SpectrEM: Exploiting Electromagnetic Emanations During Transient Execution

Jesse De Meulemeester, Antoon Purnal, Lennert Wouters, Arthur Beckers, and Ingrid Verbauwhede, *COSIC, KU Leuven*

https://www.usenix.org/conference/usenixsecurity23/presentation/de-meulemeester

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

SPECTREM: Exploiting Electromagnetic Emanations During Transient Execution

Jesse De Meulemeester
COSIC, KU Leuven

Antoon Purnal
COSIC, KU Leuven

Lennert Wouters
COSIC, KU Leuven

Arthur Beckers
COSIC, KU Leuven

Ingrid Verbauwhede
COSIC, KU Leuven

Abstract

Modern processors implement sophisticated performance optimizations, such as out-of-order execution and speculation, that expose programs to so-called transient execution attacks. So far, such attacks rely on specific on-chip covert channels (e.g., cache timing), instilling the hope that they can be thwarted by closing or weakening these channels. In this paper, we consider the inevitable physical side effects of transient execution. We focus on electromagnetic (EM) emanations produced by the processor and develop two lightweight and accurate EM channels to extract secret bits from the transient window. We propose SPECTREM, a Spectre variant for embedded devices exposed to physical access by an attacker. While it assumes a physical adversary, it does not fundamentally require code execution, expanding its applicability in the embedded world. We evaluate SPECTREM on an Arm Cortex-A72, leaking up to 366 bits per second at a bit error rate as low as 0.008 %. To our knowledge, this is the first practical demonstration of physical transient execution attacks.

1 Introduction

Connected and embedded devices are taking up an ever-increasing presence in modern life. As the role of these devices grows, security becomes paramount. Faced with higher computational demands, embedded processors have become increasingly more powerful by leveraging clever optimizations such as out-of-order execution and speculation. Although these constructs greatly improve the efficiency of the processor, they have been shown to be vulnerable to attacks that break common isolation properties. In particular, *transient execution attacks* invalidate implicit assumptions imposed by the developer [10, 12, 40, 49]. They can be divided into two broad categories, Spectre and Meltdown, exploiting prediction mechanisms or faulting instructions, respectively. These attacks are particularly devastating in contexts where multiple users operate on the same system, such as cloud environments, where isolation between customers is critical. Following their public disclosure in 2018, a whole range of transient execution attacks were discovered; exploiting new mechanisms [7, 11, 33, 43, 52], or leveraging different exfiltration techniques [9, 14, 20, 47, 67, 72].

In the context of embedded and internet-of-things (IoT) devices, potential adversaries may have physical access to the computing device. In some cases, the adversary may even be the intended end-user. Such physical access exposes the device to a myriad of attacks, leveraging side effects of its computations in the physical world. Indeed, the power consumption or electromagnetic (EM) emanations of a device have been shown to reveal details about the data processed by the device, as well as the computations performed on them [23, 41, 62]. Physical effects can also create covert channels (i.e., unintended communication channels) using, e.g., power [32, 48], EM [29, 68, 76], or thermal effects [55].

So far, transient execution attacks have only considered software-observable side effects, e.g., cache timing [40] or port contention [9], to extract sensitive values. With only one exception [67], this limits their attack surface to attackers with code execution on the device. Furthermore, it instills the hope that transient execution attacks can be overcome by closing or weakening specific on-chip covert channels [1, 36–38, 51, 66, 74]. Other countermeasures, like isolation [15, 17] or randomization [60, 63, 64, 73] of micro-architectural elements, do not preclude leakage in the physical domain either.

This paper explores whether the physical side effects of transient instructions can be leveraged for an attack. Despite already being proposed theoretically by Kocher et al. in 2018 [40], we have not seen any practical demonstration of such attacks. Therefore, we ask the following:

How do the constraints imposed by the transient window affect the physical covert channel? Can a physical adversary mount Spectre attacks without the need for code execution?

Leveraging physical manifestations allows for the extraction of secrets without fundamentally requiring code execution, instead requiring only physical access to the target device that exposes a gadget through an interface. As a result, such attacks can operate in regimes for which transient execution attacks previously posed only a limited threat, such as in embedded contexts where the attacker is not able to execute code. Additionally, such attacks can circumvent countermeasures focused on micro-architectural-based covert channels.

In this paper, we introduce the first transient execution attacks that do not rely on micro-architectural covert channels. Instead, we leverage the inevitable electromagnetic emanations resulting from transiently executed instructions. However, the number of transient instructions following the misprediction or fault, i.e., the *transient window*, is severely limited. Existing EM covert channels, such as GSMem [29], BitJabber [76], or EMLoRa [68], are not applicable in this context, as they require symbol lengths multiple orders of magnitude larger than those tolerated by transient execution.

To overcome this challenge, we propose two lightweight techniques to transmit secrets over EM waves. The first uses instructions with operand-dependent latency, producing a strong signal in the EM frequency spectrum. The second introduces a control flow dependency on the transient secret to produce a distinct peak in the frequency domain. Our techniques demonstrate that information can be encoded within the transient window for a physical attacker to decode.

We then embed our EM covert channels in SPECTREM, a physical Spectre attack. It shares with NetSpectre [67] that it does not require code execution on the target, and instead relies on access to an interface to trigger the relevant Spectre code patterns (so-called *gadgets*). However, by additionally considering physical access for an attacker, SPECTREM can work with more generic Spectre gadgets and reduce the required number of gadgets from two to one. We show the occurrence of these vulnerable code patterns by considering a case study of OpenSSH. Additionally, we also demonstrate MELTEMDOWN, a physical Meltdown attack.

Based on our evaluation on the Arm Cortex-A72, SPECTREM achieves a bitrate of up to 367 bit/s at a bit error rate (BER) of <0.01 % and 0.769 % for the instruction and control flow EM transmission, respectively. MELTEMDOWN, on the other hand, achieves a BER of 0.00687 % at 241 bit/s.

Contributions. Our main contributions are as follows:

- We identify two electromagnetic covert channels capable of working under the transient window constraints.
- We propose SPECTREM, a physical transient execution attack not fundamentally requiring code execution.
- We perform an initial investigation of MELTEMDOWN, a physical Meltdown attack.
- We evaluate our attacks on the Arm Cortex-A72 using supervised and unsupervised classification methods.
- We perform a case study on OpenSSH and show vulnerable code patterns occur in real-world code.
- We make our proof-of-concept implementations and measurement artifacts publicly available.

Availability. Our code and measurement artifacts are available at https://github.com/KULeuven-COSIC/SpectrEM.

Responsible disclosure. We disclosed our findings to Arm on January 16th, 2023. Arm acknowledged these attacks, but did not request an embargo as they generally consider physical attacks out-of-scope.

2 Preliminaries

2.1 Physical Side Channels

Programs written in some high-level language are compiled into instructions, i.e., machine operations the processor understands. When executed on a physical device, these instructions bring with them inevitable side effects, i.e., they consume power, emanate electromagnetic waves, and produce heat. Differences in operands or operations can manifest themselves in subtle differences in side effects. By collecting measurements, or *traces*, of these physical quantities over time, the differences can be leveraged to gain information about the exact operations or operands.

Side-channel attacks use this information to extract secrets from a victim process, which are often cryptographic implementations. While a cryptographic algorithm may be theoretically secure, subtle implementation details may expose it to leakage of secret information through side channels [41].

Covert channels use the same side effects, not to passively eavesdrop on a victim program, but instead to create a hidden communication interface, i.e., with a deliberate transmitter and receiver. Such an ad-hoc channel allows an attacker to transfer data between parties, despite not being allowed to communicate by some security policy (e.g., a sandbox).

One key difference with side channels is that, for covert channels, the leakage is intentional [45]. Transmitter and receiver are either cooperating or, as in transient execution attacks, a victim program is forced by an attacker to perform some action that leaks secret information (cf. Section 2.3).

Various physical phenomena may be used for side-channel attacks and covert channels. This paper focuses on the emanation of EM waves by the computing device to establish EM covert channels [29, 68, 76].

2.2 Out-of-Order Execution and Speculation

To improve throughput and efficiency, modern processors often have multiple independent execution units, allowing for multiple instructions to be executed at the same time. These instructions need not be executed in the order in which they were issued. If certain instructions do not depend on any previous ones, the processor may execute them as soon as their operands are available. This avoids unnecessary stalls due to data hazards, thereby improving performance.

Control hazards, on the other hand, can still stall the out-of-order pipeline and require additional mechanisms, such as speculation, to prevent. Speculation predicts the outcome of certain operations, such as branches, to prevent stalls. An accurate prediction results in a significant performance improvement as the stall is avoided, whereas a misprediction only incurs a small penalty on top of the stall. In case of a misprediction, instructions that were speculatively executed are discarded without affecting the architectural state of the processor. Even branch predictors with a moderate prediction rate will therefore see a significant performance improvement.

Modern processors leverage a variety of prediction mechanisms. In the case of branch prediction, a pattern history table (PHT) determines whether a direct branch will be taken, taking into account the (recent) history of the branch. For indirect branches, a branch target buffer (BTB) is used to predict the target address of that branch. Together, they allow the processor to predict any branch target, significantly improving performance. Other prediction mechanisms include memory dependence prediction, which aims to predict whether certain memory access operations depend on each other, and return address prediction.

2.3 Transient Execution Attacks

Prediction mechanisms are often highly effective, yet they inevitably experience mispredictions. This results in instructions that were speculated, and thus executed, but not committed, called *transient instructions* [12]. The window in time during which they occur is called the *transient window*. Another source of transient instructions are faulting instructions—instructions that raise an exception. Due to out-of-order execution, instructions architecturally following such instructions may have already been executed and will have to be discarded.

While architecturally invisible, micro-architectural elements such as caches are affected by transient instructions. As a result, transient instructions may encode information in micro-architectural components, acting as the transmitting end of a covert channel (cf. Section 2.1). By controlling which instructions are executed in the transient domain, attackers can retrieve architecturally inaccessible data. This idea forms the basis for so-called *transient execution attacks*.

Transient execution attacks are categorized into two groups depending on the source of the transient instructions [12]. They originate either from a prediction mechanism—Spectre-type attacks [7,33,40,43,52]—or from a faulting instruction—Meltdown-type attacks [10,49].

2.3.1 Spectre

Spectre attacks [40] exploit prediction mechanisms resulting in transient instructions and can be subdivided based on the exact prediction mechanism they exploit [12]. Spectre-PHT and Spectre-BTB exploit the pattern history table and branch target buffer of the branch predictor, respectively [7,40]. Other exploitable prediction mechanisms include the return stack buffer (Spectre-RSB) [43,52] and the memory dependence predictor (Spectre-STL) [33].

The first Spectre variant, Spectre-PHT, exploits memory accesses guarded by a bounds check. Such checks are typically employed to prevent buffer overreads, but can in certain scenarios be circumvented by leveraging transient execution. In particular, the adversary can train the branch predictor to predict that the provided index adheres to the bounds check, in which case an out-of-bounds value may be accessed transiently. Subsequent transient instructions can then transmit this value through a covert channel, e.g., by encoding it in the cache. The condition variable, which for instance determines the maximal value of the provided index, is typically flushed from the cache [40]. This increases the transient window size and improves the success rate of the attack.

2.3.2 Meltdown

Meltdown attacks [10,49] exploit transient instructions following a faulting instruction, e.g., an unauthorized access to kernel memory or a system register. Exceptions resulting from a faulting instruction are only raised when the instruction is committed. However, due to out-of-order execution, instructions following this faulting instruction may already have (partially) executed before this instruction is committed. These transient instructions can be used to exfiltrate the result of the faulting load through some covert channel.

The original attack [49] leverages out-of-order execution to access kernel memory from user space. Other exception mechanisms can also be exploited, allowing for different types of attacks [12], including reading system registers [26] or accessing memory in Intel SGX enclaves [10].

2.3.3 Gadgets

Transient execution attacks make use of *gadgets*, i.e., code patterns that trigger the attack and/or transmit the secret value through some covert channel. They can be either present in the victim's code base, typically in the case of a Spectre attack, or written by the adversary [12]. Existing works typically exfiltrate the secret values via a cache covert channel (e.g., FLUSH+RELOAD [75]). By accessing certain memory addresses based on the secret value, an attacker observes a change in access timings, allowing them to infer the value.

Besides the cache, other micro-architectural elements are also susceptible to modifications by transient instructions. NetSpectre [67] uses x86's Advanced Vector Extensions (AVX) to construct an AVX-based covert channel. This covert channel relies on the latency of the AVX unit, which is substantially lower when powered up. SMoTherSpectre [9] leverages port contention by timing certain instruction sequences running in parallel with the victim's transient instructions. Alternate methods include using the floating point division unit [20] or even using the branch predictor unit itself [14].

3 Electromagnetic Covert Channel Under Transient Window Constraints

In this section, we consider EM covert channels that can operate in the context of transient execution attacks. Since the unauthorized access results from a mispredicted branch or a faulting instruction, the secret value remains accessible only until the processor corrects the wrongly executed instructions by clearing the pipeline. As a result, the transient window size places an upper bound on the number of instructions that can be used to encode the secret value in electromagnetic waves.

Two factors contribute to this transient window size: the size of the micro-architectural components, such as the reorder buffer (ROB), and the time it takes for the processor to resolve the branch. The former will generally define the size for instructions occupying few clock cycles, whereas the latter will dictate the size for more expensive operations.

To determine the instruction window available to the covert channel transmitter, we extend the Speculator tool [53] for AArch64. Our experiments, which are covered in detail in Appendix A, indicate that the EM covert channel can utilize no more than around 60 instructions on the Arm Cortex-A72. Given this constraint, existing EM covert channels are not suitable for transmitting secrets from the transient domain. GSMem [29], for instance, proposes a symbol length of 1–10 ms, multiple orders of magnitude larger than the transient window size. Similarly, BitJabber [76] and EMLoRa [68] require multiple accesses to main memory for each transmitted bit, which is impossible within this window. Instead, different techniques are required. Concretely, we identify two techniques that allow the transmission of secrets within this constraint while also requiring as few traces as possible; instructions with operand-dependent behavior, and control flow dependencies.

3.1 Operand-Dependent Instruction Timing

Certain instructions are expensive to execute in hardware. Computing a division, for instance, is an inherently iterative process, which results in a much higher instruction latency than, for instance, an addition or a multiplication. To avoid excessive latency, these instructions will often return the result as soon as it is known. This avoids unnecessary performance degradation by waiting for the worst-case latency. Dividing two small integers, for instance, takes fewer clock cycles than computing the quotient of a large dividend and a small divisor. We now demonstrate this dependency.

Methodology. The division instruction is one of the instructions that displays this behavior on the Arm Cortex-A72 [3]. We divide the largest unsigned 64-bit integer $\left(2^{64}-1\right)$ by 0, 3, and itself and measure the EM traces using setup A as described in Section 5.1. Note that a division by zero on Arm does not produce an exception but rather simply returns zero. In order to assess the frequency response of two operands, the fast Fourier transform (FFT) of the mean of 1000 traces, each containing 64 identical divisions, is computed.

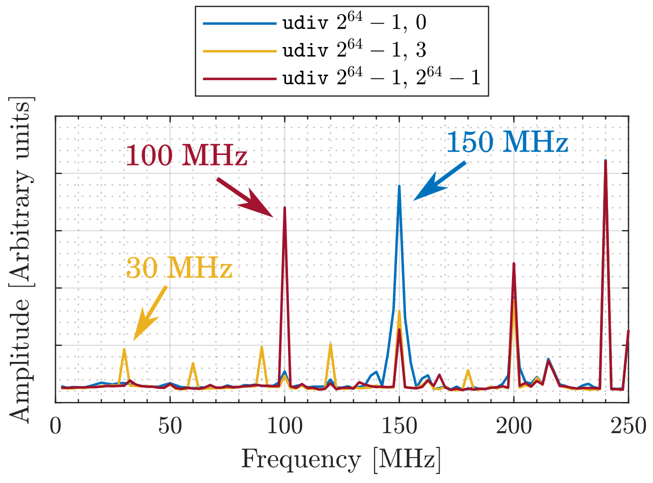


Figure 1: Frequency dependency of the unsigned division instruction on the Arm Cortex-A72.

```
bit = secret & bitmask;

res = dividend / (bit-1);
```

Listing 1: An EM covert channel where the operand dependency of the `udiv` instruction leaks the value of `bit`.

Results. Figure 1 shows the frequency spectrum resulting from the division of the three considered cases. These three divisions result in noticeably different frequency spectra. Distinct peaks can be observed at 150 MHz, 30 MHz, and 100 MHz (and their harmonics), respectively. Given the clock frequency of 600 MHz, they correspond to an execution latency of 4, 20, and 6 clock cycles, respectively.

By controlling the operands to these time-varying instructions, one can construct a rudimentary communication interface. An example using the division instruction is shown in Listing 1. In this example, the value of `bit` will determine which of the frequency spectra will be emitted (cf. Figure 1). Based on the frequency components in the recorded trace, the transmitted bit can be estimated by the receiver.

3.2 Control Flow Dependency

A second technique to transmit information via the EM channel is to create a control flow dependency on the value to be transmitted, as shown in Listing 2. The micro-architectural behavior of this branch can reveal information on the value. Consider the branch predictor to predict the branch as taken. By inferring from the EM emanations whether a pipeline clear occurred, information about the value can be learned. We now show how this can be done on the Arm Cortex-A72.

```
bit = secret & bitmask;

if (bit) {
  ...
}
```

Listing 2: Control flow dependency on the variable `bit`.

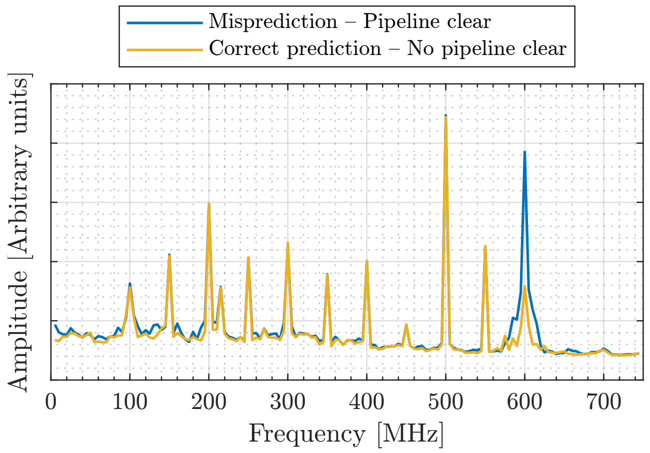


Figure 2: Frequency dependency of control flow dependencies on the ARM Cortex-A72.

Methodology. We induce both correct predictions and mispredictions in a branch in the transient domain on the Arm Cortex-A72 and measure the EM traces using setup A as described in Section 5.1. A total of 16 384 traces are collected for which the mean of the respective classes is computed in the frequency domain.

Results. Figure 2 compares the EM frequency response for correct predictions versus mispredictions. The clock frequency component, at 600 MHz, is more prominent when a pipeline clear occurs, allowing it to be detected.

Concretely, transmitting values based on this clearing behavior first involves training the branch predictor for the concerning branch. The direction it predicts is of no importance. However, it needs to be consistent such that from the detection of the pipeline clear the bit can be inferred. After training the branch predictor, the value of the bit determines whether a pipeline clear occurs. Recovering the value then involves deducing whether such a clear occurred or not.

4 Attack Overview

In this section, we discuss the assumed threat model for our physical transient execution attacks, and cover the integration of the above EM covert channels with Spectre and Meltdown.

4.1 Threat Model

Leveraging physical manifestations, instead of measuring timing differences stemming from micro-architectural elements, results in a different set of requirements for the adversary. In this context, we focus explicitly on embedded devices.

SPECTREM (Spectre attack). To mount a SPECTREM attack, the adversary is assumed to have physical access to the target device and a means of communication with the device through an interface that exposes a gadget. In a similar way to NetSpectre [67], this interface can be used to access the gadget, enabling speculative execution attacks without requiring code execution.

MELTEMDOWN (Meltdown attack). In contrast to SPECTREM, the MELTEMDOWN attack requires code execution because it cannot rely on gadgets in the victim's code. The adversary is therefore assumed to have physical access to the target device as well as unprivileged code execution on the device. MELTEMDOWN demonstrates that preventing specific micro-architectural-based covert channels is not a complete countermeasure against Meltdown attacks.

4.2 SPECTREM

The SPECTREM instance we consider is based on Spectre-PHT. The adversary, after mistraining the branch predictor, provides a data packet containing an out-of-bounds index to a gadget through an interface. Due to a misprediction, the gadget accesses architecturally inaccessible data. The transmission step of the attack can be performed with either of the covert channels from Section 3. The two resulting gadgets will be referred to as *instruction gadgets* and *control flow gadgets*. The former leverage instructions with operand-dependent timings, whereas the latter exploit control flow dependencies on the secret value.

Listing 3 provides examples of both gadget types. For the instruction gadget (Listing 3a), the operand-dependent behavior of the division instruction transmits the secret bit into the physical domain. Note that in practice, multiple operand-dependent instructions are required. For the control flow gadget (Listing 3b), leakage occurs through the control flow dependency on the bit. In these examples, the attacker is assumed to control the `index` variable. By forcing a misprediction and strategically choosing the value of `index`, the attacker can trigger the leakage of an out-of-bounds bit through the EM domain.

4.2.1 Preparing the Micro-Architectural State

The described EM transmission gadgets require the initialization of two specific micro-architectural elements before they can be exploited. First, the branch predictor needs to be trained to force a misprediction during the attack, ensuring the leakage of an out-of-bounds value. This training can take place through the very same interface we assumed for the

```
if (index < len) {
  bit = array[index] & bitmask;

  res = dividend / (bit-1);
}
```

(a) SPECTREM – Instruction gadget.

```
if (index < len) {
  bit = array[index] & bitmask;

  if (bit) {
    ...
  }
}
```

(b) SPECTREM – Control flow gadget.

```
bit = access_secret();

res = dividend / (bit-1);
```

(c) MELTEMDOWN – Instruction gadget.

```
bit = access_secret();

if (bit) {
  ...
}
```

(d) MELTEMDOWN – Control flow gadget.

Listing 3: Two types of SPECTREM and MELTEMDOWN gadgets, based on the two EM covert channels.

gadgets by simply providing an in-bounds index. In the case of a simple two-bit branch predictor, as is the case for the Arm Cortex-A72 [4], two packets with in-bounds indices will ensure the branch is predicted not taken for the next packet. Note that for the control flow gadget, the training should also take the inner branch into account. However, this training can take place together with that of the main branch. Each extracted value, therefore, requires three calls to the gadget; two to train the branch predictor and one to extract the value.

Second, the out-of-bounds value is transmitted during the transient window opened by the misprediction. This window can be prolonged by removing the condition variable corresponding to the branch to be mispredicted from the cache. When not assuming code execution, this can be performed by thrashing the cache [67], i.e., evicting all cache entries by, for instance, downloading a file from the target. While this method lacks the granularity to evict a single entry from the cache, the end result is effectively the same.

4.3 MELTEMDOWN

Both our described EM covert channels, leveraging operand-dependent instruction timings and control flow dependencies, can also be incorporated into a Meltdown attack.

The MELTEMDOWN instance we consider is based on Meltdown-GP. Listings 3c and 3d give two examples of MELTEMDOWN attacks using the instruction gadget and control flow gadget, respectively. Here, `access_secret()` is the function that accesses the secret and generates an exception. Due to out-of-order execution, the instructions following the access, while not architecturally executed, will be executed transiently, resulting in the leakage of the secret value.

5 Experimental Setup and Methodology

This section describes the experimental setup and methodology that will be used throughout the evaluation.

5.1 Experimental Setup

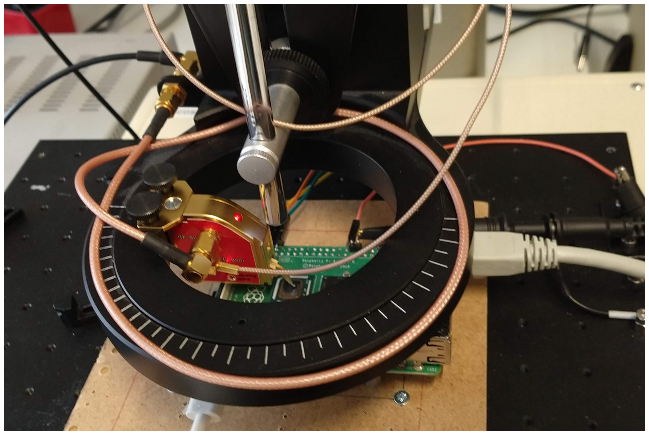
5.1.1 Target Device

The attacks will be evaluated on a Raspberry Pi 4 model B, which provides easy access to the backside of its quad-core Arm Cortex-A72 processor after removing the integrated heat spreader. The Cortex-A72 is one of Arm's high-performance CPUs geared towards mobile and embedded applications featuring a superscalar, out-of-order pipeline [2, 4]. Running on top of this device is Ubuntu 20.04 LTS (Focal Fossa). During the experiments, we implement a number of measures to simplify the signal processing and ensure more consistent and repeatable results throughout the evaluation. Specifically, we disable dynamic voltage and frequency scaling (DVFS) and lock the clock frequency at the lowest available frequency (600 MHz). Additionally, as the processor has four cores, the test program is pinned to a single core.

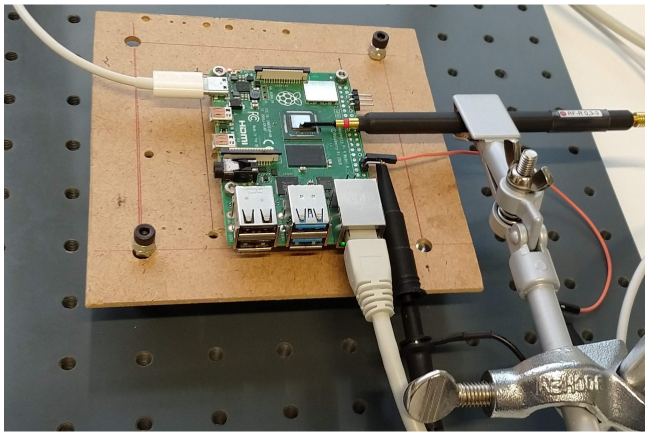
We will revisit these assumptions in Section 7, where we evaluate the impact of removing these simplifications.

5.1.2 Measurement Setup

We evaluate the proof-of-concept (POC) implementations of the attacks described in Section 4 using two different setups. The first setup, *setup A*, is relatively elaborate, allowing very localized measurements of the target chip. The second setup, *setup B*, is less precise but more accessible. The traces are captured using a Tektronix DPO70604C oscilloscope, which has an analog bandwidth of 6 GHz at a sampling rate of 25 GS/s.



(a) Setup A.



(b) Setup B.

Figure 3: The two setups used for evaluation.

To trigger the scope, a second probe is connected to one of the general-purpose input/output (GPIO) pins of the Raspberry Pi. We now describe both setups in detail.

Setup A. Setup A, shown in Figure 3a, uses a Langer ICR HH500-6 EM probe, with a spatial accuracy of 300 µm, mounted on a Langer FLS 102 stepper table that allows for the precise positioning of the probe over the processor with a resolution of 200 µm in either direction. The probe is connected to the oscilloscope using the Langer BT 706 bias-tee, which supplies the probe with power, followed by a 20 dB preamplifier (Langer PA 203) with a bandwidth of 3 GHz.

Setup B. Setup B, shown in Figure 3b, uses a Langer RF-R 0,3-3 probe mounted on a stand that positions it at a fixed location over the SoC. The probe is connected to two preamplifiers, the Langer PA 303 and Langer PA 203, with a bandwidth of 3 GHz, amplifying the signal by 30 dB and 20 dB, respectively, before acquisition by the oscilloscope.

5.2 Methodology

5.2.1 Proof-of-Concept Implementations

For the SPECTREM attack, we evaluate both the instruction and control flow gadgets. For the evaluation of MELTEMDOWN, however, we will only consider the instruction gadget. The reason for this is that in a Meltdown attack, the gadget is attacker-chosen code, and the instruction gadget will be shown to outperform the control flow gadget.

Gadgets. Instruction gadgets use instructions that exhibit operand-dependent execution times. As shown in Section 3.1, the unsigned division (`udiv`) instruction displays such behavior on the Arm Cortex-A72. Concretely, 64 `udiv` instructions are inserted to transmit the value. While a victim binary will almost certainly not contain this many leaky instructions at vulnerable locations, it assumes the best-case scenario for the attacker. In Appendix C, we study the effect of gadgets with fewer such instructions.

The second gadget leaks information through a control flow dependency on the secret. As the inference of the bit relies on the micro-architectural clearing of the pipeline, the exact operation within this branch is of no importance. We therefore simply populate this branch with a `nop`.

Evaluation. To evaluate our SPECTREM POCs, an out-of-bounds memory region is constructed within the device's memory and initialized with a random 256-bit string. A bounds check guarantees this string is architecturally inaccessible. To prolong the transient window, we flush the condition variable from the cache within the gadget itself. We do this to simplify the data acquisition and evaluation process as it ensures more consistent results. A more indirect approach to achieve this is cache thrashing, as discussed in Section 4.2.1. We will evaluate the impact of this technique in Section 7.3.

The MELTEMDOWN POC, on the other hand, is modified from a cache-based Meltdown-GP POC [26], which extracts the contents of the system registers. For the evaluation, we consider the system identification registers as they contain a constant, known value (i.e., a ground truth) and are not accessible to user programs.

Trigger. To aid in the evaluation of the POCs, a GPIO-based trigger signal is added right before or after the gadgets. Inserting such artificial triggers helps to simplify the acquisition and signal processing. Note that this is common practice in physical side-channel research [42,54,58]. In practice, an adversary can, e.g., trigger on the communication interface [5, 65], trigger on a pattern present in the EM side-channel trace [8, 56], or apply an additional signal processing step [24, 25].

5.2.2 Data Collection

An important consideration is the collection of the side-channel traces. Each trace consists of 25 000 sample points for the Spectre POC and 20 000 for Meltdown and thus captures a 1 µs and 0.8 µs window, respectively. For the Spectre attacks, the oscilloscope was set up to only capture the traces where the provided index was out-of-bounds.

Unless otherwise noted, bit error rates are based on 524 288 traces, where each trace encodes a single bit. The traces are collected in 32 batches of 16 384 traces each, where each batch consists of four frames of 4096 traces. A frame refers to a set of traces that are captured and sent to the computer simultaneously. This configuration is based on the specifications of the oscilloscope and considerations with regard to bit extraction. The oscilloscope, for instance, only has finite memory, and 4096 is roughly the maximal number of traces that can be collected at once. The batch size, on the other hand, is chosen to speed up the bit extraction by ensuring that the entire batch fits in memory.

5.2.3 Bit Extraction

For classification, we propose both supervised and unsupervised methods. This models cases where it is possible to collect sufficient training data and those where it is not. Specifically, the supervised method uses a multilayer perceptron (MLP) network. The unsupervised method is based on the Gaussian mixture model (GMM) clustering algorithm [18].

The classification is performed in the frequency domain. This makes the techniques insensitive to alignment, which is complicated by the out-of-order execution of the processor. Additionally, especially for the instruction gadget, the leakage is inherently present in the frequency domain. The optimal position of the FFT window was determined experimentally. The size of this window was chosen at 5000 samples.

Multilayer Perceptron. As we are working in the frequency domain with a limited number of inputs, we choose a fully connected network. Specifically, we use an FFT window size of 5000 and only consider the lowest 1000 frequencies. The architecture of the MLP is chosen in accordance with the architecture proposed by Pham et al. [59] as they target the same SoC and also perform classification in the frequency domain, with the only difference being the size of the input layer. The resulting MLP architecture is shown in Table 4 in Appendix B. To improve the BER, we only consider predictions with a confidence higher than 95 %.

An MLP is trained for each POC and each setup separately. For each network, 327 680 traces are collected in 20 batches of 16 384 traces. We use 14 batches to train the network, equating to 229 376 traces, and the remaining 98 304 are used for validation. The networks are trained with binary cross-entropy loss and L2 regularization using the Adam optimizer with an initial learning rate of 10^{-4} and an inverse time decay learning rate scheduler with a decay rate of one every ten epochs. The batch size is 32. Both training and inference were performed using a low-end desktop computer (Dell OptiPlex 3060, no discrete GPU). Due to the simplicity of the chosen network, training can be completed in less than an hour.

Gaussian Mixture Model. The evaluation using the GMM is performed for each batch separately and then averaged. We perform two preprocessing techniques to increase the accuracy of the GMM models. First, instead of providing all frequency components to the GMM algorithm, only the frequency components carrying leakage information are provided. These components are determined by performing a *t*-test between the frequency spectra of the traces based on the encoded secret bit. Second, we filter outliers from the GMM evaluation, which is especially important for the control flow gadget. Otherwise, strong outliers cause the algorithm to consider these outliers as one of the groups. The outliers are identified by computing the z-score corresponding to the clock frequency component and filtering out any traces that exhibit a z-score larger than 1.7 in absolute value.

6 Evaluation

6.1 SpectrEM Attack

In this section, we evaluate the performance of the two SpectrEM POCs, one implementing an instruction gadget and the other implementing a control flow gadget.

6.1.1 Probe Position

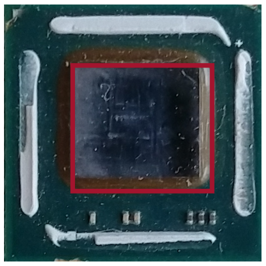
For optimal results, careful consideration must be taken when positioning the probe. Intuitively, as the leakage results from executing code on a specific physical core, placing the probe closer to said core may result in better performance. This experiment will use the stepper table of setup A to determine the optimal location.

Methodology. We enumerate a 7.20 mm by 7.95 mm grid at a resolution of 150 µm. This grid, indicated in Figure 4a, constitutes the whole SoC, as well as a small part outside.

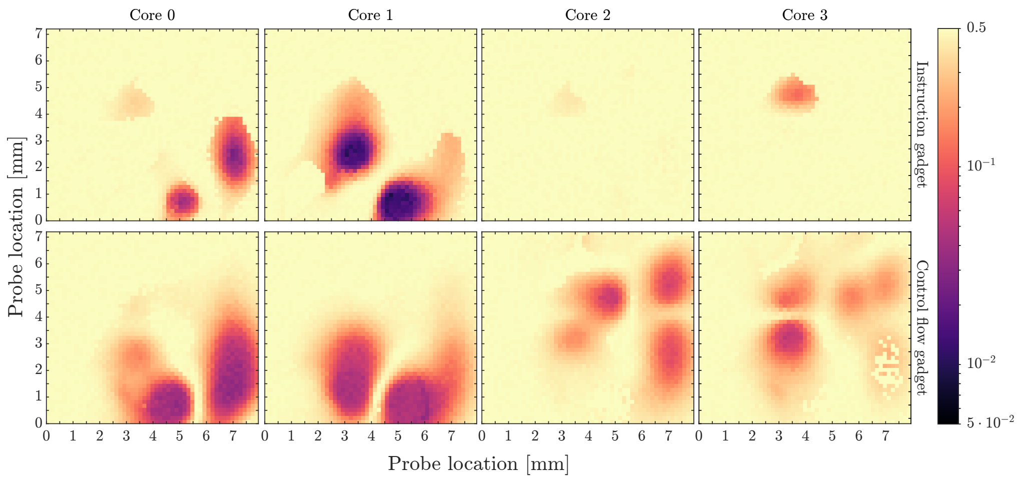
We collect 4096 traces at 2544 positions for each POC running on each core. Each location is evaluated using the unsupervised GMM clustering algorithm to avoid training a network for each location separately, which would require many more traces and significantly increase the run time.

Results. Figure 4b shows the BER at every measured location for both gadgets (row-wise) running on each core (column-wise). We use this data in the following experiments to determine the optimal probe position for each core and gadget. Interestingly, the optimal position for a specific core is slightly different for each gadget. One possible interpretation is that the gadgets have different physical leakage sources, which may reside in different locations on the core.

In what follows, we consistently consider the two resulting optimal positions for core 1, one for each type of gadget, to ensure consistency and comparability across experiments. Note that, for setup B, which does not accommodate precise positioning, we manually position the probe at (roughly) the optimal position.

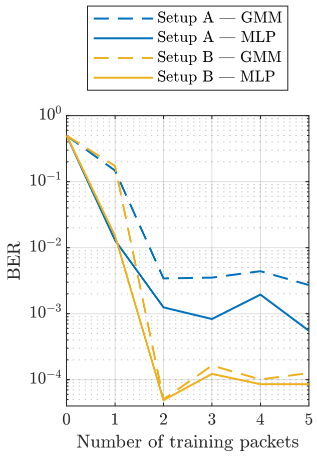


(a) Scanned area.

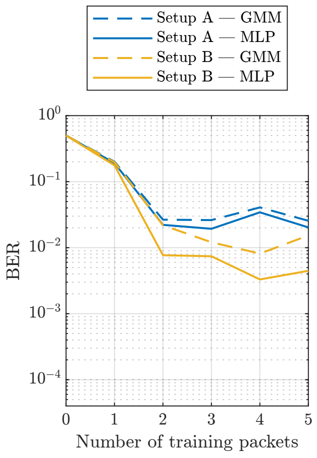


(b) BERs for both gadgets (row-wise) running on different cores (column-wise).

Figure 4: Heatmaps showing the BER at various probe positions (right) within the scanned area (left).



(a) Instruction gadget.



(b) Control flow gadget.

Figure 5: BER as function of the number of training packets.

6.1.2 SpectrEM Performance

Mistraining the branch predictor is a crucial step in mounting a Spectre-PHT attack. As our SpectrEM attacks do not assume code execution, this training has to take place through the interface. The number of packets required for mistraining determines the speed at which values can be extracted. This experiment evaluates the performance of both gadgets and quantifies the relation between the number of training packets and the acquisition rate. As discussed in Section 4.2.1, the two-bit predictor of the Arm Cortex-A72 [4] implies that, in theory, two training packets are sufficient.

Methodology. We first evaluate both gadgets using 5 training packets based on 32 batches of 16384 traces. We then evaluate the effect of reducing the number of training packets by considering zero up to five packets. For each number of training packets, we collect five batches of traces. The BERs are computed using the pre-trained MLP networks and the GMM clustering algorithm.

We also record the elapsed time, which has two main contributors: the triggering of the gadgets and the transfer of the traces from the oscilloscope to the PC used for classification. The latter is a characteristic of, on the one hand, the oscilloscope and, on the other hand, its connection to the PC. The classification time is explicitly excluded, as it does not present a bottleneck in the data collection pipeline.

Results. The bit error rates after training with five training packets are summarized in Table 1. Figure 5 displays, for both gadgets, the measured BERs as a function of the number of training packets. The error rates, as expected, drop when increasing the number of training packets. They indicate at least two training packets are required to consistently mistrain the branch predictor, indicative of a two-bit branch predictor.

Note that for the instruction gadget, the BER will also depend on the number of operand-dependent instruction present within the gadget. We evaluate the effect of fewer variable-time instructions in Appendix C.

Table 2 displays the time measurements acquired during the testing campaign. Capture time reflects the time to trigger the gadget and measure the trace, whereas transfer time covers the time to transfer the traces from the oscilloscope to the PC. Note that, while both are expressed as milliseconds per trace, they were timed on frames of 4096 traces each, where the table displays the average time per trace. The acquisition rates indicate the number of traces that can be acquired per second. This table shows that the highest realistic acquisition rate for these POCs is around 367 traces per second for this specific setup, as at least two training packets are required to obtain an acceptable BER. Note that the number of traces per second does not necessarily correspond to the number of

Table 1: Bit error rates after five training packets.

	Instruction gadget		Control flow gadget	
	Setup A	Setup B	Setup A	Setup B
GMM	0.350 %	0.011 %	3.592 %	2.441 %
MLP	0.167 %	0.008 %	2.609 %	1.037 %

Table 2: Acquisition speeds as function of training packets.

Training packets	Capture time [ms/trace]	Transfer time [ms/trace]	Acquisition rate [traces/s]
0	0.417	1.508	519
1	0.826	1.502	430
2	1.229	1.498	367
3	1.622	1.507	320
4	2.036	1.501	283
5	2.446	1.490	254

bits that can be extracted per second. This is only the case if the BER described above is considered acceptable. In cases where a lower BER is required, multiple traces will have to be collected for each bit. Additionally, these figures represent the optimal bitrates. Real-world targets will likely see a lower rate, as will be shown in Section 8.

To place the measured acquisition rates into perspective, we compare our results with previously described Spectre attacks. Our attacks are closely related to NetSpectre [67], which also overcomes the code execution requirement by operating through an interface. The NetSpectre attack based on an AVX covert channel is able to extract 60 bits per hour over a local network with an unspecified BER. When using a cache covert channel, this rate drops to 15 bits per hour. The original Spectre attack, which does assume a co-located attacker with code execution, claims a throughput of 80 kbit/s and a BER of <0.01 % when using an unoptimized implementation [40]. The different results are summarized in Table 3, where, for our attacks, the results when using the MLP on setup B corresponding to two training packets are displayed. Note that the bitrate in this table differs slightly from Table 2 as the MLP evaluation only considers confident predictions.

Note that Spectre and NetSpectre are evaluated on an x86 processor, whereas SPECTREM is evaluated on AArch64. One reason we considered AArch64 is that Arm has a bigger market share in the embedded context. Another important consideration is that NetSpectre operates with a remote adversary, whereas SPECTREM requires a physical attacker. Both of these factors make a direct comparison difficult. However, the evaluation still provides an indication of the relative performance of the respective attacks, each within their assumed threat model.

Table 3: Comparison of the extraction rates and BERs for the different Spectre scenarios leveraging different covert channels.

	Covert channel	Code Exec.	Phys. Access	Bitrate [bit/s]	BER [%]
Spectre [40]	Cache	Yes	No	80 000	<0.01
NetSpectre [67]	Cache	No	No	0.004	–
NetSpectre [67]	AVX	No	No	0.017	–
SPECTREM – Instruction	EM	No	Yes	366	<0.01
SPECTREM – Control flow	EM	No	Yes	350	0.769

6.2 MELTEMDOWN Attack

We now describe MELTEMDOWN, our physical Meltdown attack, for which the implementation details were already explained in Section 5.2.1. As mentioned, this attack will only be evaluated with the instruction gadget on setup B. We only consider the instruction gadget, as it has been shown to provide better results for the Spectre attack (cf. Table 1). We only consider setup B, as the instruction gadget performs better on setup B, and this setup is the most accessible.

Methodology. A total of 524 288 traces (32 batches) are collected, where each trace extracts one of the 248 test bits corresponding to the bits of the identified system registers. The traces are evaluated using the pre-trained MLP network as well as with the clustering algorithm.

Results. Of the 524 288 traces, only 123 out of 496 773 considered traces were misidentified when using the GMM, whereas 36 errors in 524 048 traces were recorded using the pre-trained MLP. This results in BERs of 0.02476 % and 0.00687 % for the GMM and MLP, respectively. The MELTEMDOWN attack thus performs slightly better than its SPECTREM counterpart. The reason for this can be found in the larger transient window for Meltdown, which we observed using the Speculator tool [53] (cf. Appendix A).

For this POC, the acquisition rate averages 241 traces/s. Similar to the SPECTREM attack, MELTEMDOWN cannot compete against attacks leveraging cache covert channels. The original Meltdown attack, for instance, achieves a best-case rate of 4.656 Mbps with a BER of 0.003 % [49], which is more than four orders of magnitude faster than our observed rate. Note, however, that this result is, similar to before, obtained on a different ISA, and that our MELTEMDOWN POC was not optimized, meaning higher rates should be obtainable.

7 Reducing Evaluation Assumptions

The evaluation thus far used various constructs to ease the evaluation process. We now discuss how each can be removed individually, followed by an evaluation to assess the impact on performance. We focus specifically on the control flow gadget for SPECTREM, but the same techniques can be applied for the instruction gadget and for MELTEMDOWN.

Methodology. The evaluation parameters are identical to the evaluation above. We base our evaluation on 524 288 traces (i.e., 32 batches), mistrain the branch predictor with five training packets, and evaluate the resulting traces using both techniques. The evaluation is performed on setup A.

We consider enabling frequency scaling, leaving the core affinity unspecified, and cache thrashing. For each of these individually, we discuss how they affect the extraction phase and experimentally verify their impact on performance.

7.1 Frequency Scaling

When DVFS is enabled, the processor will regulate its operating frequency to maximize performance and minimize power consumption while remaining within the thermal design limits. As the segmentation of traces is based on the frequency spectrum, the operating frequency needs to be known for every trace. This operating frequency can, however, be easily inferred from the frequency spectrum itself. A large peak will naturally be present at the instantaneous clock frequency. By identifying this peak, we can group traces based on their clock frequency and evaluate them separately.

The exact operating frequency of the processor does not appear to significantly influence the corresponding BER. After separating the traces into their corresponding frequencies and evaluating them separately, we obtained a BER of 5.560 % using the GMM algorithm and 3.237 % for the MLP network.

7.2 Core Affinity

When unconstrained, the scheduler assigns processes to a specific core and switches them around based on various factors, such as the current CPU load. This may hurt the performance of a SPECTREM attack as the EM probe position is optimized for one specific core. When the victim process runs on another core, however, the recorded traces display a detectable characteristic, allowing the attacker to filter out any traces not corresponding to the optimal core. As a result, the observed bitrate will—on average—decrease by 75 %. However, when optimal probe regions for different cores overlap, for instance, for cores 0 and 1 in Figure 4, the probe position can be chosen to minimize the combined BER.

When positioning the probe over the optimal position for cores 0 and 1, we obtain a BER of 3.760 % for the GMM and 1.791 % for the MLP. The bitrate reductions are 22.647 % and 24.844 %, respectively. These reductions include both traces running on different cores as well as outliers and uncertain predictions. While one would expect the bitrate penalty to be higher, it appears the scheduler has an inherent bias towards cores 0 and 1, as the theoretical bitrate reduction when considering two cores would be 50 %.

The presence of overlapping optimal regions is, of course, dependent on the specific processor. In the absence of any overlaps, the reduction in bitrate would be around 75 %, though a different measurement setup consisting of multiple probes may achieve a bitrate closer to the original.

7.3 Flushing

To prolong the length of the transient window, the variable describing the length of the array should not be present in the cache. As mentioned in Section 4.2.1, a remote adversary can achieve this through thrashing the cache, i.e., by accessing some other function that—through excessive memory accesses—essentially removes all entries from the cache.

We simulate this behavior by implementing a second function that iterates over a large memory buffer and expose it through the same UDP interface as the gadget. A third function simulates an access to the out-of-bounds memory region of interest to ensure it is brought back into the cache.

Due to the indiscriminating nature of the cache thrashing, other aspects of the test program will also be affected. As a result, the leakage appears more spread out in the time domain. To accommodate this, the inputs to the MLP network are slightly changed to include multiple FFT windows, i.e., a spectrogram, rather than a single one. Moreover, the cache thrashing introduces additional operations which slow down the extraction. When considering 5 training packets, for instance, the capture rate is reduced from 254 to 31.602 traces/s.

When filtering unconfident predictions, the MLP achieves a BER of 13.601 % at 16.648 traces/s, corresponding to a capacity of 7.097 bit/s. Similarly, for the GMM clustering algorithm, considering only confident predictions results in a BER of 22.172 % at a throughput of 11.279 traces/s, or a capacity of 2.670 bit/s.

This higher BER effectively means that an adversary will have to collect a few traces for each value to extract. Nevertheless, this number will still be relatively low.

8 Case Study

This section presents a case study of OpenSSH, a popular open-source implementation of the Secure Shell (SSH) protocol. OpenSSH is a relevant target as it (1) represents a security-critical application featuring interactions with external users, (2) allows for manual review due to its modest code base size, and (3) is widely deployed on embedded devices.

Methodology. We perform a high-level manual review of the latest version of OpenSSH at the time of writing (version 9.3) for control flow gadgets and assess under which circumstances these gadgets would be present in the compiled binary.

The evaluation in this section is performed identically to Section 6, introducing the same simplifications and using the GMM and MLP evaluation methods. The binaries are compiled using the default configuration with either gcc (9.4.0-1ubuntu1~20.04.1) or clang (10.0.0-4ubuntu1). The SSH daemon is instantiated on the target device using the default

```c
 1  Channel *
 2  channel_by_id(struct ssh *ssh, int id) {
 3    Channel *c;
 4
 5    if (id < 0 || (u_int)id >= ssh->chanctxt->
        channels_alloc) {
 6      logit_f("%d: bad id", id);
 7      return NULL;
 8    }
 9    c = ssh->chanctxt->channels[id];
10    if (c == NULL) {
11      logit_f("%d: bad id: channel free", id);
12      return NULL;
13    }
14    return c;
15  }
```

Listing 4: The `channel_by_id` function in `channels.c`.

configuration. The gadgets are triggered using the Paramiko Python library on a remote computer over the local network. We collect a total of 81 920 traces (i.e., 5 batches), where for each trace we mistrain the branch predictor using five training packets. After accessing the gadget, we retrieve the out-of-bounds value to compute the BER. Note that these values do not necessarily contain any relevant information and are only used for verification purposes.

8.1 SSH Server

The SSH specification allows multiple channels to be opened over a single connection. As defined in RFC 4254, messages pertaining to a specific channel always contain the recipient's channel identifier [50]. Listing 4 shows the function that processes this identifier and returns a pointer to the channel. Line 5 performs a bounds check. Line 10 introduces a control flow dependency on the variable `c`, which is loaded in based on the provided `id`, thus making it a control flow gadget.

While a gadget from a source-code perspective, both gcc and clang place the buffer pointer in the same cache line as its length due to the current structure of the `Channel` struct. As a result, this code pattern would not be exploitable in its current state. However, small changes to the source code or compilers could still introduce this gadget in the compiled binary, as we show in the evaluation.

Methodology. We modify `channels.c` to align the variable `channels_alloc` with the cache line size, thus placing it in a different cache line. We target the function `channel_input_window_adjust`, where we repeatedly supply a window adjustment of zero bytes.

Results. We obtained a capacity of 2.96 bit/s (23.87 traces/s at a BER of 29.56 %) when using the GMM clustering algorithm, and 20.94 bit/s (33.57 traces/s at a BER of 7.28 %) when using the MLP network.

8.2 SFTP Server

The SSH File Transfer Protocol (SFTP) allows users to securely access a remote file system. Each opened file or directory is assigned a unique handle, which is sent by the external user in each operation to identify its target [22]. Listing 5 shows the function that checks whether the provided handle is valid and of the correct type. When used in an if-statement, as is the case in, e.g., the `get_handle` function, a control-flow dependency is placed on the comparison.

In the specific example of the `get_handle` function, however, the presence of the gadget in the binary is not guaranteed and depends on the compiler. For instance, gcc produces a conditional select, which does not show any leakage, whereas clang does insert a branch instruction, thus creating a control flow gadget. Similar to the SSH gadget, however, clang coincidentally places the pointer to the buffer in the same cache line as the length variable.

Methodology. We modify `sftp-server.c` to place the pointer to the buffer and its length on different cache lines and compile with clang. We target the `process_read` function by repeatedly sending read commands with the length field set to zero.

Results. We obtained a capacity of 20.41 bit/s (45.67 traces/s at a BER of 12.84 %) when using the GMM clustering algorithm, and 53.39 bit/s (69.27 traces/s at a BER of 3.72 %) when using the MLP network.

8.3 Limitations

While this case study shows that vulnerable code patterns can be found in real-world code bases, we were unable to exploit the uncovered gadgets as-is, requiring minute changes to the internal representation to make them exploitable. Such changes could arise due to, for instance, benign source-code modifications. With these changes in place, the evaluation indicates that gadgets in complex, real-world binaries could still be exploited.

The gadgets we discuss, however, are not particularly powerful gadgets, only allowing comparison with a fixed value. Additionally, OpenSSH implements countermeasures that ensure the private key is not exposed when at rest by encrypting it with a large prekey, thus significantly increasing the attack effort for transient execution attacks.

Due to the absence of applicable gadget scanners, we performed a manual review. A thorough evaluation of larger code

```c
static int handle_is_ok(int i, int type) {
  return i >= 0 && (u_int)i < num_handles &&
     handles[i].use == type;
}
```

Listing 5: The `handle_is_ok` function in `sftp-server.c`.

bases would require automated tools that are able to accurately detect exploitable control flow gadgets. While tools capable of detecting the patterns that define control flow gadgets exist, e.g., uncovering 407 of these gadgets in the Linux kernel [35], these tools are typically written for x86 binaries and often do not differentiate between gadgets that are accessible to a remote attacker through an interface. We leave a more thorough exploration of the attack surface for future work.

9 Discussion

Relevance. In contrast with traditional transient execution attacks, SPECTREM and MELTEMDOWN rely on physical covert channels. While SPECTREM, for instance, requires a physical attacker, it opens the possibility of transient execution attacks on embedded devices for adversaries without code execution. Additionally, both attacks show that countermeasures against transient execution attacks must not focus on mitigating micro-architectural-based covert channels as this may fail to prevent these attacks in embedded scenarios. Instead, more fundamental approaches are required, focusing on the source of the leakage rather than its effects.

Compared to other physical attacks, the attacks described in this paper distinguish themselves by leveraging micro-architectural behavior. While profiled side-channel attacks on unprotected cryptographic implementations may require a similar or lower attack effort compared to a SPECTREM attack, controlling the micro-architectural behavior allows SPECTREM to control the disclosed memory location rather than inferring a specific cryptographic key. Additionally, while a physical attacker could potentially perform more powerful attacks, such as snooping the memory bus, performing these attacks may be more involved and may require more expensive equipment [46] compared to SPECTREM attacks.

Limitations. While a control flow gadget can be considered more generic than a conventional Spectre gadget—indeed, this pattern can be readily found in, e.g., the Linux kernel [12, 35]—a major limitation of the SPECTREM attack is the requirement for this gadget to be accessible through an external interface. While this enables attacks without requiring code execution, it limits the number of available gadgets.

Similarly, for the instruction gadget, the requirement for multiple variable-time instructions (cf. Appendix C) means that it is unlikely to occur in a real-world code base. Instead, this gadget is more relevant in contexts where an attacker can write their own gadgets, such as in a MELTEMDOWN attack.

Applicability to other interfaces. During the evaluation, we considered SPECTREM attacks with gadgets exposed through a UDP interface. This choice of interface was mainly a practical consideration. SPECTREM could, in principle, leverage any interface present on the target device. Indeed, assuming physical access to the device enables access to local interfaces such as USB, I^{2}C, or SPI.

Applicability to other ISAs. While we focused on an Arm-based CPU to evaluate our proposed attacks, we believe our results are relevant to other ISAs, including x86 and RISC-V. Even though the micro-architectural implementation details will differ between different ISAs or even between two different processors implementing the same ISA, the general concepts remain the same. For the control flow gadget, we note that most modern x86 processors also allow for nested speculation. As a result, control flow dependencies may introduce the same physical manifestations upon which the control flow gadget is built. For the instruction gadget, we note that the vast majority of processors have variable-time instructions [19]. As a result, we expect the findings of this paper to be relevant for other ISAs.

10 Related Work

Gadgets. Spectre attacks that encode the transient secret value in a micro-architectural element can also make use of a nested comparison [39], similar to the control flow gadget. The leak gadget from a NetSpectre attack [67], for instance, leverages a memory load or an AVX instruction within a nested branch to encode the secret bit in the cache or the AVX unit. One difference is that our control flow gadget does not rely on the presence of any particular instruction within the inner branch, making them more general than NetSpectre's leak gadgets. Additionally, as NetSpectre assumes a remote adversary, where we consider a stronger one with physical access, NetSpectre requires a secondary gadget to read out the encoded value, which is not the case in our attacks.

BranchSpec [14], which encodes the secret value in the branch predictor itself, considers gadgets that are identical to our control flow gadget, i.e., not requiring any particular instruction within the inner branch.

All of these examples also constitute control flow gadgets in the context of electromagnetic covert channels. The main difference is that attacks described above retrieve the secret from a micro-architectural element, whereas in the case of SPECTREM, this value is inferred through EM waves.

Countermeasures. Given the complexity and heterogeneity of the attacks and the importance of the underlying mechanisms that cause them, multiple solutions have been proposed that attempt to thwart all attacks. The most straightforward way to fend off any potential transient execution attack would be to outright disable speculation and out-of-order execution [40, 49]. While highly effective, this would come at an unacceptable cost with regard to performance. Instead, various alternative defenses have been proposed that attempt a partial or full defense against transient execution attacks while minimizing the performance degradation.

Meltdown attacks are easier to mitigate compared to Spectre attacks. They rely on the behavior that faulting instructions still pass on their results to subsequent instructions in the transient domain. Patching this behavior in hardware, e.g., by passing in a default value when the instruction faults [12], ensures that the secret value cannot be leaked.

As protecting against Spectre attacks, on the other hand, is a non-trivial task, numerous software- and hardware-based countermeasures have been proposed [13, 34]. Since all existing Spectre attacks employ micro-architectural-based covert channels, one approach that has been explored is to prevent or limit the use of these channels [1, 36–38, 51, 66, 74]. While this can stop some attacks, it does not prevent the use of physical covert channels. Instead, the micro-architecture could, for instance, be changed to prevent the use of speculatively loaded data by subsequent instructions [6, 16, 21, 72]. This can prevent Spectre attacks regardless of the used covert channel, as the instructions to transmit the value are now not allowed to be executed. Alternatively, the transient execution can be stopped when dealing with (potentially) secret data by, for instance, strategically inserting speculation barriers—that stop speculation—in vulnerable code constructs. They can stop some attacks [53], but they do not prevent leakage through every potential side channel [67] and rely on detection tools to identify the vulnerable code snippets.

Gadget Detection. Selective countermeasures for Spectre-PHT rely on detection mechanisms to select vulnerable code segments to protect. Detecting gadgets is not straightforward due to the versatility of constructs and covert channels. Tools relying on static code analysis are inherently limited by the provided templates. Any gadget not adhering to these templates, for instance, because a different covert channel is used, will not be detected [39]. To address this, tools have been proposed that utilize taint analysis [35, 61, 71], symbolic execution [27, 28, 70], fuzzing [57], and deep learning [69]. These tools do not rely on static pattern matching and will therefore be able to uncover more potential gadgets.

Not all tools can be used to detect control flow and instruction gadgets. Some tools limit their scope to cache-based covert channels [28, 61, 69, 70]. Others can detect control flow dependencies on a transient secret, allowing them to detect control flow gadgets, but not instruction gadgets [27, 35, 71]. SpecFuzz [57], on the other hand, exposes all gadgets that can result in speculative out-of-bounds memory accesses. As a result, it can detect any Spectre-PHT gadget regardless of the covert channel, including both our EM gadgets.

These tools, however, do not necessarily distinguish between gadgets accessible through an interface, limiting their applicability to find exploitable SPECTREM gadgets.

EM Covert Channels. Various electromagnetic covert channels have been proposed in the literature. These typically focus on air-gapped computers, which are physically isolated from external networks. Within this context, a desirable property of the covert channel is a large disclosure range. To achieve this, they build their covert channels upon architectural components displaying distinct EM behavior. Different building blocks for this covert channel have been investigated, for instance, monitors [30, 44], memory accesses [29, 68, 76], or peripheral devices [31]. For instance, the fastest proposed EM covert channel, BitJabber, achieves a bitrate of up to 300 kbps by modulating signals generated by the DRAM clock [76]. Similarly, EMLoRa boasts a range of up to 250 m by modulating memory accesses [68].

The requirements for an EM covert channel for transient execution attacks, on the other hand, are vastly different. The primary constraint is the time window during which a bit can be transmitted. As a result, our work necessarily builds upon short-lived micro-architectural effects, which can only be observed with very localized EM measurements with a minimal distance between the probe and the IC.

11 Conclusion

In this paper, we investigated whether transient execution is amenable to leaking secret information through physical side effects such as EM emanations. Despite the constrained transient window, we identified and evaluated suitable code patterns based on secret-dependent instruction timing and control flow. Although our attacks require physical access, they do not rely on decoding secrets from particular micro-architectural elements, circumventing countermeasures that capitalize on this property. Moreover, by assuming access through an interface, SPECTREM may overcome the requirement of executing attacker code altogether.

We demonstrated the feasibility of the proposed attacks on the Arm Cortex-A72. Our evaluation confirmed that transient execution attacks relying on electromagnetic covert channels form a practical attack, even in an unsupervised setting, where no traces are required to train the classifier. SPECTREM managed a bitrate of 366 bit/s at $<$0.01 % BER for the instruction gadget and 350 bit/s at a BER of 0.769 % for the control flow gadget. MELTEMDOWN, on the other hand, managed 241 bit/s at a BER of 0.00687 %. To our knowledge, these attacks are the first to confirm the threat of physical side-channel leakage during transient execution.

Acknowledgments

We thank the anonymous USENIX Security reviewers and our shepherd for their valuable and constructive feedback. This research is partially funded by the European Research Council (ERC #101020005 BELFORT) and the Flemish Government through the FWO project TRAPS. It was also supported by the CyberSecurity Research Flanders (#VR20192203), Horizon Europe (#101070008 ORSHIN), and the Research Fund KU Leuven. Antoon Purnal is supported by a grant of the Research Foundation - Flanders (FWO).

References

[1] Sam Ainsworth and Timothy M. Jones. Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state. In *IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[2] Arm Limited. CORTEX-A72. `https://www.arm.com/products/silicon-ip-cpu/cortex-a/cortex-a72`. Accessed: May, 2022.

[3] Arm Limited. *Cortex®A72 Software Optimization Guide*, 2015.

[4] Arm Limited. *ARM® Cortex®-A72 MPCore Processor Technical Reference Manual*, 2016. Revision r0p3.

[5] Josep Balasch, Benedikt Gierlichs, Oscar Reparaz, and Ingrid Verbauwhede. DPA, bitslicing and masking at 1 ghz. In *Cryptographic Hardware and Embedded Systems (CHES)*, 2015.

[6] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. SpecShield: Shielding speculative data from microarchitectural covert channels. In *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019.

[7] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch history injection: On the effectiveness of hardware mitigations against cross-privilege spectre-v2 attacks. In *USENIX Security Symposium*, 2022.

[8] Arthur Beckers, Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. Design and implementation of a waveform-matching based triggering system. In *Constructive Side-Channel Analysis and Secure Design (COSADE)*, 2016.

[9] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting speculative execution through port contention. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[10] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *USENIX Security Symposium*, 2018.

[11] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: hijacking transient execution through microarchitectural load value injection. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[12] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security Symposium*, 2019.

[13] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. SoK: Practical foundations for software spectre defenses. In *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[14] Md Hafizul Islam Chowdhuryy, Hang Liu, and Fan Yao. Branchspec: Information leakage attacks exploiting speculative branch instruction executions. In *IEEE International Conference on Computer Design (ICCD)*, 2020.

[15] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, 2016.

[16] Lesly-Ann Daniel, Marton Bognar, Job Noorman, Sébastien Bardin, Tamara Rezk, and Frank Piessens. PROSPECT: Provably secure speculation for the constant-time policy. In *USENIX Security Symposium*, 2023.

[17] Ghada Dessouky, Alexander Gruler, Pouya Mahmoody, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Chunked-cache: On-demand and scalable cache isolation for security architectures. In *Network and Distributed System Security Symposium (NDSS)*, 2022.

[18] Richard O. Duda and Peter E. Hart. *Pattern classification and scene analysis*. A Wiley-Interscience publication. Wiley, 1973.

[19] Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. *Technical University of Denmark*, 2022.

[20] Jacob Fustos, Michael Garrett Bechtel, and Heechul Yun. SpectreRewind: Leaking secrets to past instructions. In *ACM Workshop on Attacks and Solutions in Hardware Security Workshop (ASHES@CCS)*, 2020.

[21] Jacob Fustos, Farzad Farshchi, and Heechul Yun. Spectreguard: An efficient data-centric defense mechanism against spectre attacks. In *ACM Design Automation Conference (DAC)*, 2019.

[22] Joseph Galbraith and Oskari Saarenmaa. SSH File Transfer Protocol. Internet-Draft draft-ietf-secsh-filexfer-13, Internet Engineering Task Force, July 2006. Work in Progress.

[23] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded Systems (CHES)*, 2001.

[24] Daniel Genkin, Noam Nissan, Roei Schuster, and Eran Tromer. Lend me your ear: Passive remote physical side channels on PCs. In *USENIX Security Symposium*, 2022.

[25] Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. Stealing keys from PCs using a radio: Cheap electromagnetic attacks on windowed exponentiation. In *Cryptographic Hardware and Embedded Systems (CHES)*, 2015.

[26] Cosmin Gorgovan. spec_poc_arm. `https://github.com/gorgovan2018specpocarm/spec_poc_arm`, 2018.

[27] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled detection of speculative information flows. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[28] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. SpecuSym: Speculative symbolic execution for cache timing leak detection. In *ACM International Conference on Software Engineering (ICSE)*, 2020.

[29] Mordechai Guri, Assaf Kachlon, Ofer Hasson, Gabi Kedma, Yisroel Mirsky, and Yuval Elovici. GSMem: Data exfiltration from air-gapped computers over GSM frequencies. In *USENIX Security Symposium*, 2015.

[30] Mordechai Guri, Gabi Kedma, Assaf Kachlon, and Yuval Elovici. AirHopper: Bridging the air-gap between isolated networks and mobile phones using radio frequencies. In *IEEE International Conference on Malicious and Unwanted Software: The Americas (MALWARE)*, 2014.

[31] Mordechai Guri, Matan Monitz, and Yuval Elovici. USBee: Air-gap covert-channel via electromagnetic emission from USB. In *IEEE Conference on Privacy, Security and Trust (PST)*, 2016.

[32] Mordechai Guri, Boris Zadov, Dima Bykhovsky, and Yuval Elovici. PowerHammer: Exfiltrating data from air-gapped computers through power lines. *IEEE Transactions on Information Forensics and Security*, 2020.

[33] Jann Horn. Speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.

[34] Guangyuan Hu, Zecheng He, and Ruby B. Lee. SoK: Hardware defenses against speculative execution attacks. In *International Symposium on Secure and Private Execution Environment Design (SEED)*, 2021.

[35] Brian Johannesmeyer, Jakob Koschel, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. KASPER: Scanning for generalized transient execution gadgets in the linux kernel. In *Network and Distributed System Security Symposium (NDSS)*, 2022.

[36] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael B. Abu-Ghazaleh. SafeSpec: Banishing the spectre of a meltdown with leakage-free speculation. In *ACM Design Automation Conference (DAC)*, 2019.

[37] Sungkeun Kim, Farabi Mahmud, Jiayi Huang, Pritam Majumder, Neophytos Christou, Abdullah Muzahid, Chia-Che Tsai, and Eun Jung Kim. ReViCe: Reusing victim cache to prevent speculative cache leakage. In *IEEE Secure Development (SecDev)*, 2020.

[38] Vladimir Kiriansky, Ilia A. Lebedev, Saman P. Amarasinghe, Srinivas Devadas, and Joel S. Emer. DAWG: A defense against cache timing attacks in speculative execution processors. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[39] Paul Kocher. Spectre mitigations in Microsoft's C/C++ compiler. https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html, 2018. Accessed: March, 2022.

[40] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[41] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology (CRYPTO)*, 1999.

[42] Paul C. Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, 2011.

[43] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu-Ghazaleh. Spectre returns! Speculation attacks using the return stack buffer. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2018.

[44] Markus G. Kuhn and Ross J. Anderson. Soft tempest: Hidden data transmission using electromagnetic emanations. In *Information Hiding*, volume 1525 of *Lecture Notes in Computer Science*, pages 124–142. Springer, 1998.

[45] Corentin Lavaud, Robin Gerzaguet, Matthieu Gautier, Olivier Berder, Erwan Nogues, and Stéphane Molton. Whispering devices: A survey on how side-channels lead to compromised information. *Journal of Hardware Systems Security*, 2021.

[46] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-che Tsai, and Raluca Ada Popa. An off-chip attack on hardware enclaves via the memory bus. In *USENIX Security Symposium*, 2020.

[47] Moritz Lipp, Vedad Hadzic, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take A Way: Exploring the security implications of amd's cache way predictors. In *ACM Asia Conference on Computer and Communications (AsiaCCS)*, 2020.

[48] Moritz Lipp, Andreas Kogler, David F. Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. PLATYPUS: Software-based power side-channel attacks on x86. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[49] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium*, 2018.

[50] Chris M. Lonvick and Tatu Ylonen. The Secure Shell (SSH) Connection Protocol. RFC 4254, January 2006.

[51] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. DOLMA: Securing speculation with the principle of transient non-observability. In *USENIX Security Symposium*, 2021.

[52] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[53] Andrea Mambretti, Matthias Neugschwandtner, Alessandro Sorniotti, Engin Kirda, William K. Robertson, and Anil Kurmus. Speculator: A tool to analyze speculative execution attacks and mitigations. In *ACM Annual Computer Security Applications Conference (ACSAC)*, 2019.

[54] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.

[55] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. Thermal covert channels on multi-core platforms. In *USENIX Security Symposium*, 2015.

[56] Colin O'Flynn and Zhizhang (David) Chen. Side channel power analysis of an AES-256 bootloader. In *IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, 2015.

[57] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. Specfuzz: Bringing spectre-type vulnerabilities to the surface. In Srdjan Capkun and Franziska Roesner, editors, *USENIX Security Symposium*, 2020.

[58] Siddika Berna Örs, Elisabeth Oswald, and Bart Preneel. Power-analysis attacks on an FPGA - first experimental results. In *Cryptographic Hardware and Embedded Systems (CHES)*, 2003.

[59] Duy-Phuc Pham, Damien Marion, Matthieu Mastio, and Annelie Heuser. Obfuscation revealed: Leveraging electromagnetic signals for obfuscated malware classification. In *ACM Annual Computer Security Applications Conference (ACSAC)*, 2021.

[60] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. Systematic analysis of randomization-based protected cache architectures. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[61] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. SpecTaint: Speculative taint analysis for discovering spectre gadgets. In *Network and Distributed System Security Symposium (NDSS)*, 2021.

[62] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In *E-smart*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2001.

[63] Moinuddin K. Qureshi. Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[64] Moinuddin K. Qureshi. New attacks and defense for encrypted-address cache. In *International Symposium on Computer Architecture (ISCA)*, 2019.

[65] Thomas Roche, Victor Lomné, Camille Mutschler, and Laurent Imbert. A side journey to Titan. In *USENIX Security Symposium*, 2021.

[66] Gururaj Saileshwar and Moinuddin K. Qureshi. CleanupSpec: An "undo" approach to safe speculation. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[67] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read arbitrary memory over network. In *European Symposium on Research in Computer Security (ESORICS)*, 2019.

[68] Cheng Shen, Tian Liu, Jun Huang, and Rui Tan. When LoRa meets EMR: electromagnetic covert channels can be super resilient. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[69] M. Caner Tol, Berk Gülmezoglu, Koray Yurtseven, and Berk Sunar. FastSpec: Scalable generation and detection of spectre gadgets using neural embeddings. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2021.

[70] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. KLEESpectre: Detecting information leakage through speculative cache attacks via symbolic execution. *ACM Transactions on Software Engineering and Methodology*, 2020.

[71] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE Transactions on Software Engineering*, 2021.

[72] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. NDA: preventing speculative execution attacks at their source. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[73] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting cache attacks via cache set randomization. In *USENIX Security Symposium*, 2019.

[74] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. InvisiSpec: Making speculative execution invisible in the cache hierarchy. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[75] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium*, 2014.

[76] Zihao Zhan, Zhenkai Zhang, and Xenofon D. Koutsoukos. BitJabber: The world's fastest electromagnetic covert channel. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2020.

A Arm Cortex-A72 Transient Window Size

We extend the Speculator tool [53] to AArch64 and use it to measure the number of available instructions within the transient window on the Arm Cortex-A72. Speculator uses the CPU's hardware counters to determine the extent of the transient execution using marker instructions that appear in said counters. By iteratively increasing the number of instructions between the trigger instruction and the marker instruction, Speculator can determine which instructions were executed transiently.

Methodology. We construct two test programs, one for Spectre and one for Meltdown, to measure the transient window size resulting from both branch mispredictions and exceptions. To determine whether all inserted instructions were, in fact, executed transiently, the floating-point move immediate instruction (`fmov`) is used as a marker instruction and placed after the inserted instructions. This `fmov` instruction increments the `VFP_SPEC` performance counter, which counts the floating point operations that are executed transiently.

The Speculator tool analyzes both test snippets by inserting up to 130 addition or unsigned division instructions. Since the number of clock cycles occupied by an unsigned division differs based on the operands, three different operands are

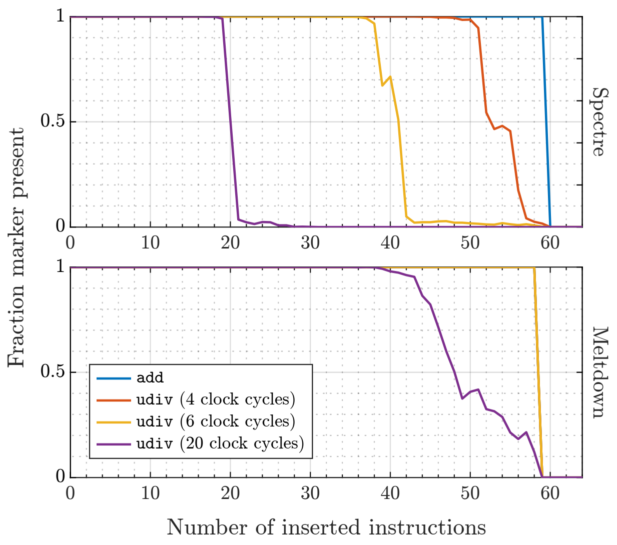


Figure 6: The fraction out of the 1000 runs for which the marker instruction was recorded by the performance counter `VFP_SPEC`, for Spectre (top) and Meltdown (bottom).

used such that the instruction takes either 4, 6, or 20 clock cycles. In all cases, the instructions only have a dependency on the transient load. This avoids stalls in the pipeline that might reduce the measured transient window size.

Note that this experiment aims to measure the window size in the best-case scenario, not necessarily in a realistic one. This provides an upper bound on the number of instructions that can be considered for an EM covert channel.

Results. Figure 6 shows the results of this analysis up to 64 instructions. For each number of inserted instructions, the fraction out of 1000 runs for which the marker was detected is plotted, both for Spectre (top) and for Meltdown (bottom).

For Spectre, the rate of additions drops off sharply after 59 inserted instructions, indicating the transient window size is limited by the micro-architectural elements. When inserting unsigned divisions, on the other hand, the transition occurs earlier, though not as sharply, indicating that the limiting factor here is the time constraint.

The results from the Meltdown snippet are shown in Figure 6 (bottom). Similar to the results for the Spectre attack, the additions see a sharp drop-off, this time after 58 instructions. Due to a pointer chase, the transient window is larger than in the case of the Spectre attack, which results in the divisions that occupy 4 and 6 clock cycles also displaying this behavior. The only case that does not reach the upper bound dictated by the micro-architectural elements is the unsigned division instruction that occupies 20 clock cycles.

These results give a rough estimate of the number of instructions that can be used in an EM covert channel. They indicate that the processor—in the used configuration—allows for up to approximately 60 transient instructions, though this number decreases the more cycles an instruction takes.

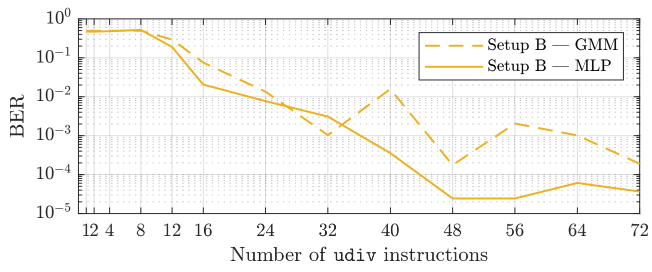


Figure 7: BER for varying number of `udiv` instructions.

B MLP Network Architecture

Table 4: MLP network for supervised classification, after Pham et al. [59].

Layer	Size	Activation
Input	1000	–
Dense	500	ReLu
Dense	200	ReLu
Dense	100	ReLu
Dense	2	Softmax

C Instruction Dependence

An important parameter when considering instruction gadgets is the number of instructions with operand-dependent timings within the gadget. Intuitively, the more such instructions are present, the larger the leakage, and thus the easier it should become to extract the value. This experiment, therefore, aims to determine how the number of instructions with operand-dependent timings affects the bit error rate and how many are required to get an acceptable BER from a single trace.

Methodology. We consider various numbers of `udiv` instructions: 1, 2, 4, 8, 12, 16, 24, 32, 40, 48, 56, 64, and 72. For each case, 81 920 traces are collected on setup B in five batches of 16 384. The BERs are computed using an MLP (supervised) and GMM clustering (unsupervised). Note that the MLP is pre-trained for a fixed number (i.e., 64) of `udiv`s. Five training packets are used to mistrain the branch predictor.

Results. Figure 7 depicts the results. As expected, the bit error rate declines with an increasing number of `udiv` instructions. It is close to 50 % for less than 8 division instructions, suggesting that the leakage for those cases is not strong enough to extract the secret bit with a single trace. After around 48 instructions, the BER levels off, indicating that no more instructions could be executed in the transient window.