From hardware to software: An end-to-end side-channel attack surface analysis

Milburn, Alyssa Anne

2025

DOI (link to publisher)
10.5463/thesis.330

document version
Publisher's PDF, also known as Version of record

VRIJE UNIVERSITEIT

FROM HARDWARE TO SOFTWARE: AN END-TO-END SIDE-CHANNEL ATTACK SURFACE ANALYSIS

PH.D. THESIS

Alyssa Anne Milburn

The research reported in this dissertation was conducted at the Faculty of Science, at the Department of Computer Science, of the Vrije Universiteit Amsterdam.

This work was supported by the Office of Naval Research (ONR) under awards N00014-16-1-2261 and N00014-17-1-2788.

VRIJE UNIVERSITEIT

FROM HARDWARE TO SOFTWARE: AN END-TO-END SIDE-CHANNEL ATTACK SURFACE ANALYSIS

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor
aan de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. J.J.G. Geurts,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Bètawetenschappen
op dinsdag 4 februari 2025 om 15.45 uur
in een bijeenkomst van de universiteit,
De Boelelaan 1105

door

Alyssa Anne Milburn

geboren te Bath, Verenigd Koninkrijk

promotor:	prof.dr.ir. H.J. Bos
copromotor:	dr. C. Giuffrida
promotiecommissie:	prof.dr. W.J. Fokkink
	prof.dr. Y. Yarom
	dr. C. Fletcher
	prof.dr. L. Batina
	dr. M. Payer

The security of our products is one of our most important priorities.

– Intel

Acknowledgements

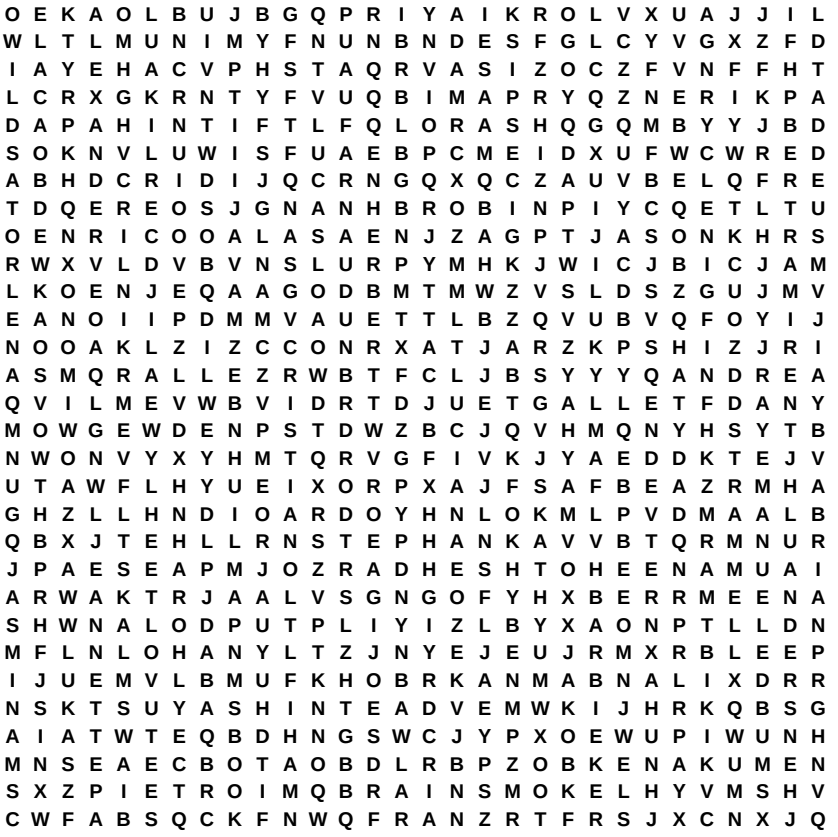
Thank you to all of these wonderful people whose knowledge, friendship and support helped me during this journey. Rather than the usual list, many of their names can instead be found in the wordsearch on the next page. I truly appreciated the company of so many brilliant, friendly and kind people over the years.

Having said that, I will take a little space to express some personal gratitude to some of those wonderful people. Emma, you are amazing in so many ways and have been a delightfully positive influence on my life. Sarah, you've been a friend and an inspiration for so many years, and my life is much better for having you in it. Koen, you not only helped me debug both SPEC and many real-life problems, but *also* you have cats. (Cats are cute.) Shiz, you consistently cheered me up when I needed it, introduced me to many new things, and have been a great friend. Peter, I learnt so much from just sitting next to you and listening to your patient explanations. Albert, you have an amazing talent to motivate me to actually do amazing things. (Or at least help you do them.) And finally, thanks to Anders, barbie, Henrique, Ke, kena, Martijn, Matthew, Stephan, Pietro, Yashin and everyone else for being there ready to help or just listen whenever I needed it.

Herbert, you are an amazing and inspiring researcher and teacher; thank you for giving me the opportunity to do a PhD in your group. Cristiano, your energy, creativity and brilliance continues to amaze me; thank you for sharing that with me. I learnt so much from both of you, as well as from the others in your group.

I wrote almost all of this thesis (except, admittedly, these acknowledgements) before joining Intel back in 2021, and promptly ignored it for many years. If you're reading this, my procrastination may have finally come to an end. I lay the blame at the feet of everyone who not only encouraged me to submit the thesis, but also to actually schedule a defense. In particular, I am grateful to my committee members – Chris, Lejla, Mathias, Wan and Yuval – for not only being willing to volunteer their time but also for their thoughtful comments and suggestions.

And finally, I will always be grateful beyond words for the love and support of my partner Bertram – as well as Sarely, who remains dearly missed.


O E K A O L B U J B G Q P R I Y A I K R O L V X U A J J I L
W L T L M U N I M Y F N U N B N D E S F G L C Y V G X Z F D
I A Y E H A C V P H S T A Q R V A S I Z O C Z F V N F F H T
L C R X G K R N T Y F V U Q B I M A P R Y Q Z N E R I K P A
D A P A H I N T I F T L F Q L O R A S H Q G Q M B Y Y J B D
S O K N V L U W I S F U A E B P C M E I D X U F W C W R E D
A B H D C R I D I J Q C R N G Q X Q C Z A U V B E L Q F R E
T D Q E R E O S J G N A N H B R O B I N P I Y C Q E T L T U
O E N R I C O O A L A S A E N J Z A G P T J A S O N K H R S
R W X V L D V B V N S L U R P Y M H K J W I C J B I C J A M
L K O E N J E Q A A G O D B M T M W Z V S L D S Z G U J M V
E A N O I I P D M M V A U E T T L B Z Q V U B V Q F O Y I J
N O O A K L Z I Z C C O N R X A T J A R Z K P S H I Z J R I
A S M Q R A L L E Z R W B T F C L J B S Y Y Y Q A N D R E A
Q V I L M E V W B V I D R T D J U E T G A L L E T F D A N Y
M O W G E W D E N P S T D W Z B C J Q V H M Q N Y H S Y T B
N W O N V Y X Y H M T Q R V G F I V K J Y A E D D K T E J V
U T A W F L H Y U E I X O R P X A J F S A F B E A Z R M H A
G H Z L L H N D I O A R D O Y H N L O K M L P V D M A A L B
Q B X J T E H L L R N S T E P H A N K A V V B T Q R M N U R
J P A E S E A P M J O Z R A D H E S H T O H E E N A M U A I
A R W A K T R J A A L V S G N G O F Y H X B E R R M E E N A
S H W N A L O D P U T P L I Y I Z L B Y X A O N P T L L D N
M F L N L O H A N Y L T Z J N Y E J E U J R M X R B L E E P
I J U E M V L B M U F K H O B R K A N M A B N A L I X D R R
N S K T S U Y A S H I N T E A D V E M W K I J H R K Q B S G
A I A T W T E Q B D H N G S W C J Y P X O E W U P I W U N H
M N S E A E C B O T A O B D L R B P Z O B K E N A K U M E N
S X Z P I E T R O I M Q B R A I N S M O K E L H Y V M S H V
C W F A B S Q C K F N W Q F R A N Z R T F R S J X C N X J Q

Contents

List of Figures

List of Tables

Publications

This dissertation includes several published research papers; details are provided below. The text differs from the published versions in minor editorial changes made to improve readability. Details of my individual contributions to each chapter can be found at the end of this dissertation.

Albert Spruyt, Alyssa Milburn, and Łukasz Chmielewski. **Fault Injection as an Oscilloscope: Fault Correlation Analysis.** *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1). [Appears in Chapter 2.]

Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. **RIDL: Rogue In-flight Data Load.** *40th IEEE Symposium on Security and Privacy*, May 2019. [Appears in Chapter 3.]

Hany Ragab[1], Alyssa Milburn[1], Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. **CrossTalk: Speculative data leaks across cores are real.** *42nd IEEE Symposium on Security and Privacy*, May 2021. [Appears in Chapter 4.]

Alyssa Milburn, Erik van der Kouwe, and Cristiano Giuffrida. **Mitigating Information Leakage Vulnerabilities with Type-based Data Isolation.** *43rd IEEE Symposium on Security and Privacy*, May 2022. [Appears in Chapter 5.]

[1]Equal contribution shared first authors.

Related publications not included in this dissertation:

Alyssa Milburn, Herbert Bos, and Cristiano Giuffrida. **SafeInit: Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities.** *Network and Distributed System Security Symposium*, February 2017.

Alyssa Milburn, Ke Sun, and Henrique Kawakami. **You Cannot Always Win the Race: Analyzing mitigations for branch target prediction attacks.** *8th IEEE European Symposium on Security and Privacy*, July 2023.

Alexander J. Gaidis, Joao Moreira, Ke Sun, Alyssa Milburn, Vaggelis Atlidakis and Vasileios P. Kemerlis. **FineIBT: Fine-grain Control-flow Enforcement with Indirect Branch Tracking.** *26th International Symposium on Research in Attacks, Intrusions and Defenses*, October 2023.

1 | Introduction

The confidentiality of data in computer systems is becoming more and more relevant, not only due to the ever-increasing presence of computing devices in everyone's lives, but also due to the increasing use of shared resources – in particular, in the 'cloud'. Information disclosure vulnerabilities threaten this confidentiality on multiple fronts: they can allow attackers to directly obtain confidential data, but they also present a risk at a systems level, where they are often a vital component of exploit chains which are used to exploit other vulnerabilities.

A variety of modern defenses have been proposed and deployed against traditional attacks. These provide differing layers of mitigation against information disclosure vulnerabilities. Although they do not prevent all information disclosure vulnerabilities, they can significantly raise the bar and make it more difficult for attackers to exploit the remaining attack surface.

For example, modern systems make extensive use of process isolation to limit the potential exposure of information between security domains, and high-risk software such as web browsers runs inside sandboxes with limited access to system resources. Similarly, typical cloud environments use virtualization to isolate different instances of operating system kernels, which allows physical machines to be securely shared between untrusted customers. Sometimes even specialized hardware support – such as secure elements, or TPMs – is also used to protect particularly sensitive data, in particular cryptographic keys.

In an ideal world, such defenses could effectively mitigate at least all cross-domain disclosure attacks – reducing real-world threats to only logical bugs in software. Unfortunately, they share a common flaw; they make unsound assumptions about the *implementations* of the computer systems they run on. In practice, so-called 'side-channels' – unintended information flows which stem from the implementations of computer systems rather than the software running on them – allow attackers to leak information even in the absence of traditional software vulnerabilities.

The unintended nature of such flows makes them difficult to even anticipate –

new side-channels can be seen as new 'surprise threat models' – and mitigations against new side-channels are necessarily reactionary.

Furthermore, these side-channels come in a wide range of different forms, and with a variety of potential consequences. Traditionally, research in this area has focused on *physical* side-channels, such as electromagnetic emissions (TEMPEST [239]), or the relationship between the data being processed by code running on a processor and the power usage of a processor, which provides power side-channels [37, 130, 131]. Such side-channels are only available to an attacker in physical proximity to a target, but they can still be a serious security threat in many situations. For example, power analysis can be used to obtain cryptographic keys from secure devices such as smart cards, even when hardware implementations are used to ensure that even the code running on these devices does not have access to these keys. In practice, physical promixity is not a requirement; for example, recent research [148] shows that software-accessible power monitors allow code running on a device to perform traditional power side-channel attacks.

However, there are also many side-channels which do not depend on the specifics of the physical implementation of a processor, but rely solely upon unintended consequences of processor designs which are unanticipated by their users. Timing side-channels [129] form the basis for many such attacks; any data-dependant timing difference allows an attacker to observe differences in execution time and thus reveals information about the data being processed. For example, branches in cryptographic code can reveal information about the private key being used, allowing compromise of the key.

Cache side-channels are an important application of such timing attacks; by timing memory accesses, an attacker can determine whether a specific region of memory is present in the processor cache, revealing information about whether that memory has recently been accessed [115, 120, 150, 179]. The resulting channel provides information about memory access patterns which can compromise modern defenses such as ASLR [82], as well as revealing cryptographic keys and other sensitive information. Since these processor caches are often shared between *all* security domains on a given system, such attacks can even compromise virtualization-based isolation.

Recent cryptographic implementations attempt to mitigate timing side-channels, and specifically cache side-channels, by using 'constant-time' implementations where the executed code and accessed memory do not depend on confidential information such as private key contents. However, this is difficult to achieve in practice, and any code which has not specifically been rewritten in this way remains vulnerable to attack.

To make matters worse, side-channels can even be used to reveal the effects of code which has not been executed. Attacks such as Spectre [128] abuse the speculative execution of modern processors by attacking functionality such as branch pre-

dictors to cause processors to speculatively execute code on incorrect flow paths. The effects of such transiently executed code are not architecturally visible, since the incorrectly-predicted code will be flushed from the pipeline before any changes are committed. However, side-channels can be used to observe any differences in processor state caused by such transient execution; in particular, memory accesses by such code can influence the contents of the processor cache, allowing the use of cache side-channels.

FLUSH+RELOAD [256] provides an example of how a cache side-channel can be easily used by a transient execution attack. An attacker flushes [87] a number of cache lines (removing them from the cache), runs some code (potentially transiently) which loads a cache line corresponding to the value of some 'secret' data, and then measures the time required to read data in each cache line. The time required to access the cache line corresponding to the secret data will be significantly less, indirectly revealing the data to an attacker. The resulting attacks had serious consequences such as allowing attackers to read arbitrary memory contents by speculatively executing incorrect code in kernels or hypervisors, and mitigating them remains a work in progress [232, 251].

This dissertation further explores the attack surface *beyond* these traditional classes of attacks (and mitigations), performing an end-to-end analysis of the side-channel attack surface spanning all the way from physical hardware attacks to Spectre-style software-based attacks. We challenge the assumptions of previous work, investigating whether we can apply power analysis attacks without directly measuring power, or whether we can use cache side-channels to expose secrets even when software has been carefully written to avoid any secret-dependant memory accesses – and even when software is isolated on separate cores. This allows us to find new avenues for investigating both hardware and software side channel attacks, and develop new attacks rooted in both categories.

It is clear that these classes of attacks are becoming more and more varied, and defending against them requires a wide range of considerations. Many of these attacks can be efficiently mitigated at hardware level, but others appear better suited to software defenses – such as Spectre variant 1, which at heart relies only on conditional branch prediction. We suspect that the best future approach may be for software defenses to consider these side-channel attacks in their threat models. As a concrete example, we propose a high-performance software defense based on type-based isolation, and show that by including Spectre v1 in the threat model, it can also efficiently mitigate many of these attacks.

1.1 Constructing power side-channels

Monitoring the power usage of a processor provides information about both the operations being performed and the data being processed. Naively, this is because

the dynamic power usage of a circuit is dominated by the cost of switching, i.e., bits flipping from 0 to 1 or vice-versa. Simple Power Analysis (SPA) takes advantage of this power side-channel by directly relating the power usage to the operations or data used; for example, simple implementations of RSA [205] cryptography can be attacked by observing the different power usage of multiplication and squaring operations, directly revealing the bits of the private key being used. More sophisticated attacks have been developed where this kind of direct relationship does not provide enough information, or where the power signal is too noisy. Countermeasures against such attacks include masking [36, 162], where secrets which are split into multiple *shares* and cryptographic implementations perform computation on individual shares.

Differential Power Analysis (DPA) [131] attacks cryptographic algorithms by performing statistical analysis of the power usage when an encryption/decryption algorithm is provided with different inputs. For example, many implementations of the AES [50] cryptographic algorithm can be attacked by guessing individual bytes of the private key. For each guess, we group the collected (power) traces into two categories, based on whether a specific bit of a XOR operation is 1 or 0 (which depends on both the data, and the key). When we average these two groups and compare them, only *correct* key guesses will have a significant difference in power usage between the two averaged traces, allowing an attacker to again determine the private key, even in the presence of noise. Many improvements upon such attacks exist, such as Correlation Power Analysis (CPA) [27] which uses Pearson's correlation coefficient to determine the most likely key.

However, these attacks all require direct measurement of the power usage of a processor. Another class of physical attack consists of Fault Attacks [21]; for example, voltage 'glitching', where we reduce the power supplied to the processor for short periods of time. This is typically used to disrupt processor activity at a specific point, allowing checks to be bypassed or specific data to be corrupted. However, this attack also allows us to *indirectly* collect data about the power usage of a processor; by performing glitching at a specific point in multiple identical executions, we can calculate the probability that the processor will fail to function correctly while deprived of power. We propose that this probability provides us with a proxy for the power usage at that point in time, and can then construct 'traces' by repeating this technique at different points in time. Is this constructed side-channel actually correlated with the power usage of the processor? In **Chapter 2**, we apply this approach and collect results on a variety of real hardware targets. We show that we can directly successfully apply existing crytographic attacks – such as those mentioned above – using our constructed traces to obtain private keys.

1.2 Stale data attacks

Recent work has shown that attackers can obtain information by observing the effects of transiently executed code – code which is executed by a processor pipeline, but which is not architecturally committed. Spectre-style attacks [128, 135, 156] take advantage of this by causing code to be speculatively executed on unintended paths. Since the effects of memory loads are not considered to be architecturally visible, they may still be executed during transient execution, allowing exposure of information via a cache side-channel.

A typical example involves manipulating branch prediction on the branch after a bounds check, resulting in transient execution of code which accesses an array out-of-bounds; if an attacker can influence the array index, and the results of the transiently executed code can be observed via a side-channel (such as cache timing), then this allows an attacker to read any memory which is available to the transiently executed code. Such code patterns are referred to as Spectre gadgets, and mitigation has focused on modifications to these gadgets – for example, by adding execution fences (preventing such gadgets from being transiently executed) or limiting the impact of transient execution (such as by masking indexes).

Transient execution can also occur in situations without any branches; in particular, modern processors often defer handling of exceptions until the commit stage, which means that code continues (transiently) to run *after an exception*. This is a particularly interesting attack vector for exceptions such as page faults, which occur when load instructions are executed. If an exception occurs during a load, the transiently loaded value – which can be used by the code which continues to transiently execute after the exception – is implementation-dependant, and depends on the specifics of the hardware design.

Meltdown [149] was the first attack based on this type of vulnerability, which took advantage of the fact that some CPUs – including those from Intel, ARM and IBM – allow transient loads to access kernel memory; an attacker can simply read from kernel addresses and use a cache side-channel to expose the contents of such a read. Soon after, Foreshadow [28, 249] demonstrated that a similar attack could read arbitrary L1 cache contents on Intel CPUs by accessing memory via invalid page table entries, even inside a virtual machine.

Note that such "domain-bypass" attacks do away with any need to find a gadget in a target's code, and thus cannot be mitigated using the methods used to defend against Spectre. As such, we consider implementations which allow transiently executed code to access data from other security domains to be (unanticipated) CPU vulnerabilities, and varied mitigations for different forms of such vulnerabilities are available [92, 99, 108, 111, 136, 220] for CPUs without hardware mitigations.

Chapter 3 presents our analysis of such vulnerabilities and the relevant mitigations in Intel CPUs, including the discovery of new vulnerabilities. We show

that some exceptions allow the use of stale pipeline data by later transiently executed code, exposing attack surfaces beyond the processor cache. This data may be from code running on another thread of the same processor core, or even from previously executed code on the same thread, including other security domains such as the kernel or hypervisor. In particular, we demonstrate techniques which allow data to be leaked despite the relatively uncontrolled nature of such data, and present several proof-of-concept attacks using our work. We also demonstrate that hardware mitigations against previous attacks (such as Meltdown and Foreshadow) only prevent a subset of these new attacks, demonstrating that such vulnerabilities need more fundamental fixes.

1.3 Cross-core stale data leaks

Isolating different security domains on different processor cores – for example, ensuring that different virtualized guests do not run on the same threads of a single core – has become a standard mitigation against many of the speculative execution attacks discussed above, including those from Chapter 3. By flushing shared resources such as buffers and caches when switching between domains, any sensitive information about other domains can be removed from the processor pipeline. Recent work has also taken this to an extreme by fully isolating domains on different cores [94].

Although some cache attacks remain a threat, due to whole-system last-level caches, this appeared to otherwise be an effective mitigation for transient execution vulnerabilities. However, our incomplete knowledge about how modern processor pipelines are implemented – typically closely-held proprietary information for processor vendors – means that we may be unaware of other shared resources, which may also provide side-channels or even have direct vulnerabilities which could be exploited by attackers.

Chapter 4 explores this potential attack surface by investigating the implementation details of complex CPU instructions, again focusing on Intel CPUs. We design a tool which combines information from performance counters and the contents of the internal memory accesses made by the processor pipeline, which can be observed using the vulnerabilities from Chapter 3. This analysis reveals a cross-core data channel which is used to access shared resources such as the hardware random number generator. To demonstrate the significance of such undocumented channels, we present an attack which can reliably expose random numbers generated by other cores, defeating core-based isolation and allowing compromise of cryptographic keys from Intel's SGX enclaves, even in the presence of all vulnerability mitigations.

1.4 Longer-term attack surface

Many of these issues are best mitigated in hardware, in particular for physical attacks and where transient execution can be used to expose 'stale' information across security boundaries (such as from kernels or hypervisors). In the meantime, such vulnerabilities can be difficult to mitigate and fix, resulting in long coordinated disclosure timelines while fixes are developed together with the (many) affected parties. For example, some of the vulnerabilities found in Chapter 3 and Chapter 4 were subject to lengthy embargo periods (often more than a year) to allow sufficient time for hardware and software mitigations to be developed and tested, and these mitigations are often high-impact (e.g., disabling use of SMT).

However, mitigating the remaining (Spectre) attack surface in hardware appears to be – at least in the medium term – unrealistic. In most environments, we are unwilling to sacrifice performance-critical features such as branch predictors, processor caches, and allowing loads in transiently executed code. As such, the mitigation of Spectre gadgets remains an ongoing challenge, years after the publication of the original research. The burden for intra-domain mitigations, in particular, continues to depend mostly or entirely on software.

At the same time, traditional memory error vulnerabilities remain a significant problem. An ideal solution would be to expand memory error solutions to consider both normal and transient execution paths; data flow analysis would then also consider potential transient flows, and defenses would use such analysis to mitigate vulnerabilities in both the architectural and speculative domains. **Chapter 5** proposes exactly such a solution, designing and evaluating a compiler-based memory error defense which prevents cross-type memory disclosure vulnerabilities while also mitigating Spectre-BCB (bounds check bypass) gadgets.

1.5 Contributions and roadmap

In this dissertation the goal is to increase our understanding of the ever-increasing attack surface exposed by recent work in side-channels. Our end-to-end analysis includes investigating potential new types of physical attacks, developing new forms of transient execution attacks which were not considered by mitigations, and exploring how to efficiently design software defenses when we include side-channel attacks within the threat models of such defenses. Specifically, we make the following contributions:

In **Chapter 2** we investigate hardware attacks by considering power side-channels, and show that the results of voltage fault injection *also* expose information about the power usage of many microprocessors, allowing existing power analysis attacks to be applied to results of a fault injection campaign – without the need to directly measure power. We explore several such attacks and

perform them against multiple real hardware targets, and demonstrate that we can use them to successfully attack both software implementations of cryptography as well as hardware cryptographic engines.

In **Chapter 3** we move to investigating software-based attacks. By analyzing transient execution vulnerabilities, we discover new vulnerabilities which allow us to leak data from internal CPU pipeline state without the need for such data to be present in the cache. We demonstrate cross-domain attacks which allow information to be leaked from other VMs and SGX enclaves, even on CPUs with hardware mitigations against existing attacks, and present an analysis of recent attacks and mitigations.

In **Chapter 4** we consider whether isolation-based approaches can mitigate these attacks. We broaden our knowledge of transient execution vulnerabilities by using performance counters to investigate the memory behavior of instructions. In particular, by analyzing instructions making unexpected off-core memory accesses, we can develop transient execution attacks that leak sensitive data used by other cores, again defeating existing mitigations which attempt to isolate security domains. For example, we can derive private keys by leaking random numbers used by other cores while doing cryptographic operations.

In **Chapter 5** we turn our attention to Spectre – which, although rooted in hardware behavior, provides a residual attack surface which must be mitigated by software. We propose an arena-based framework which uses compiler instrumentation and a customized allocator to provide fine-grained (type-based) data isolation. We reduce performance overhead to acceptable levels by using compile-time analysis made possible by our allocator design, allowing our defense to mitigate both traditional and speculative information disclosure vulnerabilities.

Finally, we conclude in **Chapter 6** by looking at lessons learned from investigating these new attacks and defenses, and consider promising directions for future work.

2 Fault Injection as an Oscilloscope: Fault Correlation Analysis

Fault Injection (FI) attacks have become a practical threat to modern cryptographic implementations. Such attacks have recently focused more on exploitation of implementation-centric and device-specific properties of the faults. In this chapter, we consider the parallel between SCA attacks and FI attacks; specifically, that many FI attacks rely on the data-dependency of activation and propagation of a fault, and SCA attacks similarly rely on data-dependent power usage. In fact, these are so closely related that we show that existing SCA attacks can be directly applied in a purely FI setting, by translating power FI results to generate FI 'probability traces' as an analogue of power traces. We impose only the requirements of the equivalent SCA attack (e.g., knowledge of the input plaintext for CPA on the first round), along with a way to observe the status of the target (whether or not it has failed and been "muted" after a fault). We also analyse existing attacks such as Fault Template Analysis in the light of this parallel, and discuss the limitations of our methodology.

To demonstrate that our attacks are practical, we first show that SPA can be used to recover RSA private exponents using FI attacks. Subsequently, we show the generic nature of our attacks by performing DPA on AES after applying FI attacks to several different targets (with AVR, 32-bit ARM and RISC-V CPUs), using different software on each target, and do so with a low-cost (<$50) power fault injection setup. We call this technique Fault Correlation Analysis (FCA), since we perform CPA on fault probability traces. To show that this technique is not limited to software, we also present FCA results against the hardware AES engine supported by one of our targets. Our results show that even without access to the ciphertext (e.g., where an FI redundancy countermeasure is in place, or where ciphertext is simply not exposed to an attacker in any circumstance) and in the presence of light jitter, FCA attacks can successfully recover keys on each of these targets.

2.1 Introduction

The physical security of efficient cryptographic implementations – in particular, protection against side-channel analysis and fault injection attacks – has become more challenging than merely preventing attacks on the security of the algorithms themselves. Many modern embedded devices are equipped with cryptographic hardware cores, and the nature of such devices means that they are physically accessible to the adversary. This means that low-cost practical physical attacks on such targets are becoming increasingly relevant. Such attacks have become a real concern for some parties due to the dramatic increase in the usage of embedded devices for mobile applications (such as banking), IoT, and home entertainment.

Side-Channel Attacks (SCA) [37, 130, 131] and Fault Attacks (FA) [21] are the most widely explored classes of such physical attacks. SCA involves the passive exploitation of the relationship between leakage – such as power usage or electromagnetic (EM) signals – and the operations (Simple Power Analysis, or SPA) or data (Differential Power Analysis, or DPA) [131] performed on a device, typically to recover a private key being used in a computation. Brier et al. [27] introduced an improved DPA-like method called Correlation Power Analysis (CPA), which continues to be a serious threat to the confidentiality of cryptographic keys in modern devices. Another important SCA technique is the use of Template Attacks [37] (TA), which enhance attacks such as SPA and CPA with the use of statistical modeling. This is performed in two stages: first profiling, and then matching. An attacker uses a device under their control to create a 'profile' of a sensitive device, a process which may take a substantial amount of time/effort. They can then 'match' this profile to measurements taken from a victim's device, to quickly perform cryptographic attacks with minimal effort. We refer to such attacks as profiled, to distinguish them from non-profiled attacks such as standard SPA/CPA.

The other major class of physical attacks – fault attacks – are active in nature. An attacker injects faults – which we will refer to as Fault Injection (FI) – in an attempt to actively influence or control the state of the target device. For example, when applied to cryptographic computations, an attacker may want to corrupt intermediate state which involves the secret key. If faults can be injected which cause bias in such intermediate state, it may be possible to use either analysis of the algorithm or statistical analysis to reduce the entropy of this key [21]. One standard fault attack is called Differential Fault Analysis (DFA) [24], which allows recovery of private keys by analyzing errors induced in the last rounds of a block cipher, such as AES [13, 78].

Developers of modern embedded devices often implement countermeasures to protect the secret keys used in sensitive cryptographic computations, especially in the context of devices which are physically accessible to adversaries. In the context of FI, detection-type countermeasures are the most common ones, which

use some kind of redundancy (e.g., in time or space) to detect faults, and then react by suppressing or randomizing the output [88, 139]. One alternative is to perform the computation in a way which results in random output in the event of an error, rather than explicitly detecting faults; this is known as an infective countermeasure [231].

Research has shown that there is a relationship between SCA and some FI attacks. Fault Sensitivity Analysis (FSA) [146, 168] proposes varying the intensity of applied faults to detect the threshold at which an observable error occurs. Specifically, FSA focuses on clock glitching, reducing the clock period until a successful setup-time violation occurs. The authors propose that the obtained fault sensitivity (FS) at that point provides information about the critical path delay (CPD), and that since the CPD is data-sensitive, it reveals information about the data being processed at the time of the glitch. Although FSA is an active FI-based attack, in many ways it is closer to passive side-channel attacks such as power analysis, rather than existing fault attacks such as DFA. For example, the leakage model used in FSA depends on the specific implementation of cryptographic components such as the S-box in AES; this is similar to the way SCA attacks depend on such implementation-specific leakage modelling, as opposed to attacks such as DFA which attack the underlying algorithm.

More recent research [74, 145] explores the relationship between fault sensitivity (FS) and power consumption. Based on FPGA experiments, the paper [145] confirms a high correlation between the power consumption and the FS, and even presents a successful key recovery using the FS profile as the leakage model for power consumption.

2.1.1 Contributions

In this chapter, we further investigate this relationship. We argue that existing attacks based on classic side-channels (such as SPA or CPA) are directly applicable to fault injection attacks, by using the observed distribution of successful faults at a certain time as a proxy of the power usage of the target at that time. Such attacks can be performed without access to ciphertext, and even without knowledge about whether or not encryption was successful. Specifically, we present a method for turning Fault Injection results into 'probability traces', based on the ratio of observed faults from an FI attack.

We demonstrate that standard side-channel attacks can be directly applied to the resulting traces, using real-world targets rather than simulations. As a basic example, we show that SPA can be performed on results of an FI attack against a software implementation of RSA running on real hardware. We refer to this technique as Simple Fault Analysis (SFA). Subsequently, to demonstrate the strength of our attacks, we present Fault Correlation Analysis, which uses a standard CPA

attack to successfully recover keys from several different targets running different software implementations of AES. We show that our approach can also use other distinguishers, such as non-profiled Linear Regression Analysis (LRA) [152], or even 'model-less' attacks such as 'correlation enhanced collision attacks' [167].

We show that FI probability traces can be generated even with restricted feedback from the target, focusing on two different classification methods: one based on successes, which requires the ability to observe whether a cryptographic operation was successful, and another based on "mutes", which requires only the ability to observe whether or not the device has failed and no longer responds. We successfully recover private keys on the majority of our targets using both of these categorizations. Note that existing ciphertext-based fault detection methods are not easily replaced using mutes, due to causes of mutes beyond data/computation-dependent faults.

We illustrate that by using voltage FI, we can perform our experiments with inexpensive equipment; our experiments use a low-end FPGA as a glitching device, at a cost of less than $50. We require only access to a trigger or an alternative synchronization mechanism; we do not need foreknowledge of the key or even details such as the target's clock speed, nor do we require access to power traces.

We allow FI attacks to be performed in an unsupervised fashion by first collecting raw results during the FI campaign and later performing post-processing when creating traces (by 'bucketing' the results). We observe that our approach requires a relatively large number of faults. However, our experiments treat targets as black boxes in many ways. In particular, our attacks do not need spatial accuracy, nor knowledge of the specific implementation of the target.

Furthermore, we do not require control of the clock input, which allows us to leave clock conditioning hardware such as Phase-locked Loops (PLL) enabled on our targets; we run our targets at standard clock speeds. Our experiments using a UART-based trigger add further jitter. We also use a wide range of voltage FI parameters, and perform FI campaigns over a lengthy time period (in particular, our FCA attacks against software cover at least the entire first AES round).

Many previous FI papers assume a high level of attacker control, where glitches are injected at a precise (known) clock cycle, using a precisely-chosen narrow ranges of parameters. Although we have not performed experiments in such controlled settings, we show that more specific timing significantly reduces the number of required faults.

Finally, we show that FCA can be used to recover several key bytes from the hardware AES engine on one of our targets, despite our use of FI attack parameters aimed to cause glitches in specific software instructions. We argue that the exposed leakage is present only when those instructions are executed, implying the hardware engine leakage is encoded in our software FI results.

2.1.2 Related work

There are many existing FI attacks on cryptographic algorithms that require access to the output ciphertexts. As discussed, DFA [24, 78] is the most famous of such attacks, combining observations of observed output errors along with knowledge of the cause of the error. Over the more than 20 years since the development of DFA, it has been applied to various cryptographic primitives using many different types of FI.

When an attacker does not have access to (faulty) output ciphertext, either due to countermeasures or due to the protocol design, such attacks can no longer be performed. The information available to an attacker is then reduced to observing whether or not ciphertext is returned, allowing them to determine only whether a fault was detected.

Some recent attacks, such as Fault Sensitivity Analysis (FSA), still work in this setting. As discussed, FSA increases the 'fault intensity' (for example, shortening the length of a clock glitch) until a difference is observed (e.g., incorrect output). An attacker can then make use of a fault model (e.g., HW of S-box input) to perform an attack without the need for faulty ciphertexts. An extension called collision FSA [168] uses the fault model combined with correlation enhanced collision side-channel attacks. This combination removes the need to define a precise fault model and can break the masking countermeasure. However, both FSA and collision FSA require the attacker to be able to target the exact clock cycle used for the operation of interest, and rely on control of the target's clock to perform clock glitching. FCA does not have such requirements, and allows the fault parameters to vary over fairly large ranges rather than using a fixed fault intensity.

Differential fault intensity analysis (DFIA) [76] employs both fault intensity and faulty values as the observables for statistical analysis. It calculates changes in 'error values' of faulty ciphertexts, on the basis that a slight change in the intensity only results in a small change in the error under a correct key assumption. However, the DFIA attack can be defeated by detection-based FI countermeasures, since it requires access to ciphertext. Both FSA and DFIA techniques require that plaintext input can be repeatedly encrypted using different intensity values, which is a limitation for attacks on nonce-based ciphers.

Blind Fault Analysis (BFA) [76] proposes an FI attack which does not require direct access to the ciphertext and even the plaintext. The secret key is found by recovering the Hamming weight of the intermediate states of two sequential rounds, which is possible if the attacker can learn the number of different results of tampered encryptions performed for the same unknown plaintext and that is only possible assuming a particular fault model. Namely, multiple bit-reset (or bit-set) needs to be possible. This attack is not effective against detection-based FI countermeasures, and the authors do not perform an experimental evaluation.

Statistical Ineffective Fault Analysis (SIFA) [57, 58] changed the widely regarded belief that fault attacks require access to faulty ciphertexts. If encryption is successful after a fault has been injected, then the fault had no effect on the computation – it was an 'ineffective fault' – and so an attacker can conclude that the intermediate value must have remained unchanged (e.g., be zero). SIFA shows that if an adversary has the power to inject a sufficient number of precise faults, the ability to observe these ineffective faults (which result in correct ciphertexts) can be exploited for an attack. However, the described SIFA attack requires brute-forcing 32-bit key chunks, which involves significant effort. SIFA is an improvement and combination of two older attacks which rely on access to the output ciphertext: the Statistical Fault Attack [70] and the Ineffective Fault Attack [42]. In a slightly different setting, Persistent Fault Analysis (PFA) [186, 260] also presents an attack which is similar to SIFA. The Safe-Error-Attack (SEA) [223] is similar to SFA, but attacking public key cryptography rather than a cipher using symmetric keys.

Dobraunig et al. [56] show that SIFA can also be used to defeat the masking countermeasure, using clock glitching to present a practical evaluation which requires only a single fault per execution. They also argue that the ability to inject precise, reliable faults is not required, a theme we further explore in our work.

Fault Intensity Map Analysis (FIMA) [198] builds on and generalizes DFIA and biased-based techniques, such as SIFA. It combines the biased distribution of correct ciphertexts under a correct key hypothesis with the variation of data distribution with fault intensity to reduce the number of faults required to recover a secret key. FIMA with neural network key distinguisher (FIMA-NN) [197, 198] builds upon this, using a neural network to rank key candidates. FIMA has similar disadvantages to SIFA: it requires the ciphertext, and requires guessing 32-bit chunks of the key.

Most recently, Saha et al. [208] introduced Fault Template Attacks (FTA), which also exploit the fact that the activation and propagation of a fault is data-dependent. These attacks can be applied even in the presence of countermeasures against FI and SCA; in particular, when masking is present. FTA is applicable even if the fault injection is made at the middle rounds of a block cipher, and it also works in a known-plaintext scenario. Their attack is performed using electromagnetic FI (EM-FI), and a practical evaluation is performed only on PRESENT. It relies on both reproducible attacks at specific points (temporal accuracy) and the ability to inject faults at multiple fault locations (spatial accuracy). Their analysis of AES is performed on a simulation, presumably due to the relatively high complexity: a successful attack would require EM-FI to be performed using at least 15 different locations. Furthermore, it is a profiled approach, and portability of the resulting template from one device to another was not discussed.

We believe that FTA benefits from the localization effect similar to the localized SCA Template Attack. Essentially, both types of attacks collect signals cor-

responding to sensitive intermediate variables. After unmasked intermediates are recovered they can be combined to obtain the secret key. Therefore, we suggest that FTA corresponds to a localized EM SCA Template Attack, similarly to how our attacks (SFA and CFA) correspond to SPA and CPA.

Table 2.1 presents a high-level comparison of many of these attacks, including our own.

Table 2.1: Comparison of various Fault Injection Attacks.

Attacked Algorithm	Main Target	Profiled	FI det.*	FI Type	Comment
DFA [24, 78] + more	Various	No	No	Various	Requires ciphertext. Last rounds attack.
SAE [223]	Sim.: RSA, ElGamal	No	Yes	N/A	Requires ciphertext correctness info. Any round attack. Can attack symmetric.
FSA [74, 145, 146]	Plain AES (FPGA)	No	Yes	clock FI	Requires ciphertext correctness info. Last rounds.
Collision FSA [168]	Masked AES (FPGA)	No	Yes	clock FI	Requires ciphertext correctness info and timing info of the S-Boxes (SCA). Last rounds.
DFIA [75, 76]	Plain AES (FPGA), sim.: PRESENT, LED	No	No	clock FI	Requires Ciphertext. Last rounds.
BFA [76]	Sim.: SPNs, inc. AES	No	No	N/A	Requires ciphertext correctness info. Any round attack.
SIFA [56, 58, 76]	protected AES†, Keyak and Ketje (FPGA), masked AES, HW AES engine (ATXmega)	No	Yes	clock FI	Requires ciphertext, and guessing 32-bits of the last round key. Last rounds.
PFA [186, 260]	DMR AES (FPGA, cache), sim.: masked AES	No	Yes	clock FI	Requires ciphertext. Last rounds.
FIMA, FIMA-NN [197, 198]	sim.: ASCON, AES	No	Yes	N/A	Like SIFA, so it has the potential to break masking, but also guesses 32-bits of the key.
FTA [76]	PRESENT, sim.: AES	Yes	Yes	EM-FI	Requires ciphertext correctness information.
Attacks from this work (part.: SFA and FCA)	Plain AES and RSA (SW), HW AES engine (ATXmega)	No	Yes	voltage FI	No ciphertext needed, mute detection is sufficient. Collision FCA might work against masking but it is not confirmed.

∗ 'FI det.' denotes the detection type of FI countermeasure;
† denotes AES protected with the infective countermeasure [231].

2.1.3 Structure of the paper

Section 2.2 introduces some background material and concepts required to understand the fault injection technique described in this chapter. Subsequently, we describe our attack method in detail in Section 2.3. Section 2.4 describes the details

of our experimental setup. In Section 2.5, we support our claims by presenting and evaluating results from practical experiments. Finally, we provide a discussion of some strengths and weaknesses of our approach in Section 2.6, together with some potential future directions.

2.2 Prerequisites

Our proposed attacks have several prerequisites, which we will discuss in this section:

First, an attacker must be able to inject faults into the target – they must be able to perform Fault Injection. In this chapter, we only consider voltage FI; we briefly discuss alternatives below.

Second, these faults must be injected at a time relative to a known event (a "trigger"). A certain amount of jitter can be tolerated, generally at the cost of requiring more attempts. However, an attacker does not require knowledge of precisely which operations are performed at which point in time.

Finally, we need a way to classify the outcome of a FI attempt – we must be able to distinguish attempts which do not cause changes in behaviour from those that do.

Since we build on SCA attacks, specific SCA attacks may impose additional requirements. For instance, they may require access to input or output of the cryptographic operation (plaintext or ciphertext), or they may require chosen input.

2.2.1 Fault Injection

Fault Injection, sometimes referred to as glitching, is the act of introducing glitches in the operating environment of a target with the intent to introduce faults. These glitches can be introduced by various means, such as by controlling the voltage supply of a target, with electromagnetic pulses (EM-FI), controlling the clock of the target, or using a laser.

For effective and repeatable attacks, the attacker requires control over not just the glitch, but also over which operation is attacked. Precisely targeting an operation requires the target and the attacking device to be synchronised, so that the attacker knows when to inject the glitch into the target. This synchronization is achieved by "triggering" the glitching device (much like an oscilloscope) at a specific point in time; the actual glitch injection is typically performed after a further delay, injecting the glitch at a specific time after the trigger.

During experiments, this trigger can be implemented by using an I/O pin on the target, which software running on the target sets high before starting the operation of interest. This allows synchronisation to be achieved up to the resolution of the update speed of the I/O pin logic on the target. Such convenient

situations may, however, be impossible to achieve in a real-world environment. One alternative is instead to trigger based on other attacker-observable behavior, such as communication protocols or power traces; some of our experiments trigger on the serial communication with the device. In both cases timing is not entirely predictable; in our attacks, we consider this as a source of measurement noise.

2.2.2 Countermeasures

Many countermeasures have been proposed to protect cryptographic keys against FI attacks. One of the most common classes is detection-type countermeasures, which detect a fault via some redundant computation (time, space, or information redundancy), and then react by either muting or randomizing the output [88, 122, 139, 189].

Another related class is infective countermeasures [188, 231], which avoid the need for explicit detection redundancy by performing the cryptographic computation in such a way that the output becomes deliberately randomized when errors occur.

Other countermeasures can be useful for mitigating both SCA and FI attacks, such as random delays [43, 230]. Some countermeasures intended for SCA can also mitigate FI attacks; examples include shuffling the execution order of independent operations [93, 204], and masking [116, 162, 173].

Finally, as a general defense against FI, some high-risk targets may trade reliability for security, disabling themselves (or access to their keys) if they detect that an FI attack may be being performed.

2.2.3 Fault detection

We propose that the probability of a fault is dependent on the data being processed by a device, as argued by previous work [146], as well as the operation being performed. This difference in fault behaviour is what we exploit in this chapter. Specifically, we hypothesize that the probability of a fault is correlated with the power consumed by the specific operations and data being processed. When considering software implementations, the classic side-channel leakage model is that this is correlated with the Hamming weight of the data being processed.

We want to categorize the results of injected glitches based on whether or not the injected glitch caused a fault in the target. However, an attacker cannot directly observe whether or not a fault has occurred; to estimate the probability that a fault has occurred, we must rely on the observable results.

Specifically, an attacker must be able to guess whether or not a fault has occurred based on the observed behavior of a device. For this purpose, we group this observed behavior into three basic categories. If a device fails to respond, we categorize the result of the glitch as a “mute”; essentially, we assume the device

or implementation has failed. If a device returns with an incorrect response, we categorize the result as a "corruption". Otherwise, we consider the device to have been unaffected by the glitch. In practice, there are more complex cases (such as reboots, and other unexpected responses); we defer discussion of those details to later in our paper.

As a concrete example of such behavior, consider an attacker performing FI on software AES running on a microcontroller. If the microcontroller no longer responds after the glitch, we consider that a mute. If corrupted output is returned, we consider it a corruption. Otherwise, nothing special has happened, and we assume it was uninfluenced by the glitch.

An attacker's ability to clearly distinguish these categories depends on the specifics of the implementation of the target implementation. To accommodate this, we will use two different probability groupings in the remainder of our paper: success-probability and mute-probability.

In the **success**-probability case, we consider the probability of successful operation. This means that glitches resulting in either a corruption or a mute are classified as faults. Note that an attacker does not need to be able to distinguish between these cases, which means this categorization can be used with targets using redundancy countermeasures. This model is used in FSA and Differential Behaviour Analysis [206].

In the **mute**-probability case, we consider only the probability of mutes. This can be useful where an attacker is unable to distinguish corruptions from successful operations.

Note that neither of these cases requires access to the (corrupted) ciphertext. In situations where the same plaintext input can be repeatedly encrypted, access to the ciphertext provides us with additional information, which allows an attacker to choose the used model. These two categorizations are practical and apply as-is to a range of targets, with no need for specific knowledge of how an algorithm is implemented. We discuss how they are distinguished in practice in Section 2.4.7. For our experiments, we found that our parameter selection excluded potentially-ambiguous behaviors (in particular, reboots) this also allows us to rule out brownout detectors as the source of the leakage; see Section 2.4.6.

2.3 Constructing traces from faults

The intuitive description of our attack is simple; we turn observed faults from fault injection attempts into a probability for a fault at a given point in time, and by repeating this at different points in time, into 'probability traces'. We can see the practicality of this approach by comparing traces taken from a hardware target (a RISC-V CPU) running the first round of AES. Figure 2.1 shows fault injection results presented using a bar chart (with each bar representing the number of mutes

in the respective time period), while Figure 2.2 shows the actual power usage during the same period, as measured by an oscilloscope.

It is clear that the same distinctive pattern of power usage is visible in both traces. As we will show in Section 2.5.1, we can see that Simple Power Analysis attacks can be performed based solely on these FI results.

The effect of unsynchronized clocks can be clearly seen in Figure 2.1. Although our voltage fault injection device and the target are aligned by a trigger (on the left side of the chart), the potential misalignment between our real-world target's clock (driven by an independent clock source) and our hardware glitcher's clock increases as time passes (towards the right side of the chart). As with SCA, this misalignment increases the number of required attempts, but does not prevent our attacks from succeeding.

Figure 2.1: Bar chart of the mutes in the first round of RISC-V AES.

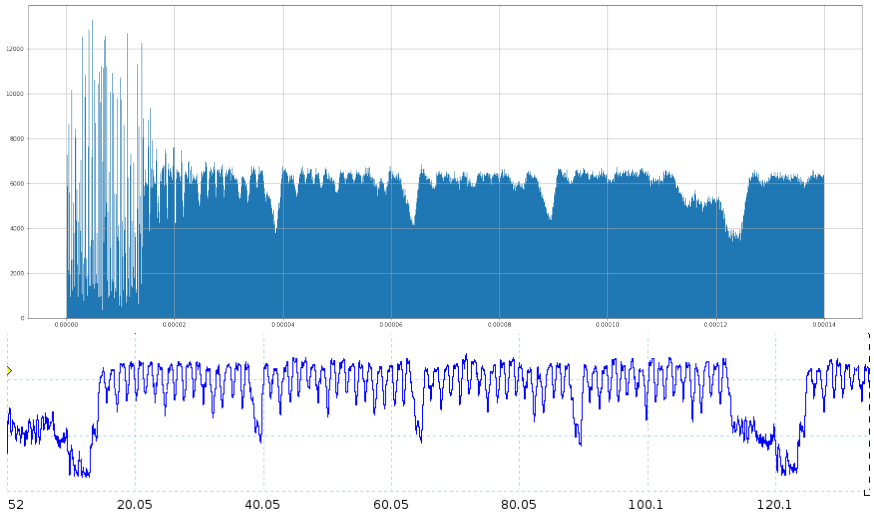


Figure 2.2: Oscilloscope power measurement of the first round of RISC-V AES.

We hypothesize that other attacks from the side-channel world, such as Correlation Power Analysis, Linear Regression Analysis and Power Collision Attacks [167], could also be used on such probability traces. In this work we show that this is indeed possible when attacking software running on a variety of hardware targets, focusing on performing CPA on such traces.

2.3.1 Data collection

We have, in effect, transformed a voltage fault injection device into a 1-bit sampling oscilloscope. Obtaining full power traces imposes the same restrictions as when

using an actual low-resolution sampling oscilloscope; we must be able to repeat the same computation multiple times. Specifically, for each identical computation (e.g., encryption of a specific plaintext with the same key), we must apply multiple faults at the same point in time, and then repeat that for different points in time.

In a real-world attack, the input data may not be fully controlled or repeatable. In such circumstances, the partitioning method could be applied [152], creating traces corresponding to specific input bytes rather than specific inputs.

No matter the method used to group input, by repeatedly applying faults at the same point in time relative to an operation and observing the ratio of their results, we obtain an estimate for the probability of a fault occurring at that point in the computation. We then perform this process at different points in time; the specific points (and the number of points) where data must be collected depends on the desired resolution and length of the probability trace. Similarly, the number of faults needed at a specific point depends on the desired precision of the fault probability.

We discuss the challenges of selecting appropriate parameters in Section 2.4.6.

2.3.2 Trace construction

Once we have obtained the fault injection results discussed in the previous section, the next step is to convert them into traces. For each input (whether partitioned or not), our results consist of a set of fault probabilities, one at each point in time. A naive conversion can be performed by simply converting each point in time to a single point on a trace.

Our FI attacks are designed to produce noisy results, and the resulting data is sparse on the time domain, since we do not attack specific known points in time, but distribute our attempts across a relatively large time period. We perform two post-processing steps to improve these results, after which they can be directly used as if they were standard power traces.

The first step involves 'bucketing' the results, which can be seen as analogous to oversampling when collecting actual power traces. We divide the attack period into 'buckets' of equal duration, and calculate the combined fault probability using all attempts which were performed during the time period corresponding to each bucket. The number of buckets defines the length of the resulting trace, much like the choice of sample rate when using an oscilloscope. Increasing the number of buckets reduces the number of faults which are placed in each bucket, which means that more faults are needed in total to achieve the same resolution.

The second step performs a simple low-pass transformation to remove aliasing due to leakage being spread across two adjacent buckets. In our experiments, we achieve this by inserting an additional sample between the samples from every pair of buckets, representing the combined probability for both buckets.

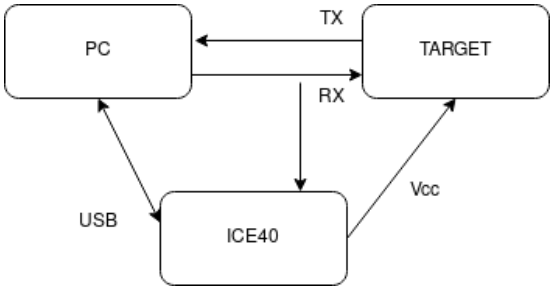


Figure 2.3: Schematic of the setup

Since these post-processing steps are applied after the FI results have been collected, they can be repeated to create improved traces without the need to repeat the experiment. Typically, we choose an arbitrary bucket size to generate an initial set of traces, and then adjust these post-processing parameters to improve correlation before performing an attack.

2.4 Practical Implementation

To applying the attack we have described in practice, we will use a voltage fault injection device attacking some hardware targets running varied software. In this section, we first describe such a device and other details of our experimental setup, and then describe details of the target hardware and software we will use to produce the results in Section 2.5.

2.4.1 Device

To inject glitches we use a custom built, inexpensive glitching device built around a cheap low-cost FPGA (the Lattice iCE40). We have made the PCB design and all source code (including FPGA bitstream and scripts) publicly available at `https://github.com/noopwafel/iceglitch`. The total cost of our setup (including FPGA, components, and all targets) is less than $50.

The device generates a glitch of arbitrary length at a given delay after a trigger input, by using an analog switch to change the output voltage connected to the target. For the purposes of this chapter, the device offers four parameters; the length of the glitch (with a resolution of less than 1ns), the delay between the trigger and the glitch (with a resolution of about 5ns), and the baseline and glitch voltage (both 9-bit values with a range between 0 and 3.3V).

2.4.2 Experiment layout

Figure 2.3 shows a schematic of a setup. The voltage supply to the target's microcontroller is provided by the glitching device. The target's UART is connected to a PC via USB to serial adapter. The trigger input to the glitching device is either an MCU GPIO pin or a MCU Rx line, the schematic shows the MCU Rx line scenario.

2.4.3 Target programs

The target MCUs run a program which receives commands over UART. For example, one command receives input which is then encrypted with AES, and then the output is returned. Receiving the output is not strictly necessary for the attack. However, collecting this data allows us to test different fault models.

The program sends a boot message after the device has been reset. This allows classifying glitches which cause resets and those that cause the target to mute or become unresponsive.

2.4.4 Fault synchronization

Our experiments use two methods to synchronize the target with our glitching device. The first trigger method requires the target to pull a GPIO pin high when starting the calculation (e.g., AES encryption). However, this is unrealistic in many settings; we also evaluate a second method, where we trigger using the last byte of serial input. Synchronizing on serial line in this way results in more jitter; note that contrary to SCA, this cannot be resolved in post-processing, since alignment is not possible.

2.4.5 Targets

We used three different hardware targets in our experiments; details can be seen in Table 2.2. We attempted to choose widely-available platforms to allow our work to be easily reproduced. The ARM Cortex-M3 target is an unmodified STM32 board typically sold as the "Black Pill". The RISC-V target is a Sipeed Longan Nano board. The AVR8 target is a small custom PCB containing only a Xmega32a4u chip, pins to expose I/O and power, and the programming interface; we use the internal oscillator rather than an external crystal. We removed decoupling capacitors from the targets to increase their susceptibility to voltage FI. No special power cuts were created; power was supplied using pre-existing power pins on all targets.

Details of the software we ran on these targets is provided in Table 2.3. On the RISC-V platform and the Cortex-M3, we run a simple implementation of AES written in C. On the Cortex-M3, we also evaluate an optimized implementation of AES written in assembly. [209] Finally, on the AVR8 platform, we use the AES/RSA code provided by AVR-Crypto-Lib.

Table 2.2: Overview of target hardware.

Architecture	Microcontroller	Clock speed
ARM Cortex-M3	STM32F103C8T6[221]	72MHz
RISC-V RV32IMAC	gd32vf103cb[71]	108MHz
AVR8	Xmega32a4u[15]	32MHz

Table 2.3: Hardware/software configurations for experiments.

Hardware	Software	Triggering method
ARM Cortex-M3	Simple AES[4]	UART(Rx) trigger
ARM Cortex-M3	SS2016 AES[209]	UART (Rx) trigger
RISC-V RV32IMAC	Simple AES[4]	GPIO trigger
AVR8	avr-cryptoLib AES[16]	GPIO trigger
AVR8	avr-cryptoLib RSA[16]	GPIO trigger

2.4.6 Parameter selection procedure

Before we can perform our attack, we need to identify two things: the time window in which to attack (corresponding to the delay between the trigger and the fault injection), and the parameters to use when injecting the fault.

Previous attacks have used a power trace collected using an oscilloscope to identify the relevant time window. For example, if the goal is to attack the first round of AES, this window would be the period between a trigger event and the second round of AES. Our attack instead uses a probability trace generated using FI, such as the one in Figure 2.1, which also allows us to identify the relevant window.

Once we have identified a window of interest, we perform a characterization step, injecting a large number of faults across the window using a broad range of parameters. Typically, we choose a voltage range from 0V up to the normal operating voltage of the target, and a glitch length ranging from 10ns up to hundreds of nanoseconds. This gives us an overview of the target's behaviour when subjected to FI. We then narrow down the parameters until we have an approximate mix of 50% mutes and 50% non-mutes (much like [168]), and have excluded regions of the parameter space which always produce the same result and thus have no variance. We also exclude regions which result in significant numbers of resets of the target, avoiding any potential interactions with brownout detection.

If the parameters are narrowed down too far, effects similar to clipping or attenuation occur. The probability of fault is dependent on the operation performed, not just the data. This dependency on the operation or time can again be seen in Figure 2.1. Since our FI campaigns can take a significant amount of time to complete, and we do not perform them in a controlled environment, we must also ensure

that any temperature variations will not cause such clipping artifacts.

Table 2.4: FI parameter ranges used in our evaluation.

Target	Delay (s)	Glitch voltage (V)	Length (ns)
ARM Simple AES	0.001 - 10e-05	1.5-2.35	170-180
ARM SS2016 AES	7.1e-05 - 8.1e-05	1.5-2.3	18-200
RISC-V Simple AES	0.0001 - 10e-05	1.4-2.1	50-100
AVR8 AES	1e-06 - 10e-06	1.83-2.02	550-700

For our evaluation, we chose a relatively wide range of parameters, with the exception of the ARM Simple AES target, where we limited the glitch length to a somewhat smaller window. When performing an actual attack, it would be possible to narrow these parameters much further. The specific parameters we used are shown in Table 2.4. We supply these numbers to provide the reader with a general idea of the ranges involved, but note that the delay (the time between the trigger and the glitch) depends on the trigger being used, and that the effect of glitches on the target can also depend heavily on the details of the glitching setup/hardware.

Once we have a suitable range of parameters, we can begin the actual FI campaign. When injecting faults, we randomly select parameters from these ranges using a uniform distribution. This reduces the biasing effect of, for instance, temperature or previous attempts.

2.4.7 Classification of outputs

As described in Section 2.2, the FI attempts need to be categorized. In our experiments, we extend the categorization to reflect the actual responses from our targets.

We receive these responses from our target and discern five classes of behaviour: normal, mute, reboot, corruption, and other. If the target no longer responds, we classify the fault as a 'mute'; if the target reboots (returns boot messages), then we classify it as a 'reboot'. If the expected behavior occurs (for example, the device returns correct results of an encryption being returned or a valid signature), we classify it as 'normal'; if an unexpected response is received (such as a response which has more or fewer bytes than expected), we classify it as 'other'. Finally, since the response contains the ciphertext and we can repeat inputs, we can easily determine the correct ciphertext for any given input, which allows us to classify responses with incorrect ciphertext as 'corruption'.

Since we know the keys used in our experiments, we can also simply use the key to determine the correct ciphertext. An attacker in the field would not need to know the ciphertext, only be able to observe that it was corrupted; if an attacker cannot do so, they are limited to the mute model.

We group the classes according to the desired model: for the success-probability model we group the correct AES outputs vs the rest. For the mute probability model we group the corrupted and normal outputs vs the rest. In this way we can compare the difference in model accuracy using the same datasets.

2.5 Results

2.5.1 Simple fault analysis on RSA

To demonstrate the generic nature of our attack, we will first demonstrate a proof-of-concept attack which shows that Simple Power Analysis (SPA) can be applied to FI probability traces. We use the AVR8 target from Table 2.2, and the RSA code from avr-crypto-lib. In our tested configuration, this uses Montgomery modular multiplication, without CRT. The relevant command decrypts a single fixed message with a (short) secret key, and then prints a message. Only the mute probability model is possible, since failed decryptions do not lead to different behavior.

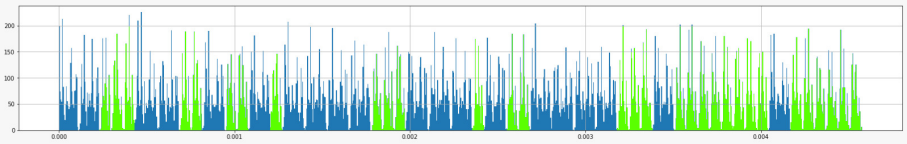


Figure 2.4: Results from glitching RSA, with individual key bits color-coded according to the operation (and thus the value of the key bit).

We performed an attack over a period of 3 hours, covering 5ms of time after a GPIO trigger (reducing jitter for this example), which is pulled high immediately before the modular exponentiation. During this period of time, the first 52 bits of the key are used. The results can be seen in Figure 2.4. The different patterns of the square and multiply operations are clearly visible, allowing the key bits to be read directly from the figure.

2.5.2 Fault correlation analysis on AES

We performed a long-running FI campaign on each of the targets from Table 2.3. For each attempt, we instructed the target to perform AES encryption, with input randomly selected from a set of 1000 plaintexts. During each attempt, we injected a glitch using (uniformly) randomly-selected parameters derived from our characterization results, as discussed above. We recorded the parameters, the observed input and the observed output in a database.

This experimental setup allowed us to explore different classification models, and their accuracy, based on the same data. By using our knowledge of the input,

output and key, we can classify different behaviours accurately. We can then choose which classification model to apply by selecting different groupings of classes. In this way, we can generate traces corresponding to success-model and mute-model, using results from only a single FI campaign.

Attacking the FCA traces is performed with a standard Hamming Weight CPA attack [27]. Specifically, we attack the S-box output during the first round, using JLsca's [118] implementation of CPA (`AesSboxAttack`).

Key rank evolution plots for all targets are shown in Figure 2.5. The X axis shows the number of attempts used, rather than the number of traces used as is customary when performing CPA. The total number of attempts includes all attempts: normal, mutes, corrupted, reboots and other. As we are primarily interested in the non-profiled case we also include attempts which are in time intervals which do not contribute to the attack.

Table 2.5: Fault attempts needed for full key recovery, using either mutes or success, both with and without profiling.

Target	success	prof success	mutes	prof mutes
ARM Simple-AES	0.714M	39K	95M	5.1M
ARM SS2016 AES	40M	2.3M	74M	4.6M
RISC-V Simple-AES	19M	2.6M	31M	7.5M
AVR8 Avr-crypto-lib AES	14M	800K	N/A*	N/A*

* only 14 key bytes were recovered during our experiment

Table 2.5 shows the number of attempts which are needed for full key recovery for the different targets. The results are presented for two models: being able to distinguish faulty outputs, and not being able to distinguish faulty outputs. Additionally, the table also contains the amount of faults which would be required for a repeat, profiled attack – specifically, if only the time intervals which contributed to key recovery were attacked. We believe that further profiling, together with optimization of the Fault injection parameters, would further reduce the required attempts.

As stated in Section 2.4.6 we selected glitch parameters which would give a good distribution over the entire length of the trace. We believe tailoring the parameters per attacked interval could reduce the amount of noise at these points.

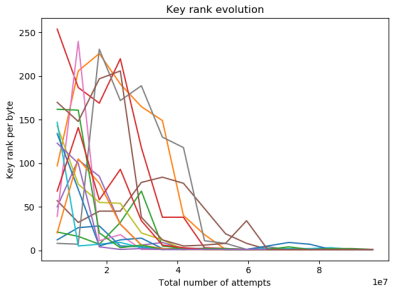
These results show that classifying faults based on success rather than mutes is a superior approach. However, using mutes is still a realistic option.

On targets where the mute model is extremely poor, it is still possible to recover keys with a sufficient number of attempts; in the case of the AVR8 we terminated our data collection before complete key recovery. However, convergence of the remaining key bytes can be seen in Figure 2.5.

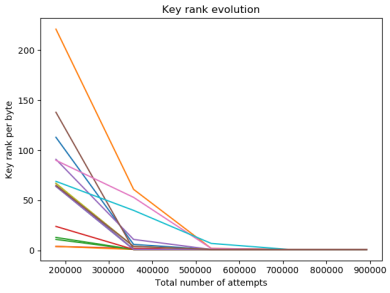
These results also clearly show that, on a pure software target, our approach is

Figure 2.5: Key rank evolution plots, showing the key rank for each key guess vs. the number of attempts. Note that the x-axis scales can differ by an order of magnitude.

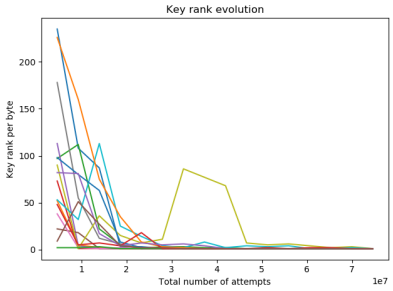
(a) ARM simple AES: mutes



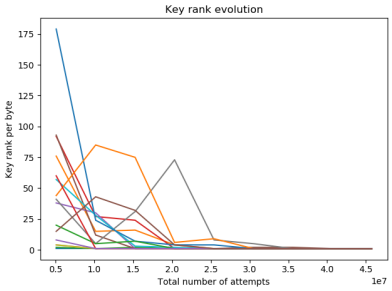
(b) ARM simple AES: successes



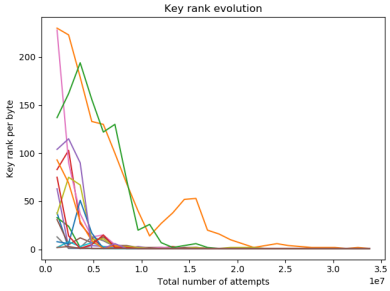
(c) ARM SS2016 AES: mutes



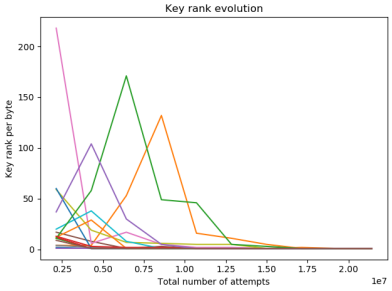
(d) ARM SS2016 AES: successes



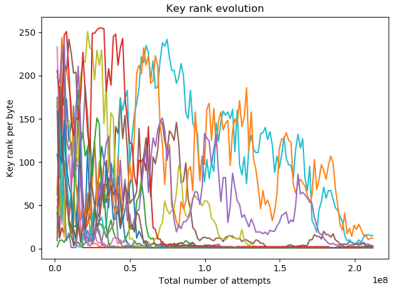
(e) RISC-V simple AES: mutes



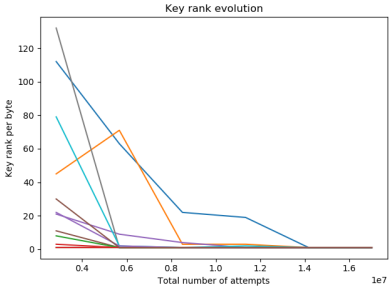
(f) RISC-V simple AES: successes



(g) AVR8 simple AES: mutes



(h) AVR8 simple AES: successes



able to overcome sources of noise such as misclassification and jitter, given sufficient attempts.

Finally, we also attempted to apply two non-CPA attacks to the same data. First, we used non-profiled Linear Regression Analysis (LRA) [152] using the implementation provided by JLsca. Using success-probability traces, we successfully recovered the full key for all our targets. Secondly, we wrote an implementation of a 'model-less' correlation-enhanced collision attack [167] in Python. When attacking the first two key bytes using the success-probability traces, we successfully recovered the correct key bytes on 2 of our targets (AVR8 and ARM simple-AES).

2.5.3 Hardware AES

The previous section has established that our attack is feasible on software implementations, where cryptographic operations run on the same processor that is responsible for the input and output of data. In this section, we show that the attack can still be applied even if a dedicated hardware engine performs the cryptographic operation, rather than software running on the processor.

Our Xmega (AVR8) target contains a hardware AES engine, which has previously been shown to be vulnerable to a CPA attack by Kizhvatov [126]. We hypothesize that we can perform the attack in the same manner as before. However, the attack is complicated because the processor core is in the same power plane as the AES engine, but the processor needs to remain running during encryption so that it can eventually return the output.

As the paper describes, the leakage from this engine indicates that AES operations are performed on a single byte at a time, presumably to optimize the number of gates. The leakage consists of the Hamming distance (HD) between pairs of intermediate states; to attack a key byte, we assume that we know the previous key byte, and then use CPA to recover the value of the next key byte. We first replicated this attack using power traces from an oscilloscope; a portion of such a trace is shown in Figure 2.6. We found that the first-round pre-sbox leakage is stronger after applying a ShiftRows permutation; the key byte numbers referred to below are *after* such a permutation has been applied.

We attempted to duplicate Kizhvatov's attack by applying FCA in the same way as our attacks on the software targets. Although the implementation of the CPA attack itself differs significantly from the software attacks described above – we attack the Hamming distance between internal states, rather than the Hamming weight of individual states – this only affects the actual SCA attack performed on the probability traces, and not the FI attack nor the generation of the traces.

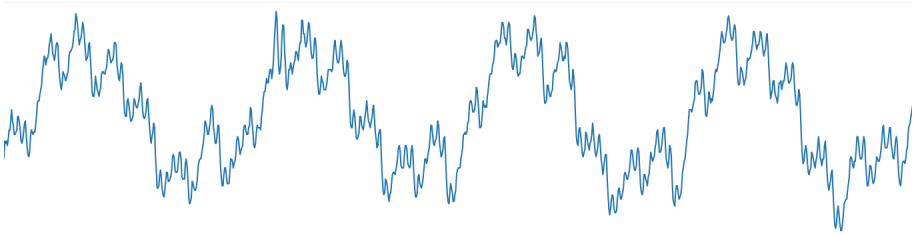


Figure 2.6: Oscilloscope power measurement during encryption with hardware AES engine.

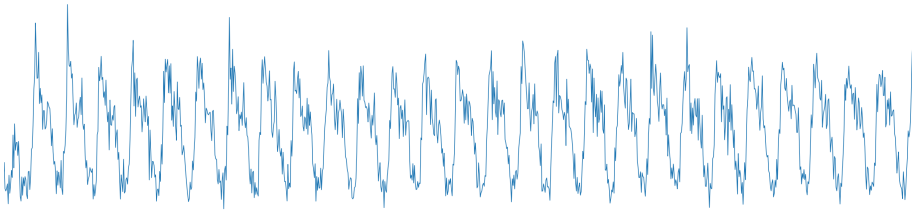


Figure 2.7: Probability trace showing iterations of software polling loop.

Results

We use the example code supplied by the chip vendor, which repeatedly checks the contents of a register to see if the engine has finished, in a loop which we will refer to as the "polling loop". During characterization, we selected FI parameters which resulted in early exits from this loop, which we determined by checking for all-zero output. This means that our attack parameters are optimized for faults in *software*, and not the hardware engine.

We performed 236 million fault injection attempts, for 1000 plaintext inputs, resulting in 236k attempts being used to generate each probability trace. At first glance, the resulting probability traces – an example for one input is shown in Figure 2.7 – appear to reflect the power usage of the software polling loop, rather than the hardware AES engine. As expected, since we are optimizing our faults to attack the software, these probability traces differ significantly from the oscilloscope power traces.

However, the power usage of the hardware engine is also present in these traces, allowing us to apply our SCA attack using the success-probability model, and obtain key bytes. We successfully recover (rank 1) the 6th and 16th key bytes, and obtain a high rank for the 11th key byte (see Figure 2.8), but fail to recover any of the other key bytes. Similarly, we saw HD correlation for only some input byte pairs: specifically, 4+5, 9+10 and 14+15. Again, we saw no significant correlation for other input byte pairs.

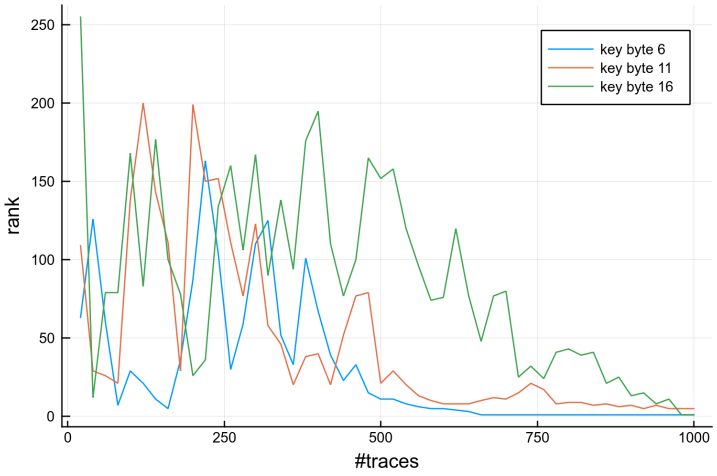


Figure 2.8: Key rank evolution for hardware AES engine FCA attack.

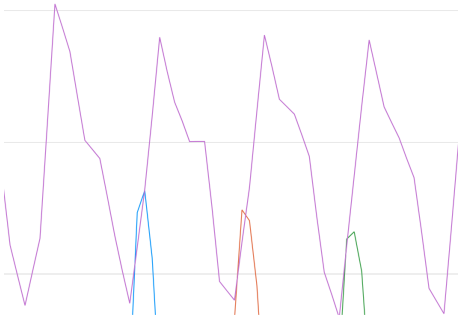


Figure 2.9: Hardware AES engine input byte pair correlation (4+5, 9+10, 14+15), with oversampled probability trace showing corresponding iterations of software polling loop.

Upon inspection, we saw that our compiler produced a polling loop which takes 5 cycles for each iteration; it seems that only *one* of the instructions in our loop was susceptible to the FI parameters we were using, and the power usage of the hardware engine is only reflected in our trace when that instruction is executing. This can be seen in Figure 2.9, where a small section of the probability trace (showing iterations of the polling loop) is shown alongside the input byte correlation spikes.

This proof-of-concept attack targeting early exits from the software polling loop demonstrates that FCA can be successfully applied to leakage from hardware engines. We have shown that the power usage of the hardware engine is encoded in the probability traces, despite the fact our underlying FI attack parameters are chosen to cause glitches in software instructions.

By analysing the corrupted outputs in more detail, using our knowledge of the

key, we can distinguish a number of other cases. We see that there are a number of cases in which there appears to be corruption which could be explained by single byte faults before and after the first and second and even the third mix column. In total, this group represents approximately 2.5% of all results. The majority of corrupted outputs contain a large number of 0 bytes. We additionally see evidence of memory dumps.

Considering a wider range of behavior (such as the above) during the characterization stage, rather than only early exits, would allow more points within the polling loop to be attacked, and thus more key bytes to be recovered. Leakage of other intermediate values could also be used to attack any remaining key bytes.

2.6 Discussion

The nature of our attack requires a large number of fault injection attempts; calculating a fault probability ratio for a given point in time necessarily requires multiple attempts, this must be repeated for multiple points in time. For attacks such as CPA where different inputs are required, this in turn must be entirely repeated for each input. As discussed in Section 2.4.6, our evaluation used a large range of parameters; further tuning of these parameters will reduce the number of attempts required in an actual attack. Despite this, we showed that a profiled attack against Simple AES running on the ARM Cortex-M3 target can recover the key with fewer than 50K attempts, despite the inherent jitter caused by triggering on the UART traffic.

Since our success probability classification does not require the attacker to observe the contents of the corrupted ciphertext, only the fact that a fault was successful in modifying the operation, countermeasures involving redundant checks are ineffective. To attack targets protected with an infective countermeasure, the input must be repeatable and the output must be available [57, 58]

Targets which are protected using random delays can also be attacked; however the number of required attempts will increase, which could render the attack infeasible. The intuition is that the probability traces will behave as power traces which are heavily averaged without any synchronization. The leakage will thus be spread over multiple samples and be combined with noise.

We do not expect masked implementations, which do not exhibit first order leakage, to be vulnerable to our attack. Again, the intuition is that the 'probability traces' behave similarly to power traces which have been averaged before a higher-order sample-combination operation has been performed. However, the probability of a fault may be dependent on both the processed intermediate and the mask at the same time, in which case squaring the samples could be effective. This would heavily depend on the method and shape of the injected glitch. Designing a second order FCA that would effectively work against masking would be interesting future

work.

Our evaluation was performed using power fault injection, which can be performed with a low-cost setup without the need to disable PLLs. Previous related work, such as SIFA and FTA, have used other glitching techniques, namely clock glitching and EM-FI. Our Xmega hardware AES engine attack could be repeated using EM-FI with the probe above the crypto engine; a related attack was performed in FTA [208]. This way, the CPU usage should be minimally affected and the FI traces would reflect mainly the engine. Similarly, the SIFA paper [57] attacks the same hardware engine using ineffective faults and clock glitching. Although both of these attacks differ significantly from our approach, these papers show that both EM-FI and clock glitching can be effectively used against the same hardware, and it would be interesting future work to attempt to apply FCA using these other techniques.

Generally, it would be easier for an attacker to perform classic power side channel attacks on a target, since they are simpler than our approach. One potential future path of investigation involves using software-activated faults, which do not require additional equipment. In recent years, these have demonstrated against high-profile targets such as TrustZone (CLKSCREW [224], VoltJockey [195]) and Intel CPUs (V0LTpwn [121], Plundervolt [169]). Finally, we also believe our work could be extended to apply other existing power or EM SCA attacks against other ciphers.

2.7 Conclusion

We have presented a general technique for performing FI attacks and classifying their results, resulting in 'probability traces' which allow us to apply existing side-channel attacks to the resulting probability traces. Specifically, we demonstrated that we can perform Simple Fault Analysis to obtain RSA key bits, by applying SPA to probability traces. Similarly, we demonstrated that we can perform Correlation Fault Analysis to obtain private AES keys, by applying CPA to probability traces.

We investigated two different classification models, assigning probabilities based on the ratios of either mutes, or successes. Although the success model appears to be superior and results in considerably faster key convergence, we have also shown that the mute model can be applied in situations where information about the success of cryptographic operations is not available to an attacker.

We demonstrated successful attacks against cryptographic libraries running on three different hardware targets, focusing on CPA against AES, but including the LRA distinguisher, collision attacks and even SPA attacks on RSA. We also presented proof-of-concept results which recover key bytes from a real-world hardware engine, where leakage from intermediate states appears to be part of the power dependency when attacking software instructions running at the same mo-

ment in time.

These results show that the relationship between FI and SCA attacks is strong enough that we can successfully apply standard, unmodified SCA attacks to the results of FI campaigns on real-world hardware.

3 RIDL: Rogue In-Flight Data Load

We present *Rogue In-flight Data Load* (RIDL), a new class of speculative unprivileged and constrained attacks to leak arbitrary data across address spaces and privilege boundaries (e.g., process, kernel, SGX, and even CPU-internal operations). Our reverse engineering efforts show such vulnerabilities originate from a variety of micro-optimizations pervasive in commodity (Intel) processors, which cause the CPU to speculatively serve loads using extraneous CPU-internal *in-flight* data (e.g., in the *line fill buffers*). Contrary to other state-of-the-art speculative execution attacks, such as Spectre, Meltdown and Foreshadow, RIDL can leak this arbitrary in-flight data with no assumptions on the state of the caches or translation data structures controlled by privileged software.

The implications are worrisome. First, RIDL attacks can be implemented even from linear execution with no invalid page faults, eliminating the need for exception suppression mechanisms and enabling system-wide attacks from arbitrary unprivileged code (including JavaScript in the browser). To exemplify such attacks, we build a number of practical exploits that leak sensitive information from victim processes, virtual machines, kernel, SGX and CPU-internal components. Second, and perhaps more importantly, RIDL bypasses all existing "spot" mitigations in software (e.g., KPTI, PTE inversion) and hardware (e.g., speculative store bypass disable) and cannot easily be mitigated even by more heavyweight defenses (e.g., L1D flushing or disabling SMT). RIDL questions the sustainability of a per-variant, spot mitigation strategy and suggests more fundamental mitigations are needed to contain ever-emerging speculative execution attacks.

Preface

This chapter is based on the officially published (camera-ready) version of the RIDL paper. Some further details were embargoed at the time of publication and were only disclosed at a later date, and thus are not present in this text:

- 14th May 2019: We published an updated version of the RIDL paper, a few days before it was presented at IEEE S&P. Along with clarifications and additional information (such as CVE identifiers), it provided a partial description of the columns of Table 3.1, which were undocumented in the original paper.
- 12th November 2019: We published Addendum 1, with details of the TSX abort variant ("TSX Asynchronous Abort", CVE-2019-11135).
- 27th January 2020: We published Addendum 2, with details of the alignment fault variant ("Vector Register Sampling", CVE-2020-0548).

The following text summarizes our original results. The 'Page Fault', 'Misaligned Read' and 'TSX' columns of Table 3.1 refer to (1), (2) and (4) respectively, and we used variant (4) for the majority of our original attacks. For more details, we refer to the addenda, which are provided as appendices to this chapter.

The behavior we observe appears to be caused by *exception deferral* (see Section 3.7.2). As such, alongside the variant we discussed above, we would expect to be able to perform these attacks using a variety of exceptions. We successfully reproduced RIDL attacks using several alternative exceptions by loading from:

1. An invalid address (our primary variant), such as the `NULL` pointer used in the example in Section 3.4, or a lazily-mapped page, which has not yet been mapped by the kernel and requires demand paging.
2. A misaligned address used with an aligned load (e.g., `movdqa`, or using the Alignment Check flag), or an address spanning two pages[1], one or both of which are invalid (e.g., demand paged).
3. A page where the `accessed` bit is not set in the corresponding page table entry, while using TSX – this aborts the transaction, and thus the load.
4. A cache line in the process of being evicted, while using TSX. This aborts the transaction, and appears to also leak *store buffers* (see Section 3.2.4).

Conclusion: we can use a variety of exceptions to trigger RIDL attack variants.

[1]The May 14th update stated that the cross-page loads leak from *load ports*.

3.1 Introduction

Since the original Meltdown and Spectre disclosure, the family of memory disclosure attacks abusing speculative execution [2] has grown steadily [28, 124, 128, 135, 149]. While these attacks can leak sensitive information across security boundaries, they are all subject to strict addressing restrictions. In particular, Spectre variants [124, 128, 135] allow attacker-controlled code to only leak within the loaded virtual address space. Meltdown [149] and Foreshadow [28] require the target physical address to at least appear in the loaded address translation data structures. Such restrictions have exposed convenient anchor points to deploy practical "spot" mitigations against existing attacks [136, 147, 190, 232]. This shaped the common perception that—until in-silicon mitigations are available on the next generation of hardware—per-variant, software-only mitigations are a relatively pain-free strategy to contain ever-emerging memory disclosure attacks based on speculative execution.

In this chapter, we challenge the common perception by introducing *Rogue In-flight Data Load* (RIDL), a new class of speculative execution attacks that lifts all such addressing restrictions entirely. While existing attacks target information at specific addresses, RIDL operates akin to a passive sniffer that eavesdrops on *in-flight* data (e.g., in the *line fill buffers* or *LFBs*) flowing through CPU components. RIDL is powerful: it can leak information across address space and privilege boundaries by solely abusing micro-optimizations implemented in commodity Intel processors. Unlike existing attacks, RIDL is non-trivial to stop with practical mitigations in software.

The vulnerability of existing vulnerabilities. To illustrate how existing speculative execution vulnerabilities are subject to addressing restrictions and how this provides defenders convenient anchor points for "spot" software mitigations, we consider their most prominent examples.

Spectre [128] allows attackers to manipulate the state of the branch prediction unit and abuse the mispredicted branch to leak arbitrary data within the accessible address space via a side-channel (e.g., cache). This primitive by itself is useful in sandbox (e.g., JavaScript) escape scenarios, but needs to resort to *confused-deputy* attacks [241] to implement cross-address space (e.g., kernel) memory disclosure. In such attacks, the attacker needs to lure the victim component into speculatively executing specific "gadgets", disclosing data from the victim address space back to the attacker. This requirement opened the door to a number of practical software mitigations, ranging from halting speculation when accessing untrusted pointers or indices [190] to not speculating across vulnerable branches [232].

Meltdown [149] somewhat loosens the restrictions of the addresses reachable

[2] Unless otherwise noted, we loosely refer to both speculative and out-of-order execution as speculative execution in this chapter.

from attacker-controlled code. Rather than restricting the code to valid addresses, an unprivileged attacker can also access privileged address space mappings that are normally made inaccessible by the supervisor bit in the translation data structures. The reason is that, while any access to a privileged address will eventually trigger an error condition (i.e., invalid page fault), before properly handling it, the CPU already exposes the privileged data to out-of-order execution, allowing disclosure. This enables cross-address space (user-to-kernel) attacks, but only in a traditional user/kernel unified address space design. This requirement opened the door to practical software mitigations such as KPTI [136], where the operating system (OS) isolates the kernel in its own address space rather than relying on the supervisor bit for isolation.

Foreshadow [28] further loosens the addressing restrictions. Rather than restricting attacker-controlled code to valid and privileged addresses, the attacker can also access physical addresses mapped by invalid (e.g., non-present) translation data structure entries. Similar to Meltdown, the target physical address is accessed via the cache, data is then passed to out-of-order execution, and subsequently leaked before the corresponding invalid page fault is detected. Unlike Meltdown, given the milder addressing restrictions, Foreshadow enables arbitrary cross-address space attacks. But this is only possible when the attacker can surgically control the physical address of some invalid translation structure entry. This requirement opened the door to practical software mitigations such as PTE inversion [147], where the OS simply masks the physical address of any invalidated translation structure entry.

A new RIDL. With RIDL, we show our faith in practical, "spot" mitigations being able to address known and future speculative execution attacks was misplaced. As we shall see, RIDL can leak in-flight data of a victim process even if that process is not speculating (e.g., due to Spectre mitigations) and it does not require control over address translation data structures at all. These properties remove all the assumptions that spot mitigations rely on. Translation data structures, specifically, enforce basic security mechanisms such as isolation, access permissions and privileges. Relaxing the requirement on translation data structures allows RIDL to mount powerful cross-address space speculative execution attacks directly from error-free and branchless unprivileged execution for the first time. In particular, by snooping on in-flight data in the CPU, attackers running arbitrary unprivileged code (including JavaScript in the browser) may leak information across arbitrary security boundaries. In essence, RIDL puts a glass to the wall that separates security domains, allowing attackers to listen to the babbling of CPU components.

To investigate the root cause of the RIDL class of vulnerabilities, we report on our reverse engineering efforts on several recent Intel microarchitectures. We show RIDL stems from micro-optimizations that cause the CPU to serve speculative loads with extraneous CPU-internal *in-flight* data. In the paper, we focus on

instances serving arbitrary, address-agnostic data from the *Line Fill Buffers* (LFBs), which we found to be exploitable in the practical cases of interest. We also report on the challenges to exploit such instances in practice, targeting specific *in-flight* data to leak in particular. Moreover, we present a number of practical exploits that leak data across many common security boundaries (JavaScript sandbox, process, kernel, VM, SGX, etc.). We show exploitation is possible in both cross-thread and same-thread (no SMT) scenarios. This applies to all the existing Intel systems with the latest (microcode) updates and all the defenses up. In particular, RIDL bypasses all the practical mitigations against existing attacks and even the more heavyweight, default-off mitigations such as periodic L1 flushing. The lesson learned is that mitigating RIDL-like attacks with practical software mitigations is non-trivial and we need more fundamental mitigations to end the speculative execution attacks era.

Contributions. We make the following contributions:

- We present RIDL, a new class of speculative execution vulnerabilities pervasive in commodity Intel CPUs. RIDL enables unprivileged attackers to craft address-agnostic memory disclosure primitives across arbitrary security boundaries for the first time and has been rewarded by the Intel Bug Bounty Program.
- We investigate the root cause of practical RIDL instances abusing Line Fill Buffers (LFBs), presenting the first reverse engineering effort to analyze LFB behavior and related micro-optimizations.
- We present a number of real-world exploits that demonstrate an unprivileged RIDL-enabled attacker can leak data across arbitrary security boundaries, including process, kernel, VM and SGX, even with all mitigations against existing attacks enabled. For example, we can leak the contents of the `/etc/shadow` in another VM using a RIDL attack. Demos of these RIDL exploits can be found at `https://ridl.eu`.
- We place RIDL in the context of state-of-the-art attacks and mitigation efforts. Our analysis shows that, unlike existing attacks, RIDL is ill-suited to practical mitigations in software and more fundamental mitigations are necessary moving forward.

3.2 Background

Figure 3.1 shows an overview of the Intel Skylake microarchitecture. It consists of three stages: ① an in-order front-end that decodes instructions into μ-ops, ② an out-of-order execution engine, ③ and the memory pipeline. Since the Intel Skylake microarchitecture is quite complex, we specifically focus on the cache hierarchy, out-of-order/speculative execution, and in-flight data.

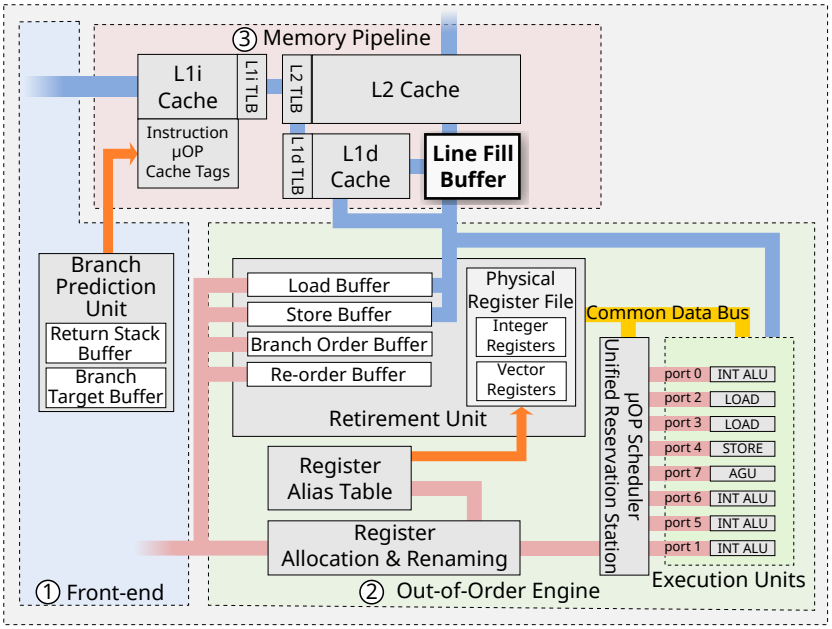


Figure 3.1: An overview of the Intel Skylake microarchitecture.

3.2.1 Caches

To overcome the growing performance gap between processors and memory, the processor contains small memory buffers, called caches, to store frequently and recently used data to hide memory latency. Modern processors have multiple levels of caches with the smallest and fastest close to the processor, and the largest but slowest being the furthest away from the processor. The Intel Core microarchitecture has three levels of CPU caches. At the first level, there are two caches, L1i and L1d, to store code and data respectively, while the L2 cache unifies code and data. Where these caches are private to each core, all cores share the L3 or last-level cache (LLC). The LLC is inclusive of the lower-level caches and set-associative, i.e., divided into multiple cache sets where part of the physical address is used to index into the corresponding cache set.

Gullasch et al. [87] use `clflush` to evict targets to monitor from the cache. By measuring the time to reload them the attacker determines whether the victim has accessed them—a class of attacks called Flush + Reload [256]. Another variant is Prime + Probe [115, 120, 150, 179], in which the attacker builds an eviction set of memory addresses to fill a specific cache set. By repeatedly measuring the time it takes to refill the cache set, the attacker can monitor memory accesses to that cache set.

3.2.2 Out-of-order execution

To improve the instruction throughput, modern CPUs implement a superscalar out-of-order execution pipeline similar to Tomasulo's algorithm [227, 229]. Out-of-order execution engines generally consist of three stages: ① in-order register allocation & renaming, ② out-of-order execution of instructions or µ-ops, ③ and in-order retirement.

Register renaming Once the decoded µ-ops leave the front-end, they pass through the register allocation & renaming unit that renames the registers to eliminate *Write-after-Read* (WAR) and *Write-after-Write* (WAW) hazards. More specifically, this unit renames source/destination operands for µ-ops by allocating pointers to freely available physical registers from the *Physical Register File* (PRF) and maintaining the corresponding mappings in the *Register Alias Table* (RAT). After renaming the µ-op, the unit allocates an entry for the µ-op in the *Re-Order Buffer* (ROB) to preserve the original programming order and sends the µ-op to the reservation station.

Out-of-order scheduling To eliminate *Read-after-Write* (RAW) hazards, the reservation station stalls each µ-op with unavailable operands. Once all the operands are available, the scheduler dispatches the µ-op to the corresponding execution unit, possibly before scheduling older µ-ops. After executing the µ-op, the reservation station stores its result, updates the µ-ops that depend on it, and marks the corresponding ROB entry as completed.

Retirement The *Retirement Unit* retires completed µ-ops in their original program order by committing the architectural state for memory/branch operations and freeing up any allocated physical registers. In case of a mispredicted branch, the *Retirement Unit* retires the offending branch instruction, flushes the ROB, resets the reservation station, and replays the execution stream from the correct branch. The *Retirement Unit* also detects faulty instructions and generates precise exceptions once the offending µ-op reaches a non-speculative state.

3.2.3 Speculative execution

To predict the target of a branch, modern processors feature a *Branch Prediction Unit* (BPU). The BPU predicts the branch target such that the processor can execute a stream of instructions speculatively. In case the predicted path is wrong, the processor reverts the state to the last known useful state and starts executing from the correct branch instead. There are several instances where speculation occurs, such as: conditional branches, indirect branches and calls, return instructions and transactions.

Recent Intel processors employ a *Branch Order Buffer* (BOB) to keep track of all in-flight branches and whether the branch is in retired or speculative state [23, 44]. The BOB is also used to implement memory transactions through *Transactional Synchronization eXtensions* (TSX). In particular, the `xbegin` instruction marks the start of a transaction and adds an entry to the BOB as a checkpoint. Transactions end once the processor encounters an `xend` instruction, an `xabort` instruction, or a fault. In case of an `xend` instruction, the processor commits the transaction, otherwise the processor rolls back the transaction by reverting back to the original state before the `xbegin`.

3.2.4 In-flight data

There are many potential sources of in-flight data in modern CPUs such as the *Re-Order Buffer* (ROB), the *Load* and *Store Buffers* (LBs and SBs) [123, 155, 161, 171], the *Line Fill Buffers* (LFBs), and the *Super Queue* (SQ) [133, 140]. We focus here on two prominent examples: store buffers and line fill buffers.

Store Buffers (SBs) are internal buffers used to track pending stores and in-flight data involved in optimizations such as *store-to-load forwarding* [155, 161]. Some modern processors enforce a strong memory ordering, where load and store instructions that refer to the same physical address cannot be executed out-of-order. However, as address translation is a slow process, the physical address might not be available yet, and the processor performs memory disambiguation to predict whether load and store instructions refer to the same physical address [258]. This enables the processor to speculatively execute unambiguous load and store instructions out-of-order. As a micro-optimization, if the load and store instructions are ambiguous, the processor can speculatively *store-to-load forward* the data from the store buffer to the load buffer.

Line Fill Buffers (LFBs) are internal buffers that the CPU uses to keep track of outstanding memory requests and perform a number of optimizations such as merging multiple in-flight stores. Sometimes, data may already be available in the LFBs and, as a micro-optimization, the CPU can speculatively load this data (similar optimizations may also be performed on *load/store buffers*, etc.). In both cases, modern CPUs that implement aggressive speculative execution may speculate without any awareness of the virtual or physical addresses involved. In this chapter, we specifically focus on LFBs, which we found particularly amenable to practical, real-world RIDL exploitation.

3.3 Threat Model

We consider an attacker who wants to abuse speculative execution vulnerabilities to disclose some confidential information, such as private keys, passwords, or ran-

domized pointers. We assume a victim Intel-based system running the latest microcode and OS version, with all the state-of-the-art mitigations against speculative execution attacks enabled. We also assume the victim system is protected against other classes of (e.g., software) vulnerabilities. Finally, we assume the attacker can only run unprivileged code on the victim system (e.g., JavaScript sandbox, user process, VM, or SGX enclave), but seeks to leak information across arbitrary privilege levels and address spaces.

3.4 Overview

Figure 3.2 illustrates the main steps and the underlying mechanism enabling the RIDL leaks. First, as part of its normal execution, the victim code, in another security domain, loads or stores some secret data[3]. Internally, the CPU performs the load or store via some internal buffers—*Line Fill Buffers* (LFBs) in the RIDL instances considered in this chapter. Then, when the attacker also performs a load, the processor speculatively uses in-flight data from the LFBs (with no addressing restrictions) rather than valid data. Finally, by using the speculatively loaded data as an index into a FLUSH + RELOAD buffer (or any other covert channel), attackers can extract the secret value.

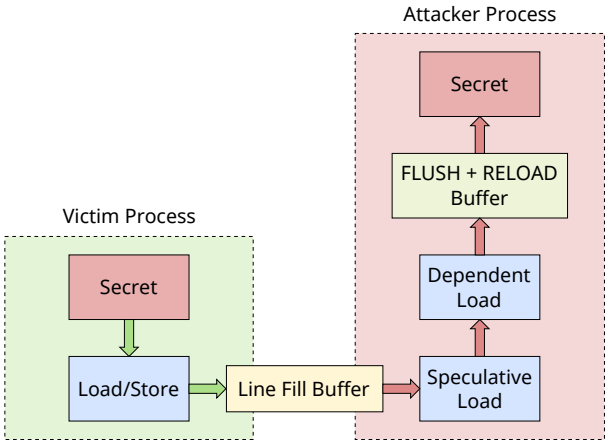


Figure 3.2: An overview of the RIDL attack.

A simple example of our attack is shown in Listing 1. As shown in the listing, the code is normal, straight-line code without invalid accesses (or, indeed, error suppression), which, as we will show, can also be implemented in managed languages such as JavaScript. Lines 2–3 only flush the buffer that we will later use in

[3]Strictly speaking, this is not even a hard requirement, as we can also leak data from inactive code by forcing cache evictions.

```
/* Flush flush & reload buffer entries. */
for (k = 0; k < 256; ++k)
  flush(buffer + k * 1024);

/* Speculatively load the secret. */
char value = *(new_page);
/* Calculate the corresponding entry. */
char *entry_ptr = buffer + (1024 * value);
/* Load that entry into the cache. */
*(entry_ptr);

/* Time the reload of each buffer entry to
   see which entry is now cached. */
for (k = 0; k < 256; ++k) {
  t0 = cycles();
  *(buffer + 1024 * k);
  dt = cycles() - t0;

  if (dt < 100)
    ++results[k];
}
```

Listing 1: An example of RIDL leaking in-flight data.

our covert channel to leak the secret that we speculatively access in Line 6. Specifically, when executing Line 6, the CPU speculatively loads a value from memory in the hope it is from our newly allocated page, while really it is in-flight data from the LFBs belonging to an arbitrarily different security domain.

When the processor eventually detects the incorrect speculative load, it will discard any and all modifications to registers or memory, and restart execution at Line 6 with the right value. However, since traces of the speculatively executed load still exist at the micro-architectural level (in the form of the corresponding cache line), we can observe the leaked in-flight data using a simple (Flush + Reload) covert channel—no different from that of other speculative execution attacks. In fact, the rest of the code snippet is all about the covert channel. Lines 8-10 speculatively access one of the entries in the buffer, using the leaked in-flight data as an index. As a result, the corresponding cache line will be present. Lines 12-21 then access all the entries in our buffer to see if any of them are significantly faster (indicating that the cache line is present)—the index of which will correspond to the leaked information. Specifically, we may expect *two* accesses to be fast, not just the one corresponding to the leaked information. After all, when the processor discovers its mistake and restarts at Line 6 with the right value, the program will also access the buffer with this index.

Our example above uses demand paging for the loaded address, so the CPU restarts the execution only after handling the page-in event and bringing in a newly mapped page. Note that this is not an error condition, but rather a normal part of the OS' paging functionality. We found many other ways to speculatively execute code using in-flight data. In fact, the accessed address is not at all important. As an extreme example, rather than accessing a newly mapped page, Line 6 could even dereference a `NULL` pointer and later suppress the error (e.g., using TSX). In general, any run-time exception seems to be sufficient to induce RIDL leaks, presumably because the processor can more aggressively speculate on loads in case of exceptions. Clearly, one can only "speculate" here, but this behavior seems consistent with existing vulnerabilities [28, 149]. Similarly, we noticed the address accessed by the attacker should be part of a page-aligned cache line to induce leaks.

While the basic concept behind in-flight data may be intuitive, successfully implementing an attack turned out to be challenging. Unlike prior work that builds on well-documented functionality such as branch prediction, page tables and caches, the behavior of internal CPU buffers such as LFBs is largely unknown. Moreover, different microarchitectures feature different types of buffers with varying behavior. Furthermore, based on publicly available documentation, it was not clear whether many of these buffers even *exist*. For our attack to succeed, we had to resort to extensive reverse engineering to gain a better understanding of these buffers and their interaction with the processor pipeline. The next section discusses how we determine exactly which in-flight data buffers are responsible for the leak, how to manipulate the processor state in such a way that we can perform a speculative load that uses the in-flight data (so that we can use our covert channel to obtain the content), and how to ensure the data we want to leak actually ends up in the buffers.

3.5 Line fill buffers and how to use them

To perform the attack described in the previous section, we first need to understand the core building blocks for the RIDL variant considered in the paper: the *Line Fill Buffers* (LFBs). Using reverse engineering and experimentation, we verify that we do indeed leak from the LFBs (and not some other buffer) and examine their interaction with the processor pipeline. After that, we discuss how attackers can control what to leak by synchronizing with the victim.

In Intel processors, the LFB performs multiple roles: it enables non-blocking caches, buffers non-temporal memory traffic [7, 8, 183, 184], and performs both load squashing [3, 22, 154] and write combining [79, 181, 207]. To help the reader understand the remainder of this chapter, we now briefly discuss each of these functions.

Non-blocking cache. Cache misses have a serious impact on performance as they

block the data cache until the data is available. To allow non-blocking reads, the LFB implements multiple *Miss Status Holding Registers* (MSHRs) to track the physical addresses of outstanding requests until the data is available [41, 49]. For example, the Haswell microarchitecture maintains 10 L1 MSHRs in its LFB to service outstanding L1 cache misses [103, 119] These MSHRs free up the L1d cache to allow load instructions that hit the L1d cache to bypass cache misses.

Load squashing. To further optimize performance, the LFB squashes multiple load misses to the same physical address. If there is already an outstanding request in the LFB with the same physical address, the processor assigns the same LFB entry to a load/store with the same address.

Write combining. For weakly-ordered memory, the processor keeps stores to the same cache line within the LFB to perform *write combining*. That is, the processor merges multiple stores in a single LFB entry before writing out the final result through the memory hierarchy.

Non-temporal requests. Finally, modern processors support non-temporal memory traffic where the programmer already knows that caching the data is of no benefit at all. In that case, the processor performs non-temporal loads and stores exclusively through the LFB.

3.5.1 Solving a RIDL: LFB leaks on loads and stores

Unaware of the source of the RIDL leaks initially, we discovered that they originate from the LFBs, rather than from other processor state, by conducting several experiments on a workstation featuring an Intel Core i7-7700K (Kaby Lake). For our experiments, we use a kernel module to mark memory pages in our victim thread as *write-back* (WB), *write-through* (WT), *write-combine* (WC) and *uncacheable* (UC) [66, 113]. We use Intel TSX to implement the attack for our analysis and perform $10,000$ rounds to leak the data during every run of the experiment. Furthermore, we run every experiment 100 times and report the average.

Our first experiment performs the attack discussed earlier against a victim running in the same or other hardware thread and repeatedly storing a secret to a fixed memory address. We then compare the number of hits in the LFB, measured using the *lfb_hit* performance counter, to the number of attack iterations. Consider the left-most (SMT) plot of Figure 3.3 which shows close correspondence between the number of LFB hits and the number of attempts for leaking (and correctly obtain the secret). This strongly suggests that the source of our leak is the LFB.

In our second experiment, a victim thread initially writes a known value A to a fixed memory address, and then reads it back in a loop where each read is followed by an `mfence` instruction (serializing all loads and stores issued prior to the `mfence`). We mark this address as WB, WT, WC and UC. Optionally, we also flush the address using `clflush` (which flushes it from the cache). Figure 3.4 shows

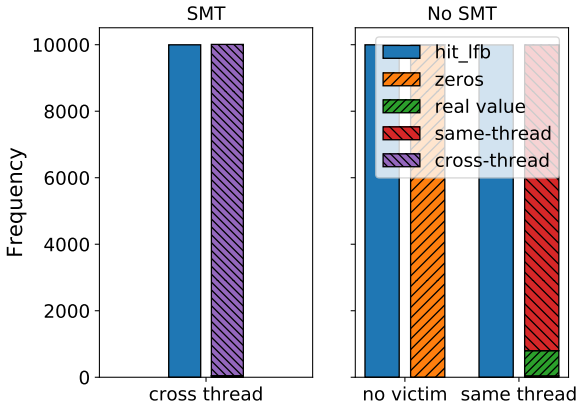


Figure 3.3: In each pair of bars, one bar shows the LFB hit count, and the other one the number of attacks. With SMT, we always leak the secret. Without SMT and no victim code, RIDL only reads zeros, but with victim and attacker in the same hardware thread, we still leak the secret in most cases (top/red bar), while occasionally finding the value the CPU *should* have loaded.

how often a RIDL attacker reads the secret value correctly. Note that we do not observe the signal for WB and WT memory if we do not flush the entry from the cache, but when we do, we observe the signal regardless of the memory type. Both observations indicate that we are not leaking from the cache. Also, since we leak from a load, this cannot be the effect of *store-to-load forwarding* either. Furthermore, since we observe the signal from WC and UC memory, which have to go through the LFB, the source of our leak must again be the LFB.

To gather further evidence that we are leaking from the LFB, we perform a third experiment where we run a victim thread which, in a loop, writes four different values to four sequential cache lines, followed by an `mfence`. Optionally, the victim thread again flushes the cache lines. We again rule out any leaks via store-to-load forwarding, by turning on *Speculative Store Bypass Disable* (SSBD [111]) for both attacker and victim. Figure 3.5 shows the RIDL results. For WB without flushing, there is a signal only for the last cache line, which suggests that the CPU performs write combining in a single entry of the LFB before storing the data in the cache. More importantly, we observe the signal regardless of the memory type when flushing. Since both flushing and the WT, WC and UC memory types enforce direct invalidation of writes, they must go through the LFB. This third experiment again indicates that the source of our leak must be the LFB.

Conclusion: our RIDL variant leaks from the Line Fill Buffers (LFBs).

To launch a RIDL attack, we still need to understand the interaction between loads, stores, the L1d cache and the LFB, such that we can massage the data from

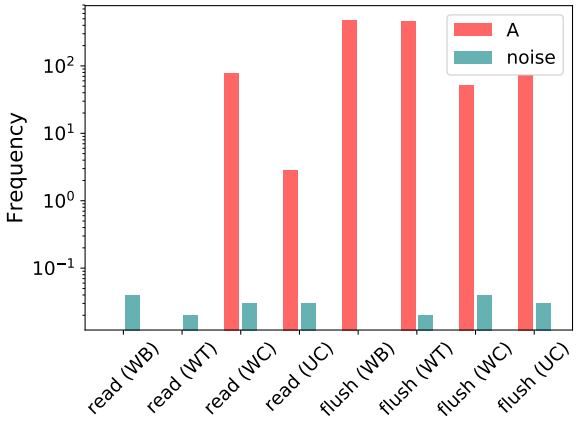


Figure 3.4: Leaking the secret A which is *read* by the victim for write-back (WB), write-through (WT), write-combine (WC) and uncacheable (UC) memory, with and without a cache flush.

the victim into the LFB. Recall that in our read experiment (Figure 3.4), we did not observe a signal if we do not flush the address, even with multiple consecutive reads like we did with writes (Figure 3.5). As the data is read constantly, all loads simply hit in the L1d cache, preventing any interaction of future loads with the LFB. In contrast, when we do flush, the future loads miss and allocate an entry in the LFB to await the data. In case of WC and UC memory, the processor avoids the L1d cache and enforces the loads to always go through the LFB. Our second experiment (Figure 3.5) shows a signal for all memory types and especially those that bypass the L1d cache, suggesting that memory writes go via the LFB.

Conclusion: reads that are not served from L1d pull data through the LFB, while writes push data through the LFB to either L1d or memory.

3.5.2 Synchronization

To leak information, the attacker must make sure that the right data is visible in the LFB at the right time, by synchronizing with the victim. We show that there are three ways to do so: serialization, contention and eviction.

Serialization. Intel CPUs implement a number of barriers to perform different types of serialization [61, 62, 180, 182]. `lfence` guarantees that all loads issued before the `lfence` become globally visible, `sfence` guarantees the same for all store instructions, and `mfence` guarantees that both load and stores before the `mfence` become globally visible. To enforce this behavior, the processor waits for these loads and stores to retire by draining the load and/or store buffers, as well as the

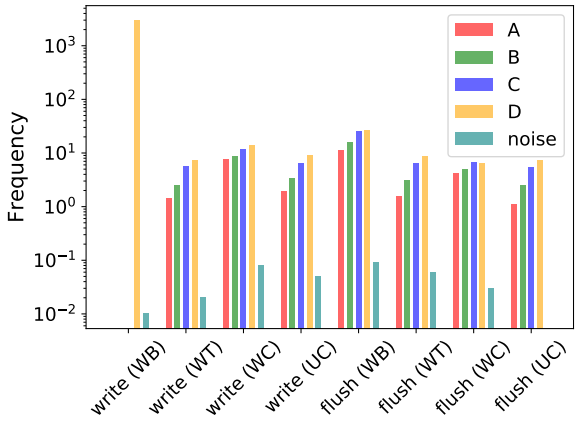


Figure 3.5: Leaking the secrets A, B, C and D, *written* by the victim, for write-back (WB), write-through (WT), write-combine (WC) and uncacheable (UC) memory, with and without a cache flush.

corresponding entries in the LFB. The `mfence` instruction therefore forms a point of synchronization that allows us to observe the last few loads and stores before the buffers are completely drained.

Contention. Another way of synchronizing victim and attacker is to create contention within the LFB, ultimately forcing entries to be evicted. Doing so allows us to obtain some control over the entries that we leak, and should not depend on SMT. To verify this, we perform the RIDL attack *without SMT* by writing values in our own thread and observing the values that we leak from the same thread. Figure 3.3 shows that if we do not write the values ("no victim"), we leak only zeros, but with victim and attacker running in the same hardware thread (e.g., in a sandbox), we leak the secret value in almost all cases.

Eviction. Finally, we can control the values that we leak from the victim by evicting cache entries from the cache set in which we are interested. To show that we can use this for synchronization, we conducted an experiment where the victim writes a value to the same cache line within a number of pages. After a while, these writes end up evicting the previous cache lines from the L1d cache. As these cache lines are dirty, the processor has to write them back through the memory hierarchy and will do this through the LFB. We extend the victim thread to alternate between two different values to write after finishing every loop and also vary the amount of pages to write during the loop. For the first test, we write the first value 1024 times and the second value 256 times (ratio 1:4) and for the second test, we write the first value 512 times and the second 1024 times (1:2 ratio). Figure 3.6 shows the results of this experiment, where we observe the first value 80% of the times and the second value 20% of the times in the case of ratio 1:4 and the first value 33.3%

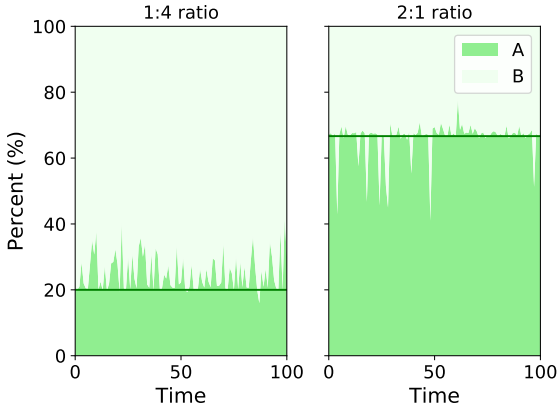


Figure 3.6: Leaking the secrets A and B *written* by the victim to a series of cache lines to trigger continuous eviction. On the left, the victim writes A then B using a 1:4 ratio. On the right, the victim writes A then B using a 2:1 ratio.

of the times and the second value 66.6% of the times in the case of ratio 2:1. Hence, we conclude that we can control the (dirty) cache entry to leak through eviction.

Conclusion: we can use serialization, contention and eviction to synchronize attacker and victim.

3.6 Exploitation with RIDL

The techniques described in the previous section allow us to leak in-flight CPU data in a controlled fashion. Since the underlying buffers are independent of address spaces and privilege levels, we can mount attacks across these security boundaries.

We have verified that we can leak information across arbitrary address spaces and privilege boundaries, even on recent Intel systems with the latest microcode updates and latest Linux kernel with all the Spectre, Meltdown, L1TF default mitigations up (KPTI, PTE inversion, etc.). In particular, the exploits we discuss below exemplify leaks in all the relevant cases of interest: process-to-process, kernel-to-userspace, guest-to-guest, and SGX-enclave-to-userspace leaks. Not to mention that such attacks can be built even from a sandboxed environment such as JavaScript in the browser, where the attacker has limited capabilities compared to a native environment.

We stress that the only requirement is the presence of in-flight secret data managed by the processor. In a non-SMT single-core attack scenario, this is data recently read/written by the victim before a mode switching instruction (`iret`, `vmenter`, etc.). In an SMT attack scenario, this is data concurrently read/written by

another hardware thread sharing the same CPU core. Once we have speculatively leaked a value, we use the techniques discussed earlier (based on FLUSH + RELOAD, or EVICT + RELOAD when `clflush` is not available) to expose the desired data. The key difference with prior Meltdown/L1TF-style attacks that cross privilege boundaries and address spaces is that the target address used by the attacker can be perfectly *valid*. In other words, the attack does not necessarily require a TSX transaction or an invalid page fault, but can also be applied to a correct, branchless execution with demand paging (i.e., a valid page fault) as we showed in Section 3.5. This bypasses side-channel mitigations deployed on all the major operating systems and extends the threat surface of prior cross-address space speculative execution attacks to managed sandboxes (e.g., JavaScript). In the next sections, we explore how RIDL can be used to leak sensitive information across different security boundaries.

Covert channel. We performed an extensive evaluation of RIDL over a number of microarchitectures, showing that it affects all recent Intel CPUs. To verify that RIDL works across all privilege boundaries, we implemented a proof-of-concept covert channel, sending data across address space and privilege boundaries.

In Table 3.1, we present the bandwidth of the covert channel. Note that our implementation is not yet optimized for all architectures. For convenience, we utilize Intel TSX on the architectures where available, as this gives us the most reliable covert channel. Using TSX, we achieve a bandwidth of 30-115 kB/s, where the limiting factor is FLUSH + RELOAD. Where TSX is not available, we present numbers from an unoptimized proof-of-concept implementation which uses either demand paging, or exception suppression using speculative execution.

Challenges. In the previous sections, we discussed how the building blocks of RIDL are used to leak in-flight data and that we can use RIDL to leak information across security domains. Applying these techniques to exploit real-world systems—leaking confidential data—presents some additional challenges that we need to overcome:

1. **Getting data in-flight**. We need to find ways to get restricted data that we want to leak into the LFB. There are some obvious mechanisms for an unprivileged user to get privileged data in-flight: interaction with the kernel (i.e., syscalls), and interaction with a privileged process (i.e., invoking a setuid binary). There are also many other possibilities, such as manipulating the page cache.

2. **Targeting**. Due to the high amount of LFB activity, getting the desired data out of the LFB poses a challenge. We describe two mechanisms for targeting the data we want to leak: *synchronizing the victim*, and *aligning the leaked data* by repeating the attack multiple times while filtering out the noise.

In the next sections, we demonstrate a number of exploits that use RIDL. We evaluated all the exploits on the Intel Core i7-7800X running Ubuntu 18.04 LTS.

3.6.1 Cross-process attacks

In a typical real-world setting, synchronizing at the exact point when sensitive data is in-flight becomes non-trivial, as we have limited control over the victim process. Instead, by repeatedly leaking the same information and aligning the leaked bytes, we can retrieve the secret without requiring a hard synchronization primitive. For this purpose, we show a noise-resilient *mask-sub-rotate* attack technique that leaks 8 bytes from a given index at a time.

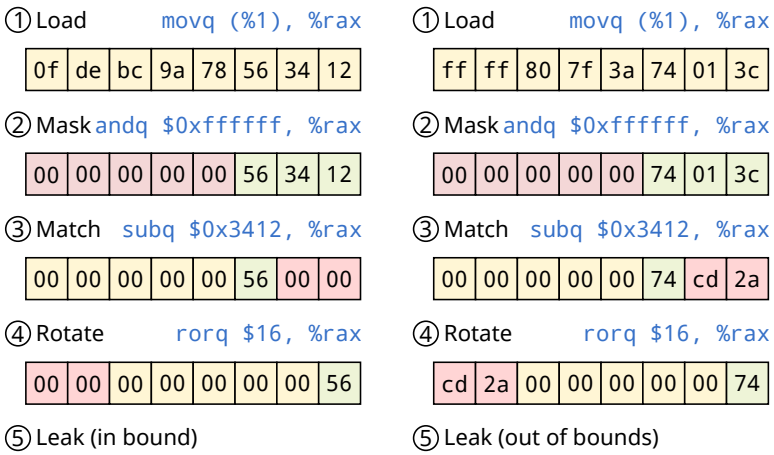


Figure 3.7: Using mask, subtract, and rotate we can selectively filter data in speculative execution to match prior observations, eliminating a large amount of noise.

As shows in Figure 3.7, ① suppose we already know part of the bytes we want to leak (either by leaking them first or knowing them through some other means). ② In the speculative path we can mask the bytes that we do not know yet. ③ By subtracting the known value, ④ and then rotating by 16 bytes, values that are not consistent with previous observations will be out of bounds in our FLUSH + RELOAD buffer, meaning we do not leak them. This technique greatly improves the observed signal.

We use this technique to develop an exploit on Linux that is able to leak the contents of the `/etc/shadow` file. Our approach involves repeatedly invoking the privileged `passwd` program from an unprivileged user. As a result the privileged process opens and reads the `/etc/shadow` file, that ordinary users cannot access otherwise. Since we cannot modify the victim to introduce a synchronization point, we repeatedly run the program and try to leak the LFB while the program reads the `/etc/shadow` file. By applying our previously discussed technique, with the additional heuristic that the leaked byte must be a printable ASCII character, we are able to leak the contents of the file even with the induced noise from creating

processes.

One observation is that the first line of the `/etc/shadow` file contains the entry for the root user. Therefore we can apply our alignment technique by fixing the first five characters to the known string `root:` to filter data from `/etc/shadow`. This approach is especially powerful as it does not require any additional information about the memory layout of the system. The attacker simply passively listens to all LFB activity, and matches the data with previous observations. As seen in Figure 3.8, we recover 26 characters (leaking 21 unknown bytes) from the `shadow` file after 24 hours. The hash entry of the root user consists of 34 characters, which leaves 8 characters (or several hours) left to leak. As this was only our initial attempt to utilize RIDL for real-world attacks, we already know we can improve the speed significantly.

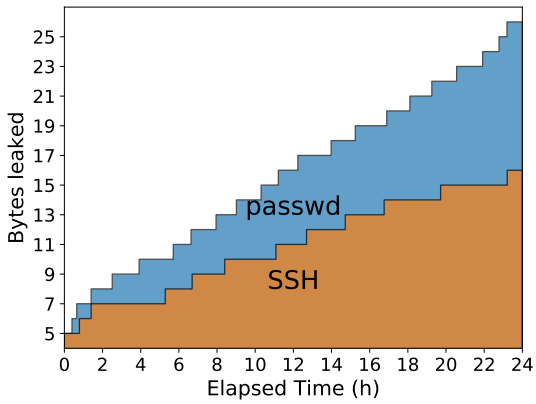


Figure 3.8: Characters leaked from the `/etc/shadow` file using the `passwd` and `SSH` attack over a period of 24 hours.

3.6.2 Cross-VM attacks

When two different virtual machines are executing simultaneously on the same physical core, a user process running inside one VM can observe in-flight data from the other VM. We verified that RIDL works on KVM [125] and even on Microsoft Hyper-V (on both Windows 10 and Windows Server 2016) with all side-channel mitigations enabled (HyperClear [92]). KVM has deployed defenses against L1TF which flush the L1D cache on `vmenter`. By default—for performance reasons—the L1D is not flushed on specific (manually) audited code paths. This defense does not hinder RIDL. In fact, flushing the L1D might actually force sensitive data to be in-flight.

We also implemented a cross-VM attack where a co-located attacker leaks the `/etc/shadow` file from the victim VM by repeatedly trying to authenticate through

SSH, confirming that virtual machine isolation does not mitigate this class of vulnerabilities. The attacker repeatedly opens a connection to the victim, trying to authenticate using invalid credentials. Similarly to the previous `passwd` attack, this strategy causes the victim to read the `/etc/shadow` file, allowing us to leak the contents. For our proof-of-concept exploit, we assume we have two co-located VMs running on co-located SMTs. We are able to retrieve 16 characters from the `passwd` file over a period of 24 hours, This is slightly slower than the previous `passwd` attack, since the execution path when SSH reads the `shadow` file is significantly longer than for the `passwd` program.

3.6.3 Kernel attacks

To verify that the privilege level does not affect our attack, we implement a user program that opens the `/proc/self/maps` file (or any other `/proc` file) and reads 0 bytes from that file. The read system call causes the kernel to generate the string containing the current mappings of the process, but copies 0 bytes to the address space of the calling program. Using the previously mentioned attacker program running on a sibling hardware thread, we are able to leak the first 64 bytes of the victim process memory mappings without these ever having been copied to user space. Our proof-of-concept exploit is able to do this reliably in a matter of milliseconds.

The kernel also provides us with a convenient target for attacks which do not require SMT. We can easily leak kernel pointers and other data stored on the stack close to the end of a system call, by executing a syscall and then performing our attack immediately after the kernel returns control to userspace. Since the kernel writes also use the LFBs, we also implemented proof-of-concept exploits that leak kernel memory writes occurring in the middle of normal execution (for example, in a `/proc` handler) in a few milliseconds. We observe the values of these writes after the kernel has already returned from the system call, as the cache lines are written back to memory via the LFB.

3.6.4 Leaking arbitrary kernel memory

RIDL can leak secrets accessed by the victim via both regular and speculative memory accesses. We demonstrate this property by implementing a proof-of-concept exploit that can leak arbitrary kernel memory. In absence of software bugs, unsanitized user-supplied pointers are never accessed by the kernel. However, speculative memory accesses are still possible. For example, we found that the function `copy_from_user()` in the Linux kernel (version 4.18) allows speculative memory accesses to user-supplied pointers. It is important to note that this attack is only possible if Supervisor Mode Access Prevention (*SMAP*) is not enabled, otherwise all accesses to user memory will be surrounded with serializing instructions

(`clac/stac`), effectively stopping speculation. Our exploit assumes that SMAP is disabled, for example due to lack of hardware/software support.

In our exploit, we use the `setrlimit()` system call to reach the `copy_from_user()` function. We start by calling `setrlimit()` multiple times with a user-land pointer to train the directional branch predictor. After the training, we call `setrlimit()` with the pointer to the kernel data we want to leak. The speculative memory access reads the data from memory, notably via the LFB, allowing us to leak it in our program after returning from the system call. To measure the performance of our exploit, we tried leaking the data from a kernel memory page (4,096 bytes) containing ASCII text. On average, leaking the entire page took us around 712 seconds, i.e., approximately 6 B/s.

3.6.5 Page table disclosure

We implemented proof-of-concept exploits to leak page table entries, since the MMU uses the LFBs to read these entries from memory at every page table walk. This discloses the physical addresses of pages in our own process, which is important information to mount other attacks (such as Rowhammer attacks [25, 45, 69, 85, 200, 212, 225, 226, 238]). This also allows us to observe page table walks and the page table entries used by a process running on a sibling hardware thread. Furthermore, by performing sliding similar to the AnC attack [82], this primitive allows us to break ASLR in a managed sandbox.

3.6.6 SGX attacks

We also verified that SGX enclaves are vulnerable to our cross-process attacks when SMT is enabled, allowing an attacker on the same physical core to leak SGX-initiated reads and writes to memory. We built SGX enclaves in pre-release mode (with debugging disabled), and successfully reproduced the cross-process experiments. Our proof-of-concept exploit trivially leaks reads and writes from a victim enclave running on a sibling hardware thread.

Our exploit can also leak the values of registers used by the enclave, since microcode reads and writes the contents of enclave registers to memory when the enclave is interrupted. By using `mprotect` to cause faults when accessing enclave pages, we repeatedly interrupt the enclave (to synchronize), allowing us to leak the contents of enclave registers. Unlike the Foreshadow attack [28], we are able to perform these attacks solely from *user space*, with no need to manipulate privileged state such as page tables. This means that SGX enclave secrets should be considered compromised on any machine where SMT is enabled, even if an attacker does not have control over the kernel or hypervisor. When an attacker is able to modify the kernel, this attack can be further improved with SGX-Step [236], using timer

interrupts to single-step through enclave code and provide a fine-grained synchronization primitive.

3.6.7 JavaScript attacks

To further demonstrate the implications of RIDL, we show that RIDL can be exploited even within restricted sandboxed environments such as JavaScript. In recent years, browser vendors have been proactively working on mitigations to protect against side-channel attacks [30, 193, 203, 244]—speculative execution side-channels, in particular. For instance, Chrome fast-forwarded the deployment of process-per-origin [30] as a mitigation against Spectre attacks. However, these mitigation efforts assume that data cannot leak across privilege boundaries, and fail to prevent in-flight data from being leaked with RIDL.

Building a RIDL attack from the browser requires a high level of control over the instructions executed by the JavaScript engine. Conveniently, *WebAssembly* allows us to generate code which meets these requirements and is available as a standard feature in modern browsers. We found that we can use WebAssembly in both Firefox and Chrome to generate machine code which we can use to perform RIDL-based attacks. Furthermore, all the major browsers try to reduce the memory footprint of the WebAssembly heap by relying on demand paging [82], which we can use to perform an attack along the lines of the one previously presented in Listing 1. That is, we can rely on the *valid* page fault generated by our memory access to trigger an exception and spill the *in-flight* data.

Generating the correct machine code and triggering the page fault are relatively straightforward. However, constructing a reliable feedback channel for speculative attacks within the browser presents some challenges. The absence of the `clflush` instruction forced our implementation to rely on an Evict + Reload channel to leak the in-flight data. Since the process of evicting entries from the L1D cache makes extensive use of the LFBs—due to TLB misses as well as filling cache lines—this adds a significant source of noise. We also need to ensure that the TLB entries for our reload buffer are still present after the eviction process, adding another source of noise to our attack. Finally, we need a reliable high-resolution timer to measure cache evictions for our Evict + Reload channel. While built-in high-resolution timers have been disabled as part of browser mitigations against side-channel attacks [30, 244], prior work has demonstrated a variety of techniques to craft new high-resolution timers [69, 82, 132], such as SharedArrayBuffer [82] and GPU-based counters [69]. The SharedArrayBuffer feature was recently re-enabled in Google Chrome, after the introduction of Site Isolation [159, 201]. Mozilla Firefox is currently working on a similar Process Isolation strategy [192].

Despite these challenges, we successfully implemented a proof-of-concept exploit on top of Firefox' SpiderMonkey JavaScript engine to reliably leak data from a

victim process running on the same system. For simplicity, our exploit uses an old-style built-in high-resolution timer in SpiderMonkey to measure cache evictions. When targeting a victim process repeatedly writing a string to memory, our exploit running in the JavaScript sandbox on a different hardware thread is capable of leaking the victim string at a rate of ~1 B/s. We also implemented a high-resolution timer in Chrome using WebAssembly threads which provided sufficient accuracy for our EVICT + RELOAD channel. At the time of writing, any site can opt into this functionality using the 'origin trials' system. Although we do not currently have a reliable RIDL exploit running inside unmodified Chrome, we believe that our results already cast doubt on the effectiveness of site isolation as a mitigation against side-channel attacks.

3.7 Speculative execution attacks

Since Horn [95] initially reported this new class of speculative execution side-channel attacks, researchers started digging into modern processor microarchitectures to spot the next generation of vulnerabilities. The result is a plethora of new attacks and attack vectors [28, 38, 124, 128, 135, 149, 156, 249].

The taxonomy of these attacks is confusing (at best) since *attacks* and *attack vectors* oftentimes have been reported as equivalent and frequently interchanged. In this section, we try to shed some light on the topic describing similarities and differences among the different classes of attacks and categorizing them based on their nature, capabilities, and constraints. We summarize our categorization in Table 3.2. We divide the currently existing attacks based on the nature of their speculation: *control speculation* vs. *data speculation*. We further introduce a sub-categorization of data speculation attacks, which we define as *exception deferral* attacks (e.g., RDCL and L1TF).

3.7.1 Control speculation

Control speculation can be triggered in multiple ways. In Section 3.2, we already described *Out-of-Order execution* and *Transactional Synchronization eXtensions* explaining how these trigger speculative execution. Here we focus on the three main forms of control instructions that can be speculated upon: ① direct branches, ② indirect branches and calls, and ③ return instruction.

Direct branches: Direct (or conditional) branches are optimized in hardware by the *Branch Prediction Unit* (BPU). This unit keeps track of the previous outcomes of a conditional branch in order to predict which code path will be taken, and the out-of-order execution engine then continues execution along the predicted path. Mistraining the BPU allows attacks known as *Bounds Check Bypass* (BCB) [124, 128], such as the one in Listing 2.

```
if (x < arr_size)
    y = probeTable[arr[x] * 4096];
```

Listing 2: An example of Bounds Check Bypass.

An attacker who controls the variable `x` can mistrain the conditional branch to always take the `if` code path. When an out-of-bounds `x` is later passed to his code, the BPU will speculate on the code path to take, resulting in a speculative OoB access which can be leaked through a cache-based covert channel. A variant of this attack targets bounds check bypass on stores (BCBS) [124], which shows the issue is not limited to speculative loads.

Indirect branches and calls: These branches are also targets of speculative execution optimizations. The *Branch Target Buffer* (BTB) is a unit embedded in modern CPUs that stores a mapping between the source of a branch or call instruction and its likely destination. An attacker running on the same physical core of the victim can poison the BTB to perform attacks known as *Branch Target Injection* (BTI). The attacker pollutes the buffer by injecting arbitrary entries in the table to divert the victim's indirect branches to the target of interest—within the victim address space. No checks are performed on the `pid`, hence the possibility of cross-process mistraining. This makes it possible to build *speculative* code-reuse (e.g., ROP) attacks to leak data. Branch target injection has been demonstrated effective to escape sandboxed environments (e.g., JavaScript sandbox) [128], to build cross-process attacks [128] and to leak data from SGX enclaves [38].

Return speculation: The use of the BTB is inefficient for the prediction of *return* instructions, as functions may have many different call sites. Therefore, modern processors employ a *Return Stack Buffer* (RSB), a hardware stack buffer to which the processor pushes the return address whenever it executes a *call* instruction. Whenever the processor stumbles upon a *return* instruction, it pops the address from the top of the RSB to predict the return point of the function, and it executes the instructions along that path speculatively. The RSB misspeculates when the return address value in the RSB does not match the one on the software stack. Unfortunately, the RSB consists of a limited number of entries and employs a round robin replacement policy. As a result, an attacker can overflow the RSB to overwrite the "alleged" return address of a function and speculatively execute the code at this address. Researchers have reported RSB attacks against sandboxes and enclaves [135, 156].

Constraints: Control speculation attacks, while powerful, are only effective in intra-address space attacks (e.g., sandboxes). Furthermore, their exploitation requires (some) cooperation from the victim's code. In situations where the attacker can generate code inside the victim (e.g., JIT compilation inside a browser, eBPF

in the Linux kernel) it is easy to meet this constraint. In the other cases (e.g., enclaves or Linux kernel without eBPF), this constraint is harder to meet. The attacker needs to mount *confused-deputy* attacks that lure the victim component into speculatively executing specific "gadgets", making exploitation more difficult. Perhaps more importantly, this class of attacks can be mitigated in software using either compiler support for emitting safe code or manually stopping speculation when deemed dangerous.

3.7.2 Data speculation

Data speculation is the second type of speculative execution. This type of speculation does not divert the control flow of the application, but instead speculates on the value to be used. As discussed in Section 3.7.1, manipulating control flow is not enough to build effective cross-privilege and cross-address space attacks. Attackers can overcome these constraints by taking advantage of data speculation.

Speculative Store Bypass (SSB) [96] takes advantage of the address prediction performed by the *Memory Disambiguator*. This unit is in charge of predicting read-after-write hazards. If the prediction fails, the attacker may be able to leak stale L1 cache lines previously stored at that address. However, this attack provides a small window of exploitation and works only within intra-address space boundaries, making it hard to exploit in practice.

Exception deferral: To bypass this limitation and allow cross-boundary leaks, researchers identified a new class of data speculation attacks that we refer to as *exception deferral* attacks. A similar distinction was previously made in the literature [124] under the nomenclature of *exception speculation*. However, as explained in Section 3.2, speculation represents a misleading terminology of the actual issue under scrutiny. The CPU does not perform any type of speculation on the validity of the operation. It simply executes instructions speculatively out-of-order and eventually retires them in-order. The flaw of this design is that the Retirement Unit is officially in charge of handling CPU exceptions. Thus, an attacker can perform loads that trigger exceptions (e.g., *Page Faults*) during the speculative execution window, but such loads will not fault until the Retirement Unit performs the compulsory checks.

Multiple attacks have exploited CPUs' exception deferral in order to circumvent different security checks. RDCL [149] (known as Meltdown) and RSRR [108] exploited the deferral of a *page fault* (`#PF`) exception caused by the presence of the supervisor bit in order to read privileged kernel memory and *Model Specific Registers* (MSRs). LazyFP [220] took advantage of the deferral of the *Device not available* (`#NM`) exception to leak *Floating Point Units* (FPUs) register state and break cryptographic algorithms. Foreshadow [249] disclosed how the deferral of a *Terminal Fault* (`#TF`) generated by a failed check on the present or reserved bits of a PTE

allows leaking arbitrary contents from the L1 cache. Given that Foreshadow operates on physical memory addresses, it can leak information across privilege and address space boundaries breaking kernel, enclaves, and VM isolation. Crafting Foreshadow attacks, however, requires control over addresses residing in PTEs as we discuss next.

Constraints: Data speculation allows attackers to operate on data they are not supposed to have access to. Furthermore, when combined with exception deferral, they gain the capability of not relying on victim code to leak data. With RDCL, for example, attackers can directly read from kernel memory without relying on "gadgets" in the kernel code. Most of these attacks, however, are still limited by the necessity of a valid address translation. That is, a valid (and known) address to leak the data from. For instance, in the case of Foreshadow, the attacker can theoretically read any arbitrary cache line in the L1d cache. However, since L1d cache lines are physically tagged, the attacker needs control over virtual-to-physical address mappings (PTEs). This constraint is easily met in situations where the attacker controls these mappings, such as inside guest VMs or SGX enclaves. In the other cases, this constraint is harder to meet, such as when attacking the kernel or another user process.

3.7.3 Comparing with RIDL

While RIDL still falls under the umbrella of data speculation attacks, it presents a unique feature that makes it stand out among the other attacks: the ability to induce leaks that are completely agnostic to address translation. All the other attacks other than LazyFP [220] (which is limited to leaking stale floating point registers) require a valid address for performing tag checks before retrieving the data. If this check fails, the speculation aborts. On the other hand, in the case of RIDL, the attacker can access any *in-flight* data currently streaming through internal CPU buffers without performing any check. As a result, address space, privilege, and even enclave boundaries do not represent a restriction for RIDL attacks.

3.8 Existing defenses

In response to the plethora of attacks described in Section 3.7, hardware and software vendors have been struggling to catch up with mitigations that can safeguard vulnerable systems. In this section, we perform an exhaustive analysis of all the existing state-of-the-art mitigations, pinpointing the current shortcomings of such solutions when applied to the RIDL family of attacks.

These mitigations can operate at three different layers: ① inhibiting the trigger of the speculation, ② protecting the secret the attacker is trying to disclose, or ③ disrupting the channel of the leakage. We focus on the first two classes, which

are specific to speculative execution attacks; the third typically applies to any timing side-channels (e.g., disabling high-precision timers in browsers [30]). We summarize all the currently deployed mitigations and their effects on currently-known attacks in Table 3.3.

3.8.1 Inhibiting the trigger

To protect against control speculation attacks, vendors have released mitigations that prevent the hardware from executing (speculatively) unsafe code paths. For instance, Intel released a microcode update with three new capabilities: IRBS, STIBP and IBPB to prevent indirect branch poisoning instructions and to protect against BTI attacks [104]. Another suggested mitigation uses the `lfence` instruction to restrict control speculation. This can be applied as a compiler-based defense, mitigating multiple families of attacks. ① To protect against BCB attacks, the compiler inserts an `lfence` instruction after conditional branches to stop the BPU from speculating on the code path taken. ② To protect against BTI attacks, the `lfence` instruction is introduced as part of the *Retpoline* [232] mitigation. Researchers have also suggested extending Retpoline to guard `ret` instructions and prevent RSB attacks [156]. The Retpoline mitigation converts each indirect jump into a direct call to a stub function, that returns to the destination of the initial indirect branch. This is achieved by altering the stack, replacing the return address of the function. Since return instructions also trigger speculation, an `lfence` loop is inserted at the expected return site of the stub, to inhibit further code execution. Retpoline can also perform RSB filling [251]. This is required for Intel architectures newer than Haswell where, in case of an empty RSB, the BTB provides the speculation address.

Software mitigations such as Retpoline do not apply for data speculation attacks since there is no need to (speculatively) divert the control flow of the application. As such, most defenses against data speculation have been in the form of microcode updates, such as:

- SSBD: *Speculative Store Bypass Disable* adds an MSR which can be used to prevent loads from executing before addresses of previous stores are known [111].
- RSRR fix: Intel's mitigation for Rogue System Register Reads patches `rdmsr` to avoid speculative L1 loads of MSR data for unprivileged users [108].

Finally, to protect against LazyFP, it suffices to enable Eager FPU context switching. This restores FPU register state when performing a context switch, preventing speculative execution on stale register contents [220].

3.8.2 Protect the secret

When preventing the CPU from speculating becomes unfeasible, the other solution is to conceal the sensitive information from the attacker. Defenses falling under this category are clearly context sensitive. That is, they highly depend on the environment they get deployed on since different environments secure different secrets. A primary example is *Kernel Page Table Isolation* (KPTI) [136]. KPTI was effectively the first mitigation deployed against speculative execution attacks and was introduced to protect the Kernel against RDCL (i.e., Meltdown) by separating kernel and user address spaces.

A similar compartmentalization approach was then deployed in other environments. ① Array index masking was deployed in the kernel and in sandboxed environments to protect against intra-address space BCB attacks. ② Multi-process isolation, similarly to KPTI, protects sandboxed environments from cross–domain attacks (e.g., JavaScript VMs) by generating a different process for every origin—hence a different address space.

These mitigations were considered effective until the disclosure of the Foreshadow attack. Foreshadow relaxed the requirement of victim cooperation for cross-address space attacks by leaking any data present in the L1d cache. Protecting against this new class of attacks requires stronger solutions targeting physical addresses. Two instances of such mitigations are PTE inversion and L1d flush [99]. PTE inversion protects kernel and enclave memory from being leaked through the L1d cache by scrambling the physical address in the PTE when a page is marked as non-present. L1d flush removes any secret information from the cache during a context switch, making it impossible to retrieve any data. The latter is part of a set of mitigations intended for environments such as the cloud, where an attacker may have control of PTE contents.

Another example of such solutions is HyperClear [92], deployed by Microsoft to safeguard Hyper-V. The mitigation consists of three units: ① The *Core Scheduler*, which performs safe scheduling of sibling logical processors by allocating resources for a single VM on the same physical processor. ② Address space isolation per virtual processor, which limits the hypervisor access to memory only belonging to the VMs running on the same physical core—preventing cross-VM leaks. ③ Sensitive data scrubbing, which protects nested hypervisors from leaking sensitive data. This is done by zeroing the latter before switching VMs and avoiding the performance impact of a complete L1d flush. Similar solutions have been deployed on other hypervisors such as KVM [147].

3.8.3 Defenses vs. RIDL

In Section 3.6, we reported the results of all our proof-of-concept exploits on fully patched systems with the latest microcode updates. As the positive results demon-

strate, the currently deployed mitigations fail to protect against RIDL attacks. As we discussed in Section 3.8.1, mitigations for data speculation attacks usually rely on microcode patches. Since the existing defenses trying to inhibit speculation do not account for the Line Fill Buffer, RIDL is not impacted by any of them.

On the other hand, defenses aiming at protecting the secret fail at defending from RIDL attacks for a different reason: they all consider a valid address a strict requirement. RIDL demonstrates that not all the sources of data speculation rely on this assumption. Our results for the i9-9900K show the risk of relying on "spot" mitigations in hardware; although the address-based page faults used by Meltdown-style attacks have been mitigated in silicon, RIDL attacks using other exceptions continue to work. Furthermore, it demonstrates for the first time a cross-address space and cross-privilege attack that relies only on in-flight, CPU-internal data, demonstrating the latent danger introduced by the related microptimizations.

3.9 New Mitigations

The response to the disclosure of speculative execution attacks has so far been the deployment of spot mitigations in software before mitigations become available in hardware [114]. For example, for Meltdown, the first deployed software mitigation (i.e., KPTI) was the separation of address spaces between user space and kernel space by the operating system. While effective, on top of increasing complexity in the kernel, KPTI has been shown to have performance penalties under certain workloads [97]. We now describe how this spot mitigation approach is not well-suited for the LFB variant of RIDL presented in this chapter.

Mitigating RIDL in software. Since sensitive information can be leaked from sibling hardware threads, it is clear that SMT must be disabled to mitigate RIDL. However, it is still possible to leak sensitive information from another privilege level within a single thread (as some of our exploits demonstrated), including information from internal CPU systems such as the MMU. To protect sensitive information in the kernel or a different address space, the kernel needs to flush the LFBs before returning to userland similar to the L1 flush in the Foreshadow mitigation. Similarly, the hypervisor needs to flush the LFBs before switching to VM execution. In the case of hardware-based components such as SGX or the MMU, the LFB flushing cannot be easily done in software.

Perhaps more importantly, while Intel could provide a L1 flush mechanism via a microcode update, it is not clear whether it is possible to expose a similar mechanism for flushing the LFBs. Furthermore, even if such a mechanism was possible, its cost will likely be even more expensive than the L1 flush on every context switch. Remember that the entire L1 cache needs to be flushed first since the entries go through the LFBs. After that, the mechanism needs to wait until the LFBs are drained before safely resuming the execution. We believe that such a mecha-

nism will be too expensive to be useful in practice.

Moving forward. In this chapter, we focused on speculation done on LFB entries. However, we believe there are several other sources of in-flight data—especially given decades of performance optimizations in the CPU pipeline. Furthermore, as discussed in this section, because these optimizations are applied deeply in the CPU pipeline, spot mitigations will likely be expensive. Moving forward, we see two directions for mitigating RIDL: 1. As Intel could release a microcode update that mitigated SSB by completely disabling speculative store forwarding, we believe it should make a similar mitigation possible for all possible sources of speculation when applying micro-optimizations. It will then be up to system software to decide which optimizations to turn off until hardware mitigations become available. 2. Finding all instances of RIDL will likely take a long time due to the complexity of these micro-optimizations. Hence, rather than spot mitigations that are often ineffective against the next discovered attack, we need to start the development and deployment of more fundamental mitigations against the many possible classes of speculative execution attacks.

3.10 Conclusion

We presented RIDL, a new class of speculative execution vulnerabilities able to leak arbitrary, address-agnostic in-flight data from normal execution (without branches or errors), including sandboxed execution (JavaScript in the browser). We showed RIDL can be used to perform attacks across arbitrary security boundaries and presented real-world process-, kernel-, VM-, and SGX-level exploits. State-of-the-art mitigations against speculative execution attacks (including the in-silicon mitigations in Intel's recent CPUs) are unable to stop RIDL, and new software mitigations are at best non-trivial. RIDL puts into question the current approach of "spot" mitigations for individual speculative execution attacks. Moving forward, we believe we should favor more fundamental "blanket" mitigations over these per-variant mitigations, not just for RIDL, but for speculative execution attacks in general.

Disclosure

The authors from VU Amsterdam (VUSec) submitted PoC exploits for the RIDL class of vulnerabilities to Intel on September 12, 2018. Intel immediately acknowledged the vulnerability and rewarded RIDL with the Intel Bug Bounty (Side Channel) Program. Since then, Intel led the disclosure process, notifying all the affected software vendors and other hardware vendors potentially susceptible to similar issues (see details below). VUSec submitted the end-to-end analysis presented in this chapter including all the exploits (except the one in Section 3.6.4) to IEEE Symposium on Security & Privacy on November 1, 2018.

Giorgi Maisuradze independently discovered the same class of vulnerabilities in June 2018 as an intern in a side-channel project at Microsoft Research. The findings were reported to Intel via the Microsoft Security Response Center. Section 3.6.4 is entirely based on his findings.

Volodymyr Pikhur independently discovered and reported a RIDL-class exploit (L1TF mitigation bypass over uncached memory) to Intel on August 25, 2018. Dan Horea Lutas' team at Bitdefender reported an issue related to the RIDL vulnerabilities to Intel on August 17, 2018.

Statements that we received from CPU vendors about RIDL are available in Appendix 3.12.

3.11 Appendix

3.12 Statements from CPU vendors

3.12.1 Statement from Intel

"We have disclosed details about the issue described in VU Amsterdam's paper with multiple parties in the computing ecosystem who have the ability to help develop mitigations. This includes operating system vendors such as MSFT and Redhat, hypervisor vendors such as VMWare and Citrix, silicon vendors or licensors such as AMD and ARM, select operating system providers or maintainers for open source projects, and others. These disclosures have been conducted under coordinated vulnerability disclosure for purposes of architecting, validating, and delivering mitigations, and all parties agreed to mutual confidentiality and embargo until 10AM PT May 14, 2019".

3.12.2 Statement from AMD

"After reviewing the paper and unsuccessfully trying to replicate the issue, AMD believes its products are not vulnerable to the described issue".

3.12.3 Statement from ARM

"After reviewing the paper and working with architecture licensees we are not aware of any Arm-based implementations which are affected by this issue. We thank VU Amsterdam for their research".

3.13 Extended results

Figure 3.9 shows a more complete diagram of the Intel Skylake microarchitecture.

Figure 3.10 shows a screenshot of our tool to test for existing vulnerabilities as well as RIDL, to check what mitigations are available and enabled and to provide a general overview including the installed microcode version. We will release this tool as open source on May 14, as well as provide binaries of this tool for various platforms including Microsoft Windows and Linux.

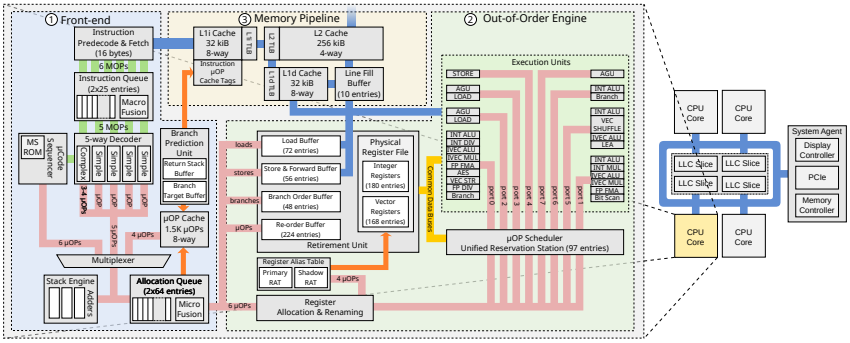


Figure 3.9: A full overview of the Intel Skylake microarchitecture.

3.14 Addenda

3.14.1 Addendum 1

Abstract: On Nov 12, 2019, we disclose *TSX Asynchronous Abort* (TAA), a "new" speculation-based vulnerability in Intel CPUs as well as other MDS-related issues. In reality, this is no new vulnerability. We disclosed TAA (and other issues) as part of our original RIDL submission to Intel in Sep 2018. Unfortunately, the Intel PSIRT team missed our submitted proof-of-concept exploits (PoCs), and as a result, the original MDS mitigations released in May 2019 only partially addressed RIDL.

At the request of Intel, and to protect their users, we redacted parts of the original RIDL paper and did not release the RIDL test suite with our PoCs on the MDS disclosure date (May 14, 2019). This addendum provides an analysis of Intel's original (flawed) MDS mitigation and an explanation for the "Misaligned Read" and the "TSX" columns in Table I, which we redacted from the original RIDL paper. Additional updated information on RIDL, TAA, the disclosure process, our now public test suite and TAA-optimized exploits can be found at `https://mdsattacks.com`.

Flawed MDS mitigation

Intel's original microcode update, which modifies the VERW instruction to clear CPU buffers and mitigate MDS, is flawed in that it clears the buffers using stale (potentially sensitive) data on several of the CPUs we used for testing (e.g., i7-7700K). Intel states this bug is only present in Skylake client CPUs. This means that data can be leaked across privilege boundaries using RIDL even if SMT has been disabled and the recommended VERW mitigation has been applied.

The non-microcode versions of the mitigation provided in the MDS whitepa-

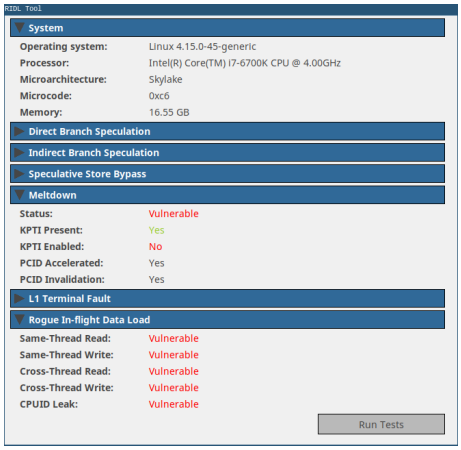


Figure 3.10: A screenshot of our tool to test for vulnerabilities and mitigations including RIDL.

per[4] appear to correctly clear the CPU buffers, but at a much higher performance cost. Our RIDL paper originally reported the intended behavior of Intel's mitigation. Unfortunately, at Intel's request, we had to withhold any comment on the flawed mitigation from the paper, in order to comply with the second embargo. The new microcode recently released by Intel still does *not* fix the issue, as we still see leaks with RIDL PoCs shared with Intel in May.

TSX Asynchronous Abort

TSX transactions can be aborted by for instance flushing a cache line before the transaction, then loading from the same cache line inside the transaction. This causes the processor to abort the transaction despite execution of instructions in the pipeline continuing until retirement, allowing information to be leaked via the load from various internal CPU buffers—including store buffers—using RIDL. Intel refers to this RIDL variant as the *TSX Asynchronous Abort* (TAA) vulnerability.

This vulnerability is present even on CPUs which Intel claims are not vulnerable to MDS, such as recent Cascade Lake CPUs. Although no microcode MDS mitigations were available on these CPUs when MDS was disclosed in May, Intel recently (September 2019) provided microcode updates. We believe this vulnerability can be mitigated by disabling TSX. Our original RIDL paper reported results for TAA for a variety of CPUs in the "TSX" column in Table 1; we withheld the explanation at Intel's request to comply with the second embargo.

[4] `https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-microarchitectural-data-sampling`

Alignment faults

Alignment faults (e.g., due to the AC flag or aligned vector instructions) can be used to cause exceptions and perform RIDL attacks. Although this vulnerability is not mitigated by the silicon fixes for Meltdown/MFBDS (page faults), it appears to be mitigated on Intel's latest CPUs. Alignment faults and split loads across cache lines can be used to leak data from a variety of sources, including load ports (as originally reported by Intel) and (indirectly) also store and fill buffers.

Our original RIDL paper reported results for both split loads and alignment faults for a variety of CPUs in the "Misaligned read" column in Table 1. Such results showcased leaks that were not explained by Intel's original MDS whitepaper (e.g., store-to-load leaks). Since then, Intel's whitepaper has undergone a number of updates. Again, at Intel's request, we withheld a full explanation of our results from the paper.

Conclusion

This research—whose details were withheld from the public version of the RIDL paper due to responsible disclosure considerations—further supports the arguments presented in our original paper. As demonstrated by the TAA vulnerability (still present in recent Intel CPUs) and the flawed MDS mitigation, RIDL-class vulnerabilities are non-trivial to fix or mitigate, and current "spot" mitigation strategies for resolving these issues are questionable. Moreover, we question the effectiveness of year-long disclosure processes and also raise concerns on their disruptive impact on the academic process. We continue to work with Intel to improve their coordinated disclosure process and collaboration with academia.

3.14.2 Addendum 2

Abstract: On Jan 27, 2020, we (VUSec) disclose two "new" RIDL/MDS variants at the end of another (third) embargo. We do not think either of these variants is particularly novel or interesting and they are simply "more RIDL" (focusing on ways to get data into microarchitectural buffers RIDL can leak from).

Flawed MDS mitigations (encore!)

Intel's original microcode update, which modifies the VERW instruction to clear CPU buffers and mitigate MDS, was flawed in that it cleared the buffers using stale (potentially sensitive) data on several of the CPUs we used for testing (e.g., i7-7700K). This meant that data could still be leaked across privilege boundaries using RIDL even if SMT has been disabled and the recommended VERW mitigation has been applied. See our first addendum (on TAA and VERW-bypass) for more details.

Then in November 2019, Intel published another set of updates to mitigate the RIDL/MDS vulnerability once and for all. Unfortunately, they failed again, as the fix still leaves multiple avenues for exploitation. None of these issues is new. They merely support the claim in the original paper that there are multiple ways to and multiple buffers to leak from. One of the issues (L1DES) was insisted on by us ever since we first shared the RIDL pre-final paper in January 2019. The other issue is a one-line of code change from one of the PoCs we originally submitted.

L1D Eviction Sampling (L1DES)

The first issue, which Intel refers to as L1D Eviction Sampling (L1DES), is a RIDL variant that leaks from L1D evictions (assigned CVE-2020-0549). It may seem that sometimes history does repeat itself, because this is again something that we had already shown in our original RIDL paper, as shown in Figure 6. In the camera-ready version of the RIDL paper, we also explicitly mentioned that, at every context switch, "the entire L1 cache needs to be flushed first since the entries go through the LFBs" to properly mitigate RIDL. We removed this sentence in the original version of the RIDL paper released on May 14, 2019, since Intel hadn't yet mitigated the RIDL variant based on L1D evictions (which would eventually become L1DES). Since then, we spent months trying to convince Intel that leaks from L1D evictions were possible and needed to be addressed.

On Oct 25, 2019, we reported to Intel that this variant would bypass their latest VERW mitigation (and so did a PoC shared with Intel on May 10, 2019), resulting in Intel finally acknowledging the L1D eviction issue and requesting another (L1DES) embargo. We learned that Intel had not found this issue internally and that the only other independent finder was the Zombieload team, which reported a PoC to Intel in May, 2019. Our RIDL-L1DES PoC is available on GitHub[5].

Vector Register Sampling (VRS)

The second issue, which Intel refers to as Vector Register Sampling (VRS), is a RIDL variant that leaks from vector registers (assigned CVE-2020-0548). This variant shows that RIDL can also leak values that are never even stored in memory. In reality, this is possible with a small, 1-line of code variation of our 'alignment write' PoC (originally leaking values stored in memory using alignment faults), which we shared with Intel on May 11, 2019. Since then, we spent months trying to convince Intel that our 'alignment write' PoC and its variations needed to be properly addressed.

On Oct 1, 2019, we reported to Intel that a 1-line modification of our 'alignment write' PoC can leak vector register values, resulting in Intel requesting a new (VRS)

[5] `https://github.com/vusec/ridl`

embargo. We are not aware of other independent finders to acknowledge for VRS. Our RIDL-VRS PoC is available on GitHub.

Conclusion

This research—some of whose details were withheld from the public version of the RIDL paper due to responsible disclosure considerations—further supports the arguments presented in our original paper. We reiterate that RIDL-class vulnerabilities are non-trivial to fix or mitigate, and current "spot" mitigation strategies for resolving these issues are questionable. Moreover, we question the effectiveness of year-long disclosure processes and also raise concerns on their disruptive impact on the academic process. We continue to work with Intel to improve their coordinated disclosure process and collaboration with academia.

Table 3.1: Our results for 15 different microarchitectures and the measured bandwidth across security domains.

CPU	Year	Microcode	Page Fault		Misaligned Read		TSX		SGX		Bandwidth(B/s)			
			ST	XT	ST	XT	ST	XT	ST	XT	XPC	SP‡	XVM	SGX
Intel Xeon Silver 4110 (Skylake SP)	2017	0x200004d	R/W	R/W	R/W	R/W	✓	✓	—	—	45k	25k	3k	—
Intel Core i9-9900K (Coffee Lake R)	2018	0x9a	✗	✗	R/W	R/W	✓	✓	✓	✓	71k	48k	10k	8k
Intel Core i7-8700K (Coffee Lake)	2017	0x96	R/W	R/W	R/W	R/W	✓	✓	✓	✓	54k	49k	46k	65k
Intel Core i7-7800X (Skylake X)	2017	0x200004d	R/W	R/W	R/W	R/W	✓	✓	—	—	37k	36k	31k	—
Intel Core i7-7700K (Kaby Lake)	2017	0x8e	R/W	R/W	R/W	R/W	✓	✓	✓	✓	65k	46k	63k	114k
Intel Core i7-6700K (Skylake)	2015	0xc6	R/W	R/W	R/W	R/W	✓	✓	✓	✓	68k	20k	76k	83k
Intel Core i7-5775C (Broadwell)	2015	0x1e	R/W	R/W	R/W	R/W	✓	✓	—	—	21k	16k	27k	—
Intel Core i7-4790 (Haswell)	2014	0x25	R/W	R/W	R/W	R/W	—	—	—	—	100	50	110	—
Intel Core i7-3770K (Ivy Bridge)	2012	0x20	R/W	R/W	R/W	R/W	—	—	—	—	92	41	89	—
Intel Core i7-2600 (Sandy Bridge)	2011	0x2e	R/W	R/W	R/W	R/W	—	—	—	—	107	73	106	—
Intel Core i3-550 (Westmere)	2010	0x07	R/W	R/W	R/W	R/W	—	—	—	—	1k	245	1k	—
Intel Core i7-920 (Nehalem)	2008	0x12	R/W	R/W	R/W	R/W	—	—	—	—	79	32	70	—
AMD Ryzen 5 2500U (Raven Ridge)	2018	0x810100b	✗	✗	✗	✗	—	—	—	—	—	—	—	—
AMD Ryzen 7 2600X (Pinnacle Ridge)	2018	0x800820b	✗	✗	✗	✗	—	—	—	—	—	—	—	—
AMD Ryzen 7 1600X (Summit Ridge)	2017	0x8001137	✗	✗	✗	✗	—	—	—	—	—	—	—	—

ST = Same-Thread, XT = Cross-Thread, SP‡ = Supervisor post-KPTI, XPC = Cross-process, XVM = Cross Virtual Machines

Table 3.2: List of currently disclosed attacks categorized by nature, capabilities and constraints. A checkmark (✓) under **capabilities** reports an attack demonstrated in the literature. A checkmark under the **constraints** represents a requirement to perform the attack. We report supervisor in both Intra/Cross- address space scenarios both pre- and post- KPTI [136].

Attacks	**Leak cause**	**Exception deferral**	**Capabilities**						**Constraints**	
			Intra-address space			Cross-address space			Victim cooperation	Valid address translation
			SB	SP†	SGX	XPC	SP‡	XVM		
Control Speculation										
BCB{S} [124, 128]	Direct branch	—	✓	—	—	—	—	—	✓	✓
BTI [38, 128]	Indirect branch	—	✓	—	✓	—	—	—	✓	✓
RSB [135, 156]	Return stack	—	✓	—	✓	—	—	—	✓	✓
Data Speculation										
SSB [96]	Memory Disambiguation	—	✓	✓	—	—	—	—	✓	✓
RDCL [149] RSRR [108]	L1D	✓	✓	✓	—	—	—	—	—	✓
LazyFP [220]	FPU	✓	—	—	—	✓	—	✓	—	—
L1TF [249]	L1D	✓	✓	✓	✓	—	✓	✓	—	✓
RIDL	LFB		✓	✓	✓	✓	✓	✓	—	—

SB = Sandbox, SP† = Supervisor pre-KPTI, SP‡ = Supervisor post-KPTI, XPC = Cross-process, XVM = Cross Virtual Machines

RIDL

Table 3.3: List of existing mitigations against currently disclosed speculative execution attacks grouped based on the nature of the defense.

Attacks	Inhibit trigger							Hide Secret					
	LFENCE [104]	IRBS, IBPB STIBP [104]	SSBD [111]	RSRR Fix [108]	Retpoline [232]	RSB Filling [251]	Eager FPU [220]	KPTI [136]	Array Index Masking [190]	Multi-Process Isolation [30]	LID Flushing [99]	PTE Inversion [99]	HyperClear [92]
Control Speculation													
BCB{S} [124, 128]	G	—	—	—	—	—	—	—	G	SB	—	—	—
BTI [38, 128]	—	G	—	—	G	G	—		—	SB	—	—	—
RSB [135, 156]	—	—	—	—	—	—	—	—	—	SB	—	—	—
Data Speculation													
SSB [96]	G	—	G	—	—	—	—	—	—	SB	—	—	—
RDCL [149]	—	—	—	—	—	—	—	SP†	—	—	—	—	—
RSRR [108]	—	—	—	G	—	—	—	SP†	—	—	—	—	—
LazyFP [220]	—	—	—	—	—	—	G	—	—	—	—	—	—
L1TF [249]	—	—	—	—	—	—	—	—	—	—	G	SB, SGX	XVM
RIDL	—	—	—	—	—	—	—	—	—	—	—	—	—

G = Generic, SB = Sandbox, SP† = Supervisor pre-KPTI, XPC = Cross-process, XVM = Cross Virtual Machines,

4 CrossTalk: Speculative Data Leaks Across Cores Are Real

Recent transient execution attacks have demonstrated that attackers may leak sensitive information across security boundaries on a shared CPU core. Up until now[1], it seemed possible to prevent this by isolating potential victims and attackers on separate cores. In this chapter, we show that the situation is more serious, as transient execution attacks can leak data across different cores on many modern Intel CPUs.

We do so by investigating the behavior of x86 instructions, and in particular, we focus on complex microcoded instructions which perform offcore requests. Combined with transient execution vulnerabilities such as Micro-architectural Data Sampling (MDS), these operations can reveal internal CPU state. Using performance counters, we build a profiler, CrossTalk, to examine the number and nature of such operations for many x86 instructions, and find that some instructions read data from a *staging buffer* which is shared between all CPU cores.

To demonstrate the security impact of this behavior, we present the first cross-core attack using transient execution, showing that even the seemingly-innocuous CPUID instruction can be used by attackers to sample *the entire* staging buffer containing sensitive data – most importantly, output from the hardware random number generator (RNG) – *across cores*. We show that this can be exploited in practice to attack SGX enclaves running on a completely different core, where an attacker can control leakage using practical performance degradation attacks, and demonstrate that we can successfully determine enclave private keys. Since existing mitigations which rely on spatial or temporal partitioning are largely ineffective to prevent our proposed attack, we also discuss potential new mitigation techniques.

[1] Or rather, up until publication of this work in 2020.

4.1 Introduction

Recent research into transient execution vulnerabilities[2] has shown that more attention should be paid to the internal details of CPU pipelines. Meltdown [149], Spectre [128], Foreshadow [28], ZombieLoad [211] and RIDL [240] collectively demonstrated direct information leakage across any and all security domains supported by modern CPU cores. This is due to the transient execution performed by modern CPU pipelines, which allows an attacker to observe side-effects of transiently executed code. Mitigations include hardware updates, microcode updates, operating system updates, and user-level defenses but they have been costly [83, 143] and incomplete [128, 240]. So far these attacks have required the attacker and victim to share the same core, fueling the belief that isolating different security domains on their own cores would prevent these transient execution attacks – leaving us only with well-understood timing attacks on shared resources such as caches. Various scheduling mechanisms in operating systems and hypervisors follow this belief and isolate different security contexts on their own cores [46, 100, 163]. In this chapter, we challenge this belief and show that sensitive information leaks across cores in modern Intel CPUs, via a staging buffer that is shared across cores.

To investigate the leakage surface of transient execution across cores, we build CrossTalk, a framework for identifying and profiling x86 instructions in different contexts. Unlike previous work [2] which characterizes the performance of instructions, CrossTalk executes instructions in a variety of different contexts (most importantly, with different operands), which allows us to investigate a wider range of instruction behavior, and collects data from a wider range of performance counters. This led us to a number of interesting observations: most importantly, the existence of a global (cross-core) shared staging buffer in a variety of Intel processors that retains information from previously executed instructions. We explore this using the second phase of CrossTalk, which uses the recently discovered MDS transient execution vulnerabilities [211, 240] to further investigate the nature of these instructions by observing which instructions modify the buffer, and leaking the data they leave behind in this buffer.

In more detail, the CrossTalk analysis focuses on x86 instructions with non-trivial behavior, which we found to be decoded to multiple micro-ops. Micro-ops for Intel processors are undocumented and have, as of yet, received relatively little scrutiny from the security community. The number and nature of these micro-ops depend on the context of the instruction (such as the operands provided), and in some of these situations, they perform offcore reads and writes using internal CPU interconnects. Two examples are the RDMSR and WRMSR instructions, which allow privileged code to read from and write to model-specific registers. We

[2] also known as speculative execution vulnerabilities

also found this behavior in instructions typically available to userspace — such as CPUID, RDRAND and RDSEED. Most crucially, we observed that Intel CPUs perform reads from certain CPU-internal sources using a shared 'staging' buffer. The contents of this buffer are visible to *any* core on the system that can execute these instructions—including non-privileged userspace applications within a virtual machine.

The security implications of this behavior are serious, as it allows attackers to mount transient execution attacks *across* CPU cores, which implies that mitigations separating security domains at the granularity of cores are insufficient. Although our attacks do not expose the contents of memory or registers, we exemplify the threat posed by this shared staging buffer by implementing a cross-core attack for leaking random numbers generated via the RDRAND and RDSEED instructions. We show that we can exploit this in practice against SGX enclaves, which are amenable to practical performance degradation attacks. The leak allows attackers to observe the output of the hardware random number generator (RNG) in other virtual machines or even SGX enclaves on the same machine, even when hyperthreading (SMT) has been disabled and all other standard mitigations have been applied. Furthermore, given that RDRAND and RDSEED are the only local sources of randomness inside SGX, the attack compromises currently-deployed SGX enclaves which rely on randomness for their cryptographic operations. Finally, we show that even recent Intel CPUs – including those used by public cloud providers to support SGX enclaves – are vulnerable to these attacks.

To summarize, our contributions are:

- We present the design and implementation of CrossTalk, a profiler for analyzing the behavior of instructions on Intel CPUs in different contexts. We use CrossTalk to perform an analysis of the behavior of instructions on Intel CPUs, with a focus on complex instructions and those performing undocumented "offcore" accesses on internal CPU buses.
- We show that some of these offcore reads can leak information *across cores* on modern Intel CPUs, due to their use of a globally shared buffer (which we refer to as the *staging buffer*). Using CrossTalk, we analyze the way in which instructions use this buffer, show that it can contain sensitive information, and demonstrate that this mechanism can be (ab)used as a stealthy cross-core covert channel.
- To demonstrate the security impact of our findings, we present the first **cross-core** attack using transient execution. By leaking RDRAND output, we obtain an ECDSA private key from an SGX enclave running on a separate physical core after just a single signature operation. More details about CrossTalk and our attack, including proof-of-concepts (PoCs) are available at `https://www.vusec.net/projects/crosstalk`.

Table 4.1: Examples of relevant CPU (Skylake) performance counters.

Name	Mask	Description
UOPS_EXECUTED	CORE	Number of micro-ops executed on a given CPU core.
UOPS_DISPATCHED	PORT_0–7	Number of cycles where micro-ops were dispatched on a specific port.
IDQ	MS_UOPS	Number of micro-ops provided from microcode.
OTHER_ASSISTS	ANY	Number of (non-FP) microcode assists invoked.
MEM_INST_RETIRED	ALL_LOADS/ _STORES	Number of load/store instructions which reached retirement.
OFFCORE_REQUESTS	ALL_REQUESTS	Number of requests which "reached the Super Queue" (not in cache).
OFF_CORE_RESPONSE	STRM_ST	Number of streaming store requests.
OFF_CORE_RESPONSE	OTHER	Number of miscellaneous requests.

- We discuss existing mitigations and argue that they are largely ineffective against our attack, and present results for Intel's new in-microcode mitigation.

4.2 Background

Ever since the public disclosure of Meltdown [149] and Spectre [128], transient/speculative and out-of-order execution attacks have stormed onto the security stage with new and often devastating vulnerabilities appearing constantly [28, 31, 128, 135, 149, 211, 240]. They leak information from a wide variety of sources, including data caches and CPU buffers such as (line) fill buffers, load ports and store buffers. What these vulnerabilities have in common is that fixing them is typically expensive [83, 143], [240], or even impossible for existing hardware [159]. In this chapter, we make use of the MDS vulnerabilities [31, 211, 240] as a vehicle for finding information leakage beyond what happens inside a single core.

4.2.1 Microarchitectural data sampling (MDS)

The vulnerability which Intel calls Microarchitectural Data Sampling (MDS), also referred to as RIDL [240], ZombieLoad [211] and Fallout [31], allows attackers to leak sensitive data across arbitrary security boundaries on Intel CPUs. Specifically,

they can obtain arbitrary in-flight data from internal buffers (Line Fill Buffers, Load Ports, and Store Buffers)—including data that was never stored in CPU caches. We briefly discuss these three buffers.

Line Fill Buffers (LFBs) are internal buffers that the CPU uses to keep track of outstanding memory requests. For instance, if a load misses the cache, rather than blocking further use of the cache, the load is placed in the LFB and handled asynchronously. This allows the cache to serve other requests in the meantime. As an optimization, when a load is executed and the data happens to be already available in the LFB, the CPU may supply this data directly. Intel CPUs also transiently supply this data when a load is aborted, due to an exception or microcode assist (e.g., setting dirty bits in a page table). An attacker who can observe side-effects from transiently executed code can take advantage of this to obtain data in LFB entries containing memory belonging to a different security domain, such as another thread on the same CPU core, or a kernel/hypervisor. This vulnerability is known as Microarchitectural Fill Buffer Data Sampling (MFBDS).

Store Buffers (SBs) track pending stores. In addition, they play a role in optimizations such as store-to-load forwarding where the CPU optimistically provides data in the store buffer to a load operation if it accesses the same memory as a prior store. Again, this transiently forwarded data may belong to another security domain, allowing an attacker to leak it.

Finally, load ports are used by the CPU pipeline when loading data from memory or I/O. When a load micro-op is executed, data from memory or I/O is first stored in the load ports before it gets transferred to the register file and or younger operations that depend on it. When load instructions are aborted during execution, they may transiently forward the stale data from previous loads, which attackers can leak using transient execution.

As an example, we consider a 'RIDL-style' MDS attack – using LFBs – performed with four steps. First, the attacker creates a Flush + Reload array, with one cache line for each possible value for the data (typically a byte) to be leaked, and flushes it to ensure that none of these lines are in the cache. Then, the attacker ensures that the processor uses some secret data, for instance by prompting the victim program to read or write such data, or by ensuring that such data is evicted from the cache. Either way, the processor moves the in-flight data into these Line Fill Buffers (LFBs). Next, the attacker performs a load causing an exception or assist, for instance from an address that causes a benign page fault. The load can forward to dependent instructions despite not completing, using the secret data from the LFB. The attacker's transiently executed code then uses the data as an index into the Flush + Reload array. The corresponding cache line will be optimistically loaded into the cache by the pipeline when it executes the transiently executed code. Finally, by loading every element of the array and timing the load, the attacker can determine which one was in the cache. The index of the cached

entry is the secret value which was obtained from the LFB.

In November 2019, Intel announced several new MDS vulnerabilities, among which TSX Asynchronous Abort (TAA) with CVE-2019-11135 is perhaps the most prominent [211, 240]. In a TAA attack, an aborted TSX transaction causes the instructions currently under execution to continue until retirement in a manner that is akin to transient execution—allowing the attacker to leak information from the internal buffers as described above.

4.2.2 Intel micro-ops/microcode

While Intel microcode is undocumented and its behavior is largely unknown, it is no secret that all x86 instructions are translated to one or more micro-ops which have a RISC-like format. Generally, the decoder performs a direct translation to a small number of micro-ops (at most 4). In rare cases, larger numbers of micro-ops are required, such as for microcode assists (handling corner cases such as marking dirty bits in page tables, or after a faulting memory load) and complex instructions (where more than 4 micro-ops are needed, or control flow is necessary). In those cases, the micro-ops are instead fetched from the microcode ROM. To allow for post-production bug fixes, Intel processors support in-field microcode updates since the mid-1990s [89].

4.2.3 Intel performance counters

Many performance counters are available on Intel CPUs, giving developers information about potential bottlenecks in their code. More generally, they can be used to gain insight into CPU behavior. Some examples can be seen in Table 4.1. The first two examples provide information about the decoding and issuing of instructions, including the number of micro-ops issued, and the number of micro-ops executed on each execution port. Since micro-ops can only be executed on specific (sets of) ports, the latter gives coarse information about the types of micro-ops being executed. For example, on Skylake, we observe [2] that the AESDEC instruction uses port 0 (used for AES operations), and that it also uses ports 2/3 (used for loads) when the input is a memory operand.

There are other counters which can provide insight into the micro-ops being executed. For example, one counter counts the number of micro-ops decoded from the microcode ROM, and another provides the number of invoked microcode assists. Finally, we can observe information about loads and stores by checking how many load/store instructions were retired, the number of hits/misses at each level of the processor cache, as well as by using the counters which provide the number and type of 'offcore' requests (such as DRAM accesses).

4.2.4 Intel software guard extensions (SGX)

Intel's Software Guard Extensions (SGX) instructions create so-called 'enclaves' to be executed using encrypted memory. This protects sensitive data (such as encryption keys) from potentially-hostile operating systems and/or hypervisors.

There have been a number of transient execution vulnerabilities allowing the contents of SGX enclaves to be exposed to a hostile attacker [28, 240]. Mitigations against these attacks have been implemented in microcode and on recent CPUs; microcode now clears the L1 cache and internal CPU buffers when leaving an enclave, and TSX transactions are automatically aborted if an SGX enclave is running on a sibling core. Enclaves can confirm that they are being run in a secure environment using attestation [47], which allows a remote party to ensure that SGX enclaves are running on machines with up-to-date microcode, and that SMT is disabled when running on hardware vulnerable to L1TF/MDS.

4.2.5 RDRAND

The RDRAND x86 instruction was first introduced in Intel's Ivy Bridge CPUs. It returns random numbers derived from a digital random number generator (DRNG), and is available at all privilege levels (including userspace and SGX enclaves). Intel's DRNG [160] outputs random seeds (processed using AES-CBC-MAC) and feeds them to a deterministic random-bit generator (DRBG), which fills the global RNG queue using AES in counter mode. More recently, the RDSEED instruction was added in Intel's Broadwell CPUs, allowing access to higher-entropy randomness (intended for seeding software PRNGs). AMD CPUs also support RDRAND and RDSEED, although with a higher performance cost (around 2500 cycles for 64-bit RDRAND output on Ryzen).

Cryptographic applications often rely heavily on the confidentiality of random numbers; an attacker who can predict or obtain these random numbers can often break and even obtain private keys. RDRAND provides a convenient mechanism for generating cryptographically-secure random numbers, to prevent such attacks. In environments such as SGX, the only available source of randomness provided by the CPU is through RDRAND and RDSEED instructions.

4.3 Threat Model

We assume an attacker who aims to abuse transient execution to disclose sensitive information from a victim that is running on the same system. We further assume that all standard hardware and software mitigations (available at the time of writing) against transient execution are in effect. Although co-location on the same physical system is required, we assume that the operating system employs conservative scheduling policies that avoid executing processes from different security

domains on the same core [46, 100, 163]. Even under these strong assumptions, we show that on many Intel processors, an attacker can abuse transient execution to leak sensitive information such as CPU-generated random numbers from the victim regardless of the placement of the attacker and the victim on different cores in the system.

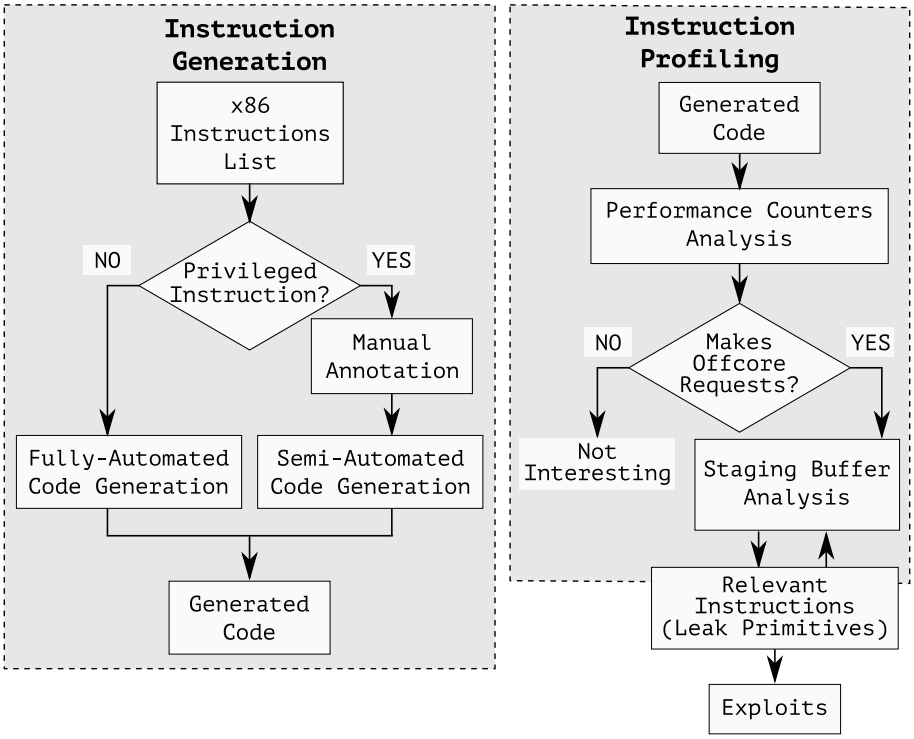


Figure 4.1: Overview of the two stages of CrossTalk.

4.4 CrossTalk

Figure 4.1 shows the components of CrossTalk. In the first stage, CrossTalk profiles all the x86 instructions that make offcore memory requests. We use the output of this first stage in combination with MDS to understand the interaction of on-core LFBs with a globally-shared offcore buffer as shown in Figure 4.2. With this knowledge, CrossTalk's second stage automatically discovers how information leaks from one instruction to another as they write to different offsets within the offcore buffer. The output of CrossTalk's second stage is a number of instructions, each capable of leaking information from other instructions that are executed on different cores in the system.

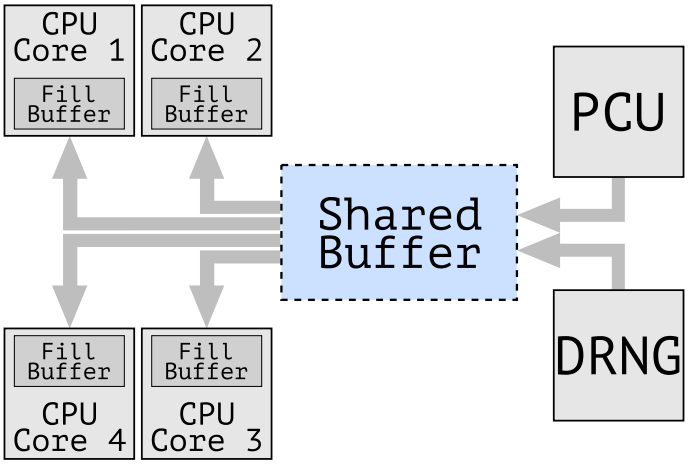


Figure 4.2: Flow via shared staging buffer to fill buffers of specific cores.

4.4.1 Instruction generation

To understand which instructions on Intel's CPU use non-obvious micro-ops and how these instructions are implemented in practice, CrossTalk attempts to execute many variants of x86 instructions, in different contexts. Previous research (uops.info [2]) provides a dataset containing performance counter information for many x86 instructions, in particular port usage information and the number of executed micro-ops. We needed to track a wider variety of performance counters, which can be done using the information in this dataset together with the tool used to generate it (nanoBench [1]).

However, although this existing dataset is sufficient for simple instructions which are translated directly to micro-ops by the hardware decoder unit, it fails to provide information about many microcoded sequences, which may contain control flow based on their context. For example, the value of the operands to some instructions drastically modifies their behavior; for example, the leaf[3] number passed to CPUID. As noted in [2], performance differences due to different register or intermediate values are not considered by their tool. Error paths may only be exercised when incorrect data is provided, and instructions behave differently in some execution environments (such as inside virtual machines, in different rings, or in SGX enclaves). Microcode assists [31] are only executed in situations where they are necessary.

As such, CrossTalk is designed to allow execution of instructions in different situations and with different operands, and allows us to profile their behavior in multiple different ways. This allows CrossTalk to obtain a more comprehensive

[3] a CPUID leaf refers to the category of data being requested

view of the behavior of the CPU, by increasing our coverage of Intel's microcode.

CrossTalk's first stage uses the uops.info dataset discussed above as a starting point to automatically generate both user and kernel mode instructions of interest. CrossTalk then executes the resulting code in different contexts to collect performance counter information, recording the values of all supported performance counters before and after running several iterations of the generated instructions. After each run, we manually examined the results, and added code to improve coverage in some cases. For example, after finding all CPUID leaves by testing all values of EAX and observing the differences as reflected in performance counters, we then updated our code to ensure we had full coverage of potential CPUID subleaves (specified by ECX) for each of these leaves. Privileged instructions will often crash machines if executed with arbitrary operands; we extended the coverage to include some of these by adding manual annotations/code, such as providing valid values for WRMSR.

Table 4.2 summarizes some representative examples of the output of this phase of our tool for some instructions on an Intel i7-7700K desktop CPU running Ubuntu 18.04.3 LTS with kernel version 5.3.0-40-generic and microcode version 0xca[4]. CPUID is a normal, userspace instruction, and is present in the uops.info dataset, which claims it executes 169 micro-ops on Skylake. As we can see in the table, the behavior of this instruction depends heavily on the value of EAX (the leaf), and only *some* of these variants make cross-core requests. Similarly, RDMSR is a privileged instruction which depends on the value of ECX, which specifies the MSR to read. We found hundreds of different valid MSRs (470 on the i7-7700K), and again we can see from the performance counters that many of them execute different flows in the microcode, many of which make cross-core requests (205 MSRs on the i7-7700K). The uops.info dataset only presents results for a single MSR. These examples demonstrate the importance of the context when analyzing these instructions, as the number and nature of the micro-ops executed changes significantly, depending on the instruction's operands.

4.4.2 Offcore requests

We do not observe unexpected performance counter values when executing non-microcoded instructions, where a small number of micro-ops are generated directly by the decoder. However, complex microcode flows with larger numbers of micro-ops are more interesting. In particular, some instructions unexpectedly perform *offcore* requests, according to the relevant performance counters. Specifically, we monitor the total number of these memory accesses performed by each instruction using the `OFFCORE_REQUESTS.ALL_REQUESTS` counter. We found that the responses to these offcore requests can be broken down into categories using

[4]Unless specified otherwise, we will use this system for our examples throughout the paper.

Table 4.2: Example results from the instruction profiling stage of CrossTalk.

Instruction	Description	Executed μOps	μOps from Microcode ROM	μOps Dispatched on Ports 2/3	Offcore Requests	Offcore Store Responses	Other Offcore Responses	Retired Insts. Loads/ Stores
CPUID	Brand String 1 (0x80000002)	104	134	5 / 6	4	0	4	1 / 0
CPUID	Thermal/Power Mgmt (0x6)	120	163	2 / 3	3	0	3	1 / 0
CPUID	SGX Enumeration (0x12) (Subleaf 0)	3677	2939	297 / 304	30	28	2	1 / 0
CPUID	SGX Enumeration (0x12) (Subleaf 1)	3694	2938	303 / 311	30	28	2	1 / 0
CPUID	SGX Enumeration (0x12) (Subleaf 2)	3694	2942	302 / 309	30	28	2	1 / 0
CPUID	Processor Info (0x1)	83	89	1 / 1	0	0	0	1 / 0
RDRAND	DRBG Output	16	16	1 / 1	1	0	1	1 / 0
RDSEED	ENRNG Output	16	16	1 / 1	1	0	1	1 / 0
CLFLUSH	Address Not Cached	4	4	1 / 0	1	0	0	0 / 1
RDMSR	Platform ID (0x17)	104	127	1 / 2	3	0	2	1 / 0
RDMSR	Platform Info (0xCE)	122	155	1 / 2	3	0	2	1 / 0

[RDRAND and RDSEED return random numbers from an Intel on-chip hardware random number generator, CPUID allows software to discover details of the processor, while RDMSR is used to read the content of model-specific registers.]

the `OFF_CORE_RESPONSE` event, which Intel provides to observe requests that miss in the L2 cache.

In particular, two counters allow us to categorize the requests made by these instructions: `STRM_ST` (which counts streaming store requests) and `OTHER` (which counts miscellaneous accesses, including port I/O, MMIO and uncacheable memory accesses), which we find sufficient to distinguish our cases of interest. For example, instructions responsible for flushing caches appear to make one offcore store request for every cache line flushed; CLFLUSH and CLFLUSHOPT make a single request. However, the `OFFCORE_RESPONSE` counter remains zero for these cases.

We encounter some unexpected behavior even when restricting our analysis to this limited subset of performance counters. For example, the VERW instruction makes as many as 28 store requests, despite the fact that VERR makes none. While this discrepancy may appear puzzling at first, the explanation is simply that VERW has recently been repurposed to flush internal CPU buffers (such as the line fill buffers), as a mitigation for the MDS vulnerabilities [106].

However, our attention was drawn to the unexpected memory accesses performed by *other* instructions, which appear to have no obvious reason to access memory at all. The majority of other requests (corresponding to reads) seem to be in the `OFFCORE_RESPONSE.OTHER` group, although there are exceptions. For example, the SGX information CPUID leaf increases the store counter by 28—the same number of accesses as incurred by VERW, which implies that the microcode for this leaf also performs CPU buffer clears.

```
for (int offset = start; offset < end;
  offset++)
{
  // Execute a leak primitive
  cpuid(0x1);

  // Perform invalid read to
  // leak from an LFB at "offset"
  char value =
    *(invalid_ptr + offset);

  // Expose result for flush+reload
  (void)reload_buf[value];
}
```

Listing 3: Simplified example of leaking offcore requests.

4.4.3 Leaking offcore memory requests

To investigate these memory reads further, we make use of MDS [211, 240] which allows us to examine the contents of internal CPU buffers. The hope is that doing so will reveal the nature of these memory accesses. MDS allows attackers to observe (normal) memory reads and writes performed by microcode. An example is the contents of page table entries fetched by the PMH. CrossTalk uses the same vulnerability to leak information about the memory accesses performed by microcoded instructions that perform offcore memory requests, as shown in Listing 3.

Consider the CPUID instruction being used to read the CPU brand string – remember, the behavior of this instruction depends on the requested (sub)leaf. Specifically, we read the first brand string leaf on our i7-7700K, which is 'Intel(R) Core(TM'. First, we use MDS to leak load port contents from a sibling thread, by performing a single vector load which span a page boundary, where one or both of the pages are invalid, and using FLUSH + RELOAD to obtain the value read during transient execution. If both pages are invalid, we leak the values 'Inte' and ' Cor'; if only the first page is invalid, then we leak the values 'l(R)' and 'e(TM'. These appear to correspond to the values on the first and second load ports, based on other experiments; in any case, it seems that the four offcore requests correspond to these four read values.

Alternatively, we can use MDS to leak the contents of the fill buffer. We saw the same results using both MFBDS and TAA variants; example code using TAA can be found in Appendix 4.11.1. Here, we consistently leak the entire value, as opposed to the individual components. This implies that not only do these loads go via the fill buffer, but also that a single fill buffer is used for the entire offcore request. Since the fill buffer is 64 bytes, we can also leak data beyond the first 16

bytes of the buffer. This produces inconsistent results; sometimes the next bytes of the buffer contain the remainder of the brand string, but it can also contain other values.

To explore this, we run the following experiment: on one core, we execute CPUID with the leaf that reads the first part of the brand string. As we saw already, this uses the first 16 bytes of the fill buffer. On another core, we use CPUID to read the second part of the brand string, which turns out to write to the next 16 bytes of the fill buffer. Interestingly, using MDS we observed the result of the instruction executed on the *other* core in the line fill buffer of the current core. Specifically, we saw the first 32 bytes of the CPUID brand string when leaking the contents of the fill buffer – but not the remaining bytes (since we did not request the third brand string leaf). Therefore, we are not just leaking the entire brand string.

This experiment implies that that we are leaking contents from an offcore global *staging buffer*; our thoughts about the nature of this buffer can be found in Appendix 4.8. After reporting our findings to Intel, they confirmed that a global staging buffer is responsible for our results.

We used the insight that we can leak the contents of this staging buffer using CPUID as a starting point for building the second stage of CrossTalk, which automatically discovers which code sequences (instructions together with the necessary context, such as register initialization) write to which offsets within this buffer.

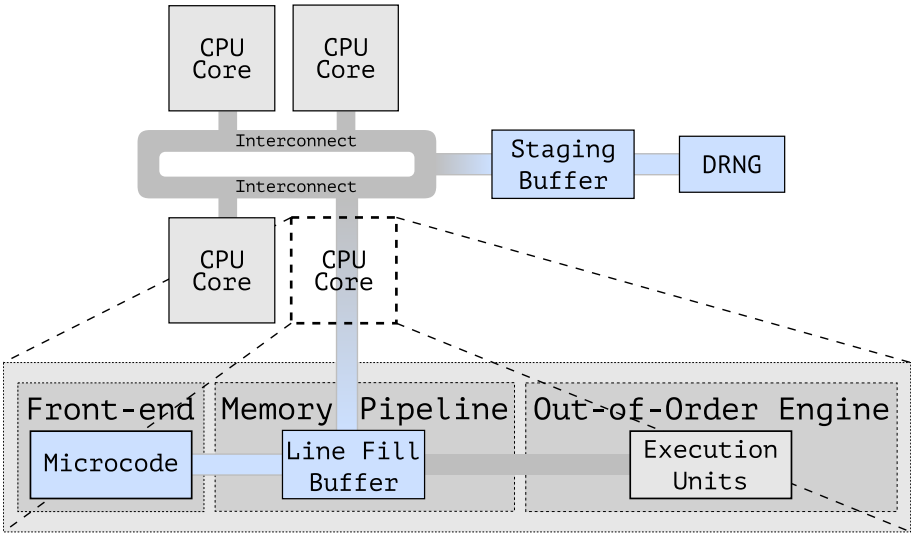


Figure 4.3: Microcode reads cause data to be read from the DRNG using per-core fill buffers, via a shared staging buffer.

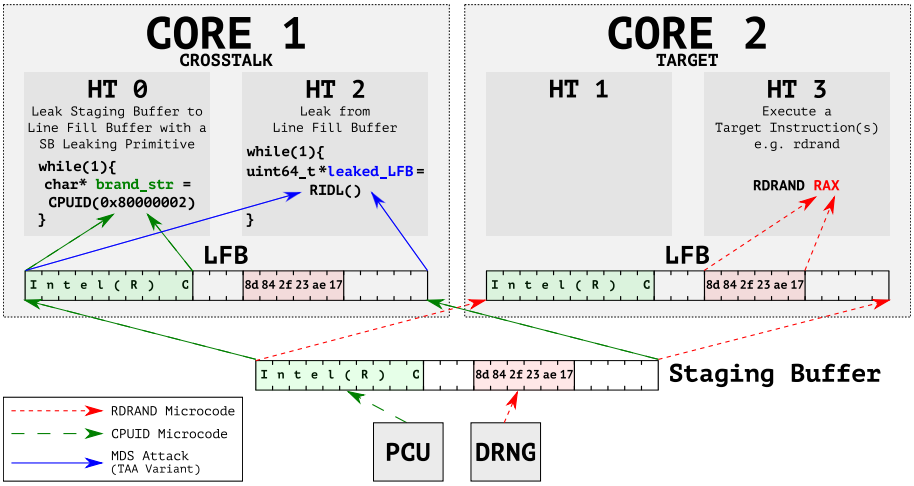


Figure 4.4: Staging buffer analysis process of the second stage of CrossTalk.

4.4.4 Profiling the staging buffer

In the second stage of CrossTalk, we aim to automatically analyze how the previously-discovered code sequences that send offcore memory requests interact with the globally-shared staging buffer. For each sequence, we want to know which values the CPU stores in the staging buffer, which offsets they use, and to find any additional staging buffers if present. Figure 4.4 shows the design of CrossTalk's second stage. On one physical core, we run the target instructions that potentially interact with the staging buffer. In the other physical core, we try to observe whether the contents of the staging buffer changes due to the execution of the target instructions. To make sure that we observe the contents of the staging buffer, we need to ensure that we continuously pull the data from the staging buffer. We use what we call *masking* primitives for this purpose.

The masking primitives overwrite a portion of the staging buffer while bringing in the data from the rest of the buffer. We refer to the region of the staging buffer overwritten by each masking primitive as its "mask". To obtain the entire contents of the staging buffer, we need at least two masking primitives with different (non-overlapping) masks. The first primitive allows us to obtain the data which is not overwritten by the mask, and the second primitive allows us to obtain the remaining data. The obvious candidates for masking primitives are the various leaves of the CPUID instructions, which provide primitives meeting these requirements.

To perform our profiling, we need masking primitives which cover all offsets in the buffer, and which write a constant (or predictable) value to these offsets. Once we have such primitives, we can profile code sequences by comparing the contents of the staging buffer to the 'expected' data at each offset. If we see a

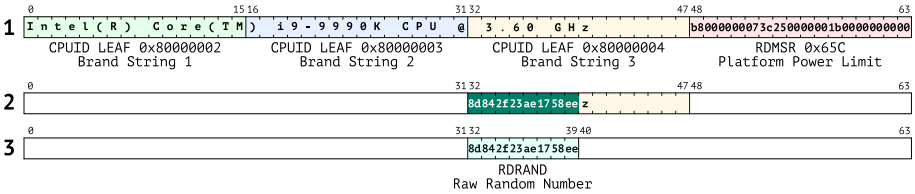


Figure 4.5: Profiling a target instruction. **Step 1**: Prime the staging buffer by executing leak primitives which write known data to known offsets within the buffer. **Step 2**: Execute the target instruction (here, RDRAND). **Step 3**: Observe any overwritten bytes (by comparing to step 1).

significant number of unexpected values at any given offset, we record that the code sequence being profiled modifies that offset of the staging buffer.

Since this is only possible if we are sure that each sequence is not overwriting the contents of the buffer with the same value written by our masking primitives, we need two masking primitives for each byte, with different values. While profiling the buffer, we search for suitable additional masking primitives which write known values to the staging buffer, gaining access to additional primitives as we profile.

An example of this process can be seen in Figure 4.5. Here, the masking primitives are three calls to CPUID, reading the three leaves corresponding to the brand string. These calls overwrite the first 48 bytes of the staging buffer with known data. After running a target instruction sequence containing RDRAND, some of the offsets in the staging buffer are overwritten with new data; our staging buffer analysis records that RDRAND modifies these offsets. We call these sequences 'leak primitives', since when executed, they potentially leak the data to an attacker who can run code on another core.

Some leak primitives will write constant values, allowing us to also record the data written by that code; for example, the CPUID brand string leaves always write the brand string itself. Other instructions, such as RDRAND, do not write predictable data, so we mark the values as unknown. If necessary, we can also use these as masking primitives, by leaking the value they write to the staging buffer before every attempt to profile a sequence. We can also build our own masks by using WRMSR to modify the value of MSRs and then reading them back; for example, RDMSR 0x395 can be used as a mask with an arbitrary 48-bit value.

Representative results obtained from CrossTalk's second stage can be found in Table 4.3. We found various leak primitives including instructions that interact with Machine-Specific Registers (MSRs), and instructions that are used for hardware random number generation. Although disclosure of the majority of this information does not seem to present a security threat, the RDRAND and RDSEED instructions are more of a concern. In Section 4.5, we discuss how we can build practical

Table 4.3: Examples of primitives we found to be using the staging buffer.

Instruction	Operand(s)	Description	Offcore Responses	Staging Buffer Offsets	Leaked Data from Staging Buffer
RDRAND	—	DRBG Output	1	32–39	Random Number
RDSEED	—	ENRNG Output	1	0–7	Random Number
CPUID	0x80000002	Brand string 1	4	0–15	Brand String part 1
CPUID	0x80000003	Brand string 2	4	16–31	Brand String part 2
CPUID	0x80000004	Brand string 3	4	32–47	Brand String part 3
CPUID	0x6	Thermal/Power Management	3	0–7, 17–28, 48–55	Unknown (includes raw MSR value)
CPUID	0x12 (Subleaf 0)	Intel SGX Enumeration	30‡	0–7, 56–63	Unknown
CPUID	0x12 (Subleaf 1)	Intel SGX Enumeration	30‡	0–7, 56–63	Unknown
CPUID	0x12 (Subleaf 2)	Intel SGX Enumeration	30‡	0–7, 56–63	Unknown
RDMSR	0x20	Bootguard Hash 1	1	0–7	Unknown
RDMSR	0x13A	Bootguard Status	3	16–23, 48–55	Unknown
RDMSR	0xCE	Platform Information	2	24–31	Raw MSR value
RDMSR	0x17	Platform ID	2	16–23	Raw MSR value

Offcore responses are 'other' except: ‡28 are strm_st

real-world exploits attacking these instructions.

4.5 Exploitation

The disclosure capabilities we identified can be used to observe the contents of the globally-shared staging buffer in combination with MDS attacks, allowing code running on one core to read buffer data belonging to a different core. These attacks can be performed on *any* core of a system and hence any mitigation isolating security domains on a per-core basis is ineffective. Given their non-trivial security impact, we focus our exploitation on the RDRAND and RDSEED instructions.

First, we discuss details and challenges involved in performing attacks based on the relevant instructions. Then, we demonstrate such attacks are realistic with an exploit that can obtain private keys by observing the staging buffer while an SGX enclave performs cryptographic operations.

4.5.1 Available primitives

Since we can sample the staging buffer contents at an arbitrary time, we can craft a probing primitive to detect when instructions touching the buffer have been used. We do this by sampling the buffer at regular intervals with a leak primitive, and then comparing the data at a specific offset to the previously-seen values. For instance, we can determine when the Linux CRNG is being used (such as filling AT_RANDOM when processes are created), since the `_extract_crng` function always mixes new RDRAND output into the state before outputting data.

We can also craft an information disclosure primitive which leaks the *contents* of the staging buffer, and discloses security-sensitive data such as the actual RDRAND output. As we shall see, this example in particular has serious consequences for code performing cryptographic operations. To do so, we can use any of the leak primitives we have identified that transiently sample data from the staging buffer, with some environment-specific constraints.

4.5.2 Constraints

In practice, not all of the leak primitives are available to attackers; we consider some typical limitations in different environments, and how they can be avoided. Our attacks can be mitigated in some environments due to such restrictions; we discuss this in Section 4.7.

Userspace: The CPUID, RDRAND and RDSEED instructions can all be executed from userspace.

Virtual machines: If attackers are only able to run code inside a virtual machine, their ability to run disclosure primitives to access the staging buffer will be limited. For example, RDMSR is likely to be restricted or prohibited entirely, and typically VMs also trap on CPUID, to allow the hypervisor to restrict the information and capabilities which will be reported to the guest. However, two disclosure primitives can still be executed from userspace in the default configurations of many virtual machines: RDRAND and RDSEED. An attacker can use one of these primitives to leak the output of the other.

When SMT is enabled, an attacker can make hypervisor requests that involve disclosure primitives (a form of 'confused deputy' attack), and then read the staging buffer from the fill buffer. For example, Xen will call RDMSR with 0x17 (platform ID) when a guest attempts to read MSR 0x17. Even if MDS mitigations (such as scheduling-based isolating strategies [67]) are in place, this allows an attacker to leak the contents of the staging buffer from the sibling thread—disclosing data of a victim running on a different core.

SGX: Although most relevant instructions are not available within SGX, a theoretical attacker inside an SGX enclave could (much as in the VM case) mount a cross-core attack using RDRAND and RDSEED.

4.5.3 Synchronization

A probing primitive allows an attacker to detect accesses to the staging buffer and synchronize with the victim. Since we only need to check whether the byte we leak is the byte we expect, this can be done with a single flush and a single reload, and the performance overhead is dominated by the execution time of our leak primitive. However, to *preserve* synchronization, an attacker armed with our information disclosure primitive needs to leak data from the buffer at a sufficiently high sampling

rate to keep up with the consumption of random numbers by the victim.

Each RDSEED or RDRAND instruction provides a maximum of 8 random bytes (one 64-bit register). Many applications require a larger amount of entropy, so these instructions can potentially be called in a loop. Both instructions take approximately 370 cycles on Skylake, so generally, an attacker will not have enough time (assuming a relatively fast victim loop) for an attacker to complete leaking from FLUSH + RELOAD before it is overwritten with the next value. Since up to two bytes can be efficiently obtained in a single 'round' on Skylake, and an attacker can use multiple cores at once, an attacker with access to sufficient CPUs/threads may be able to leak all 8 bytes at once. Even so, it appears impractical to leak the full entropy from a victim which executes several (or many) RDRAND instructions in quick succession.

In practice, a single byte (or less) is sufficient for many attacks [248]. However, where it is convenient or even necessary to leak more bytes, we can use a performance degradation attack to slow down the victim [14]. In the following, we first analyze how we will actually perform these leaks efficiently. Then, we show how an attacker can induce performance degradation on a realistic victim (an SGX enclave) to mount practical and reliable exploits.

4.5.4 Optimizing leakage

We found that an attacker can obtain better results where SMT is available; they can run the FLUSH + RELOAD loop on one logical thread, and a tight loop using a leak primitive to fetch the staging buffer (here, we used RDRAND) on the sibling thread. Both of these threads are controlled by the attacker and in the same security domain; the victim is running on a different physical core. We found this to be the best way to almost guarantee that the leaked fill buffer would contain staging buffer content. Where SMT is not available, we need to ensure that we leak the LFB which contains the staging buffer. On our i7-7700K, we determined that this occurs when we use CLFLUSH to flush 15 cache lines after running the leak primitive; note that this can be done as part of FLUSH + RELOAD. We made use of the TAA variant of the MDS vulnerabilities to actually leak the fill buffers, since it is fast and works even on CPUs with mitigations against other MDS attacks. Again, see Appendix 4.11.1 for an example code listing. Where TSX is unavailable, an attacker can instead obtain fill buffers using MFBDS [240].

4.5.5 Performance degradation

There are different ways to slow down a victim performing a target security-sensitive computation. For instance, we can use microarchitectural profiling to determine the resources the victim is bottlenecked on and flush such resources (e.g., last-level cache lines) from another core to slow down the victim [14, 254]. If

we are specifically targeting RDSEED instructions, we can attempt a more targeted performance degradation attack.

Since the amount of global entropy available is limited, calls to RDSEED are unsuccessful (returning zero) when no entropy is currently available. An unsuccessful call does not overwrite the previous contents of the staging buffer. Hence, an attacker can make their own calls to RDRAND or RDSEED, consuming entropy and increasing the time period between successful RDSEED calls by the victim. A successful call to RDRAND or RDSEED will overwrite the previous data in the buffer, which means that old data cannot be leaked after this point. However, by then, an attacker may have already read the bytes; FLUSH + RELOAD can complete after this point.

A practical avenue to mount generic performance degradation attacks is SGX, where an attacker can slow down the execution of a victim enclave at will by inducing frequent exceptions [236]. As such, and given that SGX enclaves rely on RDRAND as a source of entropy (amplifying the impact of the attack), we do not attempt to use other performance degradation techniques but instead specifically target code running in an SGX enclave in our exploit.

4.5.6 Leaking RNG output from SGX

Since an attacker is assumed to control the entire environment, enclave code running in Intel's SGX is unable to trust local sources of random data, other than RDRAND. Even typical forms of 'additional entropy', such as the timestamp counter, are unavailable in most implementations of SGX. Intel state that CPUs which support SGX2 allow it to be used inside enclaves, but even then, attackers can determine and/or control (at least within a narrow range) the value of this counter. Although a coarse "trusted clock" source is also available (`sgx_get_trusted_time`), this does not appear to be widely used and is primarily intended against replay attacks. This trusted clock is provided by CSME, which has itself been the subject of several recent vulnerabilities [64, 105], and Intel acknowledge that CSME secure time remains vulnerable to physical attackers on some platforms [112].

Many enclaves and SGX-based defenses explicitly use RDRAND [5, 51]. Other enclaves use the SGX SDK's `sgx_read_rand` function, which generates entropy in a loop using RDRAND to generate 32-bit random numbers, and copies the results directly into the output buffer.

Hence, by dumping RDRAND data, we can leak *all* the random entropy used by arbitrary security-sensitive code running inside an SGX enclave, allowing recovery of cryptographically-critical data such as random nonces. As mentioned, one option is to induce controlled exceptions on the victim SGX enclave and single step its execution using SGX-Step [236]. We could then sample the buffer after every RDRAND from the very same core. However, this exploitation strategy can

be easily mitigated in software or microcode (as we propose in Section 4.7.5). As such, we instead opt for an asynchronous exploitation strategy that is significantly harder to mitigate. In particular, we first induce exceptions on the SGX enclave only to mount a performance degradation attack and slow down the execution of the victim. Then, we use our leak primitives from a *different* core to mount a hard-to-mitigate (asynchronous) but reliable (since the victim is much slower than the attacker) cross-core attack.

As mentioned earlier, the primary challenge for an attacker is to leak the RDRAND results fast enough to keep up with the victim, since the reload step (after the buffer has already been transiently accessed) is our primary bottleneck. If we can use exceptions to prevent an enclave from executing RDRAND faster than we can leak it, then we can reliably leak all of the entropy used by the enclave.

In fact, this means that we only need to degrade the performance of an enclave when it is actively calling RDRAND– and we found that `sgx_read_rand` actually makes use of another function, `do_rdrand`, to actually perform the RDRAND calls. Due to the convenient placement of these functions in different pages in all the enclaves we inspected, we can simply use page faults on the pages containing the two different functions to enforce one RDRAND call at a time. If enclave authors attempt to mitigate our attacks by using multiple calls to RDRAND in quick succession in a single page, we can simply resort to a "standard" SGX-Step approach.

4.5.7 Attacking crypto in SGX

To exemplify the exploitation capabilities of our primitives, we present a cross-core exploit leaking random nonces used by an (EC)DSA implementation running in an SGX enclave. Previous research [248] shows that leaking a small number of *bits* of a random nonce is sufficient to recover private keys, using a small number of ECDSA signatures. We show our exploit exceeds such expectations by recovering *all* of the bits, with just a single signature.

An ECDSA signature consists of a pair (r, s), where r depends only on k, and $s = k^{-1}(z + rp)$[†], where z is based on the hash of the input, and p is the private key. By rewriting this as $p = (sk - z)/r$, an attacker who can generate a signature (r, s) with a known nonce k can simply solve for p.

The SGX SDK provides an *sgx_ecdsa_sign* function for performing ECDSA signatures with a private key. For example, it is used by the *certify_enclave* function of Intel's Provisioning Certificate Enclave.

The default configuration for Intel's SGX SDK performs cryptographic operations using the Intel IPP crypto library. When generating ECDSA signatures, it uses a nonce (ephemeral key) k based on the output of sgx_ipp_DRNGen, which in turn calls the sgx_read_rand function discussed above. This means that k can

[†]For simplicity, we omit details such as the requirement that all calculations are done modulo n.

```
void get_SignedReport(char *p_report,
         sgx_ec256_signature_t *p_sig) {

    sgx_ecc256_open_context(&handle);

    // sign g_rpt with g_priv_key
    sgx_ecdsa_sign(g_rpt, g_rpt_size,
                   &g_priv_key, p_sig,
                   handle);

    // return the signature contents
    memcpy(p_report, g_rpt, g_rpt_size);
}
```

Listing 4: SGX enclave function

be calculated by an attacker who can observe the output of RDRAND.

To demonstrate this is feasible, we attack a simple victim SGX enclave that uses *sgx_ecdsa_sign* to sign a message, and then returns both the message and the signature (r, s). A simplified listing of the function can be seen in Listing 4.

To perform the attack, we start executing the victim enclave, while our exploit running on another core collects random data from the staging buffer as described above. Specifically, we simply use SGX-Step to cause a page fault when execution enters the page containing *do_rdrand*, single-step for several instructions (to ensure RDRAND has been executed), and then wait for 1ms to ensure that our exploit code has collected the staging buffer. In practice, 1ms is enough time to collect thousands of results from the staging buffer, which allowed us to exclude noise, and differentiate the enclave-collected entropy from normal system entropy. If needed, synchronization between stepping core and the leaking core could be used to obtain better results.

Afterwards, we possess the signature (r, s), and can immediately calculate z by hashing the message. We can then attempt to recover the private key p by trying all likely values for k – in our case, to find candidates for k, we simply identified all entropy which appeared in the staging buffer at a regular interval (slightly longer than our 1ms wait period), and made a list containing all candidates.

We implemented the key recovery step in Python, using the ecdsa library. An overview of our attack can be seen in Listing 5. When the SGX enclave calls Intel's IPP crypto library, it computes k by making 8 calls to RDRAND, using 32 bits each time. We take every possible linear sequence of 8 values in the entropy observed in the staging buffer during our attack, compute the relevant value of k, and check whether it is the private key (by performing the signature again ourselves).

We performed this attack on an i7-7700K CPU with up-to-date microcode as of January 2020, and with SMT disabled. When encountering a page fault, we at-

```
msgHash, r, s = call_enclave()
recovered_entropy = get_leaked_entropy()
z = int(msgHash, 16)

for k in recovered_entropy:
  p = ((s*k - z) * inverse_mod(r, n)) % n

  if attemptSign(msgHash, p, k) == msgHash:
    print "key is: " + hex(p)
```

Listing 5: ECDSA key recovery

tempt 10 steps (the *do_rdrand* function executes at least 7 instructions), wait 1ms and then re-protect the page. Each execution of the enclave code causes exactly 29 page faults; by running the enclave in debug mode, we can determine that only 10 of these were calls to *do_rdrand*, and the remainder were other enclave code which happened to be located on the same page (in our case, the top-level *enter_enclave* function). Since our attack relies only on degrading the performance of code calling RDRAND, and does not rely on any synchronization, the presence of these other page faults makes no difference.

Our leak code collected between 200 and 250 identical values from the staging buffer for every confirmed RDRAND leak (one which successfully produced the private key), when performing 3 iterations of FLUSH + RELOAD for each byte. After making 100 unique attempts, we successfully recovered the private key (and reproduced the signature) after just this *single* enclave run in 92 of the attempts, a success rate of >90%. This success rate is without any synchronization on our entropy collection, and so without filtering out entropy which was generated by code other than the target enclave. We also do not attempt to brute-force any incorrect bytes, since we have ample time to collect the exact contents of the staging buffer.

Note that if a private key is generated by the SGX enclave itself, an even simpler attack is possible; an attacker can instead observe the random values used during creation of the key, and directly obtain the private key. This differs only from the above-described attack in that we compute candidates for p directly, rather than k, and a different approach must be taken to verify the key (e.g., computing the public key or checking a signature). We confirmed this by successfully performing such an attack against an example enclave using Intel's IPP library.

Although many cryptographic libraries perform ECDSA signatures or compute keys in this way, some (such as OpenSSL) compute the nonce k using both random entropy *and* the contents of the private key, which prevents this attack from succeeding; see Section 4.7.

Table 4.4: List of the tested microarchitectures.

(Intel) CPU	Year	Microcode	Staging Buffer Present	SMT	CrossTalk?
Xeon Scalable 4214 (Cascade Lake)	2019	0x500002c	?	✓	✗
Core i7-0850H (Coffee Lake)	2019	0xca	✓	✓	✓
Core i7-8665U (Whiskey Lake)	2019	0xca	✓	✓	✓
Xeon E-2288G (Coffee Lake)	2019	?	✓	✓	✓
Core i9-9900K (Coffee Lake R)	2018	0xca	✓	✓	✓
Core i7-7700K (Kaby Lake)	2017	0xca	✓	✓	✓
Xeon E3-1220V6 (Kaby Lake)	2017	0xca	✓	✗	✓
Core i7-6700K (Skylake)	2015	0xc2	✓	✓	✓
Core i7-5775C (Broadwell)	2015	0x20	✓	✓	✓
Xeon E3-1240V5 (Skylake)	2015	0xd6	✓	✓	✓

4.5.8 Affected processors

We ran CrossTalk on many recent Intel CPUs to check whether they are vulnerable to our attacks by checking whether RDRAND output could be leaked across cores. As shown in Table 4.4, these attacks can be performed on many Intel CPUs, even with up-to-date microcode at the time of our research.

We could not reproduce our results on our Xeon Scalable CPU, which does not appear to leak a 'staging buffer' when microcode is reading from internal resources. Intel informs us that these 'server' class CPUs, which include Xeon E5 and E7 CPUs, are not vulnerable to our attacks.

However, our results show that a variety of desktop, laptop and workstation CPUs are vulnerable to our attacks, including Xeon E3 and E CPUs. These 'client' class CPUs are used by some cloud providers to provide support for SGX, which is not yet supported on Intel's 'server' CPUs. Both Alibaba and IBM offer Xeon E3 v6 CPUs (like the Xeon E3-1220 v6 we tested) with SGX support, although they only offer them as bare-metal dedicated machines. Other cloud providers appear to use vulnerable CPUs in shared configurations; for example, Azure's preview SGX support appears to use the Xeon E-2288G, which we have shown to be vulnerable. After disclosure Intel released a complete list of the processors affected by this vulnerability [107].

4.6 Covert channel

As a proof-of-concept, we implemented a covert channel using CPUID and RDRAND, which are available to userspace applications and could be used by applications which are sandboxed or running inside a container. It implements communication

between two different physical cores.

To send a character, we call RDRAND repeatedly until the least significant 8 bits are the character we want to transmit, and then call CPUID to signal that we are ready. The receiver waits until they see CPUID output in the staging buffer, leaks the first byte of RDRAND, and then acknowledges reception by overwriting the ready signal with another CPUID leaf at the same offset. We again use the code in Appendix 4.11.1.

Even without using SMT, our naive implementation manages to transmit data between physical cores at 3KB/s, with an error rate of <5%. We only perform full (256-entry) FLUSH + RELOAD rounds until we observe a leaked character; we then perform a second FLUSH + RELOAD round for a single cache line, to verify our read was correct.

Although we need a few entries in the L1 cache to perform FLUSH + RELOAD to observe the results of transient execution, this covert channel has a minimal impact on the cache. Some calls to CPUID and short loops of RDRAND are not unusual in real-world code, but it would also be possible to use a mix of RDSEED (to pick a value to leak) and RDRAND (to leak the value), with a different synchronization method.

Short bursts of noise cannot be avoided due to other applications executing instructions (such as RDRAND) which overwrite the staging buffer themselves, but we did not encounter a significant increase in errors while running typical applications (e.g., Chrome and apache2). The covert channel can be easily disrupted by running leak primitives (which themselves overwrite the staging buffer) on another core; if only some offsets in the staging buffer are used, a one-bit covert channel could still be constructed using a leak primitive that writes to the remaining offsets.

4.7 Mitigations

4.7.1 Software changes

Since our demonstrated attacks are only relevant where RDRAND and RDSEED are used and where the resulting entropy must be kept confidential (e.g., in cryptographic algorithms), software changes may be sufficient to largely mitigate our attacks. Some software which relies on cryptographically secure random number generation has already stopped trusting hardware-based random number generators such as RDRAND. For example, the Linux kernel default is only to use them to initialize entropy stores, and OpenSSL has disabled the RDRAND 'engine' by default since 2014 (version 1.0.1f [98]).

As discussed, for SGX enclaves, RDRAND and RDSEED instructions are the only local source of trusted entropy. Nonetheless, it is often still possible to limit the

impact of our attacks. For example, an algorithm such as EdDSA can be used in place of ECDSA to eliminate the need for entropy to generate signatures. And if ECDSA is required, private data can be mixed into k when generating ECDSA signatures (as seen in OpenSSL). It may also be possible to obtain random entropy by opening a secure channel to a trusted remote server.

Countermeasures preventing performance degradation attacks against SGX enclaves exist but may be inappropriate or difficult to apply against our attack. For example, T-SGX [217] runs enclave code inside TSX transactions, which prevents single-stepping code; however, RDRAND and RDSEED always abort TSX transactions on recent CPUs, so these instructions must be run outside transactions and can be trapped. Other defenses attempt to detect high levels of interruptions (aborted transactions or enclave exits), which prevents single-stepping through SGX enclave code. One example is Déjà Vu [39], which again only protects instructions which can be run inside transactions. However, an adaptation of a non-TSX-based defense such as Varys [176] (which requires SMT) could help prevent an attack from making use of performance degradation, if tuned to an appropriately high level of paranoia.

4.7.2 Disabling hardware features

Some hardware features such as SMT and TSX (for TAA) improve the performance of our attacks. Hence, disabling SMT and TSX can frustrate (but not eliminate) exploitation attempts. These features are still in widespread use in real-world production systems, and we found both to be enabled by default in public cloud environments. Intel specifically do not recommend disabling SMT [109], but this is necessary to mitigate L1TF/MDS attacks against SGX on older CPUs.

Cloud environments, and hypervisors in general, instead attempt to mitigate SMT-based attacks by isolating code from different security domains on different physical cores [67], and flushing CPU buffers when switching between domains. However, since our attacks works across different physical cores, these mitigations are ineffective against them.

Similarly, TSX is still enabled in many environments to accelerate concurrent applications, and Intel suggest that TAA can be mitigated by using MDS mitigations to clear buffers when switching between security domains, along with microcode changes which attempt to mitigate attacks against SGX by aborting TSX transactions when a sibling thread is running an SGX enclave [100]. Intel has also updated the remote attestation mechanism to ensure the new microcode has been applied. However, since TSX transactions are still allowed on other physical cores, these mitigations are ineffective against our attacks on CPUs vulnerable to TAA.

Table 4.5: CrossTalk results after applying the microcode update containing Intel's mitigation.

		Pre Microcode Update			Post Microcode Update		
Instruction	Operands	Number of Cycles	Executed μOps	Offcore Requests	Number of Cycles	Executed μOps	Offcore Requests
RDRAND	—	433	16	1	5212	7565	6
RDSEED	—	441	16	1	5120	7564	6

4.7.3 MDS mitigations

Since our work depends on MDS-class vulnerabilities, CPUs with in-silicon mitigations against MDS-class vulnerabilities are no longer vulnerable to our attacks. Unfortunately, even these recent CPUs are still vulnerable to TAA. This can be mitigated by disabling TSX, but again, this does *not* apply to SGX, in the absence of a microcode update that disables it entirely (rather than leaving it under operating system control).

4.7.4 Trapping instructions

Trapping and emulating (or forbidding) the specific instructions our exploits need is another avenue for mitigation. Instructions that read/write MSRs are privileged and are already trapped by the operating system kernel when used by userspace. However, CPUID, RDRAND and RDSEED cannot be trapped by an operating system without use of a hypervisor.

In virtualised environments, it is possible to trap all of these instructions. First, MSR bitmaps can be used to disable access to specific MSRs from a VM, causing RDMSR and WRMSR instructions to trap. Second, hardware virtualisation extensions can be configured to cause a VM exit on a wide variety of other instructions, including RDRAND, RDSEED, and CPUID. This strategy can prevent code running in virtual machines from mounting attacks using these instructions, but may result in lower performance due to a larger number of VM exits and the need to emulate such instructions on the host.

Hypothetically, if all other relevant existing and future microcoded instructions can be disabled in VMs, and RDSEED is also disabled on the host system, then it may be possible to enable RDRAND for VMs (removing the performance penalty) without exposing RNG results. This is because running RDRAND will overwrite the relevant portion of the staging buffer and the same instruction cannot be used to leak the RDRAND results. However, the offending instructions can still be used from native execution to leak information from a VM.

Finally, trapping instructions is not a suitable mitigation strategy for SGX en-

claves, where an attacker is assumed to have control of privileged code underpinning enclaves. In fact, when SGX enclaves are run inside a VM configured to cause a VM exit on RDRAND and RDSEED, attacks are even easier. Such targeted traps allow an attacker to determine exactly when an enclave runs one of these instructions.

4.7.5 Staging buffer clearing

Similar in spirit to the VERW MDS mitigation, it is possible for microcode to clear out the staging buffer before an attacker gets a chance to leak it. However, in contrast to buffers used by existing MDS attacks, the staging buffer has cross-core visibility and an attacker can always leak RDRAND results at the same time as they are being read by another CPU. As such, existing mitigation strategies that clear out buffer content at well-defined security domain switching points are ineffective.

Nonetheless, having microcode clear out the staging buffer immediately after reading data from it would significantly reduce the time window available to an attacker, reducing the attack surface. This strategy can also work for SGX enclaves. In absence of a microcode update, software can use instructions to overwrite the sensitive regions of the staging buffer with non-confidential information after using RDRAND or RDSEED, again reducing the time window for an attacker. However, this software-only strategy is again not a suitable mitigation for SGX enclaves, where an attacker can single-step code and leak values before they are overwritten.

4.7.6 Intel's fix

Clearing the staging buffer can mitigate this vulnerability if it were possible to ensure that the staging buffer *cannot* be read while it may contain sensitive contents. Intel's proposed mitigation for these issues does just this, locking the entire memory bus before updating the staging buffer, and only unlocking it after clearing the contents. Due to the potential whole-system performance penalty of locking the entire bus, this is only implemented for a small number of security-critical instructions – specifically, RDRAND, RDSEED and EGETKEY (a leaf of the ENCLU instruction).

An MSR is provided which allows the mitigation to be disabled [101]; on CPUs which are not vulnerable to MDS, it allows an OS to instead choose to mitigate TAA (by disabling TSX). The mitigation is always applied when SGX enclaves are running, regardless of the MSR setting.

We re-ran both stages of CrossTalk on the i7-7700K with a microcode update containing this fix. Our coverage did not include EGETKEY (in an SGX enclave), but RDRAND and RDSEED are still detected by our profiling (since the buffer contents still change). However, we no longer leak RNG output from the staging buffer after running these instructions.

We observe significant differences in performance counters as shown in Table 4.5; both instructions execute far more micro-ops (around 7560, perhaps due to a busy loop), and make 6 offcore requests (rather than 1). We also observed differences with leaf 1 of CPUID, which may indicate other changes are present in the update. Post-disclosure benchmarks have shown that the deployed mitigation may reduce RDRAND's performance [142] by as much as 97% on some processors.

4.8 Discussion

We have shown that, on many Intel CPUs, reads are performed via a shared staging buffer. Microcode sometimes needs to communicate with offcore IP blocks. For example, implementing the MSRs related to power management (as discovered by CrossTalk) require communication with so-called 'PCode' running on the 'P-Unit' or PCU (Power Control Unit). Some hints can be found in Intel's patents; one patent [73] describes a fast mailbox interface, using a 'mailbox-to-PCU' interface as an example.

Intel's DRNG – the source of RDRAND and RDSEED entropy – is a global CPU resource, connected to individual cores using different buses (interconnects), depending on the platform; specifics for several platforms were described as part of an Intel presentation [48]. Originally, on the Ivy Bridge platform, the DRNG uses the so-called *message channel*. We can see evidence for this in the performance counters for Skylake-era Xeons, where the counters for RDRAND and RDSEED are in a category documented as *register requests* within the message channel. More recent CPUs directly use the sideband interface of Intel's On-Chip System Fabric (IOSF-SB) for connecting to the DRNG, which implies we may be leaking from the sideband (or some form of mailbox).

4.9 Related work

Speculative and transient execution vulnerabilities in Intel CPUs were originally reported by researchers as Spectre [128], Meltdown [149], and Foreshadow [28]. Later, MDS-class vulnerabilities (which we used in our research) were studied in RIDL [240], ZombieLoad [211], Fallout [31] and CacheOut [242]. All these papers make use of microarchitectural covert channels to disclose information. Attempts have been made to create a systematization of these vulnerabilities [32], and they have been used for other attacks, most recently LVI [234], as well as for other investigations of CPU behavior [68].

There is extensive existing research on microarchitectural covert/side-channels, with most focusing on timing. Some such attacks are only relevant in SMT situations, such as port contention [12, 20] and TLB [81] attacks, but others are more generally applicable. For example, Yarom and Falkner demonstrated

cross-core cache attacks using FLUSH + RELOAD [256]. We refer the reader to [72] for an extensive survey on microarchitectural timing side-channel attacks.

Many attacks against ECDSA using nonce leakage have been proposed [52, 255]. A systematic survey of nonce leakage in ECDSA implementations [248] discussed (among many other things) the methods used by OpenSSL, LibreSSL, and BoringSSL to generate nonces and demonstrated attacks based on partial nonce leakage. Our SGX exploit obtains the *full* nonce, making such attacks even more practical.

There have been other papers attacking cryptographic algorithms running in SGX which have not been hardened against cache or other timing-based side-channels, or memory access channels which can be observed by an attacker [253]. Recent efforts in the area include interrupt latency [91, 235], port contention [12]. and CopyCat [166] attacks. Finally, Evtyushkin and Ponomarev [65] showed that RDSEED can be used as a (one-bit) covert channel, by observing the success rate of RDSEED on a core. Since RDSEED will fail if entropy is not available, this success rate drops significantly if another core is also calling RDSEED, providing a covert channel.

In contrast to transient execution vulnerabilities, Intel and other chip vendors delegate mitigations of traditional covert/side-channels entirely to software [102], recommending the use of constant-time code manually or automatically generated [199] in security-sensitive applications.

4.10 Conclusion

We have shown that transient execution attacks can reach beyond individual CPU cores. With CrossTalk, we used performance counters to investigate the behavior of microcode and study the potential attack surface behind complex instructions whose execution may rely heavily on the operands with and context in which they are executed. We further investigated the data these instructions leave behind in microarchitectural buffers using MDS attacks and uncovered a global 'staging buffer' which can be used to leak data between CPU cores.

The cryptographically-secure RDRAND and RDSEED instructions turn out to leak their output to attackers via this buffer on many Intel CPUs, and we have demonstrated that this is a realistic attack. We have also seen that, yet again, it is almost trivial to apply these attacks to break code running in Intel's secure SGX enclaves.

Worse, mitigations against existing transient execution attacks are largely ineffective. The majority of current mitigations rely on spatial isolation on boundaries which are no longer applicable due to the cross-core nature of these attacks. New microcode updates which lock the entire memory bus for these instructions can mitigate these attacks – but only if there are no similar problems which have yet to be found.

Disclosure

We disclosed an initial PoC of staging buffer leaks to Intel in September 2018, followed by cross-core RDRAND/RDSEED leakage in July 2019. Following our reports, Intel rewarded CrossTalk with the Intel Bug Bounty (Side Channel) Program, and attributed the disclosure to our team with no other independent finders. Intel requested an embargo until May 2020 (later extended), due to the difficulty of implementing a fix for the vulnerabilities identified in this chapter.

Intel describes our attack as "Special Register Buffer Data Sampling" or SRBDS (CVE-2020-0543), classifying it as a domain-bypass transient execution attack [101]. After disclosure, Intel informed us that the issue had also been found internally, by Rodrigo Branco, Kekai Hu, Gabriel Negreira Barbosa and Ke Sun.

4.11 Appendix

4.11.1 Example code

The code in Listing 6 leaks a byte from the staging buffer using TAA, without SMT. If SMT is available to the attacker, the leaking primitive (here, CPUID) can instead be run in a tight loop on a sibling thread, and the code marked "flush some cache lines" is no longer required (see Section 4.5).

See `https://www.vusec.net/projects/crosstalk` for complete ready-to-run PoCs (proof-of-concepts).

```
/* reloadbuf, flushbuf and leak are just mmap()ed buffers */

// Flush the Flush+Reload buffer entries.
for (size_t k = 0; k < 256; ++k) {
  size_t x = ((k * 167) + 13) & (0xff);
  volatile void *p = reloadbuf + x * 1024;
  asm volatile("clflush (%0)\n"::"r"(p));
}

/* Leak primitive; as an example, here we use a CPUID leaf. */
asm volatile(
  "movabs $0x80000002, %%rax\n"
  "cpuid\n"
  :::"rax","rbx","rcx","rdx"
);

/* Flush some cache lines (until we get the right LFB).*/
for (size_t n = 0; n < 15; ++n)
  asm volatile("clflush (%0)\n" :: "r"(reloadbuf + (n + 256)*0x40));

/* Perform a TAA-based leak */
asm volatile(
  // prepare an abort through cache conflict
  "clflush (%0)\n"
  "sfence\n"
  "clflush (%2)\n"
  // leak inside transaction
  "xbegin 1f\n"
  "movzbq 0x0(%0), %%rax\n"
  "shl $0xa, %%rax\n"
  "movzbq (%%rax, %1), %%rax\n"
  "xend\n"
  "1:\n"
  "mfence\n"
 ::"r"(leak+off), "r"(reloadbuf), "r"(flushbuf)
  :"rax"
);

/* Reload from the flush+reload buffer to find the leaked value. */
for (size_t k = 0; k < 256; ++k) {
  size_t x = ((k * 167) + 13) & (0xff);
  unsigned char *p = reloadbuf + (1024 * x);

  uint64_t t0 = rdtscp();
  *(volatile unsigned char *)p;
  uint64_t dt = rdtscp() - t0;

  if (dt < 160) results[x]++;
}
```

Listing 6: Leaking a value from the staging buffer.

5 Mitigating Information Leakage Vulnerabilities with Type-based Data Isolation

Information leakage vulnerabilities (or simply *info leaks*) such as out-of-bounds/uninitialized reads in the architectural or speculative domain pose a significant security threat, allowing attackers to leak sensitive data such as crypto keys. At the same time, such vulnerabilities are hard to efficiently mitigate, as every (even speculative) memory load operation needs to be potentially instrumented against unauthorized reads. Existing confidentiality-preserving solutions based on data isolation label memory objects with different (e.g., sensitive vs. nonsensitive) colors, color load operations accordingly using static points-to analysis, and instrument them to enforce color-matching invariants at run time. Unfortunately, the reliance on conservative points-to analysis introduces overapproximations that are detrimental to security (or further degrade performance).

In this chapter, we propose Type-based Data Isolation (TDI), a new practical design point in the data isolation space to mitigate info leaks. TDI isolates memory objects of different colors in separate memory *arenas* and uses efficient compiler instrumentation to constrain loads to the arena of the intended color by construction. TDI's arena-based design moves the instrumentation from loads to pointer arithmetic operations, enabling new aggressive speculation-aware performance optimizations and eliminating the need for points-to analysis. Moreover, TDI's color management is flexible. TDI can support a few-color scheme with sensitive data annotations similar to prior work (e.g., 2 colors) or a many-color scheme based on basic type analysis (i.e., one color per object type). The lat-

ter approach provides fine-grained data isolation, eliminates the need for annotations, and enforces strong color-matching invariants equivalent to ideal (context-sensitive) type-based points-to analysis. Our results show that TDI can efficiently support such strong security invariants, at average performance overheads of <10% on SPEC CPU2006 and nginx.

5.1 Introduction

Despite advances in security engineering, information leakage vulnerabilities (or *info leaks*) remain a major security threat. Modern systems software is riddled with info leak bugs [215] and Spectre-based variations [128] have expanded the already large attack surface. Unfortunately, existing mitigations that aim to significantly reduce this attack surface incur nontrivial performance costs. In this chapter, we show such costs are not fundamental and an efficient, fine-grained data isolation strategy based on secure allocation and lightweight compiler instrumentation can mitigate info leaks, with single-digit performance overhead for practical cases of interest.

The info leak era Info leaks based on spatial (out-of-bounds, type confused reads) or temporal (uninitialized, use-after-free reads) memory errors are crucial in modern software exploitation [215]. Such vulnerabilities enable attackers to leak private data such as crypto keys (e.g., Heartbleed [178]). Moreover, they enable reliable ROP [216] by allowing attackers to bypass mitigations such as ASLR and stack cookies, or by leaking a massaged memory object location [215]. While info leaks in the architectural domain have dominated software exploitation in the last decade, the attack surface has recently expanded to the speculative domain with Spectre [128]. Spectre-BCB (Bounds Check Bypass) is a widespread example of an out-of-bounds read vulnerability using speculative execution.

Mitigating info leaks Mitigating info leaks in a practical way is notoriously difficult. Mitigations that entirely eliminate the attack surface in the architectural (e.g., memory safety [261]) and speculative (e.g., load fencing [177]) domain are expensive and normally out of reach of the performance budget available in production settings. More practical confidentiality-preserving solutions described in literature are based on data isolation: isolating memory objects in the address space to make them inaccessible from other objects vulnerable to info leaks [26, 33, 151, 185, 233]. Such solutions generally color memory objects and load operations based on the color of the objects they are allowed to access, as dictated by static points-to analysis. Loads are then instrumented to enforce such color-matching invariants at run time by means of pointer masking [33, 134], domain switching [26, 134, 151, 185, 194, 219, 233] or run-time checks [26, 33, 134]. Some (not all) of these techniques

(e.g., pointer masking) are also Spectre-safe. Some coarse-grained solutions use a few, often two (sensitive vs. nonsensitive) colors set by user annotations (e.g., labeling an allocation site for crypto keys as sensitive) [26, 33, 185], while other fine-grained solutions use many colors, based on the clusters automatically determined by points-to analysis [10, 54, 219].

Regardless of the particular scheme, existing data isolation techniques—barring those targeting very specific code patterns [134]—rely on static points-to analysis to determine the set of possible targets of load operations. Since such analysis is conservative and context-insensitive (other than having trouble scaling to large programs), the set of possible targets is often largely overapproximated even in state-of-the-art implementations such as SVF [222] or data isolation-tailored ones such as DataShield's [33]. Such overapproximations are problematic either for security, as they lead to much weaker color-matching invariants, or for performance, when additional metadata-based run-time checks are used to compensate for this weakness [10, 33]. Even without expensive run-time checks, the cost of instrumenting pervasive load operations for data isolation is nontrivial. For example, generic load pointer masking-based solutions with only two colors incur over 17% average overhead on SPEC CPU2006 [134].

TDI In this chapter, we present Type-based Data Isolation (TDI), a new design point in the data isolation space with strong performance and security guarantees against both architectural and speculative info leak vulnerabilities. TDI's key insight is that we can eliminate expensive load-based instrumentation and imprecise points-to analysis if we rearrange the address space layout and constrain pointers within specific address ranges. In particular, TDI allocates independent memory regions (*arenas*) for each memory object color, both on the heap and stack, and then uses lightweight compiler instrumentation to ensure each pointer of any given color stays within its arena (i.e., object color) by construction. TDI's design provides several benefits compared to prior work.

First, our arena-based strategy moves the compiler instrumentation from loads to pointer arithmetic operations. Not only does this eliminate any dependency on context-insensitive points-to analysis (which would degrade precision and security), it also provides much better performance. Intuitively, since many load operations depend on the same (or similar) computed pointers we can reason about, this significantly reduces the number of instrumentation points. Moreover, as we will show, such strategy is particularly amenable to other optimizations, such as efficient masking and the use of inter-arena *guard zones*. While similar optimization techniques have been explored by traditional SFI solutions [63, 138, 158, 213, 259], we show that TDI's unique pointer arithmetic design enables much more aggressive optimizations, significantly outperforming prior work. We also detail and address the challenges of making our optimizations speculation-aware.

Second, our design (and instrumentation) is entirely agnostic to the object coloring scheme. Specifically, TDI supports arbitrary coarse- or fine-grained (i.e., many-color) isolation schemes—with colors determined by explicit user annotations or static analysis—despite significantly outperforming prior (annotation-based) data isolation solutions limited to coarse-grained (e.g., 2-color) schemes. By default, TDI uses simple static type analysis [237] to isolate each individual object type in its own arena. This scheme supports annotation-free protection and provides very fine-grained isolation—well beyond object coloring based on the clusters determined by points-to analysis used by WIT [10] and others [54, 219].

With such a scheme, architectural or speculative info leak vulnerabilities cannot be exploited to leak data across any two given object types. For example, a crypto key can never be leaked by means of a vulnerable string or buffer of any other type. We show that TDI can flexibly protect such situations in OpenSSL with both coarse- and fine-grained coloring.

Third, our design eliminates the need for imprecise and hard-to-scale points-to analysis altogether. Our masking instrumentation does not rely on any particular object coloring scheme, as we simply constrain pointers within their predetermined arena rather than attempting to enforce color-matching invariants by reasoning about the targets of load operations. This strategy matches the precision of the underlying object coloring scheme, with no overapproximations. As a result, our standard configuration using per-type arenas can enforce color-matching invariants equivalent to load-side counterparts using ideal (context-sensitive) type-based points-to analysis.

To summarize, our contributions are:

- We design and implement a prototype[1] of TDI, a low-overhead Type-based Data Isolation system based on lightweight compiler instrumentation.
- We explore the challenges of efficiently implementing such instrumentation, presenting aggressive but speculation-aware optimizations allowing TDI to be applied to real-world code with low performance overhead.
- We automate TDI's object coloring using state-of-the-art type analysis, resulting in a fine-grained isolation system that aggressively contains info leak vulnerabilities in both the architectural and speculative domain.
- We evaluate our TDI prototype using standard benchmarks and the modern nginx web server. Our results show TDI incurs only single-digit average performance overhead on SPEC CPU2006 and nginx.

[1] Our current code can be found at `https://github.com/vusec/typeisolation`.

5.2 Threat model

We assume a typical modern software exploitation scenario with an attacker exploiting either spatial (out-of-bounds reads, type confused reads) or temporal (uninitialized reads, use-after-free reads) info leak vulnerabilities while all the standard modern mitigations, such as ASLR, DEP, stack cookies, etc., are in place. The attacker has not (yet) achieved control-flow hijacking, and aims to leak private data (e.g., crypto keys) or information needed to hijack control (e.g., pointers or stack cookies). The attacker can exploit both classical and speculative info leak vulnerabilities. A typical example in the former category would be a classical out-of-bounds read with an attacker-controlled value which is not bounds-checked before being used to index an array. A typical example in the latter category would be a speculative out-of-bounds read with an attacker-controlled value which is only architecturally bounds-checked before being used as an array index. An attacker may speculatively bypass the bounds check and leak data using a (e.g., cache) covert channel a la Spectre-BCB [128]. While we mostly focus on speculative out-of-bounds reads used by the widespread Spectre-BCB variant, all the other classical info leak vulnerabilities exploited in the speculative domain (e.g., speculative type confusion) are in scope. Other unrelated Spectre (e.g., Spectre-BTB [128]) or transient execution variants (e.g., MDS [240]) are out of scope and addressed by complementary (e.g., hardware) mitigations. We also consider other vulnerabilities (e.g., memory corruption) out of scope.

5.3 Overview

TDI hardens C/C++ programs by preventing pointers from escaping the memory area—'arena'—in which they were allocated. Our design, as shown in Figure 5.1, relies on both compiler instrumentation and runtime code. At compile time, we instrument all pointer arithmetic to ensure that all pointers stay in their original arena. At run time, our arena-based allocator allows programs to allocate memory in appropriate isolated heap/stack regions. Besides explicit program annotations, TDI also offers support to automatically allocate both heap and stack objects in arenas based on their type.

We constrain pointers to their original arena by *masking* pointers to preserve the upper 32 bits during pointer arithmetic (i.e., we only allow the lower 32 bits to change), as shown in Listing 5.1. As long as all arenas are at least 4GB in size and appropriately aligned, this ensures that pointers always point to the same arena both before and after pointer arithmetic.

Masking pointers after every individual instance of pointer arithmetic would be very inefficient; for example, a pointer used to access successive elements of a struct or array would need to be repeatedly re-masked. As shown in Figure 5.2,

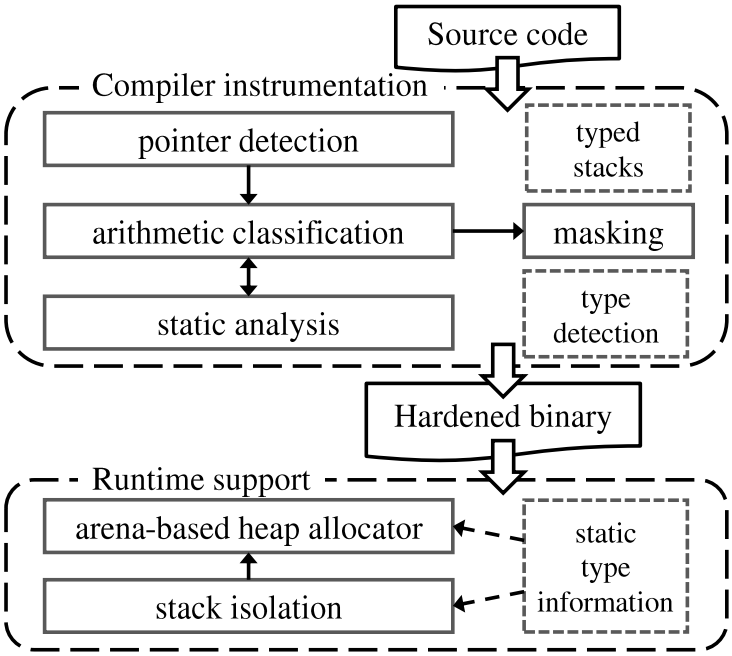


Figure 5.1: High-level overview of TDI.

we relax the need for masking by adding guard zones around each arena. If a base pointer is known to be inside an arena, then we know that any pointer within 4GB is either in the same arena, or in a guard zone. This insight allows us to optimize TDI by identifying and removing unnecessary masking.

5.4 Instrumentation

In this section, we discuss the design of TDI's compiler instrumentation, which enforces our security guarantees by preventing pointers from crossing arena boundaries. Although our design is not compiler-specific, we discuss some of the details and challenges in the concrete context of LLVM.

TDI allocates each memory object (whether stack or heap) in an arena based on its type; the 32 most significant bits of each pointer identify its arena. The result of pointer arithmetic may end up in a different arena, which would allow an attacker to break isolation. We prevent this using pointer masking; however, masking after every instance of pointer arithmetic results in unacceptable overhead. To determine which pointers we must mask, we divide pointers in three classes:

- A *valid pointer* can be proven to be in the same arena in which the pointer

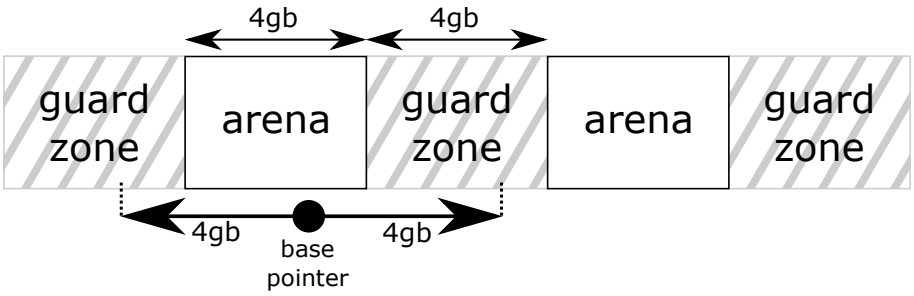


Figure 5.2: Overview of TDI's arena layout, with guard zones of ≥4GB.

from which it derives was allocated;

- A *safe pointer* either points to the same arena or to a guard zone, in which case a dereference will fault;
- An *unsafe pointer* may point to a different arena.

These three classes are illustrated in Figure 5.3.

We can identify valid pointers based on their sources; for example, allocation functions always return valid pointers. If we can prove that the result of pointer arithmetic is within 4GB of the original pointer, we can classify that result as safe; otherwise, such a result is unsafe.

Dereferencing a valid or safe pointer does not threaten security, but we must ensure that unsafe pointers are not dereferenced by loads or stores.

The *base pointer* of a pointer is the most recent known valid pointer that a pointer derives from. We can ensure that a pointer is valid by overwriting the high-order bits (which represent the arena) with those of its valid base pointer, i.e. *masking* the pointer, as described above. Such a masked pointer is always valid, since it points inside the same arena as the valid base pointer, and can then be safely dereferenced.

To ensure a valid base pointer is always available, we only allow safe pointers to be used locally within a function; any pointer which escapes a function's scope (e.g., by being stored to memory, returned from a function, or passed as a parameter to another function) must be valid (i.e., must be masked if necessary). This means we can determine pointer categories with only intraprocedural analysis, since any pointers from outside the scope of the function must be valid.

To summarize, we mask pointers in two situations:

- When an unsafe pointer may be used by a load (or optionally, as the address used by a store). This means we cannot prove that it is safe (<4GB away from a valid pointer), and may now be pointing to a different arena.

```
char myFunction(char *validPtr, size_t idx)
  char *newPtr = validPtr + idx;
  upperBits = (validPtr & 0xffffffff00000000);      ①
  lowerBits = (newPtr & 0xffffffff);
  newPtr = upperBits | lowerBits;
  char ret = *newPtr;                               ②
  return ret;
```

Listing 5.1: This pseudocode shows a potentially-unbounded memory access ②. TDI's instrumentation (marked with a dark background) preserves the upper bits of newPtr ①. This ensures that the access at ② cannot be further than 4GB from validPtr, and so cannot escape validPtr's arena.

- When a pointer value escapes the local analysis. For example, when a pointer is stored in memory, passed as a function argument or used as a return value. This ensures that all pointers entering a function are themselves valid.

5.4.1 Challenges

Our instrumentation identifies pointer arithmetic and classifies the results as valid, safe, or unsafe. Doing this analysis on real-world code must overcome the challenges below.

Arithmetic on non-pointer types

Since pointers can be cast to/from other types, such as integers, we need to distinguish pointer arithmetic from non-pointer arithmetic; we want to mask all resulting pointers, but pointer arithmetic is not always intended to result in a valid pointer. For example, pointers may be subtracted to obtain a delta. Even though the result of such arithmetic may escape local scope, the result must not be masked by our instrumentation, since that would produce incorrect code. Compiler passes also manipulate pointers, including converting them to untyped values.

We resolve this by doing *pointer detection*, allowing us to find which variables/intermediates are truly (non-)pointers.

Non-constant offsets

Array indexes or other pointer offsets are often non-constant. Our efficiency relies on being able to prove that such offsets are within 4GB of a known-valid pointer, allowing us to mark the result as safe; however, offsets are often stored in 64-bit variables.

Pointer arithmetic is often performed on the result of previous pointer arithmetic; since safe pointers are *not* necessarily valid, efficient instrumentation also

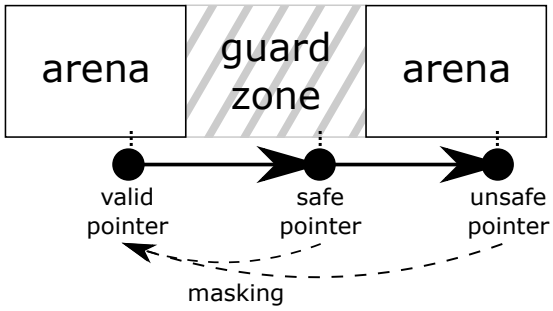


Figure 5.3: Valid, safe and unsafe pointers; safe pointers are always within 4GB of a valid pointer into the same arena, which means that while they *may* point into a guard zone (as illustrated), they will never point to a different arena. If the result of pointer arithmetic may lie in a different arena, we call the result an *unsafe* pointer and mask it before use (as shown in Listing 5.1).

requires that we handle such 'chained' arithmetic. Pointers may only be modified and/or used in some program paths, making it more difficult to reason about their behavior. A common example is a pointer which is incremented inside a loop, as we see below.

We resolve this using the static analysis we describe below, *range analysis* and *dominating pointer access analysis*.

Speculative execution

Since we include speculative execution (Spectre) in our threat model, bounds provided by existing static analysis can be unsafe. For example, LLVM's ScalarEvolution (SCEV) will use array bounds checks to prove that an array access is in-bounds, but that array may still be accessed on a transient path. We resolve this by ensuring our custom analysis is valid in the context of speculative execution.

5.4.2 Pointer detection

We found that, in many cases, variable types marked in the original source code and in the compiler IR do not accurately reflect whether a variable is used as a pointer or not. To ensure masking wherever needed while retaining compatibility with existing code, we designed static analysis to detect pointer/non-pointer status. Our main insight is that variables should be classified (where possible) based on how they are used rather than how they are declared.

Our approach marks variables as pointers, non-pointers, or negated pointers in several steps. In each step, we mark variables based on their usage (if their type is still unknown), and then we use forward and backward propagation to in-

fer the type of other variables. For example, our first step considers all pointer dereference operations and marks their operands as pointers. The main idea of our propagation is that a pointer plus an offset is still a pointer, a pointer minus another pointer (or plus a negated pointer) is an offset, and other operations typically yield non-pointers. Details are in Appendix 5.15.

5.4.3 Categorization

To be able to apply efficient masking, we categorize all pointer arithmetic results we detect as valid, safe, or unsafe. We start with the first such instructions in a given function and proceed top-down (based on domination). Since the classification of arithmetic can depend on other arithmetic, we can also temporarily place them in a group of *unknown* arithmetic. We repeat this process until all arithmetic has been classified; if there are only circular dependencies left, we mark the first remaining arithmetic as unsafe and continue.

We only need to mask pointers which are dereferenced by a load (or store) instruction, are used in pointer arithmetic, or escape the local function. This includes both direct and indirect uses (including integer casts which are later used by arithmetic). We ignore instructions which are not used in such ways, which removes almost all ambiguous cases of pointer arithmetic. We discuss some remaining cases in our evaluation.

At this stage, we apply *dominating pointer access* analysis, which we describe in Section 5.4.6, which can prove that some pointers are valid or safe. Otherwise, we trace the instruction's base pointer. If the source is outside the local scope (a function argument, returned from another function, or loaded from memory), we consider the source to be valid. If the source is a pointer arithmetic instruction (or a merge of several such instructions), then we use the classification of that instruction (and defer processing if that instruction has not yet been classified). Otherwise, we consider the source to be unsafe.

Our classification for the result of pointer arithmetic then depends on the classification of the base pointer: (1) a valid base pointer means that the result is safe; (2) a safe base pointer means that the result is unsafe (and we will mask the pointer before using it); (3) an *unsafe* base pointer must itself be masked. Since a masked pointer is valid, the result of the arithmetic is then safe. If we cannot prove that an offset is bounded to 4GB (nor, as described below, truncate the offset to enforce this), we mark the current arithmetic result as unsafe.

5.4.4 Range analysis

One important supporting component of our analysis involves determining the maximum range of offsets used in pointer arithmetic. We require a Spectre-BCB-aware analysis which calculates the worst-case (largest) distance to a known-valid

```
void func(char *validPtr, size_t idx) {
    char *ptr = validPtr;
    if (...)
①     ptr = ptr + 2;
    else
②     ptr = ptr + 4;
③ char val = *(ptr + 1);
}
```

Listing 5.2: The access at ③ is safe, because the maximum offset to validPtr is less than 4GB.

pointer on all paths within a function, which prevents using standard compiler analysis (such as LLVM's SCEV). Instead, we designed an alternative analysis which performs a recursive check of all conditional control flows. We calculate the bounds by considering instructions such as truncations, arithmetic (e.g., AND operations), the bitwidth of loads/variables, and sources such as constant values. When merging several possible bounds (e.g., phi nodes), we use the worst-case bound among all incoming values.

One special case is where we can prove a 32-bit bound, i.e. a maximum offset of 4GB, which is multiplied by the object size, such as during an array lookup. We can ensure that such an index is always safe by ensuring that the guard zone is at least 2^{32}*sizeof(type) bytes (see Section 5.6). The majority of such calculations are performed for types ≤ 8 bytes (64-bit pointers or doubles); if our arenas use 32GB guard zones, then we can classify all scaled 32-bit offsets for such types as safe, assuming the base pointer is known to be valid.

When we are using this range analysis to determine whether a pointer is safe, and the distance to a valid pointer cannot be proven to be less than 4GB, simply truncating the offset in bytes to 4GB (32 bits) is sufficient to ensure that the resulting pointer is safe. Note that masking is still required to make it valid, and that truncation is unnecessary if the analysis later decides to mask this pointer.

5.4.5 Chaining distances

A pointer is safe when the distance to a valid pointer is known to be less than 4GB. Even when pointer arithmetic is based on a safe pointer, we can compute bounds using the offsets of previous arithmetic, and use that information to prove that the result is still within 4GB of a known-valid pointer. A simple example is shown in Listing 5.2. Here, our analysis checks the phi node for `ptr` at ③, which has incoming values from ① and ②, and concludes that the maximum offset to a known-valid pointer is less than 4GB even though the intermediate pointers may not be valid.

```
void func(char *validPtr, int idx) {
    char *A = validPtr + idx;
    char *B = A + 1024;
① char valA = *A;
② char valB = *B;
}
```

Listing 5.3: After ①, pointer A is known to be valid, so when execution reaches ②, we know that Pointer B is safe.

Again, standard compiler analysis could provide this information (by calculating the distance between pointers), but it does not consider speculative flows. Instead, we use our own (simple) control-flow-insensitive distance analysis, which is also needed to support several other analysis stages.

5.4.6 Dominating pointer accesses

When a safe pointer is dereferenced to access (load or store) memory, any code after that memory access can assume that the used pointer was valid. If a safe pointer is *not* valid, then it must point into a guard zone; after such a pointer is accessed, a fault will occur, and execution will not continue.

This allows us to improve our categorization of pointers for all code dominated by (i.e., guaranteed to run after) a memory access. Existing compiler static analysis does not (easily) provide the information we need. Alias analysis focuses on proving that pointers are *never* pointing to the same *object*, while our analysis must prove that pointers are *always* pointing to the same *arena* (i.e., within 4GB of a valid object).

Listing 5.3 provides a simple example. The access at ② is dominated by the access at ①. Since the pointer B at ② is less than 4GB away from the pointer A used at ①, then B could be categorized as safe in the context of ②, and architecturally it could be accessed without masking. However, note that in this particular case, in the face of a Spectre attack, B might still be dereferenced speculatively even if A points to a guard zone. This limits the applicability of this type of optimization in a threat model where we must also consider transient attacks.

However, only a small number of loads/stores will be speculatively queued [157]. As long as we limit the distance between successive accesses (we limit it to 64kB), and ensure that base pointers are *always* valid (masking them where needed), we can make such optimizations speculation-aware.

For example, consider Listing 5.4, where `ptr` is incremented inside the loop. On the first iteration of the loop, the new `ptr` at ③ is based on the valid pointer from ①. On later iterations, the new `ptr` at ③ is dominated by the previous access

```
void func(char *validPtr, size_t size) {
① char *ptr = validPtr;
   for (size_t n = 0; n < size; n++) { ②
③    ptr = ptr + 1;
④    char val = *ptr;
  }
}
```

Listing 5.4: The pointer arithmetic at ③ is dominated by the loop header at ②. Since all candidates for ptr are valid in the context of ②, we know that ptr is safe at ④.

at ④. We can check whether all incoming values at ② (the loop header) are valid in the context of ④, i.e. whether *all* the potential values of `ptr` are valid at this point. This allows us to determine that the access at ④ is safe.

Finally, we consider Listing 5.5, which uses a loop induction variable rather than modifying the pointer in the loop. Even though we avoid use of non-speculative-safe analysis, we can analyze simple cases where pointers are being offset by a fixed stride based on a loop induction variable. In this example, we detect that the offset `n` is a loop induction variable which increments (or decrements) at each iteration, and that the distance to the known-valid pointer from the previous loop iteration is <4GB (or, again, a fraction of this due to speculative safety). This allows our analysis to confirm that the access is valid in the context of the loop. Although in practice the involved strides are often much larger (e.g., stepping over arrays of large structs), the same reasoning applies.

TDI provides static analysis for each of the three situations described above, and uses the information obtained to mark pointers as valid or safe. Importantly, we chain the analysis described above; for example, we can detect accesses which are close to a known-valid access, which in turn was based on loop induction variable analysis. Combined, this allows us to significantly improve the performance of many inner loops and other performance-sensitive code, by removing unnecessary masking where we can prove pointers to be safe.

5.4.7 Masking

Finally, although our design relies on the static analysis described above to avoid the need for masking where possible, our instrumentation also needs to efficiently emit code for applying masks where it cannot be avoided.

We can efficiently mask pointers on both x86-64 and AArch64 by making use of implicit clearing of the upper 32 bits of 64-bit registers, when they are used as 32-bit registers. A 32-bit XOR of a valid pointer with a (potentially) unsafe pointer, followed by a 64-bit XOR, will preserve the lower bits of the unsafe pointer, but overwrite the upper 32 bits.

```
void func(char *validPtr, size_t size) {
  for (size_t n = 0; n < size; n++) {
      char val = *(validPtr + n);
  }
}
```

Listing 5.5: Each access in the loop is safe, due to being at most 1 byte away from a known-valid pointer.

We found the code emitted by compilers for this sequence to be very efficient. For example, the code in Listing 5.6 is a typical x86-64 code sequence emitted by LLVM for our pointer masking; bitmask-based arithmetic is used in cases where such a sequence would be inefficient.

5.5 Arena-based allocation

The only requirement that TDI imposes on allocated objects is that preserving the upper bits of a pointer must not result in corruption of valid pointers; that means that the lower N bits of any pointer must remain constant for the entire object, and thus that allocated objects must not exceed the arena size. If necessary, larger objects can be supported by using a larger arena size, or relaxing checks in some circumstances; we discuss some real-world examples later.

TDI relies on objects being allocated within appropriate arenas. Source code annotations and runtime calls could be manually added to source code to allocate objects in appropriate arenas. However, this requires a substantial investment in time, and only protects a subset of code/data. Instead, TDI provides automatic type-based isolation based on type information from previous work [237].

For heap allocations, our arena-based allocator allows objects to be allocated in a specific arena. Where type (or callsite) information is available, TDI allocates each type of object in a different arena. Otherwise (e.g., allocations by uninstrumented libraries) we allocate in a generic untyped arena, isolated from other arenas. All arenas are allocated dynamically, allowing isolation decisions to be made at runtime.

We allocate the stack in an independent arena, isolating it from other arenas. We can also enforce type-based allocation on the stack, by isolating stack objects in typed arenas.

To support programs which use out-of-bounds pointers which are just after or before the valid memory range of an object, we also reserve some space at the start and end of every arena. This is required because our instrumentation can mask such pointers to ensure they are valid; a subsequent unmasked use (as a safe

```
; RAX: valid pointer, RCX: pointer to be masked
xor %eax, %ecx ; xor low 32 bits (clears upper bits)
xor %rax, %rcx ; xor all bits
```

Listing 5.6: Example x86-64 code for pointer masking.

pointer within 4GB) may end up accessing the guard zone rather than the intended object.

5.6 Implementation

5.6.1 Compiler instrumentation

We built our prototype implementation based on LLVM 9.0 (together with clang), using a pass ($\sim$ 2500 SLOC) that implements the static analysis and categorization described in Section 5.4.3, along with instrumentation of unsafe arithmetic and/or offset truncation. Our distance and loop analysis is currently only supported on GEPs. We perform our transformation at LLVM IR level, running our pass (and some support passes) after other LLVM IR transformation passes are complete.

Our pointer detection uses type-based alias analysis (TBAA) metadata (added by clang) to help find loads of pointers (even when typed as integers). We treat vectors of pointers as pointers, instrumenting arithmetic on vectors where needed; our masking is often overly conservative since some of our analysis does not support vectors, and we mask all pointers inserted into aggregates (structs and LLVM-level arrays).

5.6.2 Arena allocation

We implemented an arena-based allocator on top of tcmalloc [77]. We chose to build our allocator on tcmalloc to allow fair performance comparisons with a non-isolated baseline, but we could also modify an existing arena-based allocator such as PartitionAlloc to provide the needed behavior. We partially based our work on the patches from Type-after-Type [237].

We do not use the first and last tcmalloc page (8K) of each arena, so that pointers which are just before or after an allocated object are left unmasked; this also keeps stack pointers inside arenas. If an arena runs out of space, we allocate additional arenas with new guard zones.

As discussed in Section 5.4.4, we optimize typical index scaling cases by mandating 32GB guard zones; this allows $\sim$ 3600 4GB arenas in the typical 47-bit userspace address space of current x86-64 processors (see Section 5.9.4).

The addresses $\leq$32GB should be left unmapped, to prevent a (valid) NULL pointer from being used to access an arena. This requires building binaries as position-independent executables (PIE) – the standard for most recent Linux distributions. A similar relocation is required for LLVM's SLH mitigation.

5.6.3 Type-based isolation

We use Type-after-Type [237] (TAT) to obtain type information. TAT detects types at allocation sites based on C-level data type information including type casts and `sizeof` operators. When such data type analysis fails (i.e., for untyped `char` allocations), TAT resorts to using the callsite ID as the type.

To deal with custom allocator wrappers, TAT conservatively detects and aggressively inlines them *before* the type analysis. This strategy adds context sensitivity to the analysis, boosting precision and resulting in fine-grained type identification without run-time tracking [9]. In particular, this reduces the number of untyped allocations and also ensures the residual untyped allocation callsite IDs yield one type per allocation context (including custom wrappers) rather than one per allocator call.

We ported TAT to LLVM 9.0, fixed various bugs, expanded its allocation function support, and modified its runtime library to allocate safe stacks in typed arenas. Stack pointers are stored in per-thread arrays; static indices for each type are assigned during LTO (link-time optimization).

SafeStack (which TAT builds upon) uses SCEV to statically determine whether accesses are in-bounds (and thus safe). Due to speculative flows, we only allow this for constant offsets; this results in more objects being placed on typed stacks.

TDI reduces the number of such moved objects using a custom interprocedural analysis pass which marks function pointer arguments which are only accessed in-bounds. This allows some objects – in particular, pointers to variables used to store lengths/sizes – to remain on the safe stack.

5.7 Evaluation

We first consider some examples of TDI's mitigation of classical and Spectre-BCB vulnerabilities, and then provide an evaluation of benchmarks (SPEC CPU2006/2017) and a web server (nginx with OpenSSL). We consider three basic configurations in all our evaluations:

- Typed allocation: We apply type analysis and allocate all heap and stack objects in typed arenas. This could be seen as an spatial extension of Type-after-Type.
- Masking: We instrument pointer arithmetic; since this requires arena alignment, we run with our arena-based allocator placing everything in a single

heap arena; similarly, all stack contents are a single arena.

- Full protection: We apply our complete defense, including typed allocation and instrumenting pointer arithmetic.

5.7.1 Vulnerabilities

To confirm TDI's protection against real-world vulnerabilities, we checked relevant issues in the CVE database; here, we discuss a small selection to illustrate the different ways in which TDI can mitigate issues:

CVE-2016-1234 (glibc): Linear stack buffer overflow in glob. Although we cannot build all of glibc, we built glibc's glob.c using TDI after removing some clang-incompatible lines from headers. The stack buffer is identified as having a unique type and is allocated in an isolated arena, which would prevent the overflow from being exploitable.

CVE-2018-16845 (nginx): *ngx_http_mp4_read_atom* subtracts a header size from an (unchecked) value read from an mp4 file, resulting in an integer overflow. One potential exploitation path uses the 'trak' parser to recurse into the vulnerable function, allowing control of *end* and potentially adding a 64-bit offset to *buffer_pos* (a pointer to the input buffer). We confirmed that, as expected, TDI masks this (and other) pointer arithmetic. Since the buffer is allocated from a unique callsite and thus is placed in a separate arena, this appears to fully prevent exploitation of the bug.

CVE-2018-16890 (curl): Integer overflow allows an attacker to disclose data (via a NTLMv2 response) via an attacker-controlled 32-bit array index. The array is allocated by a malloc call inside `Curl_base64_decode`; again, TDI does not need to apply any masking, since heap arena isolation prevents an attacker from disclosing any data except other allocations from that callsite.

CVE-2019-3859 (libssh2): This CVE covers several different issues; we consider one in `kex.c`. Attacker-controlled 32-bit lengths provided during SHA1 key exchange are not checked, potentially leading to reads beyond the end of a buffer; the sum of the offsets can be >4GB, but TDI masks the pointer arithmetic and mitigates the vulnerability.

5.7.2 Spectre-BCB

To ensure that our instrumentation is applied to potential Spectre-BCB gadgets, we applied TDI to the corpus of 27 Spectre v1 variants provided by the authors of Spectector [86] (including the 15 examples from Kocher [127]). TDI correctly masks the potentially out-of-bounds loads for all 27 examples.

We also examined four Spectre demos from Google's Safeside [80] suite, which we modified to allocate the private (secret) string using malloc. Originally, the public and private strings were both static global strings, stored in the same arena.

We also made changes to prevent truncation or masking when calculating cross-arena offsets, which would typically be attacker-supplied rather than calculated by the code itself, and confirmed that the examples work when TDI is not applied[2].

We mitigate three of these four examples:

spectre_v1_pht_sa

This is a Spectre-BCB example, which is covered by our threat model; as expected, the private string no longer leaks when TDI is applied, since the array access is correctly masked.

spectre_v1_btb_sa

This example uses a mispredicted indirect branch to cause type confusion. Even though this is not covered by our threat model, the private string no longer leaks when TDI is applied. The transient (mispredicted) branch target uses an out-of-bounds read and TDI prevents it from accessing the private string. (If we modify the code to remove the out-of-bounds read, the example leaks the private string after TDI is applied, as expected.)

spectre_v1_btb_ca

This uses a mispredicted indirect branch to transiently execute code to read the private string. Since the transiently executed code is intended to be able to read the private string, this is outside our threat model, and the code leaks the private string even after TDI is applied.

spectre_v4

This example is intended to demonstrate Spectre-SSB. It causes an out-of-bounds array index to be transiently used while waiting for a store to complete. Since the array index is out-of-bounds, TDI masks the array access and the private string no longer leaks.

5.7.3 SPEC CPU2006 and CPU2017

We evaluated the performance of TDI using SPEC CPU2006, to aid comparison with previous work. We also present results from SPEC CPU2017 (without OpenMP). We ran these evaluations on Xeon E5-2630 v3 CPUs with 64GB of RAM. Transparent Huge Pages were disabled and the benchmarks were pinned to a single core. In both cases, we include all C/C++ benchmarks and use the

[2] We excluded the out-of-scope ret2spec demos since they do not leak any data on our Cascade Lake machine, presumably due to hardware mitigations.

reference SPECspeed data. We run each benchmark/configuration at least 5 times; the reported numbers are the median value from these runs.

We modified some of the benchmarks to make them build and run successfully with TDI. We also applied these changes to the baseline where relevant. These changes can be grouped into three categories (details are in Appendix 5.13):

- build problems: we added an #include to dealII's code.
- undefined pointer arithmetic: we apply gcc patches and exclude one perlbench function.
- large allocations in CPU2017: we disable LTO (and thus instrumentation) for the SPEC I/O wrapper for xz, which allocates a >4GB array for one test. We also annotate one struct type in deepsjeng (via flags), which is used only for a >4GB hash table (accessed via a masked index).

For our baseline, we compile the benchmarks using an unmodified LLVM, and link against an unmodified version of tcmalloc. We compiled all benchmarks with -O2, and PIE flags. All our benchmarks were compiled using LTO (via the gold linker), and the same flags were passed to the linker.

We did *not* use -fno-strict-aliasing; TBAA information improves our pointer analysis, and we did not have miscompilation issues in this version of LLVM. Otherwise, clang could be modified to output TBAA metadata despite this flag.

Figure 5.4 shows runtime overhead for our three basic configurations. Geometric means are 2.5% for typed allocation, 5.8% for masking, and 8.4% for the combined full TDI protection. This is significantly more efficient than prior load pointer masking-based solutions with only two colors (e.g., over 17% on CPU2006 for [134]). We can see that overhead is high (>15%) for two benchmarks, perlbench and xalancbmk, due to the cost of typed allocation; perlbench's overhead is due to the type-safe stack (2% heap, 13.4% heap+stack), while xalancbmk's overhead is due to both (11.5% heap, 21% heap+stack). Much of the perlbench overhead appears to be due to LLVM register allocator issues [237], and could be mitigated by limiting inlining.

We also evaluated runtime overhead for two alternatives (full results can be found in Figure 5.11 in Appendix 5.16):

(1) TDI without stores; here, we do not instrument pointers used only by stores. Although the benefit is significant for sjeng, hmmer suffers due to different base pointers being masked on the hot path (which could be resolved with runtime profiling). The geomean on CPU2006 is 8.1%, compared to 8.4% for full protection; the cost of reduced protection would seem to outweigh this minor performance gain.

(2) TDI without dominator pointer access analysis. This analysis significantly benefits some benchmarks (e.g., namd, hmmer, sjeng and xalancbmk), and reduces the geomean overhead from 10.4% to 8.4%. We believe that improving our

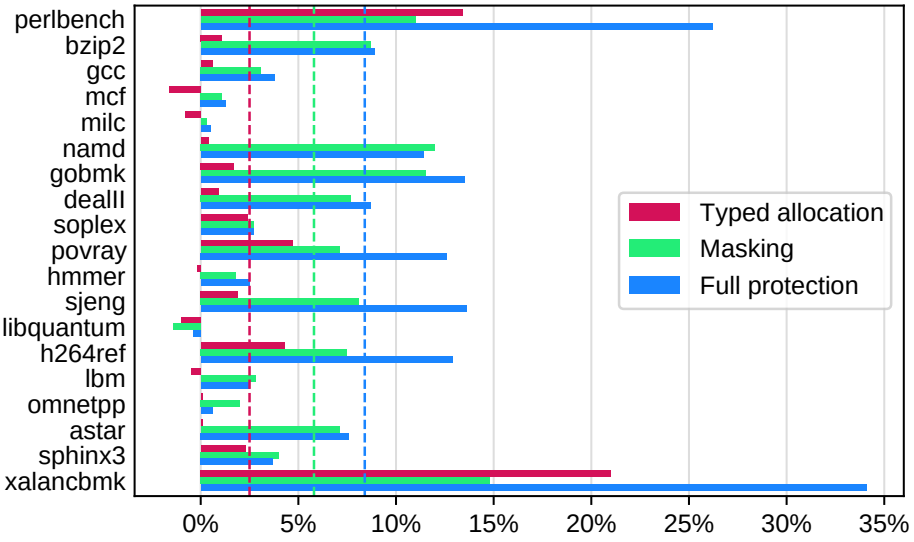


Figure 5.4: CPU2006 runtime overhead

analysis could probably improve this overhead further; in any case, the benefit seems worth the implementation effort.

The runtime overhead of TDI on SPEC CPU2017 is shown in Figure 5.5; the geomean (12.5%) is higher than that of CPU2006. Overall, masking is the source of the majority of the overhead (geomean 8.0%), although omnetpp suffers from inefficiencies in our heap allocation. This is partially due to shortcomings in the analysis of our prototype; the CPU2017 versions of x264 and imagick contain significant numbers of (non-GEP) pointer arithmetic instructions in situations unsupported by our analysis, and are conservatively masked.

Our benchmarking of xz shows high variance, with a standard deviation of ∼5% (including the baseline). Other benchmarks (e.g., mcf and lbm) have stddev <1%; the speedups shown when using typed allocation are consistent. As discussed by Mytkowicz et al. [170], measurement bias is difficult to avoid in this form of evaluation. Our instrumentation and runtime inevitably have side-effects which will influence performance. For example, arena allocations may cause more cache conflicts; allocations at the start of arenas will share lower bits, and many arenas are only used for small allocations. This could be mitigated by adding small offsets to the arena base, e.g., based on internal type IDs or allocation order. However, we did not observe any significant performance change when subtracting small (cache-line-sized) offsets from the base pointers of the typed stacks.

Full TDI's memory overhead (peak RSS) on CPU2006 has a geomean of 15.5% (vs unmodified tcmalloc); this is due to increased memory fragmentation caused

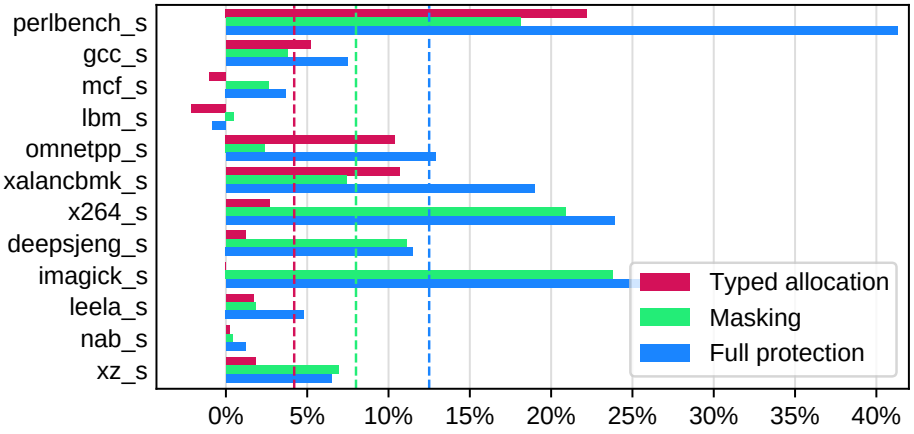


Figure 5.5: CPU2017 runtime overhead

by our allocation strategy, amplified by tcmalloc configuration (e.g., minimum page cache sizes) which are inappropriate for arenas. To ensure fairness of our baseline comparison, we left these values unmodified. Details can be found in Figure 5.9 in the appendix.

We also compared the runtime overhead of TDI to LLVM's Speculative Load Hardening (SLH) mitigation. SLH has a significantly stronger speculative threat model which aims to prevent loads from executing by mixing predicate state (from branches) into the pointers being loaded, providing a mitigation against the majority of Spectre v1 attacks. However, overhead when applying (x86) SLH to CPU2006 is prohibitively high (geomean 75.6%), and it provides only speculative safety. (Overhead should be slightly lower without indirect call/jump hardening, but we encountered code generation errors when disabling it.) Again, detailed results are in the appendix.

5.7.4 nginx

We tested TDI using the nginx 1.18.0 web server. We used default options and enabled SSL, but disabled the 'geo' module (due to undefined behavior, see Appendix 5.13). We linked against OpenSSL 1.1.1h[3] using LTO (and -O2), hardening both nginx and OpenSSL with TDI. We confirmed that the OpenSSL tests pass after full hardening, and used a hardened `openssl` binary to generate 2048-bit RSA keys for SSL.

Note that nginx does not fully benefit from our automated type-based protection, since allocations in nginx's pools (including shared memory slab pools) lose

[3] configured with `no-shared`, `no-asm` and `no-zlib`.

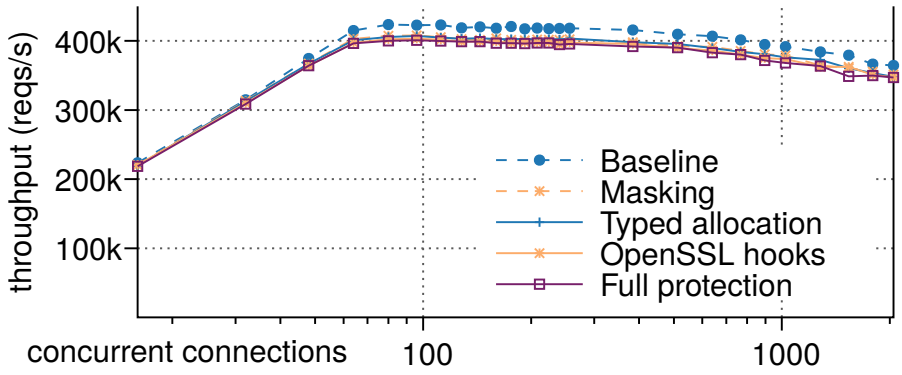


Figure 5.6: nginx throughput

the benefit of intra-pool type isolation. However, since different types of pools are identified based on callsites, pools containing disjoint types remain isolated from each other, as well as from the many other arenas identified by the type analysis (Section 5.7.6). One improvement could be to allocate each pool instance in a separate arena, providing finer-grained isolation.

The 'OpenSSL hooks' configuration uses TDI's instrumentation but assigns arenas using OpenSSL's allocator hooks; as we discuss later, such arenas are surprisingly coarse-grained.

We evaluated nginx by serving a small file (64 bytes) via SSL (with default settings), using two Xeon Silver 4110 machines with 100Gb/s Ethernet (plain HTTP is largely I/O bound). We configure nginx to use 16 workers, and use 16 threads of wrk2 [252] to make the requests.

Throughput results are shown in Figure 5.6 (median of 3 runs of 30s each); all cores are saturated for $\geq$96 connections. Saturated throughput at that point is 5.4% lower than the baseline for full TDI, 3.6% for masking, 3.8% for the typed allocator and 4.8% for the hook-based allocation. 90th percentile latency is 4.7% higher for full TDI, and 2.9%, 3.6% and 3.9% for masking, typed allocator and the hooks respectively.

5.7.5 Instrumenting system libraries

TDI's protection does not rely on complete instrumentation of system libraries, since pointers passed to external functions or stored in memory are always masked. For example, a call to `memcpy` will always be provided with valid pointers to the expected arenas, and any pointers copied *by* `memcpy` will already have escaped analysis, and so also have been masked.

Since glibc does not support clang, alternative C libraries have compatibility

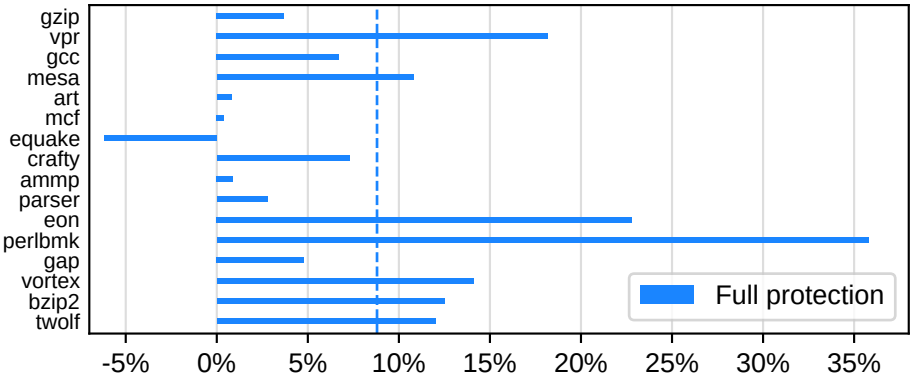


Figure 5.7: CPU2000 runtime overhead

issues, and TDI's stack instrumentation currently requires LTO, we expect TDI to be used in practice with an uninstrumented system libc. Despite this, we also evaluated the overhead of applying TDI's full stack/heap protection to libc, by using musl (and libc++) rather than glibc.

Throughput overhead for our nginx+OpenSSL benchmark is 8.4% at the point of saturation (vs 5.4% without libc instrumentation), and lower using alternative configurations such as using 64kB files (4%) or (multi)thread pools (6.8%). Geomean runtime overhead is 10.3% for SPEC CPU2006 (vs 8.4%); as before, xalancbmk and perlbench are largely responsible. Similarly, geomean overhead is 13.9% for CPU2017 (vs 12.5%). Details can be found in Appendix 5.16.

We also evaluated TDI on SPEC CPU2000, to aid comparisons with prior work. Again, details of the (mostly minor) changes are in Appendix 5.13. Figure 5.7 presents our performance results for full protection with complete instrumentation (including musl/libc++). As shown in the figure, the geomean runtime overhead is 8.8%—with the highest overhead (35.8%) for perlbmk, similar to previous results.

Our CPU2000 overhead is comparable to domain-based sandboxing solutions such as NaCl [257] ($\sim 7\%$)—despite our support for arbitrary (rather than NaCl-only) programs and intra-domain isolation. Moreover, our overhead is much lower than state-of-the-art software fault isolation techniques that rely on highly optimized address masking instrumentation on loads/stores [259] (rather than pointer arithmetic like TDI). Specifically, Zeng et al.'s solution [259], which can only support the limited number of colors allowed by load/store masking, yields 19% overhead on top of a CFI baseline and on a CPU2000 subset excluding costly benchmarks like perlbmk. More fine-grained solutions like WIT [10] can support more colors (limited by the imprecision of context-insensitive points-to analysis), but

load instrumentation can increase overhead (10% on a CPU2000 subset excluding costly benchmarks like perlbmk) "by more than a factor of three" [10]. Note that these numbers (from [10] and [259]) are not directly comparable due to the different evaluation platforms.

5.7.6 Isolation granularity

Although TDI can be used as a traditional coarse-grained (e.g., 2-color) isolation scheme even in the absence of any automated color analysis (significantly outperforming prior load/store address masking solutions, as noted), we briefly evaluated how arenas are assigned in practice by the automated type analysis in a fine-grained, many-color configuration.

For our nginx(+OpenSSL) benchmark, the automated type analysis (TAT) statically identifies 197 colors (and arenas) on the stack and 649 colors (and arenas) on the heap. On the heap, the data type analysis assigns a total of 96 types to allocations at 583 call sites, while the remaining 553 types are identified by the context-sensitive callsite ID analysis based on wrapper detection and inlining (Section 5.6.3).

We also looked at the per-arena object distribution during the execution of the benchmark. Figure 5.8 shows the number of objects allocated in each heap arena during startup and the first client request. The 'OpenSSL hooks' results manually assign arenas by using OpenSSL's built-in support for hooking allocator functions (`CRYPTO_set_mem_functions`); we used one-line wrappers which assign an arena ID based on the callsite information provided by OpenSSL. There were 178, 133, and 106 heap arenas for the automated (TAT), TAT+hooks, and (manual) hooks configurations respectively. The two arenas with the highest number of objects are used to store OpenSSL object names and their related hashes, and all configurations have a relatively large 'long tail' of arenas used only for a single object.

Notably, this shows that attempting to manually assign arenas by hooking OpenSSL's allocator functions leads to *coarser* arenas than a fully-automated approach, even when TAT is also used to assign arenas and can merge allocations of the same type. The fully-automated approach can produce finer-grained arenas because OpenSSL's allocator hooks use indirect calls and only provide direct callsite information (filename/line numbers). For example, OpenSSL provides wrapper functions for allocating and resizing 'buffers'; OpenSSL's allocation functions are called from these buffer wrapper functions, resulting in a large number of allocations from a small number of callsites. TAT instead detects the buffer code as allocator wrappers, and instead allocates arenas based on the parent callsite since the buffer data is untyped (char *).

We also inspected arena usage for some of the SPEC benchmarks. On the CPU2000 subset evaluated by WIT [10], TDI's type analysis yields a number of

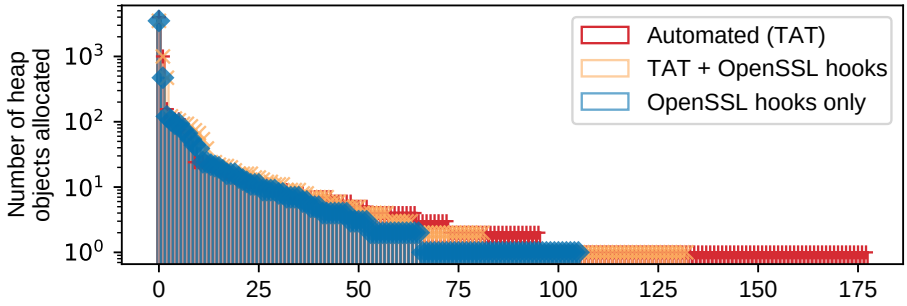


Figure 5.8: Number of objects allocated in each nginx+OpenSSL heap arena.

colors comparable to WIT's points-to analysis (which fares well on such simple benchmarks with many stack allocations). However, unlike WIT, TDI can easily handle the entirety of CPU2000 and even much more complex programs. Moreover, while WIT is limited to 256 colors, TDI uses a larger number of colors even on the slightly more complex CPU2006 benchmarks. For example, xalancbmk allocates 186 stack and 200 heap arenas, and gcc allocates 110 stack and 192–198 heap arenas (depending on the benchmark). Appendix 5.17 contains arena statistics for the other benchmarks.

5.8 Residual attack surface

5.8.1 Spatial safety

TDI's arena allocation could be applied without masking (with much lower overhead). However, non-linear memory vulnerabilities are becoming the primary form of spatial safety vulnerability in the architectural [164] and speculative [128] domain, which may allow attackers to bypass guard zones. These are exactly the situations for which we apply masking.

As for the residual attack surface with full protection, TDI cannot prevent overflows within/across objects of the same color (i.e., type). This provides strong isolation for info leaks, although in some cases there is a remaining attack surface for intra-pool leaks. For example, OpenSSL stores data involving highly confidential data (private/session keys) in the same bignum types as data related to public keys; an info leak bug specifically revealing bignum data for a public key may also allow an attacker to obtain confidential bignum data. If desired, TDI supports annotations to further improve isolation of critical objects, much like existing data isolation solutions.

TDI also offers limited protection against memory corruption exploits which are outside our threat model. For instance, if an attacker can overwrite a pointer

(e.g., in a struct), they can potentially bypass our mitigation. We make such attacks more difficult by limiting the set of pointers at reach (pointers within the same object type) and their ability to leak pointers.

5.8.2 Spectre

TDI provides the same protection against Spectre-BCB attacks as it does against non-speculative information leaks—preventing cross-arena leakage. Most other Spectre variants are clearly out-of-scope and best mitigated by techniques such as retpoline or hardware-based mitigations. However, a theoretical attack surface remains in Spectre V1 gadgets that exploit speculative issues beyond memory safety (e.g., logic bugs). We also do not prevent attacks exposing potential code/stack addresses nor secrets which are already present in registers. SLH also does not mitigate many such cases. If more comprehensive protection is required, it may be possible to use our arena-based approach to reduce the overall performance impact of a more conservative SLH-style mitigation.

5.8.3 ASLR

Our prototype allocator allots pages linearly from the base of each arena, but this is not required by our design; arenas can be placed at any 4GB aligned address and pages can be assigned non-linearly within arenas with no impact on ASLR entropy. However, our design does reduce the entropy available for *fine-grained* ASLR, since the available virtual address space is reduced (by $\sim$3 bits in our prototype), as well as the entropy for large allocations which cannot cross a 4GB boundary. If an attacker leaks a pointer of a given type, they obtain the high bits for the arena; other pointers of the same type are likely to be in the same arena. However, they obtain no information about pointers of *other* types, which are more likely to be of interest to attackers.

5.8.4 Pointer arithmetic

TDI relies on instrumenting pointer arithmetic; specifically, the *security* guarantees require that all pointer arithmetic is instrumented, while the *correctness* guarantees require that non-pointer arithmetic is not instrumented. Our prototype implementation demonstrates that balancing these needs is possible for real-world C/C++ code.

However, there are some cases where this is not possible. For example, when a union contains both a pointer and an integer value, there may not be a correct approach, if arithmetic may be relevant for both values. Similarly, code may store pointers as integers. Although we make use of sources such as TBAA, sometimes

arithmetic on such values cannot be statically detected. Such code is simply incompatible with static instrumentation, but broader analysis or approaches like tagged unions [196] may help in some cases.

Other memory safety work solves these difficulties in different ways. For example, Low-fat Pointers [60] ignores 'uglygeps', excludes 23 CPU2006 functions (including gcc and perlbench) and does not instrument integer arithmetic. Although our analysis is more complete, we still had to apply some patches; we expect similar results in other software.

5.9 Prototype limitations

5.9.1 Completeness

We instrument code at the LLVM IR level. Instructions could be reordered or modified during code generation in a way that compromises our mitigation. There are also inevitably unknown bugs in our prototype passes; however, we did not find any missing instrumentation when manually inspecting the output assembly code from TDI.

5.9.2 Type-based isolation

We rely on the type analysis of Type-after-Type [237] and the limitations mentioned in their paper may result in multiple types being placed in the same arena. Complementary approaches such as TypeClone [17] are an option for improving security guarantees or reducing the number of types.

Our prototype of TDI does not place global variables in type-based arenas; they are placed in data/BSS sections, in a shared arena. Address-taken global variables could be isolated by converting them to heap allocations. Custom memory allocators may also need changes to ensure TDI's type-based isolation guarantees are as fine-grained as possible.

5.9.3 Temporal safety

tcmalloc's design does not isolate size classes once memory is returned to central pools, so new allocations of types with a non-power-of-two size can overlap with previous allocations of such types. Our allocator allows memory to be returned to (typed) central pools, reducing temporal safety (but not isolation) when misalignment may occur. This could be solved by rearchitecting tcmalloc, or using a different baseline allocator.

5.9.4 Compatibility

Custom memory allocators which directly call `mmap` or `brk` must ensure that allocations do not span arena boundaries if they allocate memory regions >4GB. Custom memory allocation code can also reduce security. For example, OpenSSL's 'secure' allocation functions resize buffers by allocating new memory and `memcpy`ing the old contents, rather than using realloc; such functions *reduce* security if used with TDI.

TDI limits maximum object size due to pointer masking. Our prototype limits objects to 4GB. If larger objects are required, manual annotations can be used, or masking can be disabled entirely for some functions/types. We demonstrated this for two CPU2017 benchmarks. Where a huge number of types are used in a program, we can run out of virtual address space, which can be solved by increasing the coarseness of the type classification or reducing the guard zone size. In any case, x86-64's 47-bit space already allows more than 16,000 arenas, and 5-level paging (or ARMv8.2-LVA on ARM) adds support for 56-bit userspace addresses. Note that TDI imposes no limitations on how many objects can be allocated.

5.10 Related work

5.10.1 Secure allocators

Similar to TDI, secure allocators change the stack/heap allocation strategy to improve security. A common approach is to provide probabilistic security by randomizing the positions at and/or order in which allocations are made, as done by StackArmor [40] (stack), DieHard [18] (heap), and OpenBSD's allocator [165]. DieHarder [174] adds guard pages to DieHard (like e.g., Electric Fence); however, allocations remain distinguished by sizes, not types. Archipelego [153] allocates one object per page, allowing guard zones between objects. FreeGuard [218] provides probabilistic security using a combination of randomization, delayed reuse, and guard zones.

Other efforts focus on temporal memory errors such as use-after-free. MarkUs [6] delays freeing memory until pointers no longer appear in memory, while FFmalloc [250] uses one-time allocations (with no memory reuse at all).

Cling [9] mitigates heap memory reuse exploits using independent allocator regions for each allocation site. Type-after-Type [237] extends Cling's design with compile-time type detection, improved wrapper detection, and stack support. Automatic Pool Allocation [144] relies on points-to compiler analysis to split allocations into separate typed pools, which allows for temporal protection of a subset of C [55]. All these defenses use some kind of "typed" pools to enable type-safe memory reuse, but do not provide spatial data isolation.

Modern web browsers also use manual allocation-level isolation to improve security, such as IE's Isolated Heap. In particular, PartitionAlloc [187] is Chrome's default allocator (as of March 2021). It allows (manual) allocation in isolated arenas ('partitions'), mitigating some temporal and linear overflow vulnerabilities, and could be used as an alternative allocator for TDI. V8 also sandboxes WebAssembly by limiting memory offsets (to the sum of two 32-bit offsets) and allocating guard zones for ±8GB around the heap [247], with accesses always using a valid base address.

5.10.2 Data isolation

Address-based defenses [134] typically use annotations of sensitive types or data to isolate one or more specific regions of memory. *Domain-based* defenses [134] instead protect sensitive *code*, protecting the data used by that code, and only allowing access when execution has switched to the relevant domain. Existing solutions fall in either one or both classes of defenses and implement different isolation mechanisms.

DataShield [33] uses annotations and masking via instrumentation. Data-flow analysis identifies potential sensitive data accesses, needing slower metadata checks, and protects non-memory flows. Non-sensitive data is placed in memory <4GB and pointers are truncated to 32 bits. Overhead in artifical case studies, annotating a single type as sensitive, is 9.12% and 27.21% for two CPU2006 benchmarks, and lower ($\sim 0\%$ using x86 prefixes) when code provably cannot access sensitive data.

ConfLLVM [26] also uses annotations along with segmentation or Intel MPX. CPU2006 overhead is 24.5% without any private data, although this includes CFI, and excludes perlbench and xalancbmk (highest overhead in our evaluation).

MemSentry [134] evaluates a variety of these solutions, implementing domain-based (virtualization and MPK) and address-based (encryption, masking, and MPX) defenses. The authors report 17.1% overhead for load masking on CPU2006.

Palit et al. [185] encrypt sensitive data using annotations and points-to analysis. The overhead is 4-33% when protecting only keys. MemCat [172] attempts to distinguish *attacker-controlled* data using compile-time policy and allocates those objects on a separate heap/stack. CPU2006 overhead is 21%.

ERIM [233] uses MPK (also Spectre-BCB-safe) to isolate memory used by a trusted domain; they demonstrate low-overhead protection of CPI's [141] safe region. SeCage [151] uses EPT (page table switching), automatically splitting off code to protect annotated secrets, and xMP [194] uses a similar approach to manually protect kernel data structures or cryptographic data in user-space code.

Data Flow Integrity [35] (DFI) uses points-to analysis to determine which stores should be accessible to each load, enforcing fine-grained isolation at

runtime with costly instrumentation that checks/updates a metadata table on loads/stores. Write Integrity Testing [10] (WIT) reduces DFI's overhead by only protecting stores and limiting object colors to at most 256, adding guards between objects to compensate for imprecise points-to analysis; overhead is 10% on a CPU2000 subset (without eon or perlbmk, TDI's worst cases). Other variants of schemes relying on points-to analysis also exist [219].

Hardware memory tagging (e.g., MTE [84]) provides an alternative isolation primitive; it could be used as an alternative to arenas, with TDI used to protect tags from info leaks.

Finally, TDI draws inspiration from optimization strategies used by prior SFI [245] and other solutions. For instance, Zeng et al. [259] use simpler forms of range and dominating pointer analysis on x86 assembly to eliminate SFI instrumentation on loads/stores. In contrast, TDI reasons over pointer *arithmetic* at the compiler IR level, allowing simpler but more effective static analysis to aggressively remove instrumentation. Previous work also used guard areas to eliminate SFI instrumentation; in particular, on loads/stores with a fixed base pointer [213], a fixed pointer offset [158], or to only detect linear buffer overflows [19, 218]. In contrast, TDI uses guard pages to reason about whether computed pointers are *safe* with respect to arbitrary "valid" base pointers, rather than directly reasoning about load/store accesses. TDI also considers speculative flows, which limit (or prohibit) the applicability of much of this previous optimization work. Finally, TDI's instrumentation relies on efficient address masking similarly to some SFI solutions [138] (others resort to bounds checking [63]), but uses masking to *preserve* bits after pointer arithmetic, rather than using a bitmask to *remove* bits at loads/stores. This allows TDI to support fine-grained isolation, rather than only the coarse-grained (e.g., 2-color) isolation of traditional SFI solutions.

5.10.3 Bounds-checking defenses

Some bounds checking defenses have similarities to our work. Baggy Bounds Checking [11] instruments arithmetic using tagged pointers (on 64-bit), with ∼60-70% overhead. Low-Fat Pointers [60] simplifies this by encoding bounds into *valid* pointers, instrumenting arithmetic and accesses, with 113% overhead. Delta Pointers [137] also instruments accesses and arithmetic (documenting challenges similar to TDI's); by encoding the *delta* to object ends in pointers, the authors limit detection to overflows and total memory space to 4GB, with 35% overhead. Similar in spirit to TDI's guard zones, Delta Pointers offloads checks to the MMU to improve performance.

Dhurjati et al. [53] use points-to analysis (via [144]) to optimize bounds checks. By omitting checks when points-to analysis fails, this avoids compatibility problems (unlike similar work such as [54]) at the cost of security, and achieves average

overhead of ∼12% on the (simple) Olden benchmarks.

AddressSanitizer [214] is a compiler-based debugging tool, using instrumentation, shadow memory and delayed reuse, but recent overhead is still ∼80% on CPU2006. Newer sanitizers such as CUP [29] and EffectiveSan [59] detect broader ranges of threats, with significantly higher overhead.

5.10.4 Spectre mitigations

Canella et al. [32] describe three categories of Spectre mitigations: mitigating covert channels (e.g., reducing timer accuracy or hardware changes), aborting speculation (e.g., fences or retpoline), and making secret data unreachable.

Compilers can automatically insert fences after vulnerable branches to stop speculation [110], but attempts to implement this efficiently for Spectre-BCB (e.g., in MSVC) have been shown to be error-prone [127]. Blade [243] proposes fencing/masking only paths where data may speculatively leak, which the authors apply to WebAssembly. Operating systems such as Linux and other solutions [177, 246] use similar selective (and thus noncomprehensive) fencing policies based on manual annotations or results of program (i.e., gadget) analysis. State-of-the-art comprehensive solutions such as LLVM's Speculative Load Hardening [34] (SLH) mitigation offer a complete but costly alternative. SLH forces a data dependency on the control flow leading to potentially-vulnerable loads, by mixing bits of the predicates used by the control flow into the pointers used by such loads. In contrast to such mitigations, TDI provides a gadget-agnostic defense with strong and fine-grained data isolation guarantees at low overheads.

Web browsers apply similar mitigations such as masking array indexes [159, 191] and applying SLH-type poisoning [159, 228]. Such mitigations are typically easier to comprehensively deploy within JIT environments, but coarser-scale solutions such as Site Isolation are still considered more cost effective [202]. Moreover, some of the efficient masking solutions used by modern browsers such as Firefox use coarse-grained masks which still allow (limited) out-of-bounds accesses to objects of a different type [90], in contrast to TDI.

Ghostbusting [117] proposes mitigating Spectre-BCB vulnerabilities with data isolation via domain switching, and ConTExT [210] protects annotated sensitive data using hardware extensions or uncacheable ('non-transient') memory mappings. The former has been only evaluated with synthetic programs, the latter reports 71% for OpenSSL RSA vs our ∼16%, although our threat models differ significantly.

Other efforts focus on detecting Spectre-BCB and similar vulnerabilities. oo7 [246] finds potential Spectre vulnerabilities using BAP to propagate taint from untrusted sources. Spectector [86] instead applies symbolic execution to source code. SpecFuzz [177] focuses on fuzzing software to find Spectre-BCB

gadgets and seeks to reduce the overhead of SLH (but also its security guarantees) by excluding branches that do not appear vulnerable. TDI's overhead for OpenSSL's ECDSA benchmark is ∼7%, vs SLH's ∼70%; SpecFuzz improves the latter by only 5%, although the performance difference is less extreme for other cases. We could attempt to use SpecFuzz to reduce our masking, but this would remove non-speculative protection and potentially increase our speculative attack surface due to false negatives.

5.11 Conclusion

We have shown that we can efficiently harden programs against temporal and spatial (even speculative, a la Spectre-BCB) info leak vulnerabilities, by using arenas to provide N-color isolation. We have also demonstrated that our protection can be applied automatically by exploiting fine-grained type information for object coloring.

Our type-based arena allocation on the heap and stack has typical run-time overhead far below 5% and already provides a strong mitigation against classical temporal and linear (adjacent) spatial attacks. We significantly broaden this protection by masking pointers to keep them in their intended arena, mitigating non-adjacent and speculative vulnerabilities. TDI still achieves acceptable run-time overhead by minimizing the need to mask pointers. We believe this overhead could be further improved with assistance from compiler frameworks.

5.12 Appendix

5.13 Undefined pointer arithmetic in software

Although TDI is compatible with a range of software, as discussed in Section 5.9, it is still incompatible with software which performs undefined pointer arithmetic and uses the results across function boundaries. During testing, we found such pointer arithmetic issues in several pieces of software. We document these issues here to assist future researchers.

We discovered that CPU2006's version of gcc stores out-of-range pointers (pointing to *before* the start of the allocated object) in global variables, which are later dereferenced by other functions. Although TDI can handle pointers being slightly out-of-bounds (see Section 5.6), the negative delta in these cases is non-constant and can be quite large, resulting in pointers being wrapped. We investigated and found two pre-2006 gcc patches which remove these cases, in both cases since they are undefined behavior; they are r62672 from 2003-02-11, "Don't use offset pointers.", and r89543 from 2004-10-25, "avoid undefined pointer arithmetic on qty_table".

The obstack code in CPU2017's version of gcc stores cross-object pointer deltas in a temporary variable inside a struct allocated on the heap, which is undefined behavior. In fact, the default behavior of the obstack code – including the version in CPU2006's gcc – is to avoid this by using a non-standard C extension, where supported by the compiler. However, the code in CPU2017 was modified by SPEC to disable the use of this code path. We re-enable it by removing the newly-added `!defined(SPEC)` from obstack.h. Note that there is other undefined behavior present in CPU2017 (such as arithmetic using NULL pointers) which we resolve in our canonicalization pass but present issues for upstream LLVM[4].

CPU2016's soplex uses a pointer delta to adjust pointers after a call to realloc. We did not encounter this in our tested configuration (since these objects are allocated in the same arena), and patching it appears to be non-trivial and invasive. This issue has also been observed by other researchers [175].

Both the CPU2006 and CPU2017 versions of perlbench make use of cross-object delta calculation in the mergesort code. Our pointer analysis correctly handles this for CPU2006, but in CPU2017 we were forced to exclude the mergesort function. TDI warns about these issues at compile time, along with various other unexpected patterns in the code, such as casting a 0x55555555 constant to a pointer.

We only encountered issues with a single CPU2000 benchmark, 254.gap, due to a custom allocator/garbage collector ('Gasman') which would need to be modified to support arena-based allocation. We excluded (when instrumenting) the

[4]https://lists.llvm.org/pipermail/llvm-dev/2017-July/115064.html

```
u_char                *p;
ngx_http_geo_range_t **ranges;

ranges[i] = (ngx_http_geo_range_t *)
              (p - (u_char *) fm.addr);
```

Listing 5.7: Simplified code example from nginx 1.18.0's `ngx_http_geo_create_binary_base` function.

`TypHandle` struct (used by GAP's 'Gasman' allocator/garbage collector,) as well as several functions.

nginx provides an illustrative example of arithmetic which must be excluded, annotated or modified to be successfully compiled with TDI. Our pass prints an error when trying to instrument the http_geo module of nginx, due to the code in Listing 5.7, which subtracts a pointer from a pointer and stores the result in an array of pointers (*fm* is a file mapping object; we believe this code is trying to update the base address of an array of pointers, and ranges should be a *ptrdiff_t**). Since the base pointer cannot be determined, masking cannot be applied, and compilation fails.

5.14 Evaluation build details

We applied patches to musl and libc++ to fix LTO issues (such as removing weak symbols) and patched several benchmarks to fix build issues (such as missing includes). We do not instrument code which performs cross-arena arithmetic (e.g., ELF header parsing, vDSO support and stack unwinding); for similar reasons, we do not instrument our allocator itself.

We needed to pass various compatibility flags and make some minor source changes (e.g., including header files) to make the SPEC benchmarks build in our environment. A summary of the (non-trivial) source changes can be seen in Table 5.1 (together with the fixes for the issues described above).

5.15 Pointer detection

While implementing TDI, we found that in real-world programs, neither the types in the LLVM IR nor even the types in the source code accurately reflect whether a value is used as a pointer or not. As such, TDI requires pointer detection to ensure all pointer dereferences are properly masked, and no integers are corrupted by pointer masking. We describe our design at a high level in Section 5.4.2, and include the details here for transparency and reproducibility.

Table 5.1: Modifications applied to SPEC benchmark code.

Software	Reason	Solution
CPU2000 gap	Use of legacy termio	Replaced with termios
CPU2000 gap	Custom garbage collector	Excluded type
CPU2006 gcc	Undefined behavior (negative offsets)	Applied (pre-2005) upstream patches
CPU2017 gcc	Undefined behavior (cross-arena deltas)	`!defined(SPEC)` removed from obstack.h
CPU2017 perlbench	Undefined behavior (cross-arena deltas)	Excluded `S_mergesortsv`
CPU2006 dealII	Missing #include	Added #include
CPU2017 xz	>4GB allocation in SPEC wrapper	Disable LTO for `spec_mem_io.c`

Our pointer detection classifies each value in the LLVM IR as one of four groups: pointers, offsets, negated pointers, and non-pointers. Additionally, during the analysis a value can be classified as unknown or invalid. Initially, we consider every value to be in the unknown class. Our analysis proceeds in four steps that mark unknown-class values based on their usage, starting with the usages that provide most confidence about (non)pointer status. Each step is followed by forward and backward propagation, marking those values that are used to compute the newly marked values and those that are computed from those values.

Marking

We perform marking in four steps: (1) We first mark variables dereferenced in loads and stores (which must therefore be pointers). (2) We mark function arguments or return values based on types from the relevant function prototypes. (3) We then mark values which are loaded/stored based on the type of the pointer used. (4) Finally, we mark any remaining unknown values based on their (LLVM IR) type.

After *each* of these marking steps, we propagate pointer types both backwards and forwards. If we find that a value is used as both a pointer and a non-pointer

type, we mark it as having a pointer type.

Propagation

We propagate pointer types through arithmetic. We consider pointers which are used in shifts, divisions and multiplications to be transformed beyond use, as with AND operations discarding the high bits of a pointer, and mark them as non-pointers. We perform some further analysis on some specific arithmetic operations (and GEP instructions, in LLVM IR). Pointers which are added (or otherwise combined, e.g., ORed) to a non-pointer remain pointers.

If a pointer is subtracted from another pointer, it becomes a non-pointer; if a pointer is subtracted from a non-pointer, it becomes a negative pointer. If a negative pointer is added to a pointer, then we can mark the result as a non-pointer. We treat similar patterns in the same way; for example, we mark a pointer XORed with a negative constant as a negative pointer. We found this to be essential for analyzing both real-world C code and the output of some LLVM transformations.

Base pointers

If an arithmetic operation is determined to result in a pointer type, then we add it to a list of potential pointer arithmetic operations, to be considered by the remaining stages of our analysis. We then attempt to determine the base pointer for each instance of such arithmetic. If only one of the operands for an arithmetic operation is a pointer, then we take that operand as the base pointer. If neither or both operands are pointers, then we mark the base pointer as *invalid*. We found this last case to occur in code calculating pointer hashes, or in control flow paths with unused results.

However, we do not remove these invalid instances of arithmetic from the list of potential pointer arithmetic operations; we later output a warning if we conclude that such arithmetic should be masked. Although such situations are rare, they do occur; an example can be found in Appendix 5.13. During evaluation, we erred on the side of caution, producing an error rather than a warning, and manually annotated 4 cases which could encounter this error in some build configurations.

5.16 Additional results

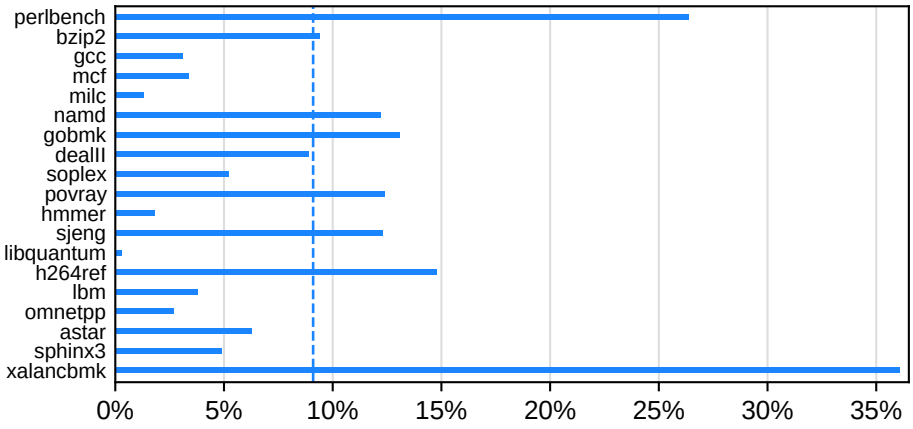


Figure 5.9: CPU2006 (peak) memory overhead

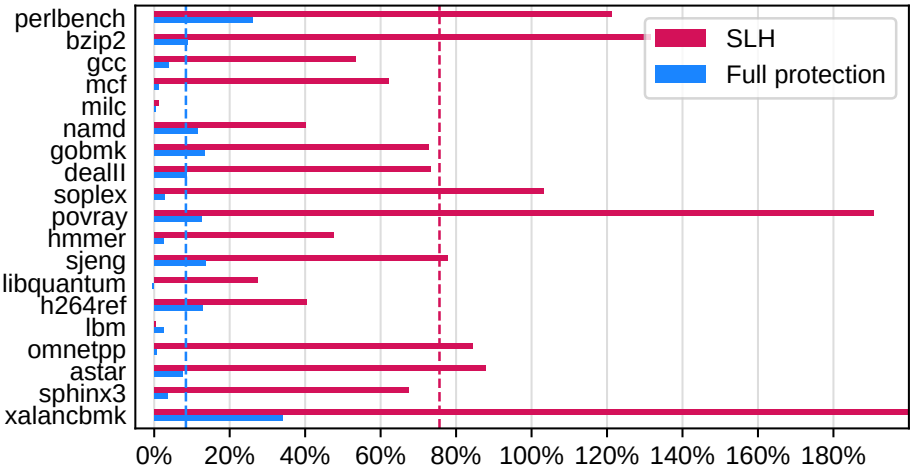


Figure 5.10: CPU2006 runtime overhead vs SLH (xalancbmk is 405%)

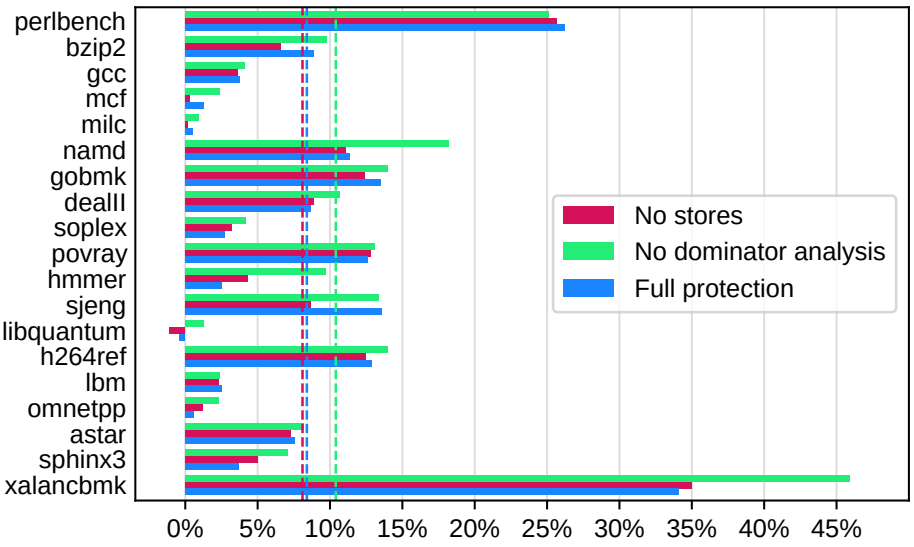


Figure 5.11: CPU2006 runtime overhead with alternative configurations

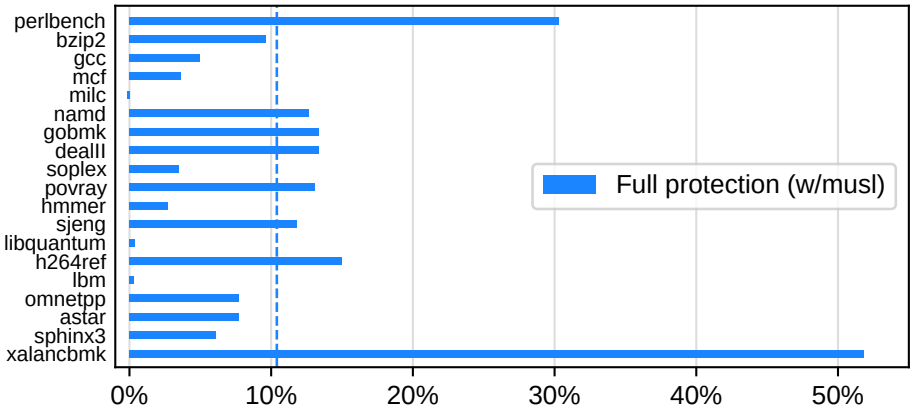


Figure 5.12: CPU2006 runtime overhead with instrumented musl/libc++

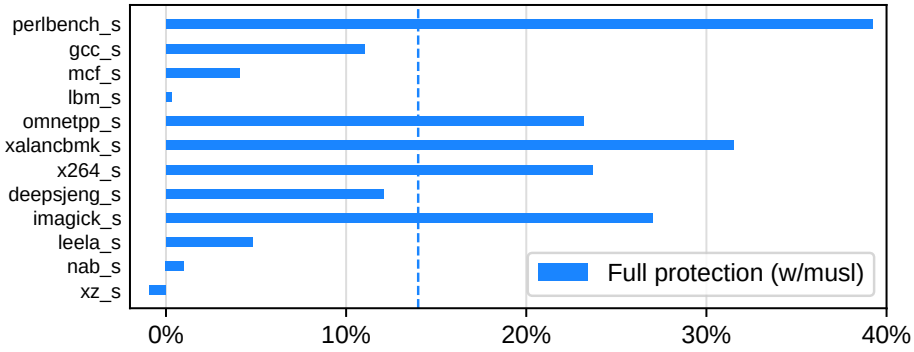


Figure 5.13: CPU2017 runtime overhead with instrumented musl/libc++

5.17 Arena statistics

The number of arenas actually allocated at runtime is the sum of the stack and heap arenas column in Table 5.2.

Table 5.2: Arena allocation statistics for SPEC CPU2000 and CPU2006 benchmarks (incl. musl/libc++).

Benchmark	Stack arenas	Heap arenas[1]	Heap type IDs[2]	Heap call-site IDs[2]
164.gzip	36	9	9	22
175.vpr	45	39–70	33	66
176.gcc	66	21	21	251
177.mesa	38	23	47	31
179.art	33	10	13	19
181.mcf	34	7	10	18
183.equake	34	9	12	18
186.crafty	35	8	10	20
188.ammp	37	15	23	18
197.parser	36	6	8	19
252.eon	68	42	30	81
253.perlbmk	52	60	22	148
254.gap	38	6	8	21
255.vortex	53	9	8	24
256.bzip2	36	11	10	22
300.twolf	34	87	37	94
400.perlbench	59	69–80	28	222
401.bzip2	35	10	9	23
403.gcc	110	192–198	98	414
429.mcf	34	7	10	18
433.milc	39	15	17	29
444.namd	40	17	12	35
445.gobmk	47	15	17	20
447.dealII	116	135	66	252
450.soplex	66	90–95	25	202
453.povray	66	98	79	147
456.hmmer	40	34–50	30	139
458.sjeng	36	8	12	18
462.libquantum	36	8	12	19
464.h264ref	37	50–53	43	37
470.lbm	33	7	8	20
471.omnetpp	55	88	61	1200
473.astar	37	33	15	40
482.sphinx3	40	98	47	95
483.xalancbmk	186	200	212	1764

[1] The number of heap arena IDs used (at least one object allocated) at runtime; this is specified as a range where some sub-benchmarks allocate objects in fewer arenas.

[2] The number of heap types and/or heap callsites (i.e., untyped) assigned unique arena IDs at compile time. The total number of potential heap arena IDs is the sum of these two columns.

6 | Conclusion

The varied nature of side channels – and the attacks which use them – makes them difficult to predict, and even more difficult to mitigate. In this dissertation, we investigated several new approaches for building, investigating and mitigating side-channel attacks; the result is not only a better understanding of the potential attack surface exposed by such attacks, but also a glimpse of how future software defenses can help efficiently mitigate them. Specifically:

Constructing power side-channels We showed that the results of voltage glitching can be aggregated to construct traces which reveal the power used by microprocessors during computations. We investigated two different classification models, both of which allow the data from this indirect power side-channel to be used to perform traditional attacks. In particular, we saw that we can apply SPA to attack RSA, and CPA to attack both software and hardware implementations of AES, without needing any changes to the attacks themselves.

Stale data vulnerabilities We presented a new class of transient execution vulnerabilities, which expose the contents of internal CPU buffers due to the way in which many CPUs implement exception deferral. We showed that these can be used to leak arbitrary data across security boundaries on Intel CPUs, despite the presence of state-of-the-art software and hardware mitigations. We proposed filtering methods and synchronization approaches allowing exposure of targeted data. We demonstrated their effectiveness with exploits which work across process, kernel, virtual machine and even SGX enclave boundaries, using different exceptions together with a cache side-channel to expose data.

Cross-core stale data leaks We designed a tool for investigating the details of how instructions are implemented on modern Intel CPUs, by combining the data exposed by transient execution attacks with information from other sources, such

as performance counters. Among other results, we discovered undocumented resources which are shared between processor cores, such as a 'staging buffer' used as temporary storage. To show the importance of understanding these details, we showed that our results can significantly broaden the impact of existing vulnerabilities, and defeat mitigations which were designed with incomplete knowledge. Specifically, we demonstrated an attack which can compromise private signing keys used by SGX enclaves despite the presence of mitigations for known vulnerabilities, by leaking random numbers across cores.

Mitigating Spectre attacks Spectre variant 1 is an example of a residual attack surface which needs to be mitigated by software, even after hardware mitigations have been implemented and deployed. Rather than reactively applying spot mitigations as individual gadgets are identified, we designed a lightweight broad mitigation based on type-based data isolation which can be applied automatically by compilers, and yet significantly reduces the attack surface for both architectural and transient attacks. Our prototype demonstrates that our design successfully defends against both architectural and transient attacks with even lower overhead (typically <10%) than similar traditional data isolation solutions.

Collectively, this work demonstrates the importance of challenging the assumptions underlying complex modern processor systems; doing so has led to the discovery of new attack surfaces, and understanding these is a critical first step for building defenses to mitigate modern attacks. We have shown that the impact of power side-channel attacks, and processor vulnerabilities such as transient execution issues, can be significantly broader than expected. The impact depends on the way in which not only pipelines but also CPUs as a whole are implemented – and the context in which this hardware is deployed.

In particular, the understandable desire of processor manufacturers to keep these details confidential makes it difficult to analyze the true security risks of modern cloud environments, especially with the future movement of hardware closer to the 'edge', where it may be exposed to physical attacks. Mitigating both existing and potential new attacks clearly demands a better understanding of this context, requiring further research on scalable approaches for investigating implementation details of existing processors. We conclude with a few thoughts on other promising directions for future research:

Physical attacks We showed that voltage (power) fault injection can be used to perform FCA attacks, but other fault injection techniques have been used in related work. Reproducing FCA-style attacks with EMFI or even clock glitching would provide more insight into the relationship between power usage and the probability of different faults. Investigating the use of software-activated faults [121, 169, 195,

224] may expose weaknesses in mitigations focused at limiting access to reliable power measurements.

Stale data Chapters 3 and 4 both demonstrate the dangerous consequences of hardware designs which fail to consider potential exposure of stale data, whether within a pipeline or at a SoC level. Clearing such data may appear to be unnecessary, especially given the negative impact on performance and power. However, such design decisions must be reevaluated in the face of modern speculative – and perhaps even architectural – attacks. Future work could focus on better tools to identify such issues (e.g., at RTL level) as well as ways to efficiently mitigate them.

Hardware support for defenses Defenses against both architectural and speculative attacks can gain significant performance and security advantages from additional hardware support. For example, modern hardware features such as ARM's pointer authentication codes (PACs) and memory tagging (MTE) which depend on pointer bits could be combined with the work presented in Chapter 5. Tagged address bits could be included in arena identifiers, allowing finer-grained isolation, while hardware tag checks further reduce the need for expensive instrumentation.

Mitigating attacks in software Extending software mitigations to cover ever-evolving architectural attacks remains a challenge. For example, the work from Chapter 5 could be extended to resolve some of the limitations as well as providing more comprehensive protection against type confusion attacks. Looking beyond the architectural attack surface, the unexpected nature of side-channel attacks and transient execution vulnerabilities makes them even more difficult to mitigate. 'Spot' mitigations in hardware or software which target specific instances of vulnerabilities are often entirely defeated by relatively minor variations, as discussed in Chapter 3. Detecting potential issues at design time, and ensuring that tools are available for mitigating a wide range of potential future issues, is sure to be a fruitful area of future research. In particular, future work could address speculative safety by considering it as part of the design of other architectural mitigations.

References

[1] Andreas Abel and Jan Reineke. nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems. *arXiv preprint arXiv:1911.03282*, 2019.

[2] Andreas Abel and Jan Reineke. Uops.info: characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *ASPLOS*, 2019.

[3] Jeffery M. Abramson, Haitham Akkary, Andrew F. Glew, Glenn J. Hinton, Kris G. Koningsfeld, and Paul D. Madland. Method and Apparatus for Performing Load Operations in a Computer System, December 1997. US Patent 5,694,574.

[4] AES algorithm implementation in C. https://github.com/dhuertas/AES. [Online; accessed 14-June-2020].

[5] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. OBFUSCURO: A Commodity Obfuscation Engine on Intel SGX. In *NDSS'19*.

[6] Sam Ainsworth and Timothy M Jones. MarkUs: drop-in use-after-free prevention for low-level languages. In *S&P '20*.

[7] Haitham Akkary, Jeffrey M. Abramson, Andrew F. Glew, Glenn J. Hinton, Kris G. Konigsfeld, Paul D. Madland, Mandar S. Joshi, and Brent E. Lince. Cache Memory System Having Data and Tag Arrays and Multi-Purpose Buffer Assembly With Multiple Line Buffers, July 1996. US Patent 5,680,572.

[8] Haitham Akkary, Jeffrey M. Abramson, Andrew F. Glew, Glenn J. Hinton, Kris G. Konigsfeld, Paul D. Madland, Mandar S. Joshi, and Brent E. Lince. Methods and Apparatus for Caching Data in a Non-Blocking Manner Using a Plurality of Fill Buffers, October 1996. US Patent 5,671,444.

[9] Periklis Akritidis. Cling: a memory allocator to mitigate dangling pointers. In *USENIX Security '10*.

[10] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with wit. In *S&P '08*.

[11] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security '09*.

[12] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *S&P'19*.

[13] Subidh Ali, Debdeep Mukhopadhyay, and Michael Tunstall. Differential fault analysis of AES: towards reaching its limits. *J. Cryptographic Engineering*, 3(2):73–97, 2013.

[14] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop Van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *ACSAC*, 2016.

[15] ATxmage32A4U Datasheet. `http : / / ww1 . microchip . com / downloads / en / DeviceDoc / ATxmega128-64-32-16A4U-DataSheet-DS40002166A.pdf`. [Online; accessed 14-June-2020].

[16] AVR Crypto lib. https://git.cryptolib.org/avr-crypto-lib.git. [Online; accessed 14-June-2020].

[17] Mohamad Barbar, Yulei Sui, and Shiping Chen. Flow-sensitive type-based heap cloning. In *ECOOP '20*.

[18] E. Berger and B. Zorn. DieHard: probabilistic memory safety for unsafe languages. In *PLDI '06*.

[19] Sandeep Bhatkar and R Sekar. Data space randomization. In *DIMVA '08*.

[20] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: exploiting speculative execution through port contention. In *CCS'19*.

[21] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *CRYPTO '97*, 1997.

[22] Milind Bodas, Glenn J. Hinton, and Andrew F. Glew. Mechanism to Improved Execution of Misaligned Loads, December 1998. US Patent 5,854,914.

[23] Darrell D. Boggs, Shlomit Weiss, and Alan Kyker. Branch Ordering Buffer, September 2004. US Patent 6,799,268.

[24] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults. In *EUROCRYPT '97*, 1997.

[25] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *SP'16*.

[26] Ajay Brahmakshatriya, Piyus Kedia, Derrick P McKee, Deepak Garg, Akash Lal, Aseem Rastogi, Hamed Nemati, Anmol Panda, and Pratik Bhatu. ConfLLVM: a compiler for enforcing data confidentiality in low-level code. In *EuroSys '19*.

[27] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *CHES 2004*, 2004.

[28] 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'18*.

[29] Nathan Burow, Derrick McKee, Scott A Carr, and Mathias Payer. CUP: Comprehensive user-space protection for C/C++. In *AsiaCCS '18*.

[30] Mathias Bynens. Untrusted Code Mitigations. `https://v8.dev/docs/untrusted-code-mitigations`, January 2018.

[31] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS'19*.

[32] 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 '19*.

[33] Scott A Carr and Mathias Payer. Datashield: configurable data confidentiality and integrity. In *AsiaCCS '17*.

[34] Chandler Carruth. Speculative load hardening. `https://llvm.org/docs/SpeculativeLoadHardening.html`, July 2018.

[35] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *OSDI '06*.

[36] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *CRYPTO'99*, 1999.

[37] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In *CHES 2002*, 2003.

[38] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SGXPECTRE Attacks: Leaking Enclave Secrets via Speculative Execution.

[39] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with Déjá Vu. In *AsiaCCS '17*, 2017.

[40] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. Stackarmor: comprehensive protection from stack-based memory error vulnerabilities for binaries. In *NDSS '15*.

[41] Yen-Kuang Chen, Christopher J. Hughes, and James M. Tuck III. System and Method for Cache Coherency in a Cache With Different Cache Location Lengths, December 2004. US Patent 7,454,576.

[42] Christophe Clavier. Secret external encodings do not prevent transient fault analysis. In *CHES*, 2007.

[43] Christophe Clavier, Jean-Sébastien Coron, and Nora Dabbous. Differential power analysis in the presence of hardware countermeasures. In *CHES 2000*, 2000.

[44] David W. Clift, Darrell D. Boggs, and David J. Sager. Processor with Registers Storing Committed/Speculative Data and a RAT State History Recovery Mechanism with Retire Pointer, October 2003. US Patent 6,633,970.

[45] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks. In *S&P'19*.

[46] Jonathan Corbet. Many uses for Core scheduling, 2019. URL: `https://lwn.net/Articles/799454/`.

[47] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, 2016.

[48] George Cox. Delivering New Platform Technologies. In *SBSeg'12*.

[49] James D. Dundas. Repair of Mis-Predicted Load Values, March 2002. US Patent 6,883,086.

[50] Joan Daemen and Vincent Rijmen. *The design of Rijndael*. 2002.

[51] Poulami Das, Lisa Eckey, Tommaso Frassetto, David Gens, Kristina Hostáková, Patrick Jauernig, Sebastian Faust, and Ahmad-Reza Sadeghi. Fastkitten: practical smart contracts on bitcoin. In *USENIX Security'19*.

[52] Elke De Mulder, Michael Hutter, Mark E Marson, and Peter Pearson. Using Bleichenbacher's solution to the hidden number problem to attack nonce leaks in 384-bit ECDSA. In *International Workshop on Cryptographic Hardware and Embedded Systems*, 2013.

[53] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for c with very low overhead. In *Proceedings of the 28th international conference on Software engineering*.

[54] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. SAFECode: enforcing alias analysis for weakly typed languages. In *PLDI '06*.

[55] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without garbage collection for embedded applications. In *TECS '05*.

[56] Christoph Dobraunig, Maria Eichlseder, Hannes Gross, Stefan Mangard, Florian Mendel, and Robert Primas. Statistical Ineffective Fault Attacks on Masked AES with Fault Countermeasures. In *Advances in Cryptology – ASIACRYPT 2018*, 2018.

[57] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Stefan Mangard, Florian Mendel, and Robert Primas. SIFA: Exploiting Ineffective Fault Inductions on Symmetric Cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018, 2018.

[58] Christoph Erwin Dobraunig, Stefan Mangard, Florian Mendel, and Robert Primas. Fault attacks on nonce-based authenticated encryption: application to keyak and ketje. In *SAC*, 2019.

[59] Gregory J Duck and Roland HC Yap. EffectiveSan: type and memory error detection using dynamically typed C/C++. In *PLDI '18*.

[60] Gregory J Duck and Roland HC Yap. Heap bounds protection with low fat pointers. In *CC '16*.

[61] Lance E. Hacking and Debbie Marr. Globally Observing Load Operations Prior to Fence Instruction and Post-Serialization Modes, January 2004. US Patent 7,249,245.

[62] Lance E. Hacking and Debbie Marr. Synchronization of Load Operations Using Load Fence Instruction in Pre-Serialization/Post-Serialization Mode, February 2001. US Patent 6,862,679.

[63] Ulfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. Xfi: software guards for system address spaces. In *OSDI '06*.

[64] Mark Ermolov and Maxim Goryachy. How to hack a turned-off computer, or running unsigned code in intel management engine. *Black Hat Europe*, 2017.

[65] Dmitry Evtyushkin and Dmitry Ponomarev. Covert channels through random number generator: mechanisms, capacity estimation and mitigations. In *CCS'16*.

[66] Andrew F. Glew and Glenn J. Hinton. Method and Apparatus for Processing Memory-Type Information Within a Microprocessor, December 1996. US Patent 5,751,996.

[67] Dario Faggioli. Core-Scheduling for Virtualization: Where are We? (If We Want It!) In *KVM Forum*, October 2019.

[68] Brandon Falk. CPU Introspection: Intel Load Port Snooping, 2019.

[69] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *S&P'18*.

[70] Thomas Fuhr, Eliane Jaulmes, Victor Lomné, and Adrian Thillard. Fault Attacks on AES with Faulty Ciphertexts Only. In *FDTC'13*, 2013.

[71] GD32VF103CBT6 Datasheet. https://www.gigadevice.com/datasheet/gd32vf103xxxx-datasheet/. [Online; accessed 14-June-2020].

[72] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 2018.

[73] Alexander Gendler, Larisa Novakovsky, and Ariel Szapiro. Communicating via a mailbox interface of a processor, January 2015. US Patent Appl. 14/609,835.

[74] Nahid Farhady Ghalaty, Aydin Aysu, and Patrick Schaumont. Analyzing and eliminating the causes of fault sensitivity analysis. In *DATE '14*, 2014.

[75] Nahid Farhady Ghalaty, Bilgiday Yuce, and Patrick Schaumont. Differential fault intensity analysis on PRESENT and LED block ciphers. In *COSADE*, 2015.

[76] Nahid Farhady Ghalaty, Bilgiday Yuce, Mostafa Taha, and Patrick Schaumont. Differential fault intensity analysis. In *FDTC '14*, 2014.

[77] Sanjay Ghemawat and Paul Menage. TCMalloc: thread-caching malloc, 2009.

[78] Christophe Giraud. AES. In *Advanced Encryption Standard – AES*, 2004.

[79] Andy Glew, Nitin Sarangdhar, and Mandar Joshi. Method and Apparatus for Combining Uncacheable Write Data Into Cache-Line-Sized Write Buffers, December 1993. US Patent 5,561,780.

[80] Google SafeSide. https://github.com/google/safeside. September 2020.

[81] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security'18*.

[82] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS'17*.

[83] Brendan Gregg. KPTI/KAISER Meltdown Initial Performance Regressions, 2018. URL: https://www.linux.com/news/kptikaiser-meltdown-initial-performance-regressions/.

[84] Matthew Gretton-Dann. Arm a-profile architecture developments 2018: armv8.5-a, 2018.

[85] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA'16*.

[86] Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. Spectector: principled detection of speculative information flows. In *S&P '20*.

[87] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games–Bringing Access-Based Cache Attacks on AES to Practice. In *S&P'11*.

[88] Xiaofei Guo, Debdeep Mukhopadhyay, Chenglu Jin, and Ramesh Karri. Security analysis of concurrent error detection against differential fault analysis. *Journal of Cryptographic Engineering*, 5:153–169, 2014.

[89] Linley Gwennap. P6 microcode can be patched. *Microprocessor Report*, 1997.

[90] Noam Hadad and Jonathan Afek. Overcoming (some) spectre browser mitigations. `https://alephsecurity.com/2018/06/26/spectre-browser-query-cache/`, 2018.

[91] Wenjian He, Wei Zhang, Sanjeev Das, and Yang Liu. Sgxlinger: a new side-channel attack vector based on interrupt latency against enclave execution. In *International Conference on Computer Design (ICCD)*. IEEE, 2018.

[92] David Hepkin. Hyper-V HyperClear Mitigation for L1 Terminal Fault. `https://blogs.technet.microsoft.com/virtualization/2018/08/14/hyper-v-hyperclear/`, August 2018.

[93] Christoph Herbst, Elisabeth Oswald, and Stefan Mangard. An AES Smart Card Implementation Resistant to Power Analysis Attacks. In *Applied Cryptography and Network Security*, 2006.

[94] Mathé Hertogh, Manuel Wiesinger, Sebastian Österlund, Marius Muench, Nadav Amit, Herbert Bos, and Cristiano Giuffrida. Quarantine: mitigating transient execution attacks with physical domain isolation. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 207–221, 2023.

[95] Jann Horn. Reading Privileged Memory with a Side-channel. `https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html`, January 2018.

[96] Jann Horn. Speculative Store Bypass. `https://bugs.chromium.org/p/project-zero/issues/detail?id=1528`, May 2018.

[97] Zhichao Hua, Dong Du, Yubin Xia, Haibo Chen, and Binyu Zang. EPTI: Efficient Defence against Meltdown Attack for Unpatched VMs. In *ATC'18)*.

[98] Intel. Changes to rdrand integration in openssl. `https://web.archive.org/web/20240000000000*/http://software.intel.com/en-us/blogs/2014/10/03/changes-to-rdrand-integration-in-openssl`, 2014.

[99] Intel. Deep Dive: Intel Analysis of L1 Terminal Fault. `https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-l1-terminal-fault` Retrieved 15.10.2018.

[100] Intel. Deep Dive: Intel Transactional Synchronization Extensions (Intel TSX) Asynchronous Abort, 2019.

[101] Intel. Deep Dive: Special Register Buffer Data Sampling, 2020. URL: https://software.intel.com/security-software-guidance/insights/deep-dive-special-register-buffer-data-sampling.

[102] Intel. Guidelines for mitigating timing side channels against cryptographic implementations, 2019.

[103] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual, June 2016.

[104] Intel. Intel Analysis of Speculative Execution Side Channels: White paper, January 2018. https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf Retrieved 15.10.2018.

[105] Intel. INTEL-SA-00307: Intel CSME Advisory, 2020.

[106] Intel. Microarchitectural Data Sampling / CVE-2018-12126,CVE-2018-12127,CVE-2018-12130,CVE-2019-11091 / INTEL-SA-00233, 2019.

[107] Intel. Processors Affected: Special Register Buffer Data Sampling, 2020. URL: https://software.intel.com/security-software-guidance/insights/processors-affected-special-register-buffer-data-sampling.

[108] Intel. Rogue System Register Read / CVE-2018-3640 / INTEL-SA-00115. https://software.intel.com/security-software-guidance/software-guidance/rogue-system-register-read, May 2018.

[109] Intel. Side Channel Vulnerabilities: Microarchitectural Data Sampling and Transactional Asynchronous Abort, 2019.

[110] Intel. Speculative execution side channel mitigations, July 2018. Revision 3.0.

[111] Intel. Speculative Store Bypass / CVE-2018-3639 / INTEL-SA-00115. https://software.intel.com/security-software-guidance/software-guidance/speculative-store-bypass Retrieved 15.10.2018.

[112] Intel. The Intel Converged Security and Management Engine IOMMU Hardware Issue – CVE-2019-0090, 2019.

[113] Intel. Write Combining Memory Implementation Guidelines, 1998.

[114] Intel Announces 9th Generation Core CPUs, Eight-Core Core i9-9900K. https://www.tomshardware.com/news/intel-9th-generation-coffee-lake-refresh,37898.html.

[115] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing–and its Application to AES. In *S&P'15*.

[116] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: securing hardware against probing attacks. In *CRYPTO 2003*, 2003.

[117] Ira Ray Jenkins, Prashant Anantharaman, Rebecca Shapiro, J Peter Brady, Sergey Bratus, and Sean W Smith. Ghostbusting: mitigating Spectre with intraprocess memory isolation. In *HoTSoS '20*.

[118] JLSCA. https://github.com/Riscure/Jlsca. [Online; accessed 14-June-2020].

[119] David Kanter. Intel's Haswell CPU microarchitecture, 2012.

[120] Mehmet Kayaalp, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Aamer Jaleel. A High-Resolution Side-Channel Attack on Last-Level Cache. In *DAC'16*.

[121] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0LTpwn: attacking x86 processor integrity from software. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[122] Mehran Mozaffari Kermani, Amir Jalali, Reza Azarderakhsh, Jiafeng Xie, and Kim-Kwang Raymond Choo. Reliable Inversion in GF(2^8) With Redundant Arithmetic for Secure Error Detection of Cryptographic Architectures. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2018.

[123] Ho-Seop Kim, Robert S. Chappell, Choon Y. Soo, and Srikanth T. Srinivasan. Store Address Prediction for Memory Disambiguation in a Processing Device, September 2013. US Patent 9,244,827.

[124] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and Defenses. In *arXiv'18*.

[125] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the Linux Virtual Machine Monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230. Dttawa, Dntorio, Canada, 2007.

[126] Ilya Kizhvatov. Side Channel Analysis of AVR XMEGA Crypto Engine. In *Proceedings of the 4th Workshop on Embedded Systems Security*, WESS '09, 2009.

[127] Paul Kocher. Spectre Mitigations in Microsoft's C/C++ Compiler. `https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html`, 2018.

[128] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre Attacks: Exploiting Speculative Execution. In *S&P '19*.

[129] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO '96*.

[130] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO'96*, 1996.

[131] Paul C. Kocher, Jushua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO '99*, 1999.

[132] David Kohlbrenner and Hovav Shacham. Trusted Browsers for Uncertain Times. In *USENIX Security'16*.

[133] Altug Koker, Thomas A. Piazza, and Murali Sundaresan. Scatter/Gather Capable System Coherent Cache, May 2013. US Patent 9,471,492.

[134] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No need to hide: protecting safe regions on commodity hardware. In *EuroSys '17*.

[135] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *USENIX WOOT'18*.

[136] KPTI - Linux Documentation. `https://www.kernel.org/doc/Documentation/x86/pti.txt` Retrieved 15.10.2018.

[137] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. Delta pointers: buffer overflow checks without the checks. In *EuroSys '18*.

[138] Joshua A Kroll, Gordon Stewart, and Andrew W Appel. Portable software fault isolation. In *2014 IEEE 27th Computer Security Foundations Symposium*.

[139] Konrad J. Kulikowski, Mark G. Karpovsky, and Er Taubin. Robust codes for fault attack resistant cryptographic hardware. In *in Fault Diagnosis and Tolerance in Cryptography, 2nd International Workshop*, pages 1–12, 2005.

[140] Tsvika Kurts, Zelig Wayner, and Tommy Bojan. Apparatus and Method for Bus Signal Termination Compensation During Detected Quiet Cycle, December 2002. US Patent 6,842,035.

[141] Volodymyr Kuznetzov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *OSDI '14*.

[142] Michael Larabel. Benchmarking The Updated Intel CPU Microcode For SRBDS / CrossTalk Mitigation, 2020. URL: `https://www.phoronix.com/scan.php?page=article&item=srbds-crosstalk-benchmark`.

[143] Michael Larabel. Looking At The Linux Performance Two Years After Spectre / Meltdown Mitigations, 2020. URL: `https://www.phoronix.com/scan.php?page=article&item=spectre-meltdown-2`.

[144] Chris Lattner and Vikram Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI '05*.

[145] Yang Li, Sho Endo, Nicolas Debande, Naofumi Homma, Takafumi Aoki, Thanh-Ha Le, Jean-Luc Danger, Kazuo Ohta, and Kazuo Sakiyama. Exploring the relations between fault sensitivity and power consumption. In *Constructive Side-Channel Analysis and Secure Design*, 2013.

[146] Yang Li, Kazuo Sakiyama, Shigeto Gomisawa, Toshinori Fukunaga, Junko Takahashi, and Kazuo Ohta. Fault sensitivity analysis. In *CHES*, 2010.

[147] Linux: L1TF - L1 Terminal Fault. `https://www.kernel.org/doc/html/latest/admin-guide/l1tf.html` Retrieved 15.10.2018.

[148] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. PLATYPUS: software-based power side-channel attacks on x86. In *S&P '21*.

[149] 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'18*.

[150] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-Level Cache Side-Channel Attacks are Practical. In *S&P'15*.

[151] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *CCS '15*.

[152] Victor Lomné, Emmanuel Prouff, and Thomas Roche. Behind the scene of side channel attacks. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 506–525. Springer, 2013.

[153] Vitaliy B Lvin, Gene Novark, Emery D Berger, and Benjamin G Zorn. Archipelago: trading address space for reliability and security. In *ASPLOS '08*.

[154] Jeffery M. Abramson, Haitham Akkary, Andrew F. Glew, Glenn J. Hinton, Kris G. Konigsfeld, and Paul D. Madland. Method and Apparatus for Blocking Execution of and Storing Load Operations during their Execution, March 1999. US Patent 5,881,262.

[155] Jeffrey M. Abramson, David B. Papworth, Haitham H. Akkary, Andrew F. Glew, Glenn J. Hinton, Kris G. Konigsfeld, and Paul D. Madland. Out-Of-Order Processor With a Memory Subsystem Which Handles Speculatively Dispatched Load Operations, October 1995. US Patent 5,751,983.

[156] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative Execution using Return Stack Buffers, 2018.

[157] Giorgi Maisuradze and Christian Rossow. Speculose: analyzing the security implications of speculative execution in cpus. *arXiv preprint arXiv:1801.04084*, 2018.

[158] Stephen McCamant and Greg Morrisett. Evaluating sfi for a cisc architecture. In *USENIX Security '06*.

[159] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. Spectre is here to stay: an analysis of side-channels and speculative execution. *arXiv preprint arXiv:1902.05178*, 2019.

[160] John Mechalas. Intel Digital Random Number Generator (DRNG). https://www.intel.com/content/www/us/en/developer/articles/guide/intel-digital-random-number-generator-drng-software-implementation-guide.html, 2018.

[161] Vineeth Mekkat, Oleg Margulis, Jason M. Agron, Ethan Schuchman, Sebastian Winkel, Youfeng Wu, and Gisle Dankel. Method and Apparatus for Recovering From Bad Store-To-Load Forwarding in an Out-Of-Order Processor, December 2015. US Patent 9,996,356.

[162] Thomas S. Messerges. Securing the AES Finalists Against Power Analysis Attacks. In *Fast Software Encryption*, 2001.

[163] Microsoft. Managing Hyper-V hypervisor scheduler types, 2019. URL: https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/manage/manage-hyper-v-scheduler-types.

[164] Matt Miller. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. In *BlueHat IL '19*.

[165] Otto Moerbeek. A new malloc (3) for openbsd. In *EuroBSDCon 2009*.

[166] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. Copycat: controlled instruction-level attacks on enclaves for maximal key extraction. *arXiv preprint arXiv:2002.08437*, 2020.

[167] Amir Moradi, Oliver Mischke, and Thomas Eisenbarth. Correlation-enhanced power analysis collision attack. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 125–139. Springer, 2010.

[168] Amir Moradi, Oliver Mischke, Christof Paar, Yang Li, Kazuo Ohta, and Kazuo Sakiyama. On the power of fault sensitivity analysis and collision side-channel attacks in a combined setting. In *CHES*, 2011.

[169] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: software-based fault injection attacks against intel sgx. In *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[170] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS '09*.

[171] Gary N. Hammond and Carl C. Scafidi. Utilizing an Advanced Load Address Table for Memory Disambiguation in an Out of Order Processor, December 2003. US Patent 7,441,107.

[172] M. Neugschwandtner, A. Sorniotti, and A. Kurmus. Memory categorization: separating attacker-controlled data. In *DIMVA '19*.

[173] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In *Information and Communications Security*, 2006.

[174] Gene Novark and Emery D Berger. DieHarder: securing the heap. In *CCS '10*.

[175] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2018.

[176] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: protecting SGX enclaves from practical side-channel attacks. In *USENIX ATC 18*, 2018.

[177] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. Specfuzz: bringing spectre-type vulnerabilities to the surface. In *USENIX Security '20*.

[178] OpenSSL. TLS heartbeat read overrun (CVE-2014-0160).

[179] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In *Cryptographers' Track at the RSA Conference*, pages 1–20. Springer, 2006.

[180] Salvador Palanca, Stephen A. Fischer, Subramaniam Maiyuran, and Shekoufeh Qawami. MFENCE and LFENCE Micro-Architectural Implementation Method and System, July 2002. US Patent 6,651,151.

[181] Salvador Palanca, Vladimir Pentkovski, Niranjan L. Cooray, Subramaniam Maiyuran, and Angad Narang. Method and System for Optimizing Write Combining Performance in a Shared Buffer Structure, March 1998. US Patent 6,122,715.

[182] Salvador Palanca, Vladimir Pentkovski, Subramaniam Maiyuran, Lance Hacking, Roger A. Golliver, and Shreekant S. Thakkar. Synchronization of Weakly Ordered Write Combining Operations Using a Fencing Mechanism, March 1998. US Patent 6,073,210.

[183] Salvador Palanca, Vladimir Pentkovski, and Steve Tsai. Method and Apparatus for Implementing Non-Temporal Loads, March 1998. US Patent 6,223,258.

[184] Salvador Palanca, Vladimir Pentkovski, Steve Tsai, and Subramaniam Maiyuran. Method and Apparatus for Implementing Non-Temporal Stores, March 1998. US Patent 6,205,520.

[185] Tapti Palit, Fabian Monrose, and Michalis Polychronakis. Mitigating data leakage by protecting memory-resident sensitive data. In *ACSAC '19*.

[186] Jingyu Pan, Fan Zhang, Kui Ren, and Shivam Bhasin. One fault is all it needs: breaking higher-order masking with persistent fault analysis. In *DATE 2019*, 2019.

[187] Partitionalloc. `https://chromium.googlesource.com/chromium/src/+/master/base/allocator/partition_allocator/PartitionAlloc.md`.

[188] Sikhar Patranabis, Abhishek Chakraborty, and Debdeep Mukhopadhyay. Fault Tolerant Infective Countermeasure for AES. In *SPACE 2015*, 2015.

[189] Sikhar Patranabis, Abhishek Chakraborty, Debdeep Mukhopadhyay, and Partha Chakrabarti. Fault space transformation: a generic approach to counter differential fault analysis and differential fault intensity analysis on aes-like block ciphers. *IEEE Transactions on Information Forensics and Security*, 2016.

[190] Filip Pizlo. What Spectre and Meltdown Mean For WebKit. `https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/`, January 2018.

[191] Filip Pizlo. What Spectre and Meltdown Mean For WebKit, January 2018.

[192] Process Isolation in Firefox. `https://mozilla.github.io/firefox-browser-architecture/text/0012-process-isolation-in-firefox.html`.

[193] The Chromium Projects. Mitigating Side-Channel Attacks. `https://www.chromium.org/Home/chromium-security/ssca` Retrieved 31.12.2018.

[194] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P Kemerlis, and Michalis Polychronakis. Xmp: selective memory protection for kernel and user space. In *S&P '20*.

[195] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. Voltjockey: breaching trustzone by software-controlled voltage manipulation over multi-core frequencies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 195–209, 2019.

[196] Jon Rafkind, Adam Wick, John Regehr, and Matthew Flatt. Precise garbage collection for C. In *ISMM '09*.

[197] Keyvan Ramezanpour, Paul Ampadu, and William Diehl. Fault intensity map analysis with neural network key distinguisher. In *ASHES'19*, 2019.

[198] Keyvan Ramezanpour, Paul Ampadu, and William Diehl. FIMA: fault intensity map analysis. In Ilia Polian and Marc Stöttinger, editors, *COSADE*, 2019.

[199] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: closing digital side-channels through obfuscated execution. In *USENIX Security'15*.

[200] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. In *SEC'16*.

[201] Re-enable SharedArrayBuffer + Atomics. `https://bugs.chromium.org/p/chromium/issues/detail?id=821270`.

[202] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site isolation: process separation for web sites within the browser. In *USENIX '19*.

[203] Tom Ritter. Firefox - Fuzzy Timers Changes, October 2018. `https://hg.mozilla.org/mozilla-central/rev/77626c8d6bee`.

[204] Matthieu Rivain, Emmanuel Prouff, and Julien Doget. Higher-order masking and shuffling for software implementations of block ciphers. In *CHES 2009*, 2009.

[205] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2), 1978.

[206] Bruno Robisson and Pascal Manet. Differential behavioral analysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 413–426. Springer, 2007.

[207] Mandar S. Joshi, Andrew F. Glew, and Nitin V. Sarangdhar. Write Combining Buffer for Sequentially Addressed Partial Line Operations Originating From a Single Instruction, May 1995. US Patent 5,630,075.

[208] Sayandeep Saha, Arnab Bag, Debapriya Basu Roy, Sikhar Patranabis, and Debdeep Mukhopadhyay. Fault template attacks on block ciphers exploiting fault propagation. In *EUROCRYPT*, 2020.

[209] Peter Schwabe and Ko Stoffelen. All the AES you need on Cortex-M3 and M4. In *International Conference on Selected Areas in Cryptography*, pages 180–194. Springer, 2016.

[210] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. ConTExT: A generic approach for mitigating Spectre. In *NDSS '20*.

[211] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: cross-privilege-boundary data sampling. In *CCS'19*.

[212] Mark Seaborn and Thomas Dullien. Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges. In *BHUS'15*.

[213] David Sehr, Robert Muth, Cliff L Biffle, Victor Khimenko, Egor Pasko, Bennet Yee, Karl Schimpf, and Brad Chen. Adapting software fault isolation to contemporary cpu architectures. In *USENIX Security*, 2010.

[214] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: a fast address sanity checker. In *USENIX ATC*, 2012.

[215] Fermin J Serna. The info leak era on software exploitation. *Black Hat USA*, 2012.

[216] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *CCS '07*.

[217] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *NDSS*, 2017.

[218] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. Freeguard: a faster secure heap allocator. In *CCS '17*.

[219] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. Enforcing kernel security invariants with data flow integrity. In *NDSS '16*.

[220] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels, 2018.

[221] STM32F103C8 Datasheet. `https://www.st.com/resource/en/datasheet/stm32f103c8.pdf`. [Online; accessed 14-June-2020].

[222] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *CC '16*.

[223] Sung-Ming Yen and M. Joye. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Transactions on Computers*, 49(9):967–970, 2000.

[224] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: exposing the perils of security-oblivious energy management. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1057–1074, 2017.

[225] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Defeating Software Mitigations against Rowhammer: A Surgical Precision Hammer. In *RAID'18*.

[226] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer Attacks over the Network and Defenses. In *USENIX ATC'16*.

[227] Michael E Thomadakis. The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms, 2011.

[228] Ben L Titzer and Jaroslav Sevcik. A year with Spectre: a V8 perspective, April 2019.

[229] R.M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 1967.

[230] Michael Tunstall and Olivier Benoit. Efficient use of random delays in embedded software. In *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems*, 2007.

[231] Harshal Tupsamudre, Shikha Bisht, and Debdeep Mukhopadhyay. Destroying fault invariant with randomization. In *CHES 2014*, 2014.

[232] Paul Turner. Retpoline: a Software Construct for Preventing Branch Target Injection. `https://support.google.com/faqs/answer/7625886`, January 2018.

[233] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: secure, efficient in-process isolation with protection keys (MPK). In *USENIX Security '19*.

[234] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *S&P*, 2020.

[235] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic. In *CCS'18*.

[236] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-step: A Practical Attack Framework for Precise Enclave Execution Control. In *SysTEX'17*.

[237] Erik Van Der Kouwe, Taddeus Kroes, Chris Ouwehand, Herbert Bos, and Cristiano Giuffrida. Type-after-type: practical and complete type-safe memory reuse. In *ACSAC '18*.

[238] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *CCS'16*.

[239] Wim Van Eck. Electromagnetic radiation from video display units: an eavesdropping risk? *Computers & Security*, 1985.

[240] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *S&P '19*.

[241] Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In *USENIX Security'18*.

[242] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. Cacheout: leaking data on intel cpus via cache evictions.

[243] Marco Vassena, Klaus V Gleissenthall, Rami Gökhan Kici, Deian Stefan, and Ranjit Jhala. Automatically eliminating speculative leaks from cryptographic code with blade. In *POPL '21*.

[244] Luke Wagner. Mitigations Landing for New Class of Timing Attack, January 2018. `https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/` Retrieved 31.12.2018.

[245] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *SOSP*, 1993.

[246] 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*, 2019.

[247] WebAssembly Out of Bounds Trap Handling, 2016.

[248] Samuel Weiser, David Schrammel, Lukas Bodner, and Raphael Spreitzer. Big Numbers—Big Troubles: Systematically Analyzing Nonce Leakage in (EC) DSA Implementations. In *USENIX Security'20*.

[249] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: breaking the virtual memory abstraction with transient out-of-order execution. *Technical report*, 2018.

[250] Brian Wickman, Hong Hu, Insu Yun Daehee Jang, JungWon Lim Sanidhya Kashyap, and Taesoo Kim. Preventing use-after-free attacks with fast forward allocation. In *USENIX Security '21*.

[251] David Woodhouse. x86/retpoline: Fill RSB on Context Switch for Affected CPUs. `https : / / git . kernel . org / pub / scm / linux / kernel / git / torvalds / linux . git / commit / ?id = c995efd5a740d9cbafbf58bde4973e8b50b4d761`, January 2018.

[252] Wrk2. `https://github.com/giltene/wrk2`. September 2019.

[253] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: deterministic side channels for untrusted operating systems. In *S&P'15*.

[254] Yuval Yarom. Mastik: a micro-architectural side-channel toolkit, 2016.

[255] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack. *IACR Cryptology ePrint Archive*, 2014.

[256] Yuval Yarom and Katrina Falkner. FLUSH + RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security'14*.

[257] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: a sandbox for portable, untrusted x86 native code. In *S&P '09*.

[258] Adi Yoaz, Mattan Erez, Ronny Ronen, and Stephan Jourdan. Speculation Techniques for Improving Load Related Instruction Scheduling. In *ISCA'99*.

[259] Bin Zeng, Gang Tan, and Greg Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *CCS '11*.

[260] Fan Zhang, Xiaoxuan Lou, Xinjie Zhao, Shivam Bhasin, Wei He, Ruyi Ding, Samiya Qureshi, and Kui Ren. Persistent fault analysis on block ciphers. *CHES*, 2018.

[261] Tong Zhang, Dongyoon Lee, and Changhee Jung. Bogo: buy spatial memory safety, get temporal memory safety (almost) free. In *ASPLOS '19*.

Contributions

This thesis is based on papers which I wrote together with other authors. A summary of my individual contribution to each paper is provided below:

Albert Spruyt, Alyssa Milburn, and Łukasz Chmielewski. **Fault Injection as an Oscilloscope: Fault Correlation Analysis.**

The majority of this research was a joint effort by Albert Spruyt and myself. I built most of the attack framework (including the hardware and FPGA code) and performed the attacks on RSA and the hardware AES engine.

Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. **RIDL: Rogue In-flight Data Load.**

I am the second author on the RIDL paper, after Stephan van Schaik, and was involved in most of the work and writing for this paper. I was responsible for designing and implementing the covert channel and attacks (except Section 3.6.4 and part of 3.6.7), and the discovery and analysis of the non-primary RIDL variants, including the TAA and VRS vulnerabilities which were used in most of these attacks. Some of the details were embargoed at the time of publication, as discussed in the chapter preface.

Hany Ragab[1], Alyssa Milburn[1], Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. **CrossTalk: Speculative data leaks across cores are real.**

I share first authorship on this paper with Hany Ragab, which we wrote together. I was responsible for the original discovery and analysis of the staging buffer, and the work described in the exploitation and covert channel sections.

Alyssa Milburn, Erik van der Kouwe, and Cristiano Giuffrida. **Mitigating Information Leakage Vulnerabilities with Type-based Data Isolation.**

I am first author on this paper, responsible for both the research and writing.

[1]Equal contribution shared first authors.

CRediT author statement

Fault Injection as an Oscilloscope: Fault Correlation Analysis

Spruyt, A.: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Writing - Original Draft, Writing - Review & Editing, Visualization. **Milburn, A.:** Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Writing - Original Draft, Writing - Review & Editing, Visualization. **Chmielewski, L.:** Conceptualization, Methodology, Writing - Original Draft, Writing - Review & Editing.

RIDL: Rogue In-flight Data Load

van Schaik, S.: Conceptualization, Methodology, Software, Validation, Investigation, Writing - Original Draft, Writing - Review & Editing. **Milburn, A.:** Conceptualization, Methodology, Software, Validation, Investigation, Writing - Original Draft, Writing - Review & Editing. **Österlund, S.:** Conceptualization, Methodology, Software, Validation, Investigation, Writing - Original Draft, Writing - Review & Editing. **Frigo, P.:** Conceptualization, Methodology, Software, Validation, Investigation, Writing - Original Draft, Writing - Review & Editing. **Maisuradze, G.:** Conceptualization, Methodology, Software, Writing - Review & Editing. **Razavi, K.:** Supervision, Writing- Original Draft, Writing - Review & Editing, Methodology. **Bos, H.:** Supervision, Writing- Original Draft, Writing - Review & Editing, Methodology. **Giuffrida, C.:** Supervision, Writing- Original Draft, Writing - Review & Editing, Methodology.

CrossTalk: Speculative Data Leaks Across Cores Are Real

Ragab, H.: Conceptualization, Methodology, Software, Validation, Investigation, Writing - Original Draft, Writing - Review & Editing. **Milburn, A.:** Conceptualization, Methodology, Software, Validation, Investigation, Writing - Original Draft, Writing - Review & Editing. **Razavi, K.:** Supervision, Writing- Original Draft, Writing - Review & Editing, Methodology. **Bos, H.:** Supervision, Writing- Original Draft, Writing - Review & Editing, Methodology. **Giuffrida, C.:** Supervision, Writing- Original Draft, Writing - Review & Editing, Methodology.

Mitigating Information Leakage Vulnerabilities with Type-based Data Isolation

Milburn, A.: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Writing - Original Draft, Writing - Review & Editing. **van der Kouwe, E.:** Supervision, Writing- Original Draft, Writing - Review & Editing, Methodology. **Giuffrida, C.:** Supervision, Writing- Original Draft, Writing - Review & Editing, Methodology.

Summary

Computers are complicated, and the security of modern software often depends on programmers anticipating and mitigating a dizzying assortment of vulnerability classes. Both researchers and adversaries continue to add additional complications based on hardware behavior to this mix, such as data-dependant power consumption, fault injection, cache timing attacks and transient execution – adding new attack surface which needs to be taken into account when building secure systems. In this dissertation, we look *beyond* the traditional attack surface, and find that many mitigations – across a variety of layers, from hardware to software – are compromised by faulty assumptions about the actual behavior of hardware.

We begin close to the hardware, discovering that the relationship between two different forms of hardware attack – Fault Injection and Side-Channel Analysis – is so strong that we can find ways to apply power side-channel attack techniques directly to the results of voltage fault injection campaigns, without the need for an attacker to obtain power measurements.

Next, we move to microarchitectural attacks – which rely on hardware behavior, but can be performed by software. By combining transient execution with unexpected – and unintended – CPU pipeline behavior, we show that the scope of *software* side-channel attacks can be expanded far beyond previous work, allowing both in-flight and 'stale' data to be stolen from the pipeline by an attacker, bypassing software mitigations by ignoring security domains entirely.

Such attacks can be mitigated by isolating workloads on different cores – which do not share a pipeline. Unfortunately, we find that even this level of isolation can fail to prevent attackers from being able to leak critical data using side-channel attacks – such as random numbers used when generating cryptographic keys. Hence, these attacks must be mitigated in hardware.

Finally, we investigate defenses against transient execution attacks such as Spectre which depend only on well-understood hardware behavior, but require software mitigations. We show that we can build efficient and practical software-based defenses which not only mitigate many Spectre attacks, but even harden software against architectural information leaks.