is an EL0 base address
7 over
```

where:

- EL0 base address is the address of a User accessible array in memory used for the subsequent timing readings on the cache

The perturbation of the cache by the `LDR X5, [X6,X3]` (line 6) can be subsequently measured by the EL0 code for different values of the shift amount `imm` (line 4). This gives a mechanism to establish the value held in the TTBR0_EL1 register so leaking data that should not be accessible to EL0 code.

The equivalent situation can be used for AArch32, for PL0 code attempting to access say the TTBR0 under a 32-bit Kernel:

```
  LDR R1, [R2] ; arranged to miss in the cache
  CMP R1, #0
  BEQ over ; This will be taken
  MRC p15, 0, R3, c2, c0, 0 ; read of TTBR0
  LSL R3, R3, #imm
  AND R3, R3, #0xFC0
  LDR R5, [R6,R3] ; R6 is an PL0 base address
over
```

Practicality of this side-channel

This side-channel can be used to determine the values held in system registers that should not be accessible. While it is undesirable for lower exception levels to be able to access these data values, for the majority of system registers, the leakage of this information is not material.

Note: It is believed that there are no implementations of Arm processors which are susceptible to this mechanism that also implement the Pointer Authentication Mechanism introduced as part of Armv8.3-A, where there are keys held in system registers.

A definitive list of which Arm-designed processors are potentially susceptible to this issue can be found at **www.arm.com/security-update**.

Software Mitigations

In general, it is not believed that software mitigations for this issue are necessary.

For system registers that are not in use when working at a particular exception level and which are felt to be sensitive, it would in principle be possible for the software of a higher exception level to substitute in dummy values into the system registers while running at that exception level. In particular, this mechanism could be used in conjunction with the mitigation for variant 3 to ensure that the location of the VBAR_EL1 while running at EL0 is not indicative of the virtual address layout of the EL1 memory, so preventing the leakage of information useful for compromising KASLR.

www.arm.com/security-update