Applying Software-Based Side-Channels to Hardware Vulnerabilities

by

Andrew Kwong

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2023

Doctoral Committee:

Professor Daniel Genkin, Co-Chair
Professor Alex Halderman, Co-Chair
Professor Christopher Fletcher
Professor Jean-Baptiste Jeannin
Professor Hovav Shacham

Andrew Kwong
ankwong@umich.edu
ORCID iD: 0000-0002-6473-288X

DEDICATION

To my parents, David and Suzanne, and my brother, Nathan

ACKNOWLEDGMENTS

The process of earning one's PhD truly is a marathon, and not a sprint. Throughout this long and sometimes grueling process, I have been fortunate to have the support and encouragement of family, friends, and colleagues, without whom I would not have reached the finish line.

Firstly, I would like to thank my advisor, Daniel Genkin, for sparking my initial interest in what would eventually become my thesis topic. His guidance and mentorship throughout the course of my degree were invaluable, and his insightful feedback and constructive criticism have been instrumental in shaping this thesis. His dedication to fostering an environment where I could both challenge his ideas and defend my own has been crucial to my development as a scientist. I also want to thank Alex Halderman for stepping in as a mentor that I could seek advice from at Michigan. His wisdom and insights have always proven to be relevant and timely.

I would also like to thank the members of my thesis committee: Christopher Fletcher, Hovav Shacham, and Jean-Baptiste Jeannin. Their constructive feedback, suggestions, and critiques were of great use during the process of writing this thesis. And to Kevin Fu, for guiding me through the first steps of my PhD.

I am also grateful for the support of my colleagues and classmates; my academic journey would not have been the same without their encouragement and camaraderie.

And last but not least, I want to express my deepest appreciation to my family and friends for their unwavering support, love, and understanding. Their encouragement and belief in me have been the driving force behind my accomplishments, and have made this experience meaningful.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

LIST OF TABLES

TABLE

ABSTRACT

The discrepancy between the abstract model used to reason about the security of computer systems and their actual hardware implementation has led to a myriad of security issues. Security researchers have demonstrated how to use shared microarchitectural components, speculative execution attacks, and even hardware based memory integrity attacks to extract sensitive information across nearly all hardware backed security domains. These "side-channel" attacks have violated boundaries necessary for the isolation and security of virtual machines, browser tabs, kernel memory, and even Trusted Execution Environments, making it clear that side-channels present a formidable challenge to safely multiplexing hardware.

This thesis explores these side-channel attacks at the intersection of software, hardware, and applied cryptography. By leveraging shared microarchitectural components, memory integrity vulnerabilities, and even physical effects, this thesis demonstrates new classes of attacks in which software-level attackers abuse vulnerabilities present in hardware

More specifically, this thesis investigates the threat of side-channels to secure execution environments (e.g. Intel's SGX), cryptographic schemes (both standardized by NIST and candidates for future post quantum crypto schemes), web browsers, micro architectural systems, and hardware, such as memory modules and storage devices.

By surveying the landscape of potential side-channel threats and discovering new classes of attacks, this thesis contributes novel insights into adversarial capabilities and how to develop effective mitigations. Thus, the works contained in this thesis aim to serve as stepping stones towards securing the next generation of computers against these side-channel attacks in a principled manner.

CHAPTER 1

Introduction

Traditionally, attackers have compromised computer systems by targeting vulnerabilities in software. Buffer overflows, SQL injections, and other common attacks exploited vulnerabilities present in software applications and their underlying operating systems. Accordingly, the security community spent considerable effort towards trying to understand how and why these systems were vulnerable, and how to fix these software bugs in order to build secure systems.

The security community bore witness to a paradigm shift, however, upon the realization that the underlying abstractions beneath the software don't always perfectly conform to their theoretical specifications; as a result, the hardware can actually leak secrets to software-level attackers, thereby exposing systems to a new class of attacks that undermine the system's security through a hitherto unknown vector.

These "side-channel" attacks allow attackers to steal and extract information through indirect channels such as timing, cache activities, speculative execution, and more. Thus, even if the interfaces at the abstraction level are designed properly and securely, information can still unintentionally leak between these layers, resulting in even the most secure, well designed software systems being compromised.

Concerningly, these vulnerabilites are much more than simply theoretical concepts. High profile side-channel attacks such as Spectre, Rowhammer, and Meltdown have led to industry wide mitigation efforts, demonstrating how devastating this breakdown between abstractions can be. The common theme underlying these side-channels was that the computer systems failed when a

hitherto unknown vector compromised the system; this is because systems cannot defend against threat vectors that have not yet even been conceived of. Thus, in order to secure computers against side-channels in a principled manner, we must first survey the landscape of potential side-channel threats and acquire an adequate understanding of adversarial capabilities and what classes of attacks devices need to be resilient against.

By leveraging shared microarchitectural components, memory integrity vulnerabilities, and even physical effects, this thesis demonstrates novel techniques by which attackers can extract information through these indirect channels. More specifically, this thesis adds to this body of knowledge by uncovering new classes of attacks regarding the Rowhammer bug, cryptographic side-channels, and even analog cyber-physical side-channels.

The following Subsections provide brief overviews of what topics this thesis covers.

1.1 Rowhammer

The trend towards increasing DRAM cell density and decreasing capacitor size over the past decades has given rise to a reliability issue known as the Rowhammer bug [160]. With Rowhammer, repeated activation of DRAM rows can cause DRAM cells in neighboring rows to lose their charge before the memory controller can refresh them, resulting in logical bit flips. This allows an unprivileged adversary to flip the values of bits in neighboring rows on the memory module, completely violating OS enforced isolation between security domains and resulting in capability to change values of individual bits without the need to access them. Concerningly, Rowhammer induced bit flips have even been demonstrated from Javascript running within a browser tab [117] and by remote attackers over the network [176, 257].

Confidentiality Implications of Rowhammer (Chapter 2). In this chapter I discuss my Rambleed [168] paper. While prior work investigated how Rowhammer could be used as a limited write primitive to break memory integrity, we show that the threat of Rowhammer also extends to *memory confidentiality*. That is, we showed how an unprivileged attacker can exploit the data

dependence between Rowhammer-induced bit flips and the bits in nearby rows to deduce these bits. In concert with novel memory massaging primitives and new techniques for finding bit flips, this effect yields a read primitive that can potentially read arbitrary memory.

To empirically demonstrate the implications of this, we developed an end-to-end key extraction attack against OpenSSH wherein we completely recovered a 2048-bit RSA private key; additionally, we proved that RAMBleed remains a threat even when error-correcting memory (ECC-RAM) successfully corrects every Rowhammer induced bit flip. We accomplished this through use of a timing side-channel on ECC-RAM's correction process; reads issued to the ECC-RAM return more slowly when the ECC-RAM has to correct an error, which leaks whether or not a bit flip occurred; this is sufficient information for RAMBleed to read bits across security domains, as RAMBleed does not require any persistent bit flips.

Rowhammering Post Quantum Cryptography (Chapter 3). In this chapter I discuss my FrodoFlips [95] paper. The common number-theoretic asymmetric cryptosystems in use today are known to fail catastrophically under a quantum attack. To address this, NIST announced the beginning of their Post Quantum Cryptography (PQC) standardization process in 2016. Despite intense scrutiny from the cryptographic community, little was known about the candidate algorithms' resistance to side-channels.

We addressed this gap, and used Rowhammer to demonstrate the first practical failure-boosting attack against a post quantum KEM. In particular, we developed a key extraction attack against FrodoKEM, one of 7 Key Encapsulation Mechanism (KEM) algorithms to be selected as a round 3 candidate in the NIST PQC contest.

At a high level, we used Rowhammer-induced flip bits to corrupt a FrodoKEM public key, thereby increasing the decryption failure rate to be high enough that we could practically find failing ciphertexts. Each failing ciphertext yields an inequality on the master key; given a sufficient number of these inequalities, we can eventually recover the learning-with-errors (LWE) secret. After finding over half a million failing ciphertexts using a supercomputer, we successfully completed an end-to-end key recovery attack that enabled the attacker to completely extract session keys in under

2 minutes on a commodity laptop.

Of particular note is the fact that our Rowhammer attack weakens the FrodoKEM public key in such a way that it is both stealthy and persistent. This means that the attacker can extract all session keys for the remainder of the public key's life, all without the victim ever knowing; this result demonstrates a shortcoming of designing PQ KEMs such that their public keys cannot be verified to have been generated properly.

1.2 Cryptographic Side-Channels

Over the past decade, we have seen the cryptography community's focus shift away from studying how to encrypt data to investigating new ways of *computing* over encrypted data. From zero-knowledge proofs (ZK), to fully homomorphic encryption (FHE) and private set intersection (PSI), academia and industry together have made tremendous advancements in understanding their security and performance. This thesis aims to help guide the development of these primitives in a principled manner, incorporating lessons learned from the substantial body of research on side-channel security in encryption and digital signature algorithms.

Private Set Intersection (Chapter 4). One cryptographic system that illustrates this is Chrome's Password Leak Detection service, which was deployed and enabled by default on Chrome just within the last year. In order to combat the rising frequency of credential stuffing attacks, Chrome automatically checks users' credentials against a list of compromised credentials upon each login. To accomplish this in a privacy preserving manner, where clients do not reveal any information about their passwords to the server and the server reveals nothing about its password database to clients, Google developed a custom protocol called Password Leak Detection that combines anonymity sets, memory-hard hashing, and Private Set Intersection (PSI).

In my Usenix Security Symposium 2023 paper [169], my colleagues and I demonstrated attacks against Chrome's Password Leak Detection protocol's usage of the hash function *scrypt*, its hash-to-elliptic curve function, and its modular inversion algorithm. In particular, we developed an

end-to-end exploit that allows an attacker to recover the user's password on the first guess in most casees, thereby empirically demonstrating the dangers of using memory-hard hashing on secret data. Furthermore, we developed a novel cryptanalysis of the Binary Extended Euclidian Algorithm (BEEA) that is able to completely recover the inputs, given a single, noisy trace. Beyond the implications for Chrome's Password Leak Detection, our attack has broader implications on BEEA's susceptibility to side-channel leakage; our single-trace, noise resistant key extraction attack demonstrates that extreme care must be taken when using BEEA on secret inputs.

Pseudorandom Number Generation (Chapter 5). Despite randomness being one of the most fundamental cryptographic primitives, we found yet again that hard-learned lessons regarding side-channel resilience had not been applied to pseudorandom number generators. In my Black Swans [79] paper, my colleagues and I systematically explored the side-channel leakage from common PRGs and demonstrated the pitfalls of neglecting side-channel countermeasures in PRG design.

In particular, we developed end-to-end cache-attacks to recover the PRG state from the `CTR_DRBG` PRG, NIST's most popular recommended PRG. At a high level, the underlying counter mode's usage of T-Table AES reveals enough information for an attacker to recover the state after observing roughly 2000 bytes of output. We used these attacks to compromise the random number generation in the OpenSSL FIPS module, the NetBSD kernel, the FortiOSv5 operating system, and the `nist_rng` library. Then, we demonstrated how to extract TLS keys after recovering the state of the PRG used during the TLS handshake. This highlights the importance of reseeding PRGs such that forward secrecy in PRGs may be preserved, even when subjected to side-channels.

We also demonstrated that using `CTR_DRBG` within an SGX enclave does not mitigate our attack; in fact, it actually amplifies it. To attack a TLS client running within the SGX enclave, we developed a novel differential cryptanalysis technique that recovers the PRG state after observing just 3 AES operations. This in turn allows the attacker to recover the premaster secret, master secret, and symmetric encryption keys for any TLS connection made by the mbedTLS-SGX library, to any TLS server. Moreover, no amount of PRG reseeding is sufficient to prevent this.

1.3 Analog Side-Channels

This thesis also investigates a novel vector for an acoustic side-channel attack on storage devices. Much like the other chapters in this thesis, this attack assumes a software-only attacker that exploits a hardware vulnerability.

Acoustic Eavesdropping (Chapter 6). In my IEEE S&P 2019 paper [170], my colleagues and I explored the cyberphysical attack surface exposed by hard disk drives (HDDs) and demonstrated how the mechanical components in HDDs approximate microphones with sufficient precision to extract and parse human speech. By turning a non-sensing device, the HDD, into a crude microphone, an adversary can mount an acoustic side-channel attack, thereby allowing the attacker to eavesdrop on speech within the vicinity of the disk. I first modeled how a hard drive's physical characteristics lend themselves to measuring acoustic waves, and then validated the model by successfully extracting and filtering human speech such that it is recognizable to a human listener's ear. Furthermore, I extracted audio samples through the hard drive with sufficient fidelity for Shazam to automatically recognize songs.

1.4 Acknowledgments

While I was the lead author on the majority of the papers contained in this thesis, and conducted most of the work contained within, I would like to explicitly acknowledge my collaborators. In particular, I would like to give special thanks to Shaanan Cohney for conducting the analysis of the NIST CTR_DRBG pseudorandom generator and developing the long-term key extraction attack. I would also like to acknowledge my collaborators on our FrodoFlips [95] paper for conducting the cryptanalysis on FrodoKEM. And finally, I would like to thank each and every one of my collaborators by name: Daniel Apon, Jonathan Berger, Connor Bolton, Shaanan Cohney, Dana Dachman-Soled, Thinh Dang, Michael Fahr, Kevin Fu, Daniel Genkin, Daniel Gruss, Nadia Heninger, Ingab Kang, Jason Kim, Hunter Kippen, Chaohao Li, Jacob Lichtinger, Marina Minkin, Alexander Nelson, Shahar Paz, Ray Perlner, Sara Rampazzi, Eyal Ronen, Stephan Van Schaik,

Hovav Shacham, Kang Shin, Youssef Tobah, Riad Wahby, Walter Wang, Wenyuan Xu, Yuval Yarom, and Arkady Yerukhimovich.

CHAPTER 2

RAMBleed: Reading Bits in Memory Without Accessing Them

The Rowhammer bug is a reliability issue in DRAM cells that can enable an unprivileged adversary to flip the values of bits in neighboring rows on the memory module. Previous work has exploited this for various types of fault attacks across security boundaries, where the attacker flips inaccessible bits, often resulting in privilege escalation. It is widely assumed however, that bit flips within the adversary's own private memory have no security implications, as the attacker can already modify its private memory via regular write operations.

We demonstrate that this assumption is incorrect by employing Rowhammer as a *read* side channel. More specifically, we show how an unprivileged attacker can exploit the data dependence between Rowhammer-induced bit flips and the bits in nearby rows to deduce these bits, including values belonging to other processes. Thus, the primary contribution of this work is to show that Rowhammer is a threat to not only integrity, but to confidentiality as well.

Furthermore, in contrast to Rowhammer write side channels, which require persistent bit flips, our read channel succeeds even when ECC memory detects and corrects every bit flip. Thus, we demonstrate the first security implication of successfully-corrected bit flips, which were previously considered benign.

To demonstrate the implications of this read side channel, we present an end-to-end attack on OpenSSH 7.9 that extracts an RSA-2048 key from the root level SSH daemon. To accomplish this, we develop novel techniques for massaging memory from user space into an exploitable state, and

use the DRAM row-buffer timing side channel to locate physically contiguous memory necessary for double-sided Rowhammering. Unlike previous Rowhammer attacks, our attack does not require the use of huge pages, and it works on Ubuntu Linux under its default configuration settings.

2.1 Introduction

In recent years, the discrepancy between the abstract model used to reason about computers and their actual hardware implementation has lead to a myriad of security issues. These range from microarchitectural attacks [104] that exploit contention on internal components to leak information such as cryptographic keys or keystroke timing [119, 210, 291], through transient execution attacks [68, 163, 175, 264, 279] that break down fundamental OS isolation guarantees, to memory integrity attacks [66, 157, 160, 167] that exploit hardware limitations to change the contents of data stored in the device.

Rowhammer [117, 160, 243] is a fault attack, in which the attacker uses a specific sequence of memory accesses that results in bit flips, i.e., changes in bit values, in locations *other* than those accessed. Because the attacker does not directly access the changed memory location, the change is not visible to the processor or the operating system, and is not subject to any permission checks. Thus far, this ability to reliably flip bits across security boundaries has been exploited for sandbox escapes [117, 243], privilege escalation attacks on operating systems and hypervisors [115, 117, 229, 243, 266, 284], denial-of-service attacks [115, 149], and even for fault injection in cryptographic protocols [44].

A common theme for all past Rowhammer attacks is that they break memory *integrity*. Namely, the attacker uses Rowhammer to obtain a (limited) write primitive into otherwise inaccessible memory, and subsequently modifies the contents of that memory in a way that aligns with the attacker's goals. This observation has led to various mitigation proposals designed to secure the target's memory by using integrity checks [271], or by employing ECC (error-correcting code) memory to ensure memory integrity. The latter, in particular, has long been touted as a

defense against Rowhammer-based attacks. Even when an attacker flips a bit in memory, the ECC mechanism corrects the error, halting the attack. While recent work has demonstrated that an attacker can defeat the ECC mechanism, resulting in observable bit-flips after error correction [81], successfully corrected flips are still considered benign, without any security implications. Thus, in this paper we pose the following questions:

- *Is the threat posed by Rowhammer limited only to memory integrity and, in particular, can the Rowhammer effect be exploited for breaching confidentiality?*
- *What are the security implications of* corrected *bit flips? Can an attacker use Rowhammer to breach confidentiality even when ECC memory corrects all flipped bits?*

2.1.1 Our Contributions

In this paper, we answer these questions in the affirmative. More specifically, we present *RAMBleed*, a new Rowhammer-based attack that breaks memory confidentiality guarantees by acquiring secret information from other processes running on the same hardware. Remarkably, RAMBleed can break memory confidentiality of ECC memory, even if all bit flips are successfully corrected by the ECC mechanism. After profiling the target's memory, we show how RAMBleed can leak secrets stored within the target's physical memory, achieving a read speed of about 3–4 bits per second. Finally, we demonstrate the threat posed by RAMBleed by recovering an RSA 2048-bit signing key from an OpenSSH server using only user level permissions.

Data-Dependent Bit Flips. The main observation behind RAMBleed is that bit flips depend not only on the bit's *orientation*, i.e., whether it flips from 1 to 0 or from 0 to 1, but also on the values of neighboring bits [160]. Specifically, true bits tend to flip from 1 to 0 when the bits above and below them are 0, but not when the bits above and below them are 1. Similarly, anti bits tend to flip from 0 to 1 when the bits above and below them are 1, but not when the bits above and below them are 0. While this observation dates back to the very first Rowhammer paper [160], we show how attackers can use it to obtain a read primitive, thereby learning the values of nearby bits which they

might not be allowed to access.

RAMBleed Overview. Suppose an attacker wants to determine the value of a bit in a victim's secret. The attacker first *templates* the computer memory to find a flippable bit at the same offset in a memory page as the secret bit. (For the rest of the discussion we assume a true bit, i.e., one that flips from 1 to 0.) The attacker then manipulates the memory layout to achieve the arrangement depicted below:

Row Activation Page	Secret
Unused	Sampling Page
Row Activation Page	Secret

Here, each memory row spans two memory pages of size 4 KiB. The attacker uses the *Row Activation* pages for hammering, the *Sampling* page contains the flippable bit, which is initialized to 1, and *Secret* pages contain the secret victim data that the attacker aims to learn. If the value of the secret bit is 0, the layout results in a flippable 0-1-0 configuration, i.e., the flippable bit is set to 1, and the bits directly above and below it are 0. Otherwise, the secret bit is 1, resulting in a 1-1-1 configuration, which is not flippable.

Next, the attacker repeatedly accesses the two activation pages she controls (left top and bottom rows), thereby hammering the middle row. Because the Rowhammer effects are data dependent, this hammering induces a bit flip in the sampling page in the case that the secret bit is 0. The attacker then accesses the sampling page directly, checking for a bit flip. If the bit has flipped, the attacker deduces that the value of the secret bit is 0. Otherwise, the attacker deduces that the value is 1. Repeating the procedure with flippable bits at different offsets in the page allows the attacker to recover all of the bits of the victim's secret.

We note here that neither the victim nor the attacker access the secrets in any way. Instead, by accessing the attacker-controlled row activation pages, the attacker uses the victim's data to influence Rowhammer-induced bit flips in her own private pages. Finally, the attacker directly checks the sampling page for bit flips, thereby deducing the victim's bits. As such, RAMBleed is a cross address space attack.

ECC Memory. ECC memory has traditionally been considered an effective defense against Rowhammer-based attacks. Even when an attacker flips a bit in memory, the hardware's ECC mechanisms simply revert back any Rowhammer-induced bit flips. However, recent work has demonstrated that an attacker can defeat the ECC mechanism by inducing enough carefully-placed flips in a single codeword, resulting in observable bit-flips after error correction [81].

In this paper, however, we show that even ECC-corrected bit flips may have security implications. This is because RAMBleed does not necessarily require the attacker to read the bit to determine if it has flipped. Instead, all the attacker requires for mounting RAMBleed is an indication that a bit in the sampling page has flipped (and subsequently corrected). Unfortunately, as [81] show, the synchronous nature of the ECC correction algorithm typically exposes such information through a timing channel, where memory accesses that require error correction are measurably slower than normal accesses.

Thus, we can exploit Rowhammer-induced timing variation to read data even from ECC memory. In particular, our work is the first to highlight the security implications of successfully corrected flips, hitherto considered to be benign.

Memory Massaging. One of the main challenges for mounting RAMBleed, and Rowhammer-based attacks in general, is achieving the required data layout in memory. Past approaches rely on one or more mechanisms which we now describe. The first practical Rowhammer attack relied on operating system interfaces (e.g., `/proc/pid/pagemap` in Linux) to perform virtual-to-physical address translation for user processes [243]. Later attacks leveraged huge pages, which give access to large chunks of consecutive physical memory [117], thereby providing sufficient information about the physical addresses to mount an attack. Other attacks utilized memory grooming or massaging techniques [266], which prepare memory allocators such that the target page is placed at the attacker-chosen physical memory location with a high probability. An alternative approach is exploiting memory deduplication [58, 229], which merges physical pages with the same contents. The attacker then hammers its shared read-only page, which is mapped to the same physical memory location as the target page.

However, many of these mechanisms are no longer available for security reasons [187, 230, 250, 266]. Thus, as a secondary contribution of this paper, we present a new approach for *massaging* memory to achieve the desired placement. Our approach builds on past works that exploit the Linux buddy allocator to allocate blocks of consecutive physical memory [77, 266]. We extend these works by demonstrating how an attacker can acquire some physical address bits from the allocated memory. We further show how to place secret-containing pages at desired locations in the physical memory.

Finally, we note that this method may have independent value for mounting Prime+Probe last-level cache attacks [177]. This is since it allows the attacker to deduce physical addresses of memory regions, thereby aiding eviction set construction.

Extracting Cryptographic Keys. To demonstrate the effectiveness of RAMBleed, we use it to leak secrets across process boundaries. Specifically, we use RAMBleed against an OpenSSH 7.9 server (newest version at time of writing), and successfully read the bits of an RSA-2048 key at a rate of 0.3 bits per second, with 82% accuracy. We combine the attack with a variant of the Heninger-Shacham algorithm [130, 132, 212] designed to recover RSA keys from partial information, achieving complete key recovery.

Summary of Contributions. In this paper we make the following contributions:

- We demonstrate the first Rowhammer attack that breaches confidentiality, rather than integrity (Section 2.4).
- We abuse the Linux buddy allocator to allocate a large block of consecutive *physical* addresses, and show how to recover some of the physical address bits (Section 2.5.1).
- We design a new mechanism, which we call *Frame Feng Shui*, for placing victim program pages at a desired location in the physical memory (Section 2.5.3).
- We demonstrate a Rowhammer-based attack that leaks keys from OpenSSH while only flipping bits in memory locations the attacker is allowed to modify (Section 2.7).

- Finally, we demonstrate RAMBleed against ECC memory, highlighting security implications of successfully-corrected Rowhammer-induced bit flips (Section 2.8).

2.1.2 Responsible Disclosure

Following the practice of responsible disclosure, we have notified Intel, AMD, OpenSSH, Microsoft, Apple, and Red Hat about our findings. The results contained in this paper (and in particular our memory massaging technique) were assigned CVE-2019-0174 by Intel.

2.1.3 Related Works

Security Implications of Rowhammer. The potential for sporadic bit flips was well known in the DRAM manufacturing industry, but was considered a reliability issue rather than a security threat. [160] were the first to demonstrate a reliable method for inducing bit flips by repeatedly accessing pairs of rows in the same bank. Subsequently, [243] showed that Rowhammer is a security concern by using Rowhammer-induced flips to break out of Chrome's Native Client sandbox [294] and obtain root privileges.

Since the initial Rowhammer-based exploit of [243], researchers have demonstrated numerous other avenues for Rowhammer exploitation. [117] demonstrated that page-table bits can be flipped via Rowhammer from JavaScript, while [58] flipped the types of JavaScript objects through the browser. [33] also demonstrated Rowhammer flips without the use of CLFLUSH, and with a halved DRAM refresh interval. [266] used Rowhammer to gain root on mobile phones, while [176] and [257] used network requests to induce Rowhammer flips via a completely remote attack. [98] managed to induce bit flips from the browser's interface to the GPU. ECC memory was shown to be vulnerable to Rowhammer by [81].

[178] systematically categorize Rowhammer attacks in a framework to better understand the problem and uncover new types of Rowhammer attacks. Their methodology, however, is limited and completely ignores the possibility of using Rowhammer as a read side channel.

Defenses. Various defenses have been proposed for Rowhammer attacks, aiming to detect ongoing attacks [33, 78, 118, 144, 213, 300], neutralize the effect of bits being flipped [117, 266], or eliminate the possibility of Rowhammer bit flips in the first place [60, 152, 159, 160].

2.2 Background

This section provides the necessary background on DRAM architecture, the row-buffer timing side channel described by Pessl et al. [219], and the Rowhammer bug. We begin by briefly overviewing DRAM organization and hierarchy.

2.2.1 DRAM Organization

DRAM Hierarchy. DRAM (dynamic random access memory) is organized in a hierarchy of cells, banks, ranks, and DIMMs, which are connected to one or more channels.

More specifically, at the lowest level DRAM stores bits in units called *cells*, each consisting of a capacitor paired with a transistor. The charge on the capacitor determines the value of the bit stored in the cell, while the transistor is used to access the stored value. For *true cells*, a fully charged capacitor represents a '1' and a discharged capacitor represents a '0' while the opposite holds true for *anti cells*.

Memory cells are arranged in a grid of rows and columns called a *bank*. Cells in each row are connected via a *word line*, while cells in each column are connected across *bit lines*. Banks are then grouped together to form a *rank*, which often corresponds to one side of a DIMM. Each DIMM is inserted, possibly with other DIMMs, into a single *channel*, which is a physical connection to the CPU's memory controller.

DRAM Operation. Access to a DRAM bank operates at a resolution of a row, typically consisting of 65536 cells, or 8 KiB. To *activate* a row, the memory controller raises the word line for the row. This produces minute currents on the bit lines, which depend on the charge in the cells of the active row. *Sense amplifiers* capture these currents at each column and amplify the signal to both copy

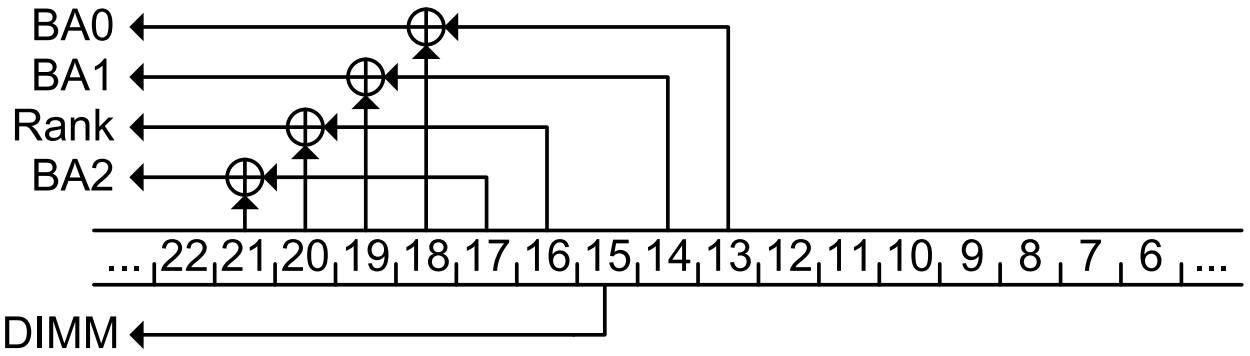


Figure 2.1: **Reverse engineered DDR3 single channel mapping (2 DIMM per channel) for Ivy Bridge/Haswell (from Pessl et al. (USENIX Security '16)).**

the logical value of the cell into a latch and refresh the charge in the active row. Data can then be transferred between the CPU and the *row buffer*, which consists of the latches that store the values of the cells in the active row.

Over time, the charge in the cell capacitors in DRAM leaks. To prevent data loss through leakage, the charges need to be refreshed periodically. Refreshing is handled by the memory controller, that ensures that each row is opened at least once every *refresh interval*, which is generally 64 ms [151] for DDR3 and DDR4. LPDDR4 defines temperature-dependent adaptations for the refresh interval [153].

DRAM Addressing. Modern memory controllers use a complex function to map a physical address to the correct physical location in memory (i.e., to a specific channel, DIMM, rank, bank, row, and column). While these functions are proprietary and undocumented for Intel processors, they can be reverse engineered through both software- and hardware-based techniques [219]. For example, Figure 2.1 shows the DRAM mapping for a typical configuration found in Ivy Bridge and Haswell systems. As the figure shows, the bank and the rank are determined based on bits 13–21 of the physical address. We have verified that the mapping matches the Haswell processor we use in our experiments.

Row Addressing. As discussed above, DRAM rows have a fixed size of typically 8 KiB. However, from the implementation side, it is usually more important to know what amount of memory has the

same row index. This is sometimes referred to as same-row [117, 243]. If the address goes to the same row and the same bank, it is called *same-row same-bank*; if it goes to different banks but has the same row index, it is called *same-row different-bank* [243].

In our experimental setup, we have a total of 32 DRAM banks, and thus an aligned block of 256 KiB $= 2^{18}$ B of memory has the same row index. In other words, the row index on our system is directly determined by bits 18 and above of the physical address. [219] provide a more extensive discussion.

2.2.2 Row-Buffer Timing Side Channel

Opening a row and loading its contents into the row buffer results in a measurable latency. Even more so, repeatedly alternating accesses to two uncached memory locations will be significantly slower if these two memory locations happen to be mapped to different rows of the same bank [219]. In Section 2.5, we use this timing difference to identify virtual addresses whose contents lie within the same bank, and also uncover the lower 22 physical addressing bits, thereby enabling double-sided Rowhammer attacks.

2.2.3 Rowhammer

The trend towards increasing DRAM cell density and decreasing capacitor size over the past decades has given rise to a reliability issue known as *Rowhammer*. Specifically, repeated accesses to rows in DRAM can lead to bit flips in neighboring rows (not only the direct neighbors), even if these neighboring rows are not accessed [160].

The Root Cause of Rowhammer. Due to the proximity of word lines in DRAMs, when a word line is activated, crosstalk effects on neighboring rows result in partial activation, which leads to increased charge leakage from cells in neighboring rows. Consequently, when a row is repeatedly opened, some cells lose enough charge before being refreshed to drop to an uncharged state, resulting in bit flips in memory.

Performing Uncached Memory Accesses. A central requirement for triggering Rowhammer bit flips is the capability to make the memory controller open and close DRAM rows rapidly. For this, the adversary needs to generate a sequence of memory accesses to alternating DRAM rows that bypass the CPU cache. Several approaches have been suggested for bypassing the cache.

- **Manually Flush Cache Lines.** The x86 instruction set provides the CLFLUSH instruction, which flushes the cache line containing its destination address from all of the levels of the cache hierarchy. Crucially, CLFLUSH only requires read access to the flushed address, facilitating Rowhammer attacks from unprivileged user-level code. On ARM platforms, prior to ARMv8, the equivalent cache line flush instruction could only be executed in kernel mode; ARMv8 does, however, offers operating systems the option to enable an unprivileged cache line flush operation.

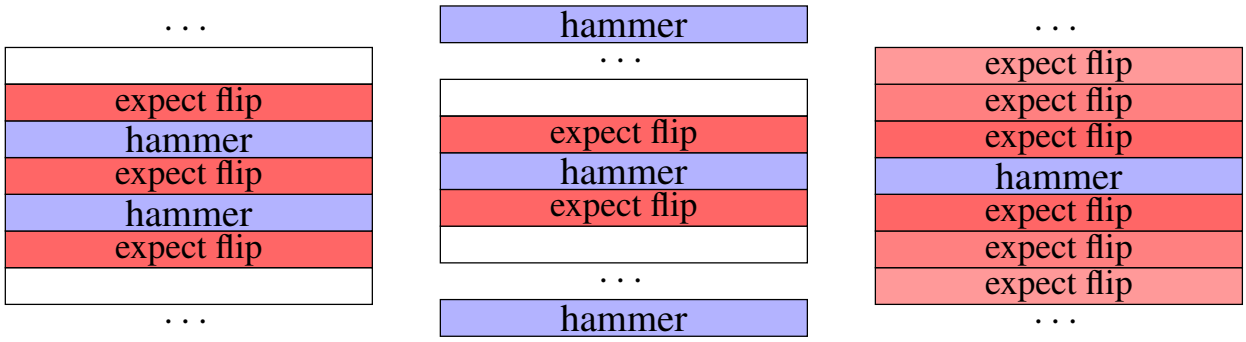
- **Cache Eviction.** In cases where the CLFLUSH instruction is not available (e.g. in the browser), an attacker can force contention on cache sets to cause cache eviction [33, 117].

- **Uncached DMA Memory.** [266] report that the cache eviction method above is not fast enough to cause bit flips on contemporary ARM-based smartphones. Instead, they used the Android ION feature to allocate uncacheable memory to unprivileged userspace applications.

- **Non-temporal instructions.** Non-temporal load and store instructions direct the CPU not to cache their results. Avoiding caching means that subsequent accesses to the same address bypass the cache and are served from memory [224].

Another important distinction between Rowhammer attacks is the strategy in which DRAM rows are activated, i.e., how aggressor rows are selected. See Figure 2.2.

Double-sided Rowhammer. The highest amount of Rowhammer-induced bit flips occur when the attacker *hammers*, that is repeatedly opens and closes, the two rows adjacent to a target row. This approach maximizes the number of neighboring row activations, and consequently the charge leakage from the target row (Figure 2.2a). However, for double-sided hammering, the attacker needs to locate addresses in the two adjacent rows, which may be difficult without knowledge of



(a) Double-sided (b) Single-sided (c) One-location

Figure 2.2: **Different hammering techniques as presented by Gruss et al. (IEEE S&P '18).**

the physical addresses and their mapping to rows. Previous attacks exploited the Linux *pagemap* interface, which maps virtual to physical addresses. However, to mitigate the [243] attack, recent versions of Linux only allow root access to the pagemap interface.

Another avenue used by previous works for finding adjacent rows is to use huge pages, e.g., transparent huge pages (THP), to obtain large blocks of physically contiguous memory [117].

Single-sided Rowhammer. To avoid the need for finding the two rows adjacent to the target row, an adversary can take a more opportunistic approach, which aims to cause bit flips in any row in memory (Figure 2.2b). This can be achieved by guessing several addresses at random, e.g., 8 addresses, in the hope that some fall within two rows in the same bank. With B banks, the probability of having at least one such a pair is $1 - \prod_{i=1}^{n} \frac{B-i}{B}$, i.e., 61.4% for 8 addresses and 32 banks.

Alternatively, the adversary can take a more disciplined approach and use the row-buffer timing channel (Section 2.2.2) to identify rows in the same bank [44, 266].

Because only one of the rows being hammered is located near the target row, single-sided Rowhammer results in fewer bit flips than double-sided Rowhammer [33].

One-location Rowhammer. Finally, one-location hammering [115], is the least restrictive strategy, but also generates the fewest number of bit flips (Figure 2.2c). Here, the attacker repeatedly flushes and then reads from a single row. The presumed cause of flips, in this case, is that newer memory controller policies automatically close DRAM rows after a small amount of time. This obviates the need to open different rows in the same bank.

2.2.4 RSA Background

As the end-to-end attack described in this paper recovers RSA private keys from an OpenSSH server, we now briefly overview the RSA [234] cryptosystem and signature scheme.

A user creates an RSA key pair by first generating two random primes, p and q, a public exponent e, and a private exponent d such that $e \cdot d \equiv 1 \pmod{(p-1)(q-1)}$. The public key is then set to be (e, N) where $N = pq$, and the private key is set to be (d, N). To sign a message m, the signer uses its private key to compute $\sigma \leftarrow z^d \bmod N$, where z is a collision resistant hashing of m. To verify a signature σ, the verifier first hashes the message by herself and obtains a digest z'. She then computes $z'' \leftarrow \sigma^e \bmod N$ using the public key and verifies that $z' = z''$, and rejects the signature otherwise.

The Chinese Remainder Theorem. A common optimization used by most applications to compute $\sigma \leftarrow z^d \bmod N$ is the Chinese Remainder Theorem (CRT). Here, the private key is first augmented with $d_p \leftarrow d \bmod (p-1)$ and $d_q \leftarrow d \bmod (q-1)$. Next, instead of computing $z^d \bmod N$ directly, the signer computes $\sigma_p \leftarrow z^{d_p} \bmod p$ and $\sigma_q \leftarrow z^{d_q} \bmod q$. Finally, the signer computes σ from σ_p and σ_q using the CRT.

Partial Key Recovery. Cryptographic keys recovered through a side channel are typically subject to some measure of noise. Often only a fraction of the key bits are recovered, and their values are not known with certainty. Various researchers have exploited the redundancy present in private key material to correct the errors [43, 141, 207, 212, 214, 292].

2.3 Threat Model

We assume an attacker that runs unprivileged software within the same operating system (OS) as the victim software. The OS maintains isolation between the victim program and the attacker. In particular, we assume that the OS works correctly. We further assume that the attacker cannot exploit microarchitectural side channel leakage from the victim, either because the victim does not leak over microarchitectural channels or because the OS enforces time isolation [103]. We

do assume that the machine is vulnerable to the Rowhammer attack. However, we assume that the attacker only changes its own private memory to bypass any countermeasures and detection mechanisms. Finally, we assume that the attacker is able to somehow trigger the victim to perform allocations of secret data (for example using an incoming SSH connections for the OpenSSH attack in Section 2.7).

2.4 RAMBleed

Previous research mostly considers Rowhammer as a threat to data integrity, allowing an unprivileged attacker to modify data without accessing it. With RAMBleed, however, we show that Rowhammer effects also have implications on data confidentiality, allowing an unprivileged attacker to leverage Rowhammer-induced bit flips in order to *read* the value of neighboring bits. Furthermore, as not every bit in DRAM can be flipped via Rowhammer, we also present novel memory massaging techniques that aim to locate and subsequently exploit Rowhammer flippable bits. This enables the attacker to read otherwise inaccessible information such as secret key bits. Finally, as our techniques only require the attacker to allocate and deallocate memory and to measure instruction timings, RAMBleed allows an unprivileged attacker to read secret data using the default configuration of many systems (e.g., Ubuntu Linux), without requiring any special configurations (e.g., access to pagemap, huge pages, or memory deduplication).

2.4.1 The Root Cause of RAMBleed.

RAMBleed exploits a physical phenomenon in DRAM DIMMs wherein the likelihood of a Rowhammer-induced bit flip depends on the values of the bits immediately above and below it. Bits only flip when the bits both immediately above and below them are in their discharged state [81]. This is in agreement with observations by [160] that hammering with a striped pattern, where rows alternate between all zeros and all ones, generates many more flips than with a uniform pattern.

Data-Dependent Bit Flips. Put simply, bits tend to flip to the same value of the bits in the adjacent rows. That is, a charged cell is most likely to flip when it is surrounded by uncharged cells. This is likely due to capacitors of opposite charges inducing parasitic currents in one another, which cause the capacitors to leak charge more quickly [35]. For our attack to work, is it also crucial that bit flips are influenced only by bits in the same column, and not by the neighboring bits within the same row. This isolation is what allows us to deduce one bit at a time. [81] experimentally demonstrate this to be the case.

A Toy Example. To illustrate with a concrete example, we introduce the notation of an x-y-z configuration to describe the situation in which three adjacent bits in the same column have the values x, y, and z, respectively, where $x, y, z \in \{0, 1\}$. The key reasoning behind our attack is as follows.

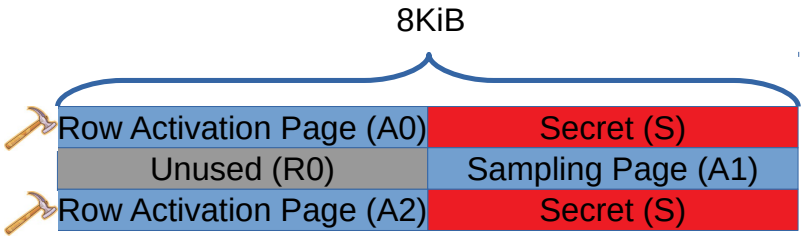
- **True Cells.** For cells where a one-valued bit is represented as the cell being *charged*, the 0-1-0 configuration is the most likely to flip, changing to an all zero configuration (0-0-0) when rows of the first and the last zero-valued cells are hammered. In this case, the surrounding zero-bits in the aggressor rows *enable* the bit flip in the victim row.

- **Anti Cells.** For cells where a one-valued bit is represented by an *uncharged* cell, a 1-0-1 configuration is more likely to flip and change to an all one configuration (1-1-1) when rows of the first and the last one-valued cells are hammered.

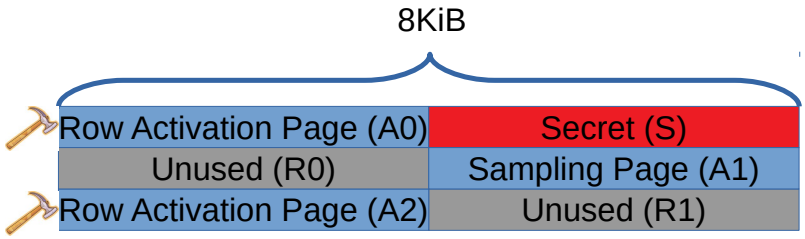
Notation. We adopt Cocojar et al.'s [81] terminology of calling 0-1-0 and 1-0-1 configurations "stripe" patterns, and naming 1-1-1 and 0-0-0 configurations "uniform" patterns. Given this data dependency, we now proceed to build a read side channel in which we read the bits in surrounding rows by observing flips, or lack thereof, in the attacker's row.

2.4.2 Memory Scrambling

One potential obstacle to building our read channel is that modern memory controllers employ *memory scrambling*, which is designed to avoid circuit damage due to resonant frequency [296]



(a) Double-sided Rambleed. Here, the sampling page (A1) is sandwiched between two copies of S.



(b) Single-sided Rambleed. Here, the sampling page (A1) is neighbored by the secret-containing page (S) on a single side.

Figure 2.3: **Page layout for reading out the victim's secret.** Each cell represents a 4 KiB page, meaning that each row represents an 8 KiB row in a DRAM bank. The attacker repeatedly accesses her row activation pages A0 and A2, activating the top and bottom rows. She then reads out corresponding bits in page S by observing bit flips in the sampling page A1.

as well as to serve as a mitigation to cold-boot attacks [128]. Memory scrambling applies a weak stream cipher to the data prior to sending it to the DRAM. That is, the memory scrambler XORs the data with the output of a pseudo-random number generator (PRNG). The seed for the PRNG depends on the physical address of the data and on a random number generated at boot time [143, 196]. The PRNG is cryptographically weak, and given access to the physical data in the DRAMs, an adversary can reverse engineer it and recover the contents of the memory [38, 296].

Bypassing Memory Scrambling. Under our threat model we cannot use the techniques of [296], as we do not assume physical access. However, we can take advantage of the weaknesses of the PRNG. In particular, The boot-time random seed is identical for all rows, and the physical address bits included in the seed are such that several adjacent rows can have the same bits in their addresses. Thus, adjacent rows typically use the same seed, and have the same mask applied. Applying the same mask across multiple rows means that adjacent bits either remain unchanged or are all inverted. Either way, as observed by [81], striped configurations remain striped after scrambling. Hence,

writing a striped configuration to memory results in a striped configuration appearing in the DIMM, maintaining the crucial property that a bit will only flip if the bits immediately above and below have the opposite value.

2.4.3 Exploiting Data-Dependent Bit Flips

We now show how to exploit the data-dependent bit flips presented above to read data without accessing it.

A Leaky Memory Layout. We begin by considering the memory layout presented in Figure 2.3a, where every DRAM row contains two 4 KiB pages. In this layout, we assume that A0, A1, and A2 are the attacker-controlled pages containing known data, S is a page with the victim's secret, and R0 is an arbitrary page. All three rows reside in the same bank. Next, note that attacker pages A0 and A2 reside in the same rows as the copies of S. Since DRAM row-buffers operate at an 8 KiB granularity, accessing a value in A0 activates the entire first row, including the page containing the secret S. Similarly, accessing a value in A2 activates the entire third row, again including the page that contains S. Thus, by repeatedly accessing A0 and A2, the attacker can indirectly use the victim pages containing S for hammering, despite not having any permissions to access them.

Hammering. By hammering the attacker-controlled pages A0 and A2, the attacker induces analog disturbance and interaction between S and A1. Page A1 also belongs to the attacker, who can therefore detect bit flips in it. From these bit flips, the attacker can infer the values of bits in S.

Reading Secret Bit Values. Given a page P, we denote by $P[i]$ the i-th bit in P, where $i \in \{0, 1, \ldots, 32766, 32767\}$. At a high level, given a known flippable bit A1[i] in the page A1, we can read the corresponding bit S[i] (i.e., the bit at the same offset within the frame) in S as follows:

1. **Initialize.** Assuming that the bits are true cells, the attacker first populates all of A1 with ones before hammering.

2. **Hammer.** The attacker repeatedly reads her own pages A0 and A2, thereby using the victim's secret-containing pages to perform double-sided hammering on A1.

3. **Observe.** After hammering, the attacker reads the value of the bit A1[i], which is accessible to her because the page A1 is located inside the *attacker's* own private memory space. We argue that after hammering, the value of A1[i] is equal to the value of S[i]. Indeed, if S[i] equals 0, then before hammering A1[i] would have been in the center of a 0-1-0 stripe configuration. Since A1[i] sits in the center of a flippable stripe configuration, A1[i] will flip from one to zero after hammering. Conversely, if S[i] equal to 1, then A1[i] will be in the center of a 1-1-1 uniform configuration, and will retain its value of 1 after hammering. Thus, in both cases, the attacker reads A1[i] from her own private memory after hammering, which directly reveals S[i].

Double-sided RAMBleed. In the case of anti-cells, the only change we make is that in step 1, we populate A1 with zeros instead of ones. Thus, by observing bit flips in her own pages, the attacker can deduce the values of surrounding cells. Since the secret S surrounds A1 from both sides, we call this "double-sided RAMBleed".

Single Sided RAMBleed. Figure 2.3b presents the memory layout for what we call "single-sided RAMBleed", which differs from the double-side case only in the bottom right frame; instead of another copy of S, an arbitrary page R1 resides below A1. With this configuration, we can still read out bits of S by following the same steps as in the double-sided scenario, albeit with reduced accuracy. The reduction in accuracy is because the value of R1[i] may differ from that of S[i]. Assuming a uniform distribution of bits in R1, in half of the cases, the starting configuration is one of 1-1-0 and 0-1-1, which are neither striped nor uniform. With such configurations, bits tend to flip less than with striped configurations introducing uncertainty to the read values. Yet, in half of the cases R1[i]=S[i], resulting in the same outcome as for the double-sided RAMBleed scenario.

While double-sided RAMBleed maximizes the disturbance interactions between the secret bits and A1, it is also more challenging to execute in practice because it requires two copies of the same data in memory. Nevertheless, in Section 2.7 we show how an attacker can reliably obtain two copies of S, demonstrating an end-to-end attack on OpenSSH.

2.5 Memory Massaging

The descriptions from Section 2.4 assume that the attacker can place the victim's secrets in the layout shown in Figure 2.3, where A0–A2 are allocated to the attacker, and that the attacker knows which bits can flip and in which direction. We now present novel memory massaging primitives that achieve both goals without requiring elevated permissions or special operating system configuration settings (i.e., avoiding huge pages, page map access, memory deduplication).

2.5.1 Obtaining Physically Consecutive Pages

As we can see in Figure 2.3, the attack requires pages located in three consecutive 8 KiB rows in the same bank. While this task was previously achieved using the Android ION allocator [266], no such interface is available in non-Android Linux. Instead, we exploit the Linux buddy allocator [108] to allocate a 2 MiB block of physically consecutive memory. As the same-row-index size (See Section 2.2.1) on our system is 256 KiB, we are guaranteed to be able to build the layout of Figure 2.3 using some of the pages in the block provided by the allocator. We now proceed to provide a short overview of Linux's buddy allocator. See [108] for further details.

Linux Buddy Allocator. Linux uses the buddy allocator to allocate physical memory upon requests from userspace. The kernel stores memory in physically consecutive blocks that are arranged by *order*, where the nth order block consists of $4096 \cdot 2^n$ physically consecutive bytes. The kernel maintains free lists for blocks of orders between 0 and 10. To reduce fragmentation, the buddy allocator always attempts to serve requests using the smallest available blocks. If no small block is available, the allocator splits the next smallest block into two "buddy" halves. These halves are coalesced into one block when they are both free again.

The user space interface to the buddy allocator, however, can only make requests for blocks of order 0. If, for example, a user program requests 16 KiB, the buddy allocator treats this as four requests for one 4 KiB block each. This means that irrespective of their size, user space requests are first handled from the free list of 0 order blocks. Only once the allocator runs out of free 0

order blocks, it will start serving memory requests by splitting larger blocks to generate new 0 order blocks. Thus, while obtaining a virtually consecutive 2 MiB block is trivial and only requires a single memory allocation, obtaining a *physically* consecutive block requires a more careful strategy, which we now describe.

Obtaining a Physically Consecutive 2 MiB Block. We now exploit the deterministic behavior of the buddy allocator to coerce the kernel into providing us with physically consecutive memory, using the following steps:

- **Phase 1: Exhausting Small Blocks.** First, we allocate memory using the mmap system call with the MAP_POPULATE flag, which ensures that the kernel eagerly allocates the pages in physical memory, instead of the default lazy strategy that waits for them to be accessed first. Next, we use the `/proc/pagetypeinfo` interface to monitor available block sizes in the kernel free lists, and continue to allocate memory until less than 2 MiB of free space remains in blocks of order less than 10.

- **Phase 2: Obtaining a Consecutive 2 MiB Block.** Once free space in blocks of order below 10 is less then 2 MiB, we make two requests of size 2 MiB each. Thus, to serve the first request after exhausting the smaller blocks, the kernel needs to split one of the 10th order blocks (whose size is 4 MiB each). This leaves more than 2 MiB in the free list, where all such space comes from the newly-split 4 MiB block, and is served in-order. Thus, the memory allocated for the second request consists of consecutive physical memory blocks, which is exactly what we require.

While the region we obtain in the second allocation is physically consecutive, this approach does not guarantee that the obtained area will be 2 MiB-aligned in the physical memory. Thus, to use the obtained region for Rowhammer, we require an additional step to recover more information about the physical address of the obtained 2 MiB region.[1]

Recovering Physical Addressing Bits. Next, for double-sided hammering, we need to locate addresses in three consecutive rows within the same bank. As some of the physical address bits of

[1]The more naive strategy of first exhausting all smaller blocks and then using one larger request in the hope that it is served from a single large block tends not to work in practice. Any block of order 0 released during the exhaustion phase will be recycled before splitting the large block and will result in a non-consecutive allocation.

the 2 MiB block are used for determining the banks of individual 4 KiB pages, we must somehow obtain these addressing bits for every 4 KiB page in our block.

Since 2 MiB$= 2^{21}$ bytes, and our 2 MiB block is physically sequential, obtaining the low 21 bits of the physical addresses amounts to finding the block's offset from being 2 MiB aligned (where the low 21 bits are 0). In older Linux kernels, an attacker could use the pagemap interface to translate virtual addresses to physical addresses. However, in the current Linux kernel, the interface requires root privileges due to security concerns [243]. Instead of using the pagemap interface, we exploit the row-buffer timing channel of [219] to recover the block offset.

Computing Offsets. To find a block's offset from a 2 MiB aligned address, we take advantage of the fact that our 2 MiB block is physically contiguous and that the set of *distances* between co-banked addresses uniquely defines the block's offset. Figure 2.4 illustrates this concept. The blue block is a 2 MiB aligned block originally found in the fragmented order 10 block, while the red, 2 MiB unaligned block is the region we have obtained from our attack on the allocator. The colored vertical stripes are 4 KiB pages, where two pages of the same color indicate that they reside in the same bank.

The distances d_i, $i \in \{0, 1, 2, \ldots, n\}$ are the differences between the addresses of the i-th page in our block and the very next address located in the same bank. Together, the set $\{d_0, d_1, d_2, \ldots, d_n\}$ forms a *distance pattern* for our block. There are 512 possible offsets for a 4 KiB page within a 2 MiB block; simulations of DRAM addressing confirm that these patterns uniquely identify the block's offset.

Recovering Distance Patterns. We can now use Pessl et al.'s [219] row-buffer timing side channel to find the distances $\{d_0, \cdots, d_n\}$ between pages located in the same bank. Once we have uncovered enough of the distance pattern to uniquely identify a single offset, we have succeeded in computing the offset of our 2 MiB block. This typically occurs after finding fewer than ten distances.

We compute a distance d_i by alternating read accesses between p_i and p_j for $j \in \{i+1, i+2, \ldots, i+2n-2, i+2n-1\}$, where p_i is the page at the i-th offset within the block, and n is the

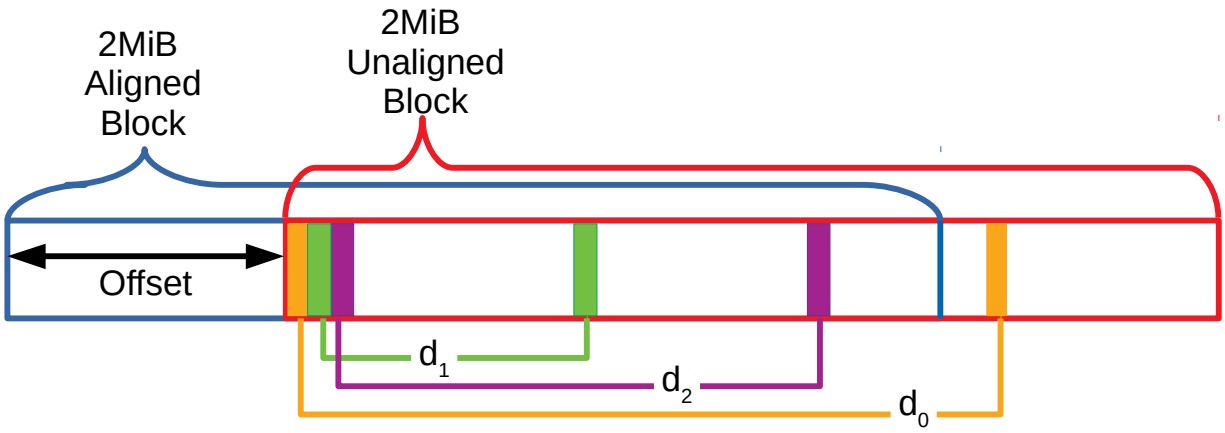


Figure 2.4: **Determining the block's offset.** The blue block is the 2 MiB aligned block that was originally found in the fragmented order 10 block, while the red, 2 MiB unaligned block is the block we have obtained from our attack on the allocator. We compute the offset by finding the distances between co-banked pages d_i, $i \in \{0, 1, 2, .., n\}$, which uniquely identify the offset.

number of pages with the same row index. We then time how long it takes to access both addresses, and average the results over 8,000 trials; the page that corresponds to the greatest read time is identified as residing in the same bank as p_i. The distance d_i is then equal to the difference in the page offset between the two.

The reason we search over the next *two* rows of any bank (i.e., 512 KiB), and not just the next, is that the nature of the DRAM addressing scheme means that the two co-banked pages in consecutive rows can potentially lie anywhere within the memory range with the same row index. When we compute the distances, we make use of Schwarz's [241] optimizations for confusing the memory controller to obtain accurate timing measurements. We empirically find over many trials that this method works with a 100% success rate.

Recovering Bit 21. So far, we have uncovered bits 0–20 of the physical address. As [219] show, however, DRAM addressing on our system depends on bits 0–21. The naive solution is to simply adjust our attack on the memory allocator to obtain a physically contiguous 4 MiB block. This solution, however, is infeasible as the buddy allocator does not track 8 MiB blocks, and thus cannot split an 8 MiB block into two contiguous 4 MiB blocks. Another solution is to simply guess the value of bit 21, doubling the attack's running time.

We can, however, overcome this through an insight into the DRAM addressing scheme. On our

system (a Haswell machine with two DIMMs on a single channel) there are three *bank addressing bits* used to select between the eight banks within a single rank. As specified by [219], bit 21 is only used for computing the third bank addressing bit by XORing bits 17 and 21 of the physical address. Thus, to find two physical addresses a^0, a^1 located in the same bank in consecutive 8 KiB rows, we need to ensure that

$$a^0_{17} \oplus a^0_{21} = a^1_{17} \oplus a^1_{21}$$

where a^i_j is the j-th least significant bit in the i-th physical addresses (a^0, a^1). Then, given a physical address a^0 in the 2 MiB block, when we want to find another physical address a^1 in the same bank, but located in the row above. First we set a^1 to be a^0 plus the size until the next row index. Then, we adjust a^1_{17} to preserve the above equation. Even though we do not know a^0_{21} nor a^1_{21}, we can examine bits 0 till 20 in a^0 to see if the addition of the size of row index done for computing a^1 had resulted in a carry for bit 21. If so, we compensate by flipping a^1_{17} in order to preserve the above equation.

2.5.2 Memory Templating

After obtaining blocks of contiguous memory, we proceed to search them for bits that can be flipped via Rowhammer. We refer to this as the *templating* phase, which is performed as follows. We first use our technique to obtain 2 MiB blocks of physically contiguous memory. Then, we locate addresses that belong to the same bank using the method described above. Next, we perform double-sided hammering with both 1-0-1 and 0-1-0 striped configurations. Finally, we record the locations of these flips for later use with RAMBleed.

2.5.3 Placing Secrets Near Flippable Bits

After templating memory, we exploit Linux's page frame allocation scheme to place the victim's page in the desired physical locations as outlined in Figure 2.3. While a similar task was achieved in [266] on Android's ION allocator by exhausting most of the available memory to control the

placement of the victim, we achieve the same result without memory exhaustion. Following the convention of [266][229][254], we call this technique "Frame Feng Shui", as we are coercing the allocator into placing select pages into a frame of our choosing.

Exploiting Linux's Page Frame Cache. The page frame cache is a structure that caches requests to the buddy allocator. It stores frames in a first-in-last-out (FILO) stack-like data structure, and upon receiving a request for a single page frame, it returns the most recently deallocated page frame [59].[2] Thus, if we assume that the victim, after being triggered, allocates a predictable number of pages before allocating the secret-containing page, we can force Linux's memory allocation mechanisms to place the victim's secret containing page in a page frame of our choice by the following:

- **Step 1: Dummy Allocations.** The attacker allocates n 4 KiB pages by calling mmap with the MAP_POPULATE flag, where n is the number of pages that the victim will allocate before allocating its secret containing page.

- **Step 2: Deallocation.** The attacker inspects her own address space and chooses the target page frame for the victim's secret to land on (one that neighbors the flippable bits). Next, the attacker calls munmap and deallocates the selected frame. The attacker then immediately unmaps all the pages mapped during Step 1. After doing so, the page frame cache's will have the n pages on top, followed by the target page.

- **Step 3: Triggering the Victim.** After Steps 1 and 2, the attacker immediately triggers the victim process, letting it perform its memory allocations. In Section 2.7, we accomplish this by initiating an SSH connection, which is served by the SSH daemon. After being triggered, the victim allocates n pages, which then land in the frames vacated by the pages mapped in Step 1. Finally, the victim allocates its secret-containing page, which then lands in the desired frame, as it will be located on top of the page frame cache's stack-like data structure at this point.

[2] An earlier version of this paper incorrectly attributed this behavior to the buddy allocator. As [71] noted in followup work, it is actually the page frame cache that is exploited by RAMBleed.

2.5.4 Putting It All Together

With the above techniques in place, we can now describe our end-to-end attack. which consists of two phases.

Offline. The attack starts by allocating 2 MiB blocks and dividing them into physically consecutive pages as described in Section 2.5.1. The attacker then templates her blocks and locates Rowhammer-induced bit flips using the methodology described in Section 2.5.2. Notice that this phase is done offline, entirely within the attacker's address space, and without any interaction with the victim. Finally, after the attacker obtains enough Rowhammer-induced bit flips to read the victim's secret, the attacker begins the online phase described below.

Online. In this step, the attacker uses Frame Feng Shui to get the victim to place his secret in the physical memory locations desired by the attacker (e.g., using the layout in Figure 2.3). The attacker then performs the RAMBleed attack described in Section 2.4.3 to exploit the data-dependency with the victim's bits, and subsequently deduces some of their values. Finally, the attacker repeats the online phase step until a sufficient number of secret bits where leaked from the victim (e.g., around 66% percent of the victim's RSA secret key, which is sufficient to mathematically recover of the remaining bits).

2.6 Experimental Evaluation

To measure RAMBleed's capacity as a read side channel, we measure the rate and accuracy of RAMBleed's ability to extract bits across process boundaries and address spaces under ideal conditions and predictable victim behavior.

Next, after evaluating both double-sided and single-sided RAMBleed, in Section 2.7 we evaluate RAMBleed against an OpenSSH 7.9 server (which is a popular SSH server), extracting the server's secret RSA signing keys.

The Victim Process. In the proof-of-concept victim code, the victim waits for an incoming TCP connection, and then copies the secret key into a freshly allocated page (using an anonymous

mmap) upon each TCP connection request. This behavior is akin to a server that runs a decryption routine every time the attacker makes a request, thereby using its secret key.

The Attacker Process. The attacking process uses the techniques described in Section 2.5.1 to obtain 2 MiB physically consecutive blocks, and subsequently templates memory for flippable cells using the methods outlined in Section 2.5.2. Finally, the attacker uses Frame Feng Shui to place the secret-containing page above and below a flippable bit (for single-sided, we only place it above). Concretely, we accomplish this by unmapping the target location and then initiating a TCP connection with the victim. Since $n = 0$ in this case, meaning that the secret is the first allocation upon context switching, the secret-containing page should land in the recently vacated frame. The attacker then hammers the surrounding rows and leaks the secret bits by reading out the flips from its own page. We run both processes as taskset with the same CPU affinity.

Hardware. We use an HP Prodesk 600 desktop running Ubuntu 18.04, featuring an i5-4570 CPU and two Axiom DDR3 4 GiB 1333 MHz non-ECC DIMMs, model number 51264Y3D3N13811, in a single-channel configuration.

Experimental Results. While [81] report that bit flips are deterministic with regards to the surrounding bits (i.e. a bit flips if and only if it is in a striped configuration), on our systems we observe the more general case where the bit flips are probabilistic. Next, the probability of a bit flip highly depends on the type of configuration (striped or uniform). This uncertainty adds noise to our read-channel, which we handle with a variant of the Heninger-Shacham technique [132].

Memory Templating. The time required to template memory and find the needed flips is entirely dependent upon how easily the underlying DIMMs yield bit flips. While [171] and [115] report finding thousands of flips within minutes, we found flips at a more modest rate of 41 flips per minute.

Reading Secret Bits. After templating the memory with a striped 0-1-0 pattern, our experimental code can read out the victim's secret at a rate of 3–4 bits/second. As we can see from the results in Table 2.1, this works with 90% accuracy overall, and 95% accuracy when it comes to identifying 1-bits. This is because "false positive" events, that is, when a 1-1-1 uniform configuration still

Type	Read Accuracy Percents		
	Overall	***False Positive***	***False Negative***
Double-sided	90%	5%	15%
Single-sided	74%	19%	29%

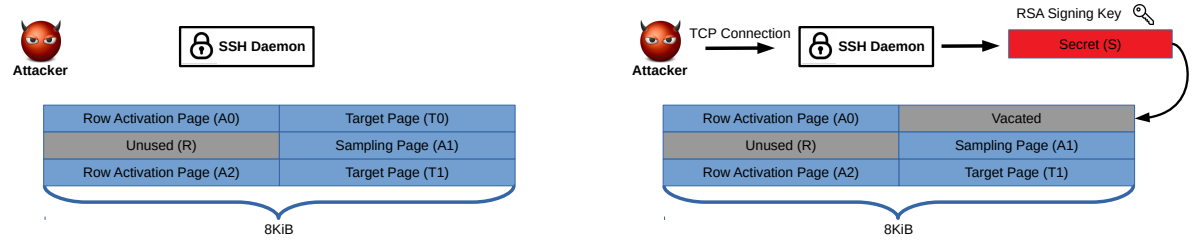
Table 2.1: **RAMBleed Accuracy.** "false positive" events, where a uniform configuration still flips are more rare than "false negative" events, in which a striped configuration refuses to flip.

results in the center bit flipping from one to zero, are much rarer than "false negative" events, in which a 0-1-0 stripe refuses to flip. We can then template with the opposite stripe pattern (1-0-1) and achieve a 95% accuracy rate on the zero-valued bits.

2.7 Attacking OpenSSH

To demonstrate the practical risk that RAMBleed poses to memory confidentiality, in this section we present an end-to-end attack against OpenSSH 7.9 that allows an unprivileged attacker to extract the server's 2048-bit RSA private signing key. This key is what allows an SSH server to authenticate itself to incoming connections. As such, a break of this key enables the attacker to masquerade as the server, thereby allowing her to conduct man-in-the-middle (MITM) attacks and decrypt all traffic from the compromised sessions.

At a high level, our attack operates by coercing the server's SSH daemon to repeatedly allocate and place its private key material at vulnerable physical locations. We then use double-sided RAMBleed to recover a portion of the bits that make up the server's RSA key. Finally, we utilize the mathematical redundancy in RSA keys to correct for errors in extracted bits, as well as recover missing bits that we were unable to read directly. Before describing our attacks, we now describe how OpenSSH manages and uses its keys in response to incoming SSH requests, and how we adapted the techniques from Section 2.5 to specifically target OpenSSH.



(a) The attacker initially owns both target pages T0 and T1.

(b) The attacker makes an SSH connection and performs Frame Feng Shui to land the secret S in the target page T0, which lies above the sampling page (A1).

(c) The attacker repeats the Frame Feng Shui process to land S in the target page T1, below the sampling page (A1).

(d) After achieving the double-sided RAMBleed position, the attacker now hammers the activation pages (A0 and A2) to induce flips in the sampling page (A1).

Figure 2.5: Overview of our attack on OpenSSH

2.7.1 Overview of OpenSSH

The OpenSSH daemon is a root-level process that binds to port 22 and has access to a root-accessible file, which stores the server's RSA private key. As shown in Figure 2.5, when a TCP connection arrives on port 22, the daemon spawns a child process that handles the authentication phase of incoming SSH connections. The child is responsible for both authenticating the server to the client as well as authenticating the client to the server. While the latter can be done either via public-private key pair, or by supplying a password, the former is done by having the server use its RSA private key to sign a challenge issued by the client. Finally, once authentication is complete, the child process spawns an unprivileged grandchild for handling the user's connection. See Figure 2.5.

Key Memory Management. The child process that is spawned by the SSH demon for mutually authenticating an incoming SSH request must first read in the server's private key from the key file into a temporary buffer. At this point, the key will actually be located in memory in two places:

namely, the temporary buffer and the OS' page cache. Unfortunately, we cannot read either of these memory locations via RAMBleed. For the former, this buffer gets overwritten immediately, before we have any chance to read even a single bit using RAMBleed. The latter copy is also inaccessible as it is stored inside the OS' page cache, which is located in a static region of physical memory that is not moved around. Luckily, OpenSSH's authentication process then proceeds to copy the keys into a new buffer maintained by a global structure, aptly named "sensitive_data". This buffer remains in physical memory for the duration of the connection. Thus, our attack aims to read the private key material from this structure.

We now proceed to describe our attack on OpenSSH.

2.7.2 Attack Overview

Our first step is to profile memory, looking for flippable bits. We do this in the same manner described in Section 2.5.2. After finding a sufficient number of flips, we begin the *reading* phase, in which we perform RAMBleed to leak a single bit at a time. At a high level, for each templated bit, we use Frame Feng Shui to place private key material in the configuration shown in Figure 2.3, where A1 is the page containing the flippable bit. We then perform double-sided RAMBleed to leak the bit's value and proceed to the next bit.

2.7.3 Overcoming OpenSSH's Memory Allocation Pattern

To use Frame Feng Shui against OpenSSH, we must determine the value n, which is the number of pages we must unmap after vacating the target frame in order to cause OpenSSH to place the secret in the targeted frame location. Examining the behavior of OpenSSH 7.9 on our system, we found that its allocations pattern is predictable, which allows us to use Frame Feng Shui with a high success rate. More specifically, we found that OpenSSH uses the default RSA key size of 2048 bits, with the following allocation pattern.

- First, the page containing d, the RSA private exponent, is allocated 101 pages after the daemon

accepts a new TCP connection. See Section 2.2.4 for RSA notation.

- Next, a single page containing both p and q is allocated 102 pages after the daemon accepts a new connection.
- Finally, a single page that contains both d_p and d_q is allocated 104 pages after accepting a new connection.

Furthermore, all the private key values mentioned above are located at the same offset within their page upon every incoming connection. Thus, we fix $n = 100, 101$, and 104 respectively for d, p and q, and d_p, d_q. Next, to obtain the configuration in Figure 2.3, we call munmap on the page above A1 and follow it with n munmaps on random pages. We then immediately make a TCP connection, causing the SSH daemon to make n allocations, followed by allocating the secret-containing page, which will then be placed in the target frame. By holding the TCP connection open, we can repeat the process to place the page in the frame below A1, thereby creating two copies of the secret in memory to facilitate double-sided RAMBleed.

Accounting for Allocation Noise. The memory placement technique described above is much more susceptible to noise, as many CPU cycles pass between the point of the original unmapping by the attacker and when the victim maps the key-containing page. Thus, if any pages are allocated or deallocated in that time frame by another process, the key-containing pages will not be placed in the desired locations. To minimize this noise, the attacker yields the scheduler before performing the page deallocaitons, allowing other scheduled system activity to execute. Next, we also use a busy loop after unmapping the pages and before reading the bits, waiting a fixed amount of time for OpenSSH to perform the required allocations. We note here that if we replace the busy loop with a sleep operation, this will likely cause the system to schedule another process and destroy the memory layout. After using RAMBleed to read the bit(s), we close the connections, triggering the daemon to kill the two children.

After mitigating noise in this manner, the memory placement process succeeds against OpenSSH with 83% probability. This means that we will be in the double-sided-RAMBleed situation $0.83^2 =$

Type	Probability
Double-sided RAMBleed	68.89%
Single-sided RAMBleed	28.22%
Unable to place victim	2.39%

Table 2.2: **Probability of OpenSSH placing pages containing private key material into double-sided, single-sided, or unable-to-place situations.**

68.89% of the time, in single-sided RAMBleed $2 \cdot 0.83 \cdot 0.17 = 28.22\%$ of the time, and $0.17^2 = 2.39\%$ of the time we will be unable to place the target page near the flipping row, resulting in random guessing. This, along with potential for RAMBleed to misread bits, gives us an overall accuracy of 82% when reading the OpenSSH host key.

Key Recovery. To recover the key from the noisy bits, we use a variant of [212]'s algorithm, an adaptation of the Heninger-Shacham algorithm [132] for the case that key bits are only known with some probability. Specifically, the algorithm aims to reconstruct the key, bit by bit, starting from the least significant bit. By relating the public (N, e) and private (d, p, q, d_p, and d_q) key components, the algorithm prunes potential keys and dramatically reduces the search space. The algorithm explores a search tree of potential keys while pruning branches that contradict known bits or have a large number of mismatches with probabilistically recovered bits. Our approach is similar to [212], but instead uses a depth-first search in place of a bread-first search.

Through a series of simulations on random RSA 2048 bit keys, we empirically found that our amended Heninger-Shacham algorithm requires 68% recovery of the private key material (d, p, q, d_p, d_q) with an 82% accuracy. This implies that 4200 distinct bits of private key material is sufficient to extract the complete key.

2.7.4 Overall Attack Performance

Memory Templating. We begin our attack by locating the flippable bits in the memory of the target machine. Using the techniques presented in Sections 2.4 and 2.5, we profiled the machine's memory to locate Rowhammer induced bit flips. We note here that the time required to template memory and find the required flips is entirely dependent upon the susceptibility of underlying

DIMMs to Rowhammer attacks. While [115, 171] report finding thousands of flips within minutes, we found flips at a more modest rate of 41 flips per minute, giving us a running time of 34 hours to locate the 84K bit flips required for the next phase of the attack.[3]

We note here that this phase can be performed ahead of time and with user level permissions, without the need to interact with the victim application or its secrets.

Removing Useless Bits. Next, we note that not all of these bitflips are useful for key extraction. First, given OpenSSH memory layout and the location of the key elements in their respective pages, only a $\frac{6144}{32768} = \frac{3}{16}$ fraction of the bits (corresponding to offsets of d, p, q, d_p and d_q) are useful for key recovery. Out of the 84K bit flips recovered in the previous phase, this leaves approximately 15750 bits flips which have the potential to reveal bits of the secret key. Next, we note that these bit flips also contain repetitions in their locations in the page, meaning that two or more bit flips might actually correspond to the same bit of the secret key. After removing such duplicates, we are left with 4.2K bit flips in distinct locations that are useful for key extraction.

Reading Private Key Material. After placing the key containing pages in the desired locations to achieve one of the RAMBleed configurations, we then proceed to hammer A0 and A2 (See Figure 2.3). We have no way of determining if we are in the double-sided, single-sided, or unable-to-place RAMBleed situation, but given the probabilities in Section 2.7.3, it is likely that the bit flip in A1 will depend upon the secret bit values. Overall, this process resulted in recovering 68% of the private key, or 4200 key bits, at a rate of 0.31 bits/second at an accuracy rate of 82% against OpenSSH. We conjecture that the deceased accuracy is due to the combined noise from both the inaccuracy of RAMBleed and Frame Feng Shui.

Key Recovery. As mentioned above, we recover 68% of the key bits with 82% accuracy. Using our amended Heninger-Shacham algorithm, we recover the entire RSA private key in about 3 minutes on a consumer laptop (Dell XPS 15 featuring an Intel i7-6700 3.4 GHz CPU and 32 GiB of RAM).

[3]We empirically found that 84K bit flips was approximately the threshold for locating 4200 usable, unique, flippable bits.

2.8 RAMBleed on ECC Memory

In this section we show how to use RAMBleed to read secret information stored on DIMMs that use ECC memory. Unlike Section 2.4, which shows how RAMBleed can exploit visible bit flips to read secret information, here we show how an attacker can exploit bit flips that were successfully corrected by ECC to read information from the victim's address space.

We begin by providing background on ECC memory.

2.8.1 ECC Memory Background

Memory manufacturers originally designed ECC memory for correcting rare, spontaneous bit flips, such as those caused by cosmic rays. As such, ECC memory uses error correcting codes that can only correct a small number of bits in a single code word, typically only one or two. This is commonly known as SECDED (Single error correction double error detection).

Correction Mechanism. When an ECC enabled system writes data to DRAM, the memory controller writes both the data bits and an additional string of bits, called the *check bits*. These bits offer the redundancy that enables detection and correction of errors. Together, the data and check bits make up a codeword, where the typical sizes for data and check bits are 64 and 8 bits, respectively. Upon serving of a read request from DRAM, the memory controller reads both the data and check bits, and checks for errors. If an uncorrectable error is detected, the controller typically crashes the machine, rather than letting the software operate on corrupted data. Alternately, if the error can be corrected, the memory controller first corrects the error, and only then passes the corrected value to the software. We note that ECC correction and detection occurs only during read requests, and that a bit flip will go undetected until a codeword is read from the DIMM.

Detecting Bit Flips. As [81] describe, this synchronous error correction results in a timing side channel that allows an attacker to determine if a single-bit error has occurred. They found that the overhead incurred by correctable bit flips is on the order of hundreds of thousands of cycles, which the attacker can easily measure.

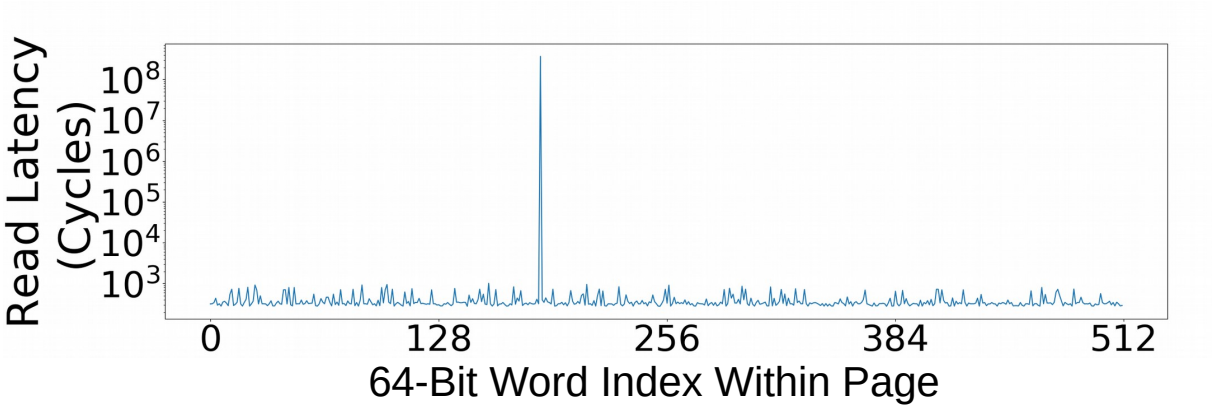


Figure 2.6: **Read latencies for the 64-bit words in a single page.** When ECC corrects an error, the latency is 5 orders of magnitude greater than the common case. This can be seen by the peak for the 186th word, which indicates a bit flip.

Concretely, we can detect the presence of a bit flip in any given word by measuring the read latency from the word. When we read from a word with a single-bit error, the hardware must first complete the ECC algorithm, and often log the error in the firmware log, before the value from the read is returned. If we observe a much longer read latency, it indicates that a bit flip occurred sometime after the last time that the same 64 bit word was read from. This effect is illustrated in Figure 2.6; after performing double-sided hammering on the two aggressor rows, we read from the victim row and observe a crisp peak for the 186th word, clearly indicating a bit flip.

2.8.2 RAMBleed on ECC Memory

We now show how we can leverage the ability to detect the presence of corrected bit flips to read information from the victim's address space. To the best of our knowledge, this is the first demonstration of security implications of corrected bit flips.

Experimental Setup. Following the Intel-1 setup of [81], we demonstrate the RAMBleed attack on ECC memory on a Supermicro X10SLL-F motherboard (BIOS version 3.0a) equipped with an Intel Xeon E3-1270 v3 CPU and a using a pair of Kingston 8GB 1333 MHz ECC DIMMs, model number KVR1333D3E9SK2.

Templating. As with the non-ECC attack, we begin by first templating memory to locate bit flips.

We do so in much the same manner of [81], only with an algorithmic improvement for determining which bit in a row is the flippable bit.

[81] locate bit flips by performing double sided Rowhammer, and then using the timing side channel to locate a word containing a bit flip. They determine which of the 64 bits flipped by setting exactly one of the bits to its charged state, while all the rest are discharged. This results in the targeted bit being in the middle of a striped configuration, while all the other bits in the word are part of a uniform configuration. Next, a long read latency indicates that the single charged bit flipped. Finally, they repeat the process for each bit to determine which bits can be flipped.

To speed up the process of templating memory for bit flips, we replace the single-bit iteration phase with a binary search over the possible locations for the bit flip. That is, after locating a word with a bit flip, we set half of the bits to their charged state, with the other half discharged. We then hammer the aggressor rows again, and record the read latency. If it is long, then the bit flip lies in the half with the charged bits; otherwise, it lies in the other half. We repeatedly reduce the search space by half in this manner, until we have pinpointed the location of the bit flip. Overall this speeds up the templating phase of [81] by a factor of 10.

Reading Bits. After profiling memory and recording the precise locations of flippable bits, we use the memory massaging and Frame Feng Shui techniques described in Section 2.5 to achieve the double-sided RAMBleed configuration. In the non-ECC RAMBleed case, we hammered the aggressor rows and subsequently directly read the victim row for a Rowhammer-induced bit flip, thereby leaking values of secret bits. With ECC, we cannot observe the flips directly. Instead we use the timing side channel and look for long read latencies. As such latencies occur only due to Rowhammer-induced flips, they can be used to reveal the value of the secret bit as described in Section 2.4.

Experimental Results. We can successfully read bits via RAMBleed against ECC memory with a 73% accuracy at a reading rate of 0.64 bits/second in our setup. Since ECC DIMMs are typically built using the same chips as used on non-ECC DIMMs, but with an additional chip for storing the check bits, we attribute the drop in accuracy to the fact that they are simply different sets of DIMMs.

2.9 Mitigations

Unlike previous Rowhammer attacks which compromise integrity, RAMBleed is an attack which compromises confidentiality. Moreover, to leak information cross process and cross address space, RAMBleed only requires that the attacker can read and hammer her own private memory, and does not involve any access or modification to the target's data, code, or address space. As such, RAMBleed can bypass software-based integrity checks that might be applied to the target, such as using message authentication codes (MAC) to protect the target's data. Moreover, techniques designed to protect cryptographic systems against fault attacks (such as Shamir's countermeasure [246]) are also ineffective as they again protect the integrity of the cryptographic computation and not its confidentiality. Other software defenses, such as Brasser et al.'s [60] memory partitioning scheme do not mitigate our attack, as we are not trying to read from kernel memory.

2.9.1 Hardware Mitigations

There are, however, a few commonly proposed hardware-based mitigations that have the potential to mitigate RAMBleed. [160] propose PARA (probabilistc adjacent row activation), wherein activating a row causes nearby rows to activate with some probability. Repeated hammering of an address then increases the likelihood that nearby victim rows will be refreshed, thereby restoring their cells' charges and preventing Rowhammer. PARA has not been widely adopted, as it can only provide a probabilistic security guarantee.

Targeted Row Refresh (TRR). The more recent LPDDR4 standard supports the ability to refresh a targeted row with TRR, where after a row is accessed a set number of times, the nearby rows are automatically refreshed [152]. Despite this mitigation, [115, 266] already report the ability to induce Rowhammer bit flips in the presence of TRR.

Increasing Refresh Intervals. Doubling DRAM refresh rate by halving the refresh interval from 64ms to 32ms is an attempt at reducing the number of bit flips by refreshing victim rows. However,

this is impractical on mobile systems due to the increased power demands. Worse yet, [33] and [115] demonstrate bit flips even under this setting.

Using Error Correcting Codes (ECC). An oft-touted panacea for Rowhammer is the usage of ECC memory, as any bit flip will simply be corrected by the hardware without affecting the software layer. However, as we show in Section 2.8, the hardware error correction implementation actually produces sufficient side channel information for mounting RAMBleed. Thus, while ECC significantly slows RAMBleed, it does not offer complete protection.

2.9.2 Memory Encryption

One defense that does in fact protect against RAMBleed is memory encryption. This is because RAMBleed reads bits directly from memory, which are ciphertext bits in the case that memory is encrypted. Trusted execution environments, such as Intel's Software Guard Extensions (SGX), ARM's Trust Zone, and AMD's Secure Encrypted Virtualization (SEV), in fact fully encrypt the enclave's memory, thereby protecting them from RAMBleed. It should be noted, however, that some enclaves, such as SGX, perform integrity checking on encypted memory; [149] and [115] show that Rowhammer-induced flips in enclave memory halt the entire machine, necessitating a power cycle.

2.9.3 Flushing Keys from Memory

For systems that use sensitive data for a short amount of time (e.g., cryptographic keys), zeroing out the data immediately after use [128] would significantly reduce the risk from RAMBleed. This is because RAMBleed cannot accurately read bits of keys that do not remain in memory for at least one refresh interval (64ms by default). While this countermeasure is effective for protecting short lived data, it cannot by used for data that needs to stay in memory for long durations.

2.9.4 Probabilistic Memory Allocator

Our Frame Feng Shui technique exploits the deterministic behavior of the page frame cache to place the victim's pages in specific locations. Consequently, introducing a sufficient amount of non-determinism into the allocation algorithm will prevent the attacker from placing secrets into vulnerable locations. Such a defense would not, however, necessarily defeat a RAMBleed attacks that use probabilistic memory spraying techniques similar to [243]. The attacker could potentially keep many SSH connections open at once, and then hammer and read from the locations with the correct RAMBleed configurations. The attacker could use the row-buffer timing side-channel to detect the correct configurations.

2.10 Limitations and Future Work

RAMBleed's primary limitation is that it requires the victim process to allocate memory for its secret in a predictable manner in order to reliably read bits of interest. Otherwise, the Frame Feng Shui process described in Section 2.5.3 will not place the secret page in the intended frame. It may be possible, however, to bypass this limitation by using Yarom and Falkner's [291] Flush and Reload technique to determine when the secret page is about to be allocated.

Another limitation is that our attack against OpenSSH 7.9 required the the daemon to allocate the key multiple times. We conjecture, however, that it may be possible to read secrets even when they are never reallocated by the victim. If the secret lies in the page cache, it is likely possible to use Gruss et al.'s [115] memory waylaying technique to repeatedly evict the secret and then bring it back into memory, thereby changing its physical address. Even if it does not lie in the page cache, the attacking process can still evict it by exhausting enough memory to start paging memory to disk. Both of these strategies would, however, be defeated by using Linux's `mlock` system call to lock secret pages into memory, thereby preventing them from ever being evicted to disk.

Next, while we demonstrated our attack on a system using DDR3 DRAM, we do not suspect DDR4 to be a fundamental limitation, assuming that DDR4 memory retains the property

that Rowhammer-induced bit flips are data-dependent. Our techniques for recovering physically sequential blocks depend only on the operating system's memory allocation algorithm, and are thus hardware agnostic. With regard to finding pairs of addresses in different rows of the same bank, [219] have already demonstrated how to reverse engineer the DRAM addressing scheme in DDR4 systems. Furthermore, Rowhammer-induced bit flips in DDR4 have been demonstrated by [16, 115, 171]. We leave the composition of these results to achieve RAMBleed on DDR4 memory to future work.

Finally, RAMBleed's rate of reading memory is modest, topping at around 3–4 bits per second. This allows sufficient time for memory scrubbing countermeasures to remove short-lived secret data from the target's memory. We thus leave the task of improving RAMBleed's read rate to future work.

2.11 Conclusion

In this paper, we have shifted Rowhammer from being a threat only to integrity to also being a threat to confidentiality. We demonstrated the practical severity of RAMBleed by conducting and end-to-end exploit against OpenSSH 7.9, in which we extracted the complete 2048 bit RSA private signing key. To do so, we also developed memory massaging methods and a technique called *Frame Feng Shui* that allows an attacker to place the victim's secret-containing pages in chosen physical frames. By uncovering another channel for Rowhammer based exploitation, we have highlighted the need to further explore and understand the complete capabilities of Rowhammer.

CHAPTER 3

When NIST PQC FIPS Flips: End-to-End Key Recovery on FrodoKEM via Rowhammer

In this work, we recover the private key material of the FrodoKEM key exchange mechanism as submitted to the NIST Post Quantum Cryptography (PQC) standardization process. The new mechanism that allows for this is a Rowhammer-assisted *poisoning* of the FrodoKEM Key Generation (KeyGen) process. The Rowhammer side-channel is a hardware-based security exploit that allows flipping bits in DRAM by "hammering" rows of memory adjacent to some target-victim memory location by repeated memory accesses. Using Rowhammer, we induce the FrodoKEM software to output a higher-error Public Key (PK), $(\mathbf{A}, \mathbf{B} = \mathbf{AS} + \widetilde{\mathbf{E}})$, where the error $\widetilde{\mathbf{E}}$ is modified by Rowhammer.

Then, we perform a decryption failure attack, using a variety of publicly-accessible supercomputing resources running on the order of only 200,000 core-hours. We delicately attenuate the decryption failure rate to ensure that the adversary's attack succeeds practically, but so honest users cannot easily detect the manipulation.

Achieving this public key "poisoning" requires an extreme engineering effort, as FrodoKEM's KeyGen runs on the order of 8 *milli*seconds. (Prior Rowhammer-assisted attacks against cryptography require as long as 8 hours of persistent access.) In order to handle this real-world timing condition, we require a wide variety of prior and brand new, low-level engineering techniques,

including e.g. memory massaging algorithms – i.e. "Feng Shui" – and a precisely-targeted performance degradation attack on the extendable output function SHAKE.

We explore the applicability of our techniques to other lattice-based KEMs in the NIST PQC Round 3 candidate-pool, e.g. Kyber, Saber, etc, as well as the difficulties that arise in the various settings. To conclude, we discuss various simple countermeasures to protect implementations against this, and similar, attacks.

3.1 Introduction

In recent years, the possible emergence of a cryptographically relevant quantum computer (CRQC), a device that exploits quantum-mechanical phenomena to break cryptographic systems, has become more of a reality. A substantial amount of research has gone into the construction of such machines, resulting in increasingly larger quantum computers. For example, in 2019, researchers at Google announced an experimental realization of "quantum supremacy" [29], the demonstration of a programmable quantum device solving a problem that is infeasible for any conventional computer. If fully realized, a CRQC would be capable of undermining the security of digital communications on the Internet and elsewhere [18].

The only practical counter to this threat is the development and deployment of quantum-resistant (or post-quantum) cryptography. In 2016, the U.S. National Institute of Standards and Technology (NIST) announced the beginning of its Post-Quantum Cryptography (PQC) standardization process [206] aimed to standardize quantum-resistant public-key cryptographic algorithms. The conclusion of Round 3 of this project is imminent, resulting in NIST announcing its decisions for the first post-quantum public-key encryption / key establishment mechanism (KEM) and digital signature standards. The pool of candidate-algorithms in the NIST process has been reduced from a large field of 69 submissions in 2017 to a small set of Round 3 finalists and alternates. Among these remaining candidates, lattice-based cryptography plays an especially prominent role, as 5 of the 7 finalists are lattice-based.

Among the lattice-based constructions considered in the 3rd Round, the FrodoKEM encryption protocol has the most mathematically conservative, security-conscious design. It is the only lattice candidate that has no special algebraic structure but instead bases its security on the *plain* Learning With Errors (LWE) problem [231]. In particular, FrodoKEM was recommended by the German Federal Office for Information Security (BSI) [101] in 2020 as "suitable for long-term confidentiality protection."

Next, in addition to security against cryptanalysis, NIST has also made clear throughout the standardization process that PQC candidates also should be resilient against side-channel attacks [57, 136, 218, 227, 228, 244, 259, 273]. While some side-channel attack vectors can be defended against using constant-time coding techniques, other actively-induced effects, such as Rowhammer induced bit flips, cannot be as easily mitigated through careful coding practices. Relatively little is known about the resistance of current PQC constructions to active side-channels like Rowhammer. There are only two prior works investigating such attacks against NIST PQC algorithms, and they examined only the case of digital signatures: specifically, the LUOV and Dilithium signature schemes [147, 198]. In this work, we embark on the task of investigating the Rowhammer resilience of NIST's post-quantum KEM candidates, focusing foremost on FrodoKEM as a representative "hard target."

Specifically, we ask the following main questions:

How resilient are PQC KEM constructions to Rowhammer attacks? What would it take for an adversary to mount such attacks, and what information can be extracted using them?

3.1.1 Our Contributions

We demonstrate the first end-to-end implementation of a successful key recovery attack against FrodoKEM using Rowhammer. At a high level, our attack works as follows. First, we use Rowhammer to poison FrodoKEM's KeyGen process. Then, we use a supercomputer to extract private-key material. Next, we synthesize this data using a tailored key-recovery algorithm. Finally,

any individual FrodoKEM-encrypted session-key can be recovered in around 2 minutes on a commodity laptop.

Our work demonstrates the near-term and high importance of protecting lattice-based cryptography's key generation processes from active side-channel attacks. Especially, we highlight the scenario of running a lattice KEM's KeyGen in a cloud computing environment, where an adversary will have access to a common, shared-memory architecture on which honest users run KeyGen: In such a setting, honest users are particularly vulnerable to our line of attack and should aim to protect themselves with intention.

3.1.2 Overview of Our New Attack

The main ideas underlying our attack on FrodoKEM are as follows.

Decryption Failure Attacks. We begin with an observation made previously [161] that many lattice-based KEMs, including FrodoKEM, have a non-zero decryption failure rate (DFR),[1] and this may lead to a security vulnerability. That is, validly encrypted ciphertexts may occasionally fail to decrypt properly, and moreover, such failing ciphertexts reveal information about the secret key used in decryption. This has led to multiple *decryption failure attacks* and related attack variants in the literature, beginning with Fluhrer's attack on Ring-LWE [96] with many improvements later (e.g. [37, 87, 88, 225]) targeting CPA-secure schemes. These attacks typically make adaptive decryption queries to the receiver of a KEM using carefully crafted ciphertexts. They rely on information of whether a failure occurred or not to gradually recover the secret key.

Protecting FrodoKEM Against Decryption Failures. Decryption failure attacks are mitigated in FrodoKEM and other NIST PQC Round 3 KEMs by use of a Fujisaki-Okamoto (FO) transform [100, 135, 239] that converts a base (IND- or OW-) CPA-secure encryption scheme into a CCA-secure KEM. At a high level, an FO-like transform samples the randomness used to construct a given ciphertext by applying a hash function modeled as a random oracle to its plaintext message. After decryption of a candidate-message, the receiver (deterministically) recomputes a ciphertext

[1] This is not an inherent requirement. Our attack may also succeed against rigid lattice-based KEMs with perfect correctness. We defer the details to the body of the paper.

encrypting that message and checks if this rederived ciphertext exactly matches the initial ciphertext it received. This makes it impossible for an adversary to craft arbitrary ciphertexts (or maul honestly-constructed ciphertexts), as an overwhelming fraction of these will fail the FO re-encryption check.

Therefore, an adversary against such FO-transformed CCA-secure KEMs is forced to run the honest encryption procedure to produce ciphertexts that will not be rejected outright. Indeed, the parameters of KEM candidates in the 3rd round of the NIST PQC process are chosen to balance the work of an adversary who attempts a decryption failure type of attack and an adversary who attempts a direct cryptanalytic attack on the mathematics of the system.[2]

Failure-Boosting Attacks. Further works [48, 93, 94] have explored failure-boosting attacks against the CCA-secure forms of NIST PQC candidate-KEMs. The idea here is that the receiver in a real-world KEM will perform only so many decryptions on behalf of various senders attempting to establish a common session key. (The NIST PQC Call For Proposals [203] sets an absolute upper limit of 2^{64} decryptions.) If a random ciphertext fails to decrypt with probability (say) 2^{-128} or 2^{-256}, it's clearly highly unlikely that even 2^{64} decryption-query attempts will yield a single failing ciphertext. But an adversary can do better by querying the random oracle locally, and generating a large list of candidate-ciphertexts to query for decryption. For example, lattice-based ciphertexts with a higher-norm or heavier weight in certain integer-coordinates are more likely to fail to decrypt than a random ciphertext. A failure-boosting adversary chooses a subset of the most-likely-to-fail ciphertexts, and queries only on those. While the above description summarizes the the current state-of-the-art for decryption failure attacks, modern analyses show that the carefully-set parameters of NIST Round 3 KEMs prevent such lines of attack from succeeding in practice.

Intuition for Our Attack. This is where our new Rowhammer-assisted attack comes in. The Rowhammer side-channel is a hardware-based security exploit that allows flipping bits in DRAM by "hammering" rows of memory adjacent to some target-victim memory location by repeated memory accesses. Specifically, physically-adjacent memory cells interact electrically between themselves, and when a sufficient charge builds up in a given capacitor, it may discharge into adjacent capacitors

[2] As an interesting historical note, CRYSTALS-Kyber-512 changed a certain binomial-sampling parameter from 2 to 3 between the 2nd and 3rd Rounds of the NIST PQC process explicitly to better balance the cost of these attacks.

(where the adversary does not have read/write permissions), causing their logical bits to flip.

By targeting the locations in RAM where the LWE secret key and error are stored during the key generation procedure, we can use Rowhammer to flip some of the higher-order bits of the LWE secret key material. This, in turn, means that the magnitude of certain secret and error coordinates are far higher than would be naturally generated by an honest run of the key generation algorithm. The effect is that decryption failures become somewhat more likely (although not so much that honest users will perceive the difference) – but for an adversary who knows precisely which bits were flipped, obtaining decryption failures via the failure-boosting approach becomes *extremely* more likely (e.g. 2^{-128} becomes $\approx 2^{-12}$).

While this constitutes the core intuition for our new attack, a plethora of additional issues need to be considered to make this approach feasible in the real world.

Rowhammer Timing. A first complication is that Rowhammer bit-flipping requires a minimum window of time at least as long as the electrical refresh rate of the DRAM device being targeted. On typical devices, this is 64 milliseconds, yet FrodoKEM (the slowest of the NIST lattice-based KEM candidates) runs in around 8 milliseconds. We note that FrodoKEM and other NIST PQC KEMs also compute a hash of the generated public key and publish this hash with the associated algebraic material. In turn, this requires that Rowhammer manipulation completes very shortly after (or before!) the natural KeyGen computation completes. In other words, the algebraic computation in the key generation process inherently must last for at least 64 milliseconds for Rowhammer to work. To overcome this challenge, we rely on an additional performance degradation side-channel (to sufficiently delay the execution of the FrodoKEM key generation procedure), so that our Rowhammer attack can do its work.

Memory Profiling. But this step is not enough. A second concern is that the Rowhammer is sensitive to the physical characteristics of individual DRAM modules in the victim device, decided arbitrarily due to process variation during manufacturing at a foundry. For any given page of RAM, some bits will flip frequently when subjected to hammering, and other bits will flip only very slowly. The solution is to profile the target device (before the Lattice KEM algorithm comes online), by

hammering every position in the RAM up front, to determine which bits on which pages are more likely to flip.

Memory Massaging. Given such knowledge of particularly "flippy" locations in various pages of the victim DRAM device, we now want to ensure that the FrodoKEM algorithm allocates its memory in a specific manner – exactly aligning with the precise bits of the FrodoKEM algebraic material that we want to manipulate. The mechanism for doing so is "Feng Shui," or memory massaging. Here, we force the victim to allocate its memory in the exact pages of DRAM that we want, by exploiting the low-level details of the data structure underlying the allocation of Linux page frame caches.

Side Stepping Memory Bit Masks. Yet even this is not quite enough. Our Rowhammer procedure physically flips bits in a one-sided manner: from 0 to 1. There are two problems that can arise with this construction. First, logical bits stored in DRAM are distinct from the physical bits stored in DRAM. In particular, every time the device is booted up, a random bit-mask is applied (akin to a one-time pad). This pad will be consistent, but for N possible pads, the attack succeeds with $1/N$ probability if the machine is reset between profiling and the attack. The solution is to re-profile after any restarts to ensure that the mask does not affect bit locations. Second, 2's complement signed representation dramatically reduces the success probability. For each column, we want to place a single 256-bit flip. The value prior to the flip is equally probable to be negative or positive. The consequence of this representation is that a negative value being targeted to induce a 256-bit flip from 0 to 1 will be unsuccessful because the bit is already a 1. This means that each column can be attacked with probability 2^{-1}, and that each of these locations is an independent probability. So our attack succeeds with additional probability 2^{-N} where N is the number of columns that need to be poisoned, depending on the random sampling of E in the key-generation process.

Generating Failing Ciphertexts. Now we have properly poisoned a FrodoKEM public key. Once the key material has been poisoned, we next need to find sufficiently many failing ciphertexts. For this purpose, we use supercomputing resources (described in Section 3.5) to run an honest encryption procedure on over a trillion distinct messages. Rather than requesting decryptions on this

many messages – which would be outside the bounds of an honest receiver's acceptable workload, and thus certainly detected and rejected by a server – we use our knowledge of the hammered positions to filter out ciphertexts that are unlikely to cause a decryption failure. Specifically, we only keep ciphertexts that have either a large positive or a large negative value in the targeted bit-locations, thus maximizing the probability of failure. By asking that only such ciphertexts be decrypted, we also significantly reduce the probability of detection.

Key Recovery. To perform key recovery given the failing ciphertexts, we present and analyze the following simple algorithm: (1) Construct a set of vectors from the failing ciphertexts generated in the previous stage, (2) Scale these vectors up by a constant factor that depends on the FrodoKEM key distribution and on the rowhammered locations, (3) Take the component-wise average of these scaled-up vectors, (4) Round the resulting vector component-wise to the nearest integer and output this as the candidate key. Comparing our approach with the lattice reduction approach of [82], we find that incorporating the same number of failing ciphertexts using the Toolkit from [82] yields an SVP instance with an estimated hardness of 150 bikz (corresponding to a bit-security of approximately 40), which would not be solvable on a commodity laptop. Further, even obtaining this 150-bikz SVP instance by incorporating decryption failure information using the Toolkit of [82] (as opposed to just computing the hardness *estimates* referenced above), was too computationally intensive for us to run on a commodity laptop due to the large number of failing ciphertexts required in our setting (the required large number of failing ciphertexts is due to the fact that the decryption failure threshold is effectively lowered in our attack, making failures more common, and resulting in less information gained about the secret key from each failure). Thus, for the current setting, our key recovery algorithm outlined above is far more computationally efficient than the approach of [82].

Attacking Session Keys. In our experiment, the attack described thus far allowed us to recover 7/8 columns of the master secret key. However, it is actually computationally hard to recover the remaining bits of the master key. So, instead, we turn to recover the session keys. We show a simple procedure that, when most of the master secret key is known, can recover the full session

key by simply trying all possible values for the remaining bits of the session key. We show that this incremental session key recovery step can complete in only a couple minutes on a commodity laptop.

3.1.3 Source Code and Data

For completeness, we provide all of our source code and our full database of generated ciphertexts at this URL: https://www.dropbox.com/sh/ksfd38as1p21fa0/AAAHWP_hisRM8tZgOmClREIOa?dl=0

See the `Readme.txt` file first.

3.1.4 Paper Organization

The rest of this paper is organized as follows. First, in Section 3.2, we review some necessary background and prior work as well as notation and definitions. Then, in Sections 3.3 - 3.6, we dive into the details of each component of our attack. Specifically, in Section 3.3, we describe how decryption failures can be used to recover a FrodoKEM secret key. Then, in Section 3.4, we describe how to use a Rowhammer attack to enable creation of decryption failures in FrodoKEM. In Section 3.5 we describe how to use a supercomputer to find sufficiently many failing ciphertexts for our attack. Finally, in Section 3.6, we conclude our attack by showing how we can recover a FrodoKEM session-key. We then discuss how our techniques can be extended to other lattice-based KEMs in Section 3.7 and possible countermeasures in Section 3.8.

3.2 Background

We now provide background on the FrodoKEM protocol as well as on the Rowhammer attack that we use to break its security.

3.2.1 Notation

We use bold lower case letters to denote vectors, and bold upper case letters to denote matrices. We use column notation for vectors, and start indexing from 0. We denote by $\mathbf{I}_n$ the n-dimensional identity matrix and denote by $\mathbf{x} \cdot \mathbf{y}$ the inner product of vectors $\mathbf{x}, \mathbf{y}$ of the same dimension. In addition, we make use of Einstein notation to delineate between rows and columns of a matrix (e.g. For a matrix $\mathbf{A}$, $\mathbf{a}_i$ is the i^{th} row of the matrix, and $\mathbf{a}^j$ is the j^{th} column of the matrix). We denote by $(\mathbf{x}||\mathbf{y})$ the concatenation of two column vectors $\mathbf{x}, \mathbf{y}$, which is a column vector whose dimension is the sum of the dimensions of $\mathbf{x}$ and $\mathbf{y}$. Random variables—i.e. variables whose values depend on outcomes of a random experiment—are denoted with lowercase calligraphic letters e.g. $\mathcal{a}$, $\mathcal{b}$, $\mathcal{e}$, while random vectors are denoted with uppercase calligraphic letters e.g. $\mathcal{C}, \mathcal{X}, \mathcal{Z}$.

3.2.2 Statistics

Definition 1 (Univariate normal distribution)**.** We denote by $\mathcal{N}(\mu, \sigma^2)$ the univariate normal (Gaussian) distribution with mean μ, variance σ^2, and probability density function (pdf)

$$x \mapsto \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right).$$

Definition 2 (Multivariate normal distribution)**.** Let $d \in \mathbb{Z}$, $\boldsymbol{\mu} \in \mathbb{Z}^d$ and let $\boldsymbol{\Sigma}$ be a positive definite matrix of dimension $d \times d$. We denote by $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ the multivariate normal (Gaussian) distribution with mean $\boldsymbol{\mu}$, covariance $\boldsymbol{\Sigma}$, and probability density function (pdf)

$$\mathbf{x} \mapsto \frac{1}{\sqrt{(2\pi)^d \cdot \det(\boldsymbol{\Sigma})}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})^T\right).$$

Definition 3. The error function, denoted `erf` is defined as:

$$\texttt{erf}(z) := \frac{2}{\sqrt{\pi}} \int_0^z \exp(-t^2)\, dt.$$

$\texttt{erf}(z)$ is the probability that an x sampled from $\mathcal{N}(0,1)$ falls in the range $[-z, z]$.

3.2.3 FrodoKEM

Definition 4 (Public-key encryption scheme (PKE))**.** A public-key encryption scheme is a tuple of algorithms (KeyGen, Enc, Dec):

- KeyGen() outputs (pk, sk), where pk is the public key and sk is the secret key.
- Enc(pk, μ) takes as input a public key pk and a message μ and outputs a ciphertext c.
- Dec(sk, c) takes as input a secret key sk and a ciphertext c and outputs a message μ, or fails.

A public-key encryption scheme is correct if an honest in-order execution of KeyGen, Enc, and Dec, results in Dec outputting the same message that is input into Enc (with overwhelming probability). The relevant notion of security for a public-key encryption scheme in our discussion is IND-CPA. A public-key encryption scheme is IND-CPA if any adversary cannot distinguish between ciphertexts of two adversarially-chosen messages with non-negligible advantage.

Definition 5 (Key Establishment Mechanism (KEM))**.** A key establishment mechanism (KEM) is a tuple of algorithms (KeyGen, Enc, Dec):

- KeyGen() outputs (pk, sk), where pk is the public key and sk is the secret key.
- Enc(pk) takes as input a public key pk and outputs (c, k), where c is the ciphertext that encapsulates the shared session key k.
- Dec(sk, c) takes as input a secret key sk and a ciphertext c, and outputs a shared session key k, or fails.

A key-encapsulation mechanism is correct if an honest in-order execution of KeyGen, Enc, and Dec results in the same shared session key output by Enc and Dec (with overwhelming probability). Moreover, a KEM is IND-CCA if any adversary with access to a decapsulation oracle, on input of a

random challenge session key and a challenge ciphertext, cannot distinguish between the case when the session key is encapsulated in the ciphertext and the case when the session key is independent and uniformly random, with non-negligible advantage.

FrodoKEM is an IND-CCA key-encapusulation mechanism. FrodoKEM is constructed as a Fujisaki-Okamoto transform of FrodoPKE, an IND-CPA public-key encryption scheme whose security is based on the hardness of Learning with Errors. Learning with Errors was first defined and studied as the basis for a public-key cryptosystem by Regev in [231]. Lindner and Peikert later in [174] showed a more efficient public-key encryption scheme based on the hardness of Learning with Errors. FrodoPKE is an instantiation of Lindner and Peikert's construction. We give the definitions of Learning with Errors, both the search and decision variants, below. We note that the definitions we state here are of so-called normal-form Learning with Errors introduced in [183], which shows that normal-form Learning with Errors is at least as hard as Learning with Errors as originally defined by Regev in [231].

Definition 6 (LWE distribution)**.** The LWE distribution is parametrized by positive integers (m, n, q) and an error distribution χ over $\mathbb{Z}$. The LWE distribution is sampled by sampling a uniformly random $\mathbf{A} \leftarrow \mathbb{Z}_q^{m \times n}$, $\mathbf{s} \leftarrow \chi^n$, $\mathbf{e} \leftarrow \chi^m$, and outputting $(\mathbf{A}, \mathbf{b} \coloneqq \mathbf{A}\mathbf{s} + \mathbf{e} \bmod q)$.

Definition 7 (The Search-LWE Problem)**.** Given $(\mathbf{A}, \mathbf{b})$ drawn from the LWE distribution, search-LWE problem asks to find $\mathbf{s}$ in the support of χ^n such that $\mathbf{e} \coloneqq \mathbf{b} - \mathbf{A}\mathbf{s}$ is in the support of χ^m modulo q.

Definition 8 (The Decision-LWE Problem)**.** The decision-LWE problem asks to distinguish between the LWE distribution and the uniform distribution over $(\mathbb{Z}_q^{m \times n}, \mathbb{Z}_q^m)$.

It's been shown (e.g. in [231]) that there is a reduction from search-LWE to decision-LWE.

We give a high-level description of FrodoPKE in algorithms 1, 2, and 3, where we omit details about (pseudo)random generation using symmetric primitives, sampling from the error distribution χ, and bit-level representations. We point readers to the Frodo specification [23] for

Algorithm 1 FrodoPKE.Keygen()

1: $\mathbf{A} \leftarrow \mathbb{Z}_q^{n \times n}$, $\mathbf{S} \leftarrow \chi^{n \times \bar{n}}$, $\mathbf{E} \leftarrow \chi^{n \times \bar{n}}$
2: $\mathbf{B} = \mathbf{AS} + \mathbf{E} \bmod q$
3: **return** $(pk = (\mathbf{A}, \mathbf{B}), sk = \mathbf{S})$

Algorithm 2 FrodoPKE.Enc(pk, μ)

1: $\mathbf{A}, \mathbf{B} = pk$
2: $\mathbf{S}', \mathbf{E}' \leftarrow \chi^{\bar{m} \times n}$
3: $\mathbf{E}'' \leftarrow \chi^{\bar{m} \times \bar{n}}$
4: $\mathbf{B}' = \mathbf{S}'\mathbf{A} + \mathbf{E}'$
5: $\mathbf{V} = \mathbf{S}'\mathbf{B} + \mathbf{E}''$
6: $(\mathbf{C}_1, \mathbf{C}_2) = (\mathbf{B}', \mathbf{V} + Encode(\mu))$
7: **return** $c = (\mathbf{C}_1, \mathbf{C}_2)$

Algorithm 3 FrodoPKE.Dec(sk, μ)

1: $\mathbf{S} = sk$
2: $(\mathbf{C}_1, \mathbf{C}_2) = c$
3: $\mathbf{M} = \mathbf{C}_2 - \mathbf{C}_1\mathbf{S}$
4: **return** $\mu = Decode(\mathbf{M})$

details. In the following, $n, \bar{m}, \bar{n}, B$, and q are integer parameters. Specifically, for Frodo640, $n = 640, \bar{m} = \bar{n} = 8, B = 2$, and $q = 32768$.

We now describe the $Encode$ and $Decode$ functions to complete our description of FrodoPKE. The $Encode$ function encodes an integer k such that $0 \leq k < 2^B \leq q$ as an element in $\mathbb{Z}_q$:

$$Encode(k) \coloneqq k \cdot q/2^B.$$

In Frodo, q is a power of 2, and therefore $q/2^B$ is an integer. The $Decode$ function extracts a B-bit integer from an element of $\mathbb{Z}_q$:

$$Decode(k) \coloneqq \lfloor c \cdot 2^B/q \rceil \bmod 2^B.$$

We extend the domain of $Encode$ and $Decode$ to vectors and matrices by entry-wise application. The $Encode$ and $Decode$ functions have an error-correcting property stated in [23, Lemma 2.18].

We re-state the result below.

Lemma 1. Let $q = 2^D$, $B \leq D$. Then $Decode(Encode(k) + e) = k$ for any $k, e \in \mathbb{Z}$ such that $0 \leq k < 2^B$ and $-q/2^{B+1} \leq e < q/2^{B+1}$.

The correctness of FrodoPKE follows because the decryption computes

$$\begin{aligned}\mathbf{M} &= \mathbf{C}_2 - \mathbf{C}_1\mathbf{S} \\ &= Encode(\mu) + \mathbf{S}'\mathbf{E} - \mathbf{E}'\mathbf{S} + \mathbf{E}''.\end{aligned}$$

Let $\mathbf{E}''' = \mathbf{S}'\mathbf{E} - \mathbf{E}'\mathbf{S} + \mathbf{E}''$. By Lemma 1, if the entries of $\mathbf{E}'''$ are sufficiently small, we have that $Decode(\mathbf{M}) = \mu$. A lower bound on the probability of this event can be computed explicitly using Frodo's parameter search script in Frodo submission available at [205]. Conversely, if an entry in the decoding error $\mathbf{E}'''$ exceeds the decoding threshold $[-q/2^{B+1}, q/2^{B+1})$, a decoding failure will occur in the corresponding entry of μ.

Given an IND-CPA public-key encryption scheme such as FrodoPKE above, a generic Fujisaki-Okamoto transform can be applied to obtain an IND-CCA key-encapsulation mechanism. We give a high-level description of the Fujisaki-Okamoto transform used by FrodoKEM in algorithms 4, 5, and 6, where H is a cryptographic hash function (e.g. SHAKE) modeled as a random oracle.

For our attack, we consider Frodo640, the NIST level 1 (as hard as brute-force search on AES-128) parameter set of FrodoKEM. For Frodo640, the parameters are as follows: $n = 640, \bar{n} = \bar{m} = 8, q = 2^{15}, B = 2$. Therefore, the shared session key is decoded from an 8×8 matrix and the decoding threshold is $[-4096, 4096)$.

3.2.4 The Rowhammer Bug

Rowhammer is a phenomenon wherein repeated accesses to a row in DRAM can induce bit flips in the neighboring rows [160, 199]. This is because activations of the row's wordline cause the capacitors storing bit values in neighboring rows to discharge slightly due to parasitic current. If this occurs a sufficient number of times to drop the voltage below the "charged" threshold before

Algorithm 4 FrodoKEM.KeyGen()

$(pk, sk) \leftarrow$ FrodoPKE.KeyGen()
return $(pk, (pk, sk))$

Algorithm 5 FrodoKEM.Enc(pk)

$\mu \leftarrow \{0,1\}^{\ell}$
$r \leftarrow H(\mu)$
$c \leftarrow$ FrodoPKE.Enc$(pk, \mu; r)$
return (c, μ)

Algorithm 6 FrodoKEM.Dec($(pk, sk), c$)

$\mu \leftarrow$ FrodoPKE.Dec(sk, c)
$r \leftarrow H(\mu)$
$c' \leftarrow$ FrodoKEM.Enc$(pk, \mu; r)$
if $c' = c$ **then**
    return μ
else
    Decapsulation fails.

the DRAM refreshes, which typically occurs every 64ms, the logical value of the bit flips. The row that is repeatedly activated, or "hammered" is called the "aggressor row," while the rows containing the induced bit flips are "victim" rows.

Double-Sided Rowhammering. In order to use Rowhammer to intentionally flip inaccessible bits, it is more effective if the attacker repeatedly triggers accesses to memory values both *above and below* the victim rows within the same bank. If the victim row contains bits susceptible to Rowhammer, this memory access pattern is far more likely to induce bit flips than single sided hammering.

Repeatability of Flips. A crucial property of Rowhammer for building attacks is that Rowhammer-induced bit flips are repeatable. This means that a bit that flips at any given point in time is likely to flip again in the future when hammered, and similarly bits that do not flip after being hammered are unlikely to flip upon further hammering. While roughly half of flippable bits can flip in the 1-to-0 direction, and the other half in the opposite direction, any given bit can only possibly flip in one direction per boot. This means that an adversary can "profile" a given machine prior to an attack by

hammering memory and building a map of where the flippable bits are and what directions they flip. This precision allows us to accurately poison the FrodoKEM key in a fairly deterministic manner.

Rowhammer History. A multitude of works from both academia and industry have helped Rowhammer evolve from its conception as a theoretical attack to a realistic attack vector with serious implications. After [160] first uncovered the Rowhammer phenomenon, researchers primarily focused on how to use Rowhammer for privilege escalation attacks and obtaining arbitrary read/writes[58, 98, 115, 165, 168, 243, 261, 266]. Rowhammer attacks from the browser [84, 117], over the network [176, 257], and against ECC memory [81] soon followed, along with new hammering patterns[99, 150, 181] enabling Rowhammer against DDR4 memory.

Cryptographic Applications of Rowhammer. In addition to compromising standard isolation primitives, Rowhammer-induced bit flips were also used to break the security of cryptographic primitives. More specifically, [229] demonstrate how Rowhammer can be used to break RSA signature validation, while [198] demonstrate an attack on the LUOV signature scheme. Finally, [168] show how to use Rowhammer-induced bit flips to directly read bits from memory, allowing for the recovery of RSA keys directly from the target's address space.

3.3 Decryption Failure Attack on Rowhammer-Poisoned FrodoKEM

In this section, we assume the reader has familiarity with decryption failure attacks (see Section 3.1.2) and Rowhammer attacks (see Section 3.2.4). It will also be helpful for the reader to reference Section 3.1.2 for an overview of our attack and our high-level strategy of combining a decryption failure and Rowhammer attack.

Given the above background, we begin by highlighting an important difference between our attack and a traditional decryption failure attack [37, 48, 87, 88, 93, 94, 96, 225]: In a traditional decryption failure attack, the failing ciphertexts contain a significant amount of information on the secret key, so that only a few thousand failures are required for a full key recovery. Because our

effective failure threshold is so much lower, the failing ciphertexts generated using our Rowhammer-altered public key contain less information. The failure event occurs orders of magnitude more often for randomly generated ciphertexts, and in turn, the fraction of secret keys that are consistent with a single failure event is far higher. This induces a trade-off in which we require significantly more failing ciphertexts to gain enough information to recover the key, but require less computation *overall*, since it is much easier to find failing ciphertexts. It is important to quantify this trade-off, as it informs which specific bits of the public key to target with Rowhammer. First, we will analyze the effects of Rowhammer on the decryption failure rate, which gives us an estimate for the total amount of work required to generate a *single* failing ciphertext.

3.3.1 Decryption Failure Rate Analysis

The decoding error in Frodo640 is an 8×8 matrix

$$\mathbf{E}''' = \mathbf{S}'\mathbf{E} - \mathbf{E}'\mathbf{S} + \mathbf{E}''. \tag{3.1}$$

The decoded message, which is also an 8×8 matrix, is correct in each of its entries if the corresponding entry in $\mathbf{E}''' \bmod q$ is in the interval $[-4096, 4096)$. Otherwise, the decoded entry is incorrect.

Consider the entry $e'''_{i,j}$ at position (i, j) of $\mathbf{E}'''$. Let $\mathbf{s}'_i$ and $\mathbf{e}'_i$ be the i-rows of $\mathbf{S}'$ and $\mathbf{E}'$ respectively, and $\mathbf{e}^j$ and $\mathbf{s}^j$ be the j-columns of $\mathbf{E}$ and $\mathbf{S}$ respectively. Then,

$$e'''_{i,j} = \mathbf{s}'_i \cdot \mathbf{e}^j - \mathbf{e}'_i \cdot \mathbf{s}^j + e''_{i,j}.$$

Let χ be the error distribution in Frodo. If key generation is executed correctly, $\mathbf{e}^j$ is sampled from χ^n. However, due to row-hammering, the true distribution of $\mathbf{e}^j$ is $\chi' \times \chi^{n-1}$, where χ' is defined as

follows. Let $p_\chi : \mathbb{Z} \to \mathbb{R}$ be the density function of χ. Then, $p_{\chi'}$ is defined by

$$p_{\chi'}(x) = \begin{cases} p_\chi(x) & \text{if } -12 \leq x < 0 \\ p_\chi(x - 256) & \text{if } 256 \leq x < 269 \\ 0 & \text{otherwise} \end{cases}.$$

For Frodo640, the support of p_χ is $\{-12, \dots 12\}$ whereas the support of $p_{\chi'}$ is $\{-12, -11, \dots, -1, 256, 257, \dots, 268\}$. The reference implementation of FrodoKEM uses 16-bit integers. Here, we are targeting an attack that only requires that one bit flip per column of $\mathbf{E}$, since requiring more bit flips can reduce the success of the Rowhammer attack. By forcing the eighth-order bit of an entry to 1, the Rowhammer attack effectively adds 256 to any positive value and leaves negative values unchanged. Thus explaining the distribution of χ' shown above. Indeed, this analysis can be repeated for other orders of bits. We also consider the failure probabilities for a sixth-order (64) bit Rowhammer and a seventh-order (128) bit Rowhammer.

With $\mathbf{e}_j$ sampled from $\chi' \times \chi^{n-1}$, an honest encapsulation has decoding error distributed as

$$e'''_{i,j} \sim \chi \cdot \chi' + (\chi \cdot \chi)^{\circledast(2n-1)} + \chi, \tag{3.2}$$

where the notations are as follows. If χ_1 and χ_2 are two distributions, then $\chi_1 \cdot \chi_2$ is the distribution of the product of 2 independent random variables having distributions χ_1 and χ_2 respectively. Moreover, $\chi_1 + \chi_2$ is the convolution of χ_1 and χ_2, which is the distribution of the sum of 2 independent random variables having distributions χ_1 and χ_2 respectively. Finally, $\chi_1^{\circledast k} := \underbrace{\chi_1 + \chi_1 + \cdots + \chi_1}_{k \text{ times}}$. Concretely, by explicitly computing the distribution in equation (3.2), we have that such $e'''_{i,j}$ causes decryption failure with probability approximately $2^{-27.8}$ ($2^{-113.2}$ for a 64-bit Rowhammer, and 2^{-76} for a 128-bit Rowhammer).

For our adversarial attack, $\mathbf{s}'_i$ is not honestly sampled from χ^n. Instead, we filter the output of the random oracle and select only the values $\mathbf{s}'_i$ that have ± 12 in the same coordinate where χ' is in $\mathbf{e}^j$. Let τ be the distribution with probability $1/2$ at 12 and -12. Then, our adversarial $e'''_{i,j}$ has the

distribution

$$e'''_{i,j} \sim \tau \cdot \chi' + (\chi \cdot \chi)^{\circledast(2n-1)} + \chi.$$

Concretely, such $e'''_{i,j}$ exceeds the decoding threshold of Frodo640 with probability approximately 2^{-13} ($2^{-98.7}$ for a 64-bit Rowhammer, and 2^{-61} for a 128-bit Rowhammer). To obtain one such $\mathbf{s}'_i$, the expected number of calls to the random oracle is 2^{15}. If there are c target columns of $\mathbf{E}$, since there are 8 rows of $\mathbf{S}'$, the expected number of calls per target column is $2^{15}/(8c)$. In our experiment, $c = 7$.

We note that there is a large variation in decryption failure rates conditioned on whether χ' is negative. The decryption failure rates conditioned on χ' being negative are negligible. Therefore, the decryption failure rates calculated above are negligibly different from decryption failure rates conditioned on χ' being positive. So we can mount our attack to recover $\mathbf{e}^j$ and $\mathbf{s}^j$ with probability $\Pr[\chi' > 0] = \Pr[\chi \geq 0]$.

Next we will describe our key recovery procedure, and analyze the number of failing ciphertexts necessary for recovering (each column of) the secret.

3.3.2 FrodoKEM Key recovery

Each failing entry of the error matrix in (3.1) gives rise to a linear inequality involving a column from each of $\mathbf{S}$, $\mathbf{E}$, a row from each of $\mathbf{S}'$, $\mathbf{E}'$, and the matching coordinate from $\mathbf{E}''$. Let $e'''_{i,j}$ be a failing entry of the error matrix. Assuming we exceed the failure threshold t in the positive direction, we obtain the following linear inequality

$$\begin{aligned} e'''_{i,j} = \mathbf{s}'_i \cdot \mathbf{e}^j + e''_{i,j} - \mathbf{e}'_i \cdot \mathbf{s}^j \geq t \\ (\mathbf{s}'_i \| - \mathbf{e}'_i) \cdot (\mathbf{e}^j \| \mathbf{s}^j) \geq t - e''_{i,j} \end{aligned}$$

Let us first consider a standard decryption failure attack with adjusted failure threshold $(t - e''_{i,j})$. In this attack, the adversary generates honestly distributed ciphertexts by running the encryption

algorithm (this is enforced in practice by the Fujisaki-Okamoto transform) and queries them to the decryption oracle. The attacker then collects all the ciphertexts that led to decryption failure. We will first show how to perform key recovery in the above setting. We will then explain why our Rowhammer attack essentially reduces to a standard decryption failure attack with a further adjusted failure threshold $(t' - e''_{i,j})$, where $t' = t - 12 \cdot e_{i,j}$, and one fewer dimension.

In the above attack, $\mathbf{S}'$ and $\mathbf{E}'$ that produce failing ciphertexts are correlated with $\mathbf{E}$ and $\mathbf{S}$ respectively. Given a set of failing ciphertexts, there are various ways to use the correlation between the failing ciphertexts and the LWE secrets to learn information about the LWE secret. For example, D'Anvers et al. [93] use the correlation to calculate explicit posterior probabilities for the values of the secrets. These then allow them to calculate revised distributions of the secret with a smaller variance.

The posterior probabilities calculated as in D'Anvers et al. [93] are difficult to analyze. We therefore instead consider the distribution of failing ciphertexts as in [82]. This distribution is parametrized by the LWE secret itself, which is, of course, unknown. We then use our supply of failing ciphertexts–which constitute random samples from this distribution–to learn the hidden parameters. Specifically, as observed in [82], failing ciphertexts can be decomposed into a single component, a, distributed as a truncated, univariate Gaussian in the direction of the secret, and components distributed as independent Gaussians in the directions orthogonal to the secret.

Note that for FrodoKEM, the secret and error distributions are discrete approximations of spherical Gaussian distributions. However as we will elaborate on below, the distribution of the failing ciphertexts can be very well approximated as a (continuous) spherical multivariate Gaussian distribution with known variance and unknown mean (where the mean depends on the LWE secret). We can thus reframe the key recovery problem as the problem of estimating the mean of a multivariate Gaussian distribution with a known covariance matrix, given samples from that distribution.

Modeling Failing Ciphertexts More formally, let us assume that $\mathbf{s}, \mathbf{e}, \mathbf{s}', \mathbf{e}'$, and e'' are as above, are all drawn coordinate-wise from a Gaussian distribution with zero mean and covariance σ_{se}^2. We

omit the explicit row and column coordinates to reduce notational clutter, as the following holds regardless of the location of the failing coordinate in $\mathbf{E}'''$. Then, let us assume the norm of $(\mathbf{e}\|\mathbf{s})$ is exactly $\ell = \sqrt{n}\sigma_{se}$, where n is the dimension of $(\mathbf{e}\|\mathbf{s})$. This assumption is rather innocuous due to the high concentration of the norm of a Gaussian vector. As $(\mathbf{s}'\| - \mathbf{e}')$ is the part of the failing ciphertext that induces the failure condition, we can condition its distribution on the learned linear inequality

$$(\mathbf{s}'\| - \mathbf{e}') \sim \mathcal{N}(\mathbf{0}, \sigma_{se}^2 \mathbf{I}_n) \mid (\mathbf{e}\|\mathbf{s}) \cdot (\mathbf{s}'\| - \mathbf{e}') \geq t - e''$$

After conditioning, $(\mathbf{s}'\| - \mathbf{e}')$ decomposes into $(\mathbf{s}'\| - \mathbf{e}') = \frac{a}{\ell}(\mathbf{e}\|\mathbf{s}) + \mathcal{W}'$ where $\mathcal{W}'$ is a random vector distributed as a Gaussian of covariance $\sigma_{se}^2 \mathbf{\Pi}^{\perp}_{(\mathbf{e}\|\mathbf{s})}$ (i.e. independent noise in all directions orthogonal to the secret) and a is a random variable that is independent of $\mathcal{W}'$ and follows a distribution that we denote $\mathcal{N}^{\geq t/\ell}_{\sigma_{se}}$—the univariate Gaussian of variance σ_{se}^2 conditioned on $a \geq (t - e'')/\ell$.

In essence, each failing ciphertext gives a draw from the decomposed distribution. If we scale each sample by $\frac{\ell}{a}$, we can simulate draws from the distribution

$$\mathcal{D} := (\mathbf{e}\|\mathbf{s}) + \mathcal{W}''$$

where $\mathcal{W}'' = \frac{\ell}{a}\mathcal{W}'$. However, we do not know the value of a for a given failing ciphertext. Instead, we will instead make use of the following heuristic. We know that $a \geq (t - e'')/\ell$. Thus, we fix $\alpha = (t - e'')/\ell$ for a, which introduces additional noise into the distribution of $\mathcal{W}''$. If we substitute α for the true mean of a in the calculation of the variance, we obtain

$$\mathbb{E}_{\chi \leftarrow \mathcal{N}^{\geq t/\ell}_{\sigma_{se}}}[((t - e'')/\ell - \chi)^2]$$

This quantity can be shown to be $\leq \sigma_{se}^2$ for any $(t - e'')/\ell \geq 0$. As such, we assume the additional noise is also a Gaussian with mean $\mathbf{0}$ and variance σ_{se}^2.

Thus, we approximate the total noise $\mathcal{W}''$ as a zero-centered multivariate Gaussian with a

coordinate-wise variance of at most

$$(\ell/a)^2 = (\ell^2/(t-e''))^2\sigma_{se}^2 = n^2\sigma_{se}^6/(t-e'')^2.$$

Recovering the secret can be performed by obtaining enough failing ciphertexts to estimate the mean of $\mathcal{D}$.

To estimate the mean, we can simply draw sufficient samples from $\mathcal{D}$, take their average, and round coordinate-wise to the nearest integer. After averaging m failing ciphertexts, we obtain a sample from the distribution

$$\mathcal{D}' := (\mathbf{e}\|\mathbf{s}) + \mathcal{W}''_{(m)},$$

where the error $\mathcal{W}''_{(m)}$ is again a Gaussian and the variance of each coordinate is at most $n^2\sigma_{se}^6/(t^2m)$. Note here that after averaging, the effects of individual e'' on $\mathcal{D}'$ can be ignored as e'' is zero-centered. For the key recovery to succeed, we require the magnitude of each error coordinate to be less than 0.5. Thus, for a given value of m, the success probability of the attack is

$$\begin{aligned} P(\mathsf{Success}) &= P(|w''_{(m)i}| < 0.5 \;\forall\; i) \\ &= P(|w''_{(m)i}| < 0.5)^n \\ &= (1 - 2 \cdot P(w''_{(m)i} < -0.5))^n \\ &= \left(-\mathtt{erf}\left(\frac{-0.5 \cdot t\sqrt{m}}{\sqrt{2} \cdot n\sigma_{se}^3}\right)\right)^n \end{aligned} \tag{3.3}$$

Given a target success probability, we then solve for m numerically. Note that it is certainly possible to compute a more accurate value for a numerically, and use it in the calculation of the mean of $\mathcal{D}'$. In a practical attack scenario, both $\alpha = (t-e'')/\ell$ and $\mathbb{E}[a]$ should be used to calculate the candidate secret, as the specific values of the true secret (e.g. if it has a higher than expected norm) can affect the number of failing ciphertexts required when using the heuristic value for a.

Using the heuristic for estimation of success probability is preferred, as it is more conservative (i.e. a smaller a results in a larger variance for $\mathcal{W}''$).

Comparison with the 'Hint' framework of [82] We note that decryption failure attacks can also be handled by the DBDD and 'hint' framework of [82], where a single failing ciphertext is viewed as a full dimensional approximate hint on the LWE secret/error. In general, integrating full dimensional approximate hints in the framework of [82] requires explicit inversion of matrices defined over the rationals, which is computationally prohibitive. Due to this, Dachman-Soled et al. provided a lightweight version of their framework, which allows one to estimate the concrete hardness of performing a lattice reduction-based attack in the decryption failure setting. However, the lightweight version is not suitable for our purposes here since we are interested in a full key recovery, not just hardness estimation. We attempted to use the full hints framework of [82] since the matrices that needed to be inverted were all diagonal. Therefore, integrating a single full dimensional approximate hint could be performed reasonably quickly. However, as discussed previously, our attack requires significantly more failing ciphertexts than a traditional decryption failure attack (on the order of 2^{17} failing ciphertexts see Figure 3.1). This additional factor resulted in the computational cost of using the full framework being untenable. An improved implementation, using parallelism or accelerators would be required to be competitive with our simple average-then-round approach.

3.3.3 Attack Modifications for Rowhammer Assisted Failures

We next explain why our Rowhammer-assisted attack can be viewed as a standard decryption failure attack with a lower threshold and one fewer dimension. More precisely, the exact Rowhammer pattern alters the effective decryption failure threshold. In our attack, one coordinate—$e_{i,j}$—in each column of $\mathbf{E}$ is increased by 256. To increase the chances of failure, we also search for 12s in the corresponding coordinates of $\mathbf{S}'$. This lowers the failure threshold for the remaining coordinates by $12 \cdot (256 + \bar{e}_{i,j})$, where $\bar{e}_{i,j}$ is the original, unhammered value of $e_{i,j}$. We denote this new, smaller threshold as $t' = t - 12 \cdot e_{i,j} = t - 12 \cdot (256 + \bar{e}_{i,j})$.

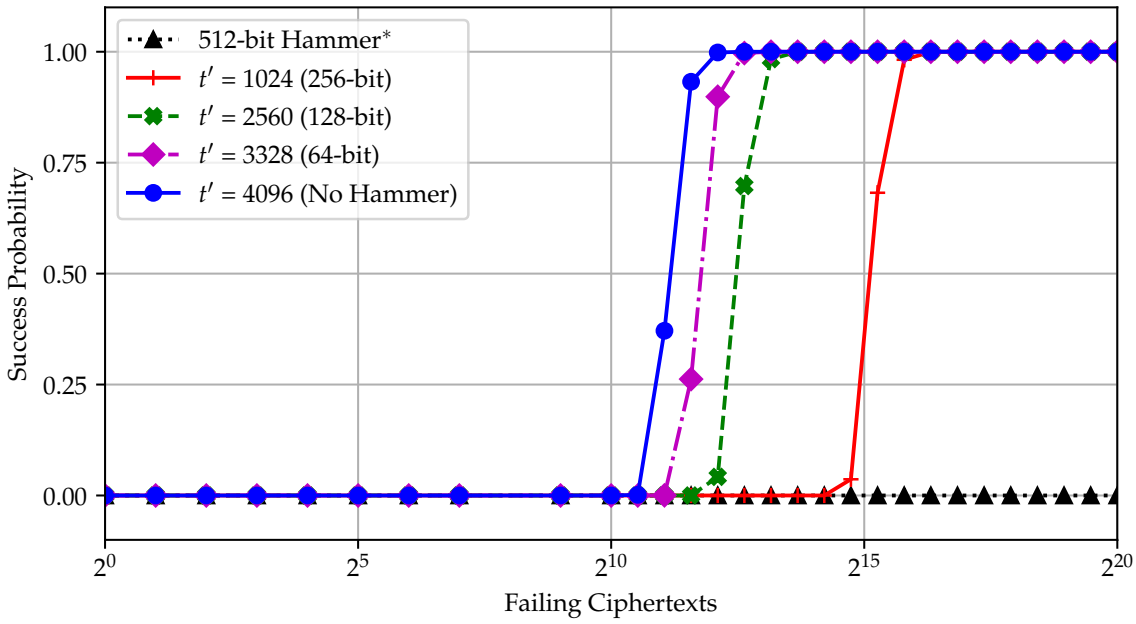


Figure 3.1: **Probability of recovering a single column of the Frodo-640 secret (Equation 3.3), for given numbers of failing ciphertexts.** Each line denotes a different (adjusted) threshold t' corresponding to rowhammer bit position.

Note that since we fix one coordinate of $\mathbf{s}'$, the distribution of that coordinate does not depend on the LWE secret as described in the decomposition given in 3.3.2. As such, we simply ignore that coordinate when performing the averaging, thereby reducing the dimension by one. To calculate the proper value for $t' - e''_{i,j}$, we must guess this value in advance for each column. As the error distribution for Frodo-640 has 25 possible values, we simply try all values for $\bar{e}_{i,j}$, and check our recovered key using the LWE equations. The same failing ciphertexts can be used for each candidate value of t', so only the averaging step (which is computationally trivial) needs to be run 25 times. See Figure 3.1 for the relationship between the failure threshold $t' - e''_{i,j}$, and the number of failing ciphertexts m required to mount our attack with various success probabilities. Approximately $100k$ failing ciphertexts are required to succeed with probability close to 1 for a 256-bit rowhammered column. Note that flipping the 512-bit of an entry in $\mathbf{E}$ results in a completely unrecoverable secret, as decryption failure is virtually guaranteed assuming the same filtering behavior.

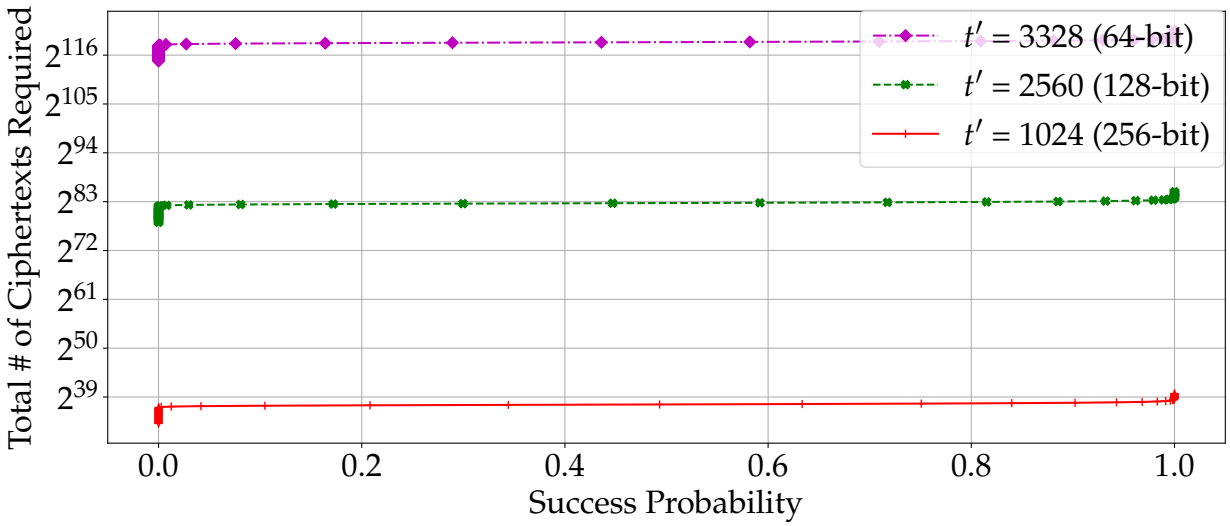


Figure 3.2: **Total number of ciphertexts during failure-boosting (not decryption queries) required to recover a single column of the Frodo-640 secret, for (adjusted) thresholds t' corresponding to rowhammer bit position.**

3.3.4 Total Attack Cost

We present the cost of the attack in the total number of ciphertexts (both succeeding and failing) needed to recover a single column of the secret key. We combine the prior analysis in 3.3.1 and 3.3.2 for 64, 128, and 256-bit rowhammers to produce the graph seen in Figure 3.2. From this, we can conclude that it is indeed best to target the 256-bit position with the rowhammer. The total number of ciphertexts we require to generate under failure-boosting to succeed with probability close to 1 is 2^{39} for the 256-bit rowhammer, compared to 2^{86} for the 128-bit rowhammer.

3.4 Poisoning Hobbits: Rowhammering a FrodoKEM Public Key

As outlined above, the main idea of our attack is to "poison" FrodoKEM's public key generation using Rowhammer. More specifically, we aim to flip specific bits inside the error term $\mathbf{E}$ computed during FrodoKEM's public key generation, obtaining a public key with a higher error. For FrodoKEM-640, the error term is an array of 640 x 8 values where each value is 2 bytes. Each of

the values in the error matrix $\mathbf{E}$ has about an equal likely chance to be either positive or negative. By adding values to known locations in each of the matrix's columns, an attacker can increase the key's decryption failure rate. Finally, using the knowledge of flipped bit positions, the attacker can feasibly find a large number of failing ciphertexts, thereby enabling cryptanalytic attacks on FrodoKEM and ultimately recovering the secret key.

A Balancing Act. While the above description is conceptually simple, we note that to extract the secret key in practice, a delicate balance of Rowhammer-induced bit flips is required to execute our attack. More specifically, too many high-order bit flips in the error term boosts the decryption failure rate to an excessive rate, causing failures to be prevalent for both the attacker and honest users. This will alert the target to the possibility that they are under attack, causing them to retire the poisoned key. On the other hand, too few bit flips will cause the decryption failure to be too low, giving the attacker only a slight advantage over honest decryption failures. This will result in an infeasible amount of computation to determine the key exchanged by the KEM.

3.4.1 Experimental Setup

Unless noted otherwise, we performed all of our experiments using a desktop containing a 3.4 GHz i7-4770 CPU and two Samsung DDR3 4 GiB 1333 MHz non-ECC DIMMs (part number M378B5273CH0-CH9). Out machine was running a fully updated Ubuntu 20.04. To simplify the attack, we disabled the machine's address-space layout randomization (ASLR). We note that ASLR was breached on numerous prior occasions [70, 109, 116, 253] making this assumption not overly-restrictive.

The victim process is the FrodoKEM reference code, compiled using the reference optimization level and Shake-128 flags. Additional code was added to FrodoKEM to output files for the public key, secret key, and error matrix. These files were used for confirming the results of the attack. We took care to add these additional lines of code only after the attack was complete, to not artificially increase the window of opportunity for the Rowhammer attack.

3.4.2 Determining Useful Bit Flip Locations

Prior to mounting a Rowhammer attack, we must first determine which bits inside FrodoKEM's error matrix $\mathbf{E}$ are useful for the attacker to flip. To that aim, we used code provided in the FrodoKEM submission package in order to examine how specific combinations of bit flips affect the decryption failure rate. As shown in Figure 3.1, we found that flipping bit 8, henceforth called a 256-bit flip because it adds $2^8 = 256$ to the value, in one value per column increased the decryption failure rate to about 2^{-27}, compared to a failure rate of $2^{-138.7}$ mentioned in Frodo-640's specification [23]. This achieves a nice balance of minimizing the number of required bit flips, while still raising the decryption failure rate enough to ensure the feasibility of our attack.

Given that there are 8 columns, the attack then requires a total of 8 256-bit flips in order to fully recover the secret key. Without all 8 flips, parts of the secret key must be recovered through a form of brute force search.

3.4.3 Profiling Memory

Prior to starting the Rowhammer attack itself, we must profile the machine's memory in order to locate the physical locations of pages containing bits susceptible to Rowhammer. We accomplished this using the memory massaging techniques described by [168]. More specifically, the method of [168] involves exploiting Linux's buddy allocator to obtain 2 MB of contiguous memory regions required for precise double-sided Rowhammering. Unfortunately, following the disclosure of [168], modern Linux versions restrict access to the `/proc/pagetypeinfo` file used by [168] to root users. While this does required us to use elevated privileges to run our attack, a concurrent work by [261] demonstrated how to perform the same attack on the allocator by using the world readable `/proc/buddyinfo` file, avoiding the use of root privileges.

After attacking the allocator to obtain tuples of memory locations for double-sided rowhammering, we iteratively hammer each pair of aggressor rows and then check the victim row for bit flips. Since bit flips are repeatable, we store the map of which bit flips were within each page so that we can select which pages are best to use for the attack.

Filtering Candidate Pages. For FrodoKEM-640, the error matrix has dimensions 640 x 8 of 2 bytes values, resulting in the matrix spanning 2.5 pages of size 4 KiB. Therefore, the error matrix can either be spread across 3 or 4 pages, depending on whether the array's page offset is past 0x800. Because of this, there must be multiple columns containing a 256-bit flip per page to satisfy all 8 required flips. Additionally, the first and last page storing the error matrix also store other nearby variables in the program; on our victim, this ended up being the S matrix and the array for storing randomness, respectively. While it may be possible to tolerate bit flips in S, we only considered pages containing no flips in these locations.

When choosing which 3 pages to use for our attack, we also must filter out both pages that have insufficient bit flips in the desired locations, and pages that have too many bit flips in undesired locations. Firstly, we filter out all sets of pages that don't provide exactly one 256-bit flip in at least 7 of the 8 columns. Otherwise, the Rowhammer attack will not increase the decryption failure rate enough for our attack to succeed. Equally important is that the pages do not exhibit bit flips during the attack in locations that will increase the decryption failure rate to be *too high*. For our attack, this means that any bit flips in bit positions 9 through 15, henceforth referred to as *high-order* bit flips, are not allowed. This is because increase in the error term will be very large in magnitude, and have an overly strong affect on the decryption failure rate. Additionally, only a small number of bits in positions 5 through 7 can be tolerated, since they will also have a substantial effect on the decryption failure rate, albeit less than the higher order bit flips. All other lower bits have an insignificant effect on the decryption failure rate.

Bit Suppression. With regards to requiring a bit to not flip, this can be accomplished by either choosing a page that doesn't contain a bit flip in that location, or by *suppressing* the undesired bit flip. As noted by [81], *stripe patterns*, where the bits above and below a given bit are the opposite value (i.e. 0-1-0 and 1-0-1 configurations), are the most likely to yield bit flips, while *uniform patterns* (i.e. 0-0-0 or 1-1-1) are unlikely to result in bit flips. Since bits can only ever flip in one direction, this means that we can use 1-1-1 patterns to suppress any 1-to-0 bit flip, and 0-0-0 patterns to suppress 0-to-1 bit flips.

By using the appropriate uniform patterns in the positions where the undesired bit flips are located, we significantly reduce the chance that those bit flips occur during the Rowhammer attack. We used this technique to suppress 117 bit flips in the chosen pages, leaving only 4 bits that could not be suppressed.

We note that not all undesired bit flips need to be suppressed, as there is a 50% chance that the randomly chosen value in $\mathbf{E}$ of the bit will already be the value that the bit can flip to. This means that for each unsuppressed bit flip, the probability of the attack succeeding decreases by a factor of 2. For example, if 3 bits cannot be suppressed, the attack succeeds 1 out of every 8 attempts.

3.4.4 Allocating Pages to Victim Process

After profiling the memory, the attacker must force the FrodoKEM victim to allocate its memory in a way that stores the secret term E in the attacker's chosen pages. This is accomplished through exploiting the Linux page frame cache in a technique termed "Frame Feng Shui" [168]. The Linux page frame cache stores frames in a first-in-last-out data structure, and returns the most recently deallocated page when receiving a request for a page frame. Therefore, if FrodoKEM allocates a predictable number of frames before the target $\mathbf{E}$, the target can be forced into a page of our choice.

First, multiple 4 KiB junk pages are allocated using mmap and the MAP-POPULATE flag to fill the bottom of the page frame cache. The number of junk pages is determined by how many pages the victim process allocates before allocating the target value. Next, the attacker chooses pages from the profiling phase that contain vulnerable bits in the correct positions. The attacker then deallocates the selected page frames using `munmap`, putting them at the top of the page frame stack. Immediately after this, the attacker unmaps all of the previously mapped junk pages, putting these pages on top of the selected pages in the stack. Finally, the the attacker induces the victim process to run its key generation process, during which the FrodoKEM process requests memory to store E, and the buddy allocator returns the pages chosen by the attacker. We used Frame Feng Shui against the FrodoKEM-640 scheme with a total of 297 junk pages to land the E error matrix into the selected pages.

3.4.5 Performance Degradation

Now that the attacker has forced the victim FrodoKEM to allocate the selected pages for the error matrix, and the spatial precision of the bits is sufficient, we must now ensure that the temporal precision of the bit flips is sufficient. That is, the attacker must flip the bits within a precise window during the execution of the FrodoKEM key generation. By examining the source of the reference code, we identified that the bit flips must occur after the generation of E by sampling the Gaussian in the `crypto_kem_keypair` function, and before E is added to AS in the `frodo_mul_add_as_plus_e` function. In our setup, this window takes roughly 8ms, whereas we empirically found that we required 1300 ms to reliably succeed with the Rowhammer attack. To bridge this gap, we conducted a *performance degradation* attack against FrodoKEM.

A performance degradation attack [24] functions by rapidly forcing the most frequently executed FrodoKEM instructions to be evicted from the cache, which causes the CPU's pipeline to be flushed in a machine clear upon detecting the invalid tag in the L1i cache. This can result in dramatic decreases in performance if it occurs frequently enough. To evict the cacheline, we used the `CLFLUSH` instruction, available on x86 machines to all unprivileged users, to completely evict cachelines from the cache hierarchy.

Choosing a Cacheline. We statically analyzed the FrodoKEM victim's binary to find the most frequently executed cache line of code, and found that the usage of Shake-128 involves a tight loop that executes the `store64` instruction, resulting in 880992 calls to `store64` within the execution window.

This makes `store64` the optimal choice for performance degradation, so we implemented our performance degradation attack by running FrodoKEM-640 on a single core, with performance degraders running on both the sibling core and 2 other virtual cores, including the core running FrodoKEM-640. Without any performance degradation, the time hammering window is 8.2 ms. With performance degradation on only the sibling core, this can be increased to around 663 ms. With additional cores running the degrade code, this reached about 1300 ms. Using the performance degradation code with multiple cores, this allowed for sufficient time to complete the Rowhammer

attack within the alotted window.

3.4.6 Results

For the attack against FrodoKEM-640, the page offset of the error matrix $\mathbf{E}$ was 0x9b0, meaning that it spanned a total of 4 pages. To prevent hammering any of the shared values on the first page, only the last 3 pages were targeted for the rowhammer attack. Using the criteria listed above, we found 3 pages that satisfy 7 of the 8 256-bit flips. Therefore, additional steps are required at the end for full key recovery. Also, bits in 4 matrix locations were found that could not be suppressed. Thus, accounting for the 7 256-bit locations that must be positive for the flip and the 4 locations that could not be suppressed, the attack succeeded 1 out of every 2^{11} attempts.

3.5 Searching for Failing Ciphertexts with a Supercomputer

In the construction of the poisoned public key, we intentionally keep the failure rate for honestly generated ciphertexts low so that the attack may persist for the duration of a generated key without the victim identifying the issue. This has the effect that finding a failing honestly generated ciphertext is still exceedingly rare. In the key established in Section 3.4, the failure rate of honestly generated ciphertexts is approximately 1:2,700,000. In order to generate approximately 100k failing ciphertexts per column, that would necessitate the creation of over 1 trillion connection requests. Alternatively, the adversary could generate ciphertexts that are likely to fail, but that would fail the Fujisaki-Okamoto transform.

To meet these constraints, our attack is predicated on a setup phase to generate honest ciphertexts that are *likely to fail* based on the profiled machine and the targeted columns. We use the $\mathbf{A}$ matrix from the public key, and then iterate on the random seed that initializes μ as input to the Frodo.Encode function. Ordinarily, μ is initialized by a randomly sampled 16-byte seed, and thus there are 2^{128} potential seeds. The encode function continues as specified until the $\mathbf{S}'$ matrix is generated. The superset of the rows and columns targeted by Rowhammer are checked to determine

if they have high values (i.e. -12 or +12) that would make them candidates to potentially fail to decode. These candidate ciphertexts are saved to attempt decoding. For the poisoned key established in section 4, otherwise honestly generated ciphertexts (i.e. those that pass the FO transform) pass this filter at a ratio of approximately 1:1175.

For our proof-of-concept attack, all candidate ciphertexts are passed as input to the Frodo.Decode function. Ciphertexts that fail are saved to a file including the μ seed to generate the ciphertext, the $\mathbf{S}'$, $\mathbf{E}'$ and $\mathbf{E}''$ matrices, and the ciphertext itself. The attacker would have access to all of these aspects of the ciphertext in a distributed attack. The components are concatenated to files named for the row/column pair where the high value in the $\mathbf{S}'$ matrix is located (e.g. 'mu-4-8.csv'). This helps to identify the likely column that created the failing condition, which aids the key-recovery process. So, we only considered matrices that had a single ± 12 in the checked locations.

To expedite the profiling stage we leveraged resources from two supercomputer clusters — namely Pittsburgh Supercomputing Center (PSC) [64] and Open Science Grid (OSG) [221, 245]. Total ciphertext generation took 237,806 hours, distributed as follows: 104,856 core-hours on OSG through the Open Science Pool, and 132,950 core-hours on the Bridges-2 Regular Memory nodes from PSC. For PSC, the modified Frodo-KEM encode and decode functions were compiled directly on each compute node. For OSG, the modified code was compiled in a container for each compute node. Jobs were launched with an incrementing seed to generate 150 million ciphertexts per seed. The start seed was shifted 28 bits to the left and stored in μ as the start of the search. Each job produced approximately 128 thousand ciphertexts that passed the filter, from which approximately 20 ciphertexts failed to decrypt. In total 665,343 failing ciphertexts were used to recover 7 of the 8 columns, necessitating 1.53 billion decryption requests—less than 10% of the absolute maximum 2^{64} decryption queries identified by NIST to prevent failure attacks.

3.6 Completing the Attack: Session-Key Recovery

If the full master secret key is unable to be recovered, it is still possible to recover encapsulated session keys from honestly generated ciphertexts. The recovery procedure is enabled by the limited number of bits present in the message μ. Due to the Fujisaki-Okamoto transform, encapsulation is deterministic given the value of μ. If we can determine the value of μ, we can easily recover the session key.

In FrodoKEM.Decaps [23], μ is decoded from an 8x8 message matrix $\mathbf{M}$, which is computed from an honestly generated ciphertext and the secret key. As we are unable to compute $\mathbf{M}$ due to our lack of the full secret key, instead we compute $\mathbf{M}'$ by filling in the missing columns of the secret key with 0's in the computation of $\mathbf{M}$. Note that $\mathbf{M}' = \mathbf{M}$ except in the missing columns. For Frodo-640, the most significant 2 bits of each entry of $\mathbf{M}$ are extracted to produce μ in FrodoKEM.Decode (resulting in a message size of 128 bits). Thus, for every missing column of the secret key, we must brute-force 16 bits of μ. To check correctness, we simply pass each μ' to a modified version of FrodoKEM.Encaps that accepts μ as a parameter rather than sampling it at random. The deterministic nature of the encryption ensures that the output ciphertext, will be equal to the intercepted ciphertext from the victim when we pass in the correct value for μ.

The total cost of the session key recovery is $2^{16 \cdot \mathsf{missingcols}}$ times the cost of FrodoKEM.Encaps. In our attack, we managed to recover 7/8 of the columns of the secret key, and only needed to brute-force 16 bits. This procedure takes at most a couple of minutes on a commodity laptop. In a scenario where super computers are enlisted for this brute-force search in the online phase of the attack, more missing columns could be tolerated at the discretion of the attacker. Sessions could be captured in the meantime for later decryption once the brute-force completes.

Algorithm 7 SessionKeyBF($\mathbf{SExp}, pk, col, ct = (ct_1, ct_2)$)

$\mathbf{C_1}$:= FrodoKEM.unpack(ct_1)
$\mathbf{C_2}$:= FrodoKEM.unpack(ct_2)
Compute $\mathbf{M'} = \mathbf{C_2} - \mathbf{C_1} \cdot \mathbf{SExp}$
μ' := FrodoKEM.Decode($\mathbf{M'}$)
for $i := 0 \rightarrow 2^{16}$ **do**
    Insert(μ', col, i) ▷ Insert i into col's bit positions in μ'.
    ct', ss := FrodoKEM.Encaps(pk, μ')
    if $ct' = ct$ **then return** ss
return `FAIL`

Figure 3.3: **Session key brute-force algorithm given the experimentally derived secret and missing column index.**

3.7 Attacking Other NIST Lattice KEMs

While we only experimentally demonstrated an attack against FrodoKEM, we believe that similar attacks are theoretically possible on other latice-based KEMs; however, there are various reasons these attacks may be more difficult in practice.

In this section we will look at the other lattice KEMs in the NIST PQC Round 3 candidate pool: Kyber, Saber, NTRU, sNTRUprime, and NTRU-LPRime. In all these cases, the strategy is similar: We note that decapsulation failures occur with high probability when the product between a noise vector chosen during key generation and a second noise vector chosen during encapsulation is large. The strategy is then to:

1. Introduce known additional noise during key generation using Rowhammer techniques, producing a ”poisoned key”

2. Use knowledge of the additional noise to select valid ciphertexts that are likely (but not too likely) to produce a decapsulation failure when the honest party attempts to decrypt them using the ”poisoned” key.

3. Accumulate information about the unknown parts of the private key by observing which of the ciphertexts are and are not successfully decapsulated.

In order to accomplish step 1, it is likely that one would need to do performance degradation similar to what was done to FrodoKEM in our experiments. For Kyber and Saber, we identify places in the code where the SHAKE function is used between the time when noise is sampled and when it is used. Cryptographic random number generation is also done in sNTRUprime and NTRU-LPRime in a location that appears similarly useful. However, the reference implementation of each uses AES instead of SHAKE as the core primitive. Since AES has native hardware support, it is likely harder to do performance degradation against the reference implementation of NTRUprime. If the cryptographic random number generation could be performance degraded, sNTRUprime seems like a better candidate for attack than NTRU-LPRime, because in the latter case, the random number generator is called only once, while in the former case it is called repeatedly.

An additional difficulty arises when trying to adapt the attack to NTRU. In that case, the fact that algebraic operations switch between the rings $\mathbb{Z}_q[x]/(x^n - 1)$ and $\mathbb{Z}_q[x]/((x^n - 1)/(x - 1))$ means that any rowhammering which fails to preserve the sum mod q of the coefficients of a polynomial will cause decryption to fail with overwhelming probability for all ciphertexts. This makes it very likely that any attack will be detected, and means that decryption failures will give no information about the private key. It is unclear how to design an attack which avoids this issue.

With regard to step 2, the filtering step, there is an additional quantitatve challenge compared to our attack on FrodoKEM. In the case of FrodoKEM, we were able to Rowhammer a single bit per column of $\mathbf{E}$. This was possible because the parameters of the attack could be set so that a decryption failure only occurred with significant probability when one of 56 locations in the ciphertext noise attained a maximal value. Since this only happened with low probability for an honestly generated ciphertext, filtering was possible. In all the other KEMs, each position in the key generation noise multiplies more positions in the ciphertext noise, and the ciphertext noise distribution has the maximum value occur much more frequently. As a result we expect that attacking any of the other schemes will require a somewhat higher density of rowhammerable bits. The filtering criterion in this case would require the products of several coefficients of the ciphertext noise with the rowhammered coefficients to be large and to have the same sign. We give more details

on the methods that would need to be employed to attack the other schemes in the full paper [95].

3.8 Possible Countermeasures

As described, our attack shows that Frodo is vulnerable against a real-world Rowhammer attack. This justifies further study of the Rowhammer security of any post-quantum protocols that will come out of the NIST standardization process. As such countermeasures are critical, we wish to highlight a few possible ones up front.

3.8.1 Rowhammer Defenses

There is a rich literature surrounding Rowhammer defenses, both proposed and deployed, that attempt to mitigate the Rowhammer bug. Despite these efforts, researchers have managed to circumvent all practically deployed mitigations and empirically demonstrate bit flips against them. Even so, a combination of such countermeasures to provide "defense in depth" may dramatically increase the effort required to mount a Rowhammer attack.

Hardware Defenses. The two hardware defenses deployed in practice are Error Correcting Code (ECC) memory and the Targeted Row Refresh (TRR) mitigation. ECC memory, most commonly found on server machines, aims to correct bit flips when they are detected upon reading from memory. The memory controller stores additional "control bits" that act as a checksum, enabling bit correction and detection up to a certain threshold of errors. While the ECC mechanism was designed to mitigate errors due to cosmic rays, it also inadvertently assists in mitigating Rowhammer bit flips. [81] defeated ECC with Rowhammer, however, by causing a sufficient number of bit flips within an ECC word such that the mechanism can no longer even detect that a bit flip has occurred.

TRR is a hardware defense implemented in DDR4 that was long touted to be a panacea for Rowhammer. It uses counters to keep track of how many times each row in memory is accessed, and once the counter reaches a specified threshold, TRR automatically refreshes the nearby rows. If this threshold is below the minimum number of hammering attempts required for a Rowhammer

attack, then this mitigation refreshes victim bits before they are hammered enough to flip. While this seems to be a perfect defense in theory, [99] were able to bypass TRR and flip bits on DDR4 due to limitations imposed by constraints in the hardware. In particular, TRR can only track accesses to a small number of rows at a time, and attackers can bypass it by hammering many rows at once.

Most recently, [181] addressed the shortcomings of TRR by proposing a scheme that tracks rows optimally, given a limited number of counters and additional refresh commands. While they demonstrate that their algorithm achieves the best trade-off between these parameters, it still suffers from the inherent limit to how many row counters can be supported by the hardware.

Software Defenses. While no software level defenses against Rowhammer have seen widespread adoption, researchers have proposed mitigations with claims of low overhead and complete Rowhammer protection. [33] aim to backport the same principles behind TRR to DDR3. They use performance counters to determine which rows are accessed most frequently, and then refresh those rows before a bit flip occurs.

[60] aim to protect the kernel from user level Rowhammer attacks by physically separating the kernel memory on the DIMM from user memory. [166] go a step further and physically isolate all data rows in memory from each other.

3.8.2 Defenses Specific to our Attack

We also present practical countermeasures derived from unique insights gleaned from our end-to-end attack against FrodoKEM.

Hardware Accelerated Cryptography. As demonstrated in Section 3.4, having a sufficiently long time window to rowhammer is critical to the success of the attack. To extend the length of this window all the way from 8ms to 1300ms, we relied on a performance degradation attack against the SHAKE hash function, wherein we rapidly flushed a cacheline containing frequently executed FrodoKEM code. The effectiveness of the performance degradation is highly dependent upon how many calls FrodoKEM makes to the flushed cacheline – the one containing `store64` in this case.

If, however, FrodoKEM were to use hardware accelerated cryptographic instructions in place

of its in-software hash function implementation, the opportunity for the attacker to conduct a performance degradation attack would be greatly diminished. This is because the tight inner loop that calls `store64` within Shake would instead be replaced, for example, by just a few instructions from Intel's AES-New Instructions (AES-NI) instruction set. We thus observe that hardware accelerated cryptographic functions seem to be more resilient against performance degradation attacks.

Algorithmic Defenses. In addition, there are two potential algorithmic defenses to our attack. The first is to limit the surface of the performance degradation through proper sequencing of operations, even when not using AES-NI. The main idea is to not perform any expensive operations between the time when $\mathbf{S}$ and $\mathbf{E}$ are sampled and when $\mathbf{B}$ is generated. For example, one could use SHAKE to expand out $\mathbf{A}$ before $\mathbf{S}$ and $\mathbf{E}$ are generated (like in Kyber and Saber) and move the SHAKE calls used to generate the randomness for $\mathbf{S}$ and $\mathbf{E}$ before the actual sampling (like in FrodoKem and Saber). Ordering the operations in this manner significantly reduces the window for performance degradation, and therefore our attack.

The second defense is to guard the key generation process. One way to do this is to regenerate $\mathbf{A}$, $\mathbf{S}$, and $\mathbf{E}$ from randomness and compute $\mathbf{B}$ again. If the two $\mathbf{B}$ are not equivalent, then abort key generation. For the Rowhammer attack to pass this check, the entire attack must succeed twice in a row, for potentially different pages of memory. In addition, to ensure the equality of the $\mathbf{B}$ matrices, the Rowhammer attack must flip *exactly* the same bits each time. It is highly unlikely, however, that the attacker would be able to find another set of pages that also exhibits the exact same bit flips under hammering as the original set.

Storing the error matrix E (Frodo overwrites E with B for efficiency) for the duration of the computation of the public key is another possible way to enforce the integrity of key generation. Then, if any value in E (or perhaps its distribution) is abnormal, recompute the public key from scratch. This has the downside of potentially alerting the attacker to the re-keying process, and enabling them to try to rowhammer the new key.

Active Defenses. Another line of defense would be to maintain an active state during the online

phase to detect an attack in progress. An adversary needs to send a vast number of filtered ciphertexts to mount a successful key recovery attempt. A potential victim could check the distribution of received ciphertexts during the decapsulation process. Indeed, during the re-encryption check, the victim would be able to see the values of the randomness used during encapsulation (at least where decapsulation would ordinarily be successful). If the distribution of incoming ciphertexts is skewed towards large values (e.g. ± 12), then it is reasonable to assume that the victim is under attack. One must be careful in how to respond to this attack, however. If the decision is to simply re-key when an attack is detected, this leads to an easy Denial of Service attack (an attacker can perform the same filtering procedure to intentionally trigger the detector). On the other hand, if the response is to discard the received ciphertexts that contain such large values then this can lead to discarding communication from honest users.

Also, while this countermeasure would make our attack harder, it would not make it very much harder – if the intensity of the rowhammering is modestly increased, the attacker can make the decryption failure on random ciphertexts sufficiently high as to not require ciphertexts to come from an unusual distribution. This would, however, result in a significantly higher decryption failure rate for honest parties, making the attack easier to detect.

CHAPTER 4

Checking Passwords on Leaky Computers: A Side Channel Analysis of Chrome's Password Leak Detection Protocol

The scale and frequency of password database compromises has led to widespread and persistent credential stuffing attacks, in which attackers attempt to use credentials leaked from one service to compromise accounts with other services. In response, browser vendors have integrated password leakage detection tools, which automatically check the user's credentials against a list of compromised accounts upon each login, warning the user to change their password if a match is found. In particular, Google Chrome uses a centralized leakage detection service designed by Thomas et al. (USENIX Security '19) that aims to both preserve the user's privacy and hide the server's list of compromised credentials.

In this paper, we show that Chrome's implementation of this protocol is vulnerable to several microarchitectural side-channel attacks that violate its security properties. Specifically, we demonstrate attacks against Chrome's use of the memory-hard hash function *scrypt*, its hash-to-elliptic curve function, and its modular inversion algorithm. While prior work discussed the theoretical possibility of side-channel attacks on *scrypt*, we develop new techniques that enable this attack in practice, allowing an attacker to recover the user's password with a single guess when using a dictionary attack. For modular inversion, we present a novel cryptanalysis of the Binary Extended Euclidian Algorithm (BEEA) that extracts its inputs given a single, noisy trace, thereby allowing a

malicious server to learn information about a client's password.

4.1 Introduction

The past few decades have witnessed a drastic increase in the amount of usernames and passwords leaked via various data breaches. This in turn led to an increase of credential stuffing attacks, where attackers try using leaked credentials from one service to breach accounts on other services. Prior works have demonstrated that even post compromise, 6.9% of credentials remain valid due to reuse, often for years [258].

However, the wide availability of datasets of breached credentials also has the potential to enable browsers and password managers to actively alert users when their specific credentials are present in the dataset, protecting their account from the risk of compromise. Indeed, most browsers have launched some type of password alerting service, automatically checking all credentials entered for prior vulnerabilities.

Their inclusion in a browser's default configuration, however, raises significant privacy concerns from users, as passwords have to be shared with a credential checking service. This prompted Google to incorporate a Private Set Intersection (PSI) protocol as part of Chrome's Password Leak Detection mechanisms [258], removing the need to share users' passwords or the server's list of compromised credentials.

Another emerging threat to modern systems is the risk of side-channel attacks. With a plethora of microarchitectural attacks on cryptographic implementations, both from native code [14, 20, 79, 95, 164, 168, 191, 210, 267, 269, 291], and from the browser [17, 98, 105, 117, 146, 249, 272], it is imperative that cryptographic protocols use side-channel hardened implementations when processing sensitive information. With Google's Password Leak Detection protocol [258] using highly customized cryptographic implementations, in this paper we ask:

Is Chrome's Password Leak Detection protocol vulnerable to side-channels? If so, how can attackers exploit the Password Leak Detection protocol to recover the users' passwords?

4.1.1 Our Contribution

We analyze Google's Password Leak Detection protocol, enabled by default in Chrome version 106 (latest at the time of writing), and find that Chrome's Password Leak Detection protocol leaks information via microarchitectural side-channels. In particular, by monitoring the cache access patterns while running this protocol, we are able to reduce the complexity of brute-forcing user credentials such that the attacker often succeeds on the very first login attempt. Since Google's Password Leak Detection protocol is active on default in nearly all modern versions of the Chrome browser, our attack is applicable to nearly every login attempt on the targeted machine.

In our analysis of Chrome's Password Leak Detection service, we found that the client side of the protocol contains three different components that are vulnerable to side-channel attacks, all leaking independently from one another: (i) using *scrypt* on users' input credentials (ii) using *hash2curve* on the output of *scrypt*, and (iii) using BEEA to compute the modular inverse of the value used to blind the output of *hash2curve*. The results of these three attacks are summarized in Table 4.1.

Section	Component	Attacker Capabilities	Average Entropy Reduction (bits)	Required # of Traces
4.4	Scrypt	Native Code	23.41	5
4.5	Hash-to-Curve	Browser Code	0.24	5
4.6	BEEA	Native Code + Blinded Hash	Entire Hash[1]	1

Table 4.1: **Summary of attacker capabilities and results.** All three attacks are mutually independent, but can be combined for greater information disclosure. The entropy reduction assumes a dictionary attack from the popular "rockyou.txt" password list, which contains 14,341,564 passwords, or 23.77 bits of entropy. The average is calculated from the experimental outcomes weighted by their probabilities, assuming that the attacker observes the number of login attempts indicated in the final column.

Attacking Memory Hard Hash Functions. In order to prevent attackers from extracting breached credentials from Google's servers by repeatedly querying the Password Leak Detection protocol, Google requires that all queries hash the checked credentials using a memory hard hash function [215]. This has the effect of slowing down attackers interested in extracting the server's credential list via dictionary attacks, as the computation of the hash digest for each dictionary entry requires a large amount of memory.

Unfortunately, Chrome uses *scrypt* to hash the user's credentials, a design that is inherently non

constant time. By observing *scrypt*'s input-dependent memory access patterns using Prime+Probe, in Section 4.4 we demonstrate how attackers can significantly reduce the cost to brute-force the target's credentials. While prior works have alerted to potential side channel issues with *scrypt*'s design [27, 97], these approaches fail in practice due to limitations in the bandwidth and accuracy of the current state-of-the-art side-channel techniques. As such, we develop novel techniques that account for noisy signals with a highly restricted view of the victim's memory access patterns, resulting in the first end-to-end cache attack against the *scrypt* algorithm.

Attacking Hash2Curve. In addition to memory hard hashing, Google's Password Leak Detection protocol also requires computing a Private Set Intersection (PSI) between the client's credentials and the server's list of compromised accounts. To that end, Google's uses a hash to curve algorithm, converting the output of *scrypt* into points on an elliptic curve. Google then computes the intersection in a homomorphic manner, avoiding the need to share user's passwords or the server's list of compromised credentials.

The *hash2curve* algorithm, however, uses rejection sampling [55], which is non-constant time and is explicitly discouraged [133, Appx. A] due to side-channel concerns. In Section 4.5 we empirically demonstrate the implications of side-channel leakage due to rejection sampling by presenting attacks on Chrome's hash to curve implementation, using both native and browser-based cache attacks. We also demonstrate that browser-based cache-attacks are still possible on the latest version of Chrome, despite multiple attempts at hardening.

Attacking Modular Inversion. As a final contribution, we analyze Chrome's modular inversion operation used during the blinding of the hash digest of the client's credentials. In Section 4.6, we attack the Binary Extended Euclidian Algorithm (BEEA) to show how a malicious password server can obtain a digest of the client's credentials using just a single side channel trace. To accomplish this, we developed a novel cryptanalysis of BEEA that allows an attacker to completely recover the inputs, given only a single, noisy trace.

This directly violates the security guarantees of Chrome's Password Leak Detection, which aims to let clients query the server without leaking information on their passwords. Using our techniques,

the server can recover a hash of the client's credentials, thereby enabling an offline dictionary attack.

Summary of Contributions: In summary, this work makes the following contributions.

- We present the first side channel analysis of Chrome's Password Leak Detection protocol.
- We empirically demonstrate the first end-to-end cache attack against Chrome's usage of the *scrypt*, allowing us to practically brute force the client's credentials (Section 4.4).
- We present an attack on Chrome's *hash2curve*, using both native code Flush+Reload and browser-based Prime+Probe (Section 4.5).
- We present a novel cryptanalysis of the Binary Extended Euclidian Algorithm (BEEA) that recovers inputs using only a single noisy trace, and show that its usage in Chrome leaks information about the client's password, allowing malicious servers to potentially breach the client's credentials. (Section 4.6).

4.1.2 Responsible Disclosure

We disclosed the vulnerabilities described in this paper to Google through a Crbug report, and shared our paper with a number of their engineers. We were able to contact the team handling the backend of Chrome's Password Leak Detection service, and suggested and discussed potential mitigations. Google stated that they intend to switch to a variant of Argon2 hash function to mitigate our attack from Section 4.4, and will use a constant-time *hash2curve* algorithm from [133] to mitigate our attack from Section 4.5. At the moment, however, Google's protocol remains unchanged, as Google believes that given the cost to mount our dictionary attack and the scope of the threat model, the risk to users is minimal.

However, in response to our paper, Chrome mitigated our attack on BEEA from Section 4.6 through computing the modular inverse by exponentiating by $p - 2$, thereby removing BEEA entirely from Password Leak Detection.

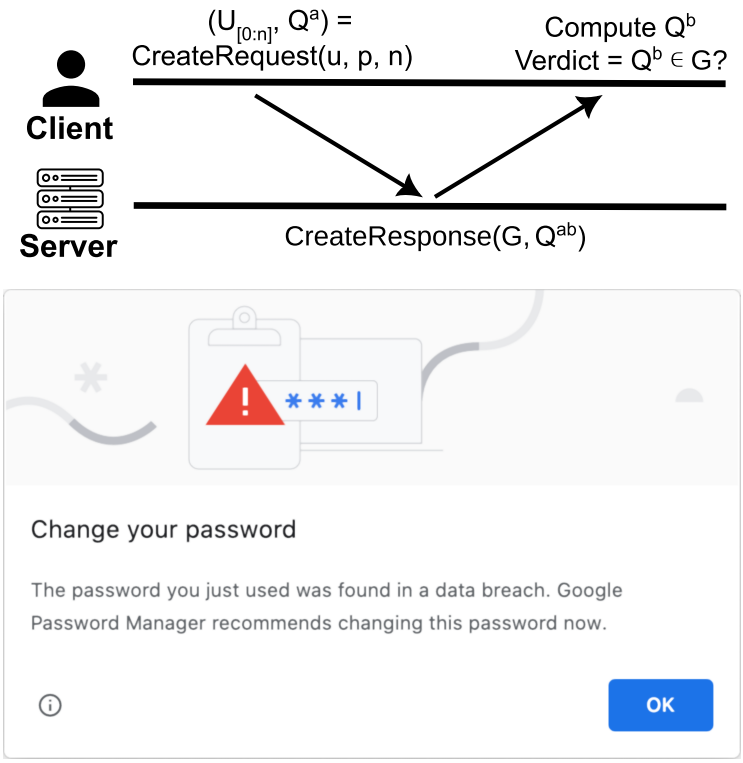


Figure 4.1: **(Top) Overview of steps in Google's Password Leak Detection protocol. (Bottom) User interface displayed by Chrome if the verdict is true.**

4.2 Background

4.2.1 Chrome's Password Leak Detection

In order to perform privacy-preserving password checking in the Chrome browser, Google uses a custom protocol called Password Leak Detection [258]. More specifically, Chrome's Password Leak Detection combines anonymity sets, memory-hard hashing, and Private Set Intersection (PSI) [139] in order to check if the given username and password pair is present in a data set of compromised credentials, all while preserving privacy against both malicious clients and malicious servers. In this section we describe the Password Leak Detection protocol as implemented in Chrome 106, which differs slighty from the description given in [258].

Protocol Overview. Figure 4.1 presents a high level overview of Google's Password Leak Detection protocol. The protocol consists of four steps, which we now outline:

CreateDatabase. Before the Password Leak Detection server can handle any requests, it must

first hash, blind, and partition its database of leaked credentials. Given the set of credentials, $S = \{(u_0, p_0), (u_1, p_1), ..., (u_\ell, p_\ell)\}$, the server first hashes each username:password pair and partitions the database on the n-bit prefix of the hashes of the usernames by computing:

$$S' = \{(H(u_i)_{[0:n]},\ H(u_i, p_i))\ :\ (u_i, p_i) \in S\}$$

where $H()$ is the same hash2curve algorithm the client used to compute Q. Then, the server generates its own secret key for blinding, b, and blinds the database by computing:

$$S'' = \{(U_{i[0:n]},\ H_i^b)\ :\ (U_{i[0:n]}, H_i) \in S'\}.$$

CreateRequest. For each request, the client hashes their username and password pair to an elliptic curve point Q using a hash2curve algorithm, and then blinds Q with a secret key a by computing Q^a. The client then constructs request $= (U'_{[0:n]}, Q^a)$, where $U'_{[0:n]}$ is the prefix of a hash of the username. Finally, the client sends request to Google's Password Leak Detection server.

CreateResponse. The Password Leak Detection server responds to the request by blinding Q^a with b to generate Q^{ab}. The server then computes G, the set of blinded credentials with username hash prefixes equal to the request's as:

$$G = \{H_i^b\ :\ (U_{i[0:n]}, H_i^b) \in S'' \text{ and } U_{i[0:n]} = U'_{[0:n]}\}.$$

The server then returns the tuple response $= (Q^{ab},\ G)$.

Verdict. Upon receiving the response, the client uses Diffie-Hellman private set intersection [139] to determine whether its credentials have been leaked. The client calculates the modular inverse of its secret key a, uses it to unblind the doubly blinded hash by computing $(Q^{ab})^{a^{-1}} = Q^b$, and finally checks if $Q^b \in G$, indicating compromise.

Our focus is on analyzing the side channel leakage from the CreateRequest phase; we now describe this phase in greater detail. See [258] for the complete protocol details.

CreateRequest in Detail. Algorithm 8 outlines the CreateRequest phase. The client begins by generating a random nonce a (Line 2), then creates the string s by canonicalizing the username u and appending the password p to the canonicalized username. More specifically, for a username password pair $(u, p) = (\text{“}user@gmail.com\text{”}, secret)$, the canonicalized username and concatenated password is $s = \text{“}usersecret\text{”}$.

The client then passes the canonicalized tuple as input to *scrypt*, a memory-hard hashing algorithm, to compute $H \leftarrow scrypt(s)$. This digest is input to Chrome's *hash2curve* algorithm, producing a point $Q \leftarrow$ *hash2curve*(H) on the NIST P-256 elliptic curve. Next, the client blinds the hashed point by computing Q^a, where we use multiplication to denote the elliptic curve group operation. Finally, the client computes $\mathsf{request} = (\mathsf{SHA256}(u)_{[0:n]}, Q^a)$ where $\mathsf{SHA256}(u)_{[0:n]}$ denotes the n least significant bits of the SHA256 hash of u.

Comparison With the Original Protocol. [258] evaluated a slightly different version of this protocol, where the client computes $\mathsf{request} = (H_{[0:n]}, Q^a)$, with $H_{[0:n]}$ being the digest resulting from computing *scrypt* over both the client's username *and* password; this has the downside that it leaks information about the user's password. To address this, the authors also propose a *Zero-Password Leakage Variant* [258] that is similar to Algorithm 8 except that it uses a memory-hard hash function for computing both U and H (lines 4–5, Alg. 8); the authors cite the cost of a second memory-hard hashing step as a disadvantage. Algorithm 8 which is deployed in modern versions of Chrome, *does not* use a memory-hard hash function to compute U, thus avoiding the downside of the original Zero-Password Leakage Variant.

4.2.2 Cache Attacks

A large body of literature examines how two processes can inadvertently reveal sensitive information to each other due to them sharing the same memory cache [90, 118, 122, 210, 291]. These works show how a *victim process*'s memory accesses influence the state of the cache, and that an attacking process, called the *spy process*, can deduce what the victim accessed by indirectly measuring the state of the cache. The three cache attacks relevant to this paper are described below.

Algorithm 8. CreateRequest: The client uses this function to hash and blind the username:password pair to send to the server. The deployed version in Chrome uses this algorithm with the prefix length of the username set to $n = 26$.

function CREATEREQUEST(u, p, n)
  $a \leftarrow \mathsf{RAND}()$
  $u' \leftarrow \mathsf{CANONICALIZE}(u)$
  $U \leftarrow \mathsf{SHA256}(u')$
  $H \leftarrow \mathsf{scrypt}(u', p)$
  $Q \leftarrow \mathit{hash2curve}(H)$
  $Q^a \leftarrow \mathsf{BLIND}(Q, a)$
  $U_{[0:n]} \leftarrow \mathsf{BYTESUBSTRING}(U, n)$
  return $(U_{[0:n]}, Q^a)$

Prime+Probe [210]. This attack only requires that the spy and victim share some level of the cache hierarchy. To carry out this attack, the spy first builds an eviction set, which is a set of addresses that are "congruent" (i.e. mapped to the same cache set) with the targeted cache line.

The attacker then *primes* the cache set by accessing each address in the eviction set, thereby bringing them into the cache. To determine if the victim accessed memory that is congruent with the targeted cache set, the spy process *probes* the cache set by timing accesses to each element in the eviction set; if any access takes longer than an L3 hit, the attacker infers that the victim brought a cache line into the probed cache set and evicted an element in the eviction set, thereby revealing the victim's access to the targeted cache set.

Flush+Reload [291]. This attack has stricter requirements than Prime+Probe, as it needs the spy and the victim to share memory (e.g., the spy and the victim access a de-duplicated library). In this attack, the spy process prepares the cache by flushing the targeted cache-line before the victim performs some action, and then reloads it after the victim finishes. By measuring the reload speed, the spy learns whether the victim brought the targeted cache-line into the cache. Compared to Prime+Probe, Flush+Reload samples more quickly, and works with cache-line, rather than cache-set, granularity.

Flush+Flush [118]. This attack is similar to Flush+Reload, with the difference that instead of reloading the targeted cache-line, the spy simply flushes it again and measures the execution time of the clflush instruction; a longer time indicates that the cache-line was present in the cache. While

Flush+Flush samples at a rate nearly 7 times faster [118] than Flush+Reload, it suffers from a lower accuracy [86].

4.3 Threat Model

In our analysis, we uncovered three separate components within Chrome's Password Leak Detection that leave the client vulnerable to three separate attacks. Each attack reveals information about the client's credentials, and the attacks can all be launched independently from one another. As such, each attack assumes a different threat model, with differing preconditions, and extracts different amounts of information from the victim. See Table 4.1.

Attack on *scrypt*. Following the standard threat model for microarchitectural attacks, in Section 4.4 we assume that the attacker can execute native code under the context of an unprivileged user process on the client's machine. Furthermore, we assume that the victim is submitting his credentials while logging into a website on a completely unmodified Chrome browser. The attacker then uses side-channels to extract information on the victim's execution of *scrypt* that will reduce the attacker's search space when conducting a dictionary attack on the victim's input credentials.

Attack on *hash2curve*. For our attack on *hash2curve* (Section 4.5), we again assume that the victim is submitting his credentials to a completely unmodified Chrome browser. We relax the assumptions on the attacker, however, and only assume that the attacker has Javascript / Web Assembly code running within the victim's browser. This can occur when the victim navigates to a web page controlled by the attacker. The attacker aims to accomplish the same goal as with the *scrypt* attack: extract information on the victim's execution of *hash2curve* to reduce the search space for a dictionary attack.

Attack on BEEA. The design of Chrome's Password Leak Detection is such that both client and server are mutually untrusting. That is, even the server should not learn anything about the clients' credentials, and Password Leak Detection is designed with a malicious server in mind.

Thus, for our attack on BEEA, we assume that the attacker has access to the blinded output of the *hash2curve*, which is true when the attacker colludes with the server. We also note that this access

to the blinded point is safeguarded only by TLS; as such, any attacker that can compromise the connection can also access the blinded point. This could occur via collusion with a TLS middlebox, or even a TLS Enterprise Root CA certificate installed on the victim's machine.

For this attack, in Section 4.6 we assume the attacker has native, unprivileged code running on the victim's machine, and that he attempts to extract information on the victim's execution of BEEA in order to recover the blinding factor.

4.4 Attacking Scrypt

In this section, we explore how to leverage a side-channel attack against *scrypt* as used in Chrome to leak information about its inputs. We empirically demonstrate how to use a combination of cache attacks to recover a small subset of the accesses into *scrypt*'s internal memory, which in turn enables an adversary to launch an efficient, offline dictionary attack against the username:password pairs used as input into *scrypt*.

After examining how *scrypt* leaks to an ideal side-channel attacker, we relax the requirements and demonstrate how we performed the attack in practice, with a much weaker attacker.

***scrypt* in Chrome.** Chrome's Password Leak Detection protocol uses *scrypt* in Line 5 of Algorithm 8, when the client hashes the username and password together. In this scenario, *scrypt* serves the function of preventing an adversarial client from efficiently using the Password Leak Detection service to confirm the validity of guessed leaked credentials.

More specifically, the *scrypt* algorithm [216] is a key-derivation function (KDF) that is memory-hard [215]. In contrast to password hashing algorithms that rely on being computationally expensive, *scrypt* was designed to require a large amount of memory, thus making parallelism impractical, assuming it is harder to scale up memory than to scale up computing power. This allows Password Leak Detection to better resist attackers employing ASICs or FPGAs to brute force the server's dataset with massive parallellism.

[1]This attack recovers all 256 bits of the password hash. In reality, however, the users' passwords have far less than 256 bits of entropy.

Algorithm 9. Scrypt: The function first uses the PBKDF2 key-derivation function to create p blocks each of length $128 * r$ bytes. Each block is then transformed by the scryptROMix function. The output is the derived key DK.

1: **function** SCRYPT($P, S, N, r, p, dklen$)
2: $B[0] \,||\, B[1] \,||\, ... \,||\, B[p-1] \leftarrow$ PBKDF2$(P, S, 1, 128 * r * p)$
3: **for** $i = 0$ to $p - 1$ **do**
4: $B[i] =$ scryptROMix$(r, B[i], N)$
5: $DK \leftarrow$ PBKDF2$(P, B[0] \,||\, B[1] \,||\, ... \,||\, B[p-1], 1, dkLen)$

Achieving Memory Hardness. At a high level, *scrypt* achieves its memory-hard property by requiring input-dependent random accesses into a very large array. While this forces attackers to store large arrays in memory, it also means that *scrypt* is non–constant-time. As noted above, previous works [27, 97] have theorized that cache attacks against *scrypt* are possible due to its inherently non–constant-time nature. Our attack, however, is the first concrete, end-to-end attack on a memory-hard hash function.

4.4.1 The Scrypt Algorithm

Before describing our attack on *scrypt*, we now outline the relevant portions of the *scrypt* algorithm as it pertains to our attack against Chrome's Password Leak Detection. We refer the reader to RFC7914 [216] for a more complete specification.

Notation. Following the notation of [216], the *scrypt* algorithm takes the following parameters: P, the passphrase to be expanded; S, the salt; N, the CPU and memory cost parameter; r, the block size parameter; p, the parallelization parameter; and $dklen$, the length of the derived key.

Scrypt Overview. Algorithm 9 is an overview of the *scrypt* algorithm. With the exception of P, the password, all parameters, including the salt, are publicly accessible values that are fixed across all users. Thus, the password value P is the only variable input to *scrypt* as used in Password Leak Detection.

In Line 2, B, an array of length p where each element is a block $128*r$ bytes in length is initialized to the output of the PBKDF2 key derivation algorithm. Then, the loop on Line 3 iterates over each block and replaces it with the value obtained by calling the function scryptROMix$(r, B[i], N)$.

Algorithm 10. scryptROMix: The *Initialization Phase* sets each $V[i]$ to $\mathsf{scryptBlockMix}^i(X)$. The *Access Phase* uses j, a function of the input to *scrypt*, as an index into V, thereby making *scrypt* non constant-time.

```
1:  function scryptROMix(r, B, N)
2:      X ← B
3:      for i = 0 to N − 1 do                          ▷ Initialization Phase
4:          V[i] ← X
5:          X = scryptBlockMix(X)
6:      for i = 0 to N − 1 do                          ▷ Access Phase
7:          j = Integerify(X) mod N
8:          T = X ⊕ V[j]                               ▷ Input-Dependent Memory Access
9:          X = scryptBlockMix(T)
10:     return X
```

Finally, the password P and the blocks B, along with the desired output length $dklen$, are passed to PBKDF2 to produce the derived key DK. Most relevant to our attack is Line 4 of Algorithm 9, as *scrypt*'s scryptROMix is highly non constant-time.

scryptROMix. The scryptROMix function (Algorithm 10) is responsible for both *scrypt*'s memory hardness and its non–constant-time-ness. X is first set to the input block B. Then, the For-loop at Line 3 initializes V, an array of size N, by setting $V[i]$ at each iteration to be equal to $\mathsf{scryptBlockMix}^i(X)$. We will refer to this first For-loop as the *Initialization Phase*. The scryptBlockMix function takes an input array of a given size, mixes the bytes, and returns an array of equal size.

The second For-loop, beginning at Line 6, iterates N times and makes a non–constant-time, input-dependent memory access (IDMA) each time. We call this second For-loop the *Access Phase*. The IDMA occurs at Line 8, where j was computed by the previous line as $j = \mathsf{Integerify}(X) \mod N$. In turn, $\mathsf{Integerify}(X)$ simply returns the final 4 bytes of X, interpreted as a little-endian unsigned integer. Since X comes from the output of PBKDF2 via B, which is dependent upon the input to *scrypt*, j is also a function of *scrypt*'s input. Thus, when j is used as the index into V in Line 8, Algorithm 10 makes an IDMA that is dependent upon the password P, making the entire hashing operation non constant-time.

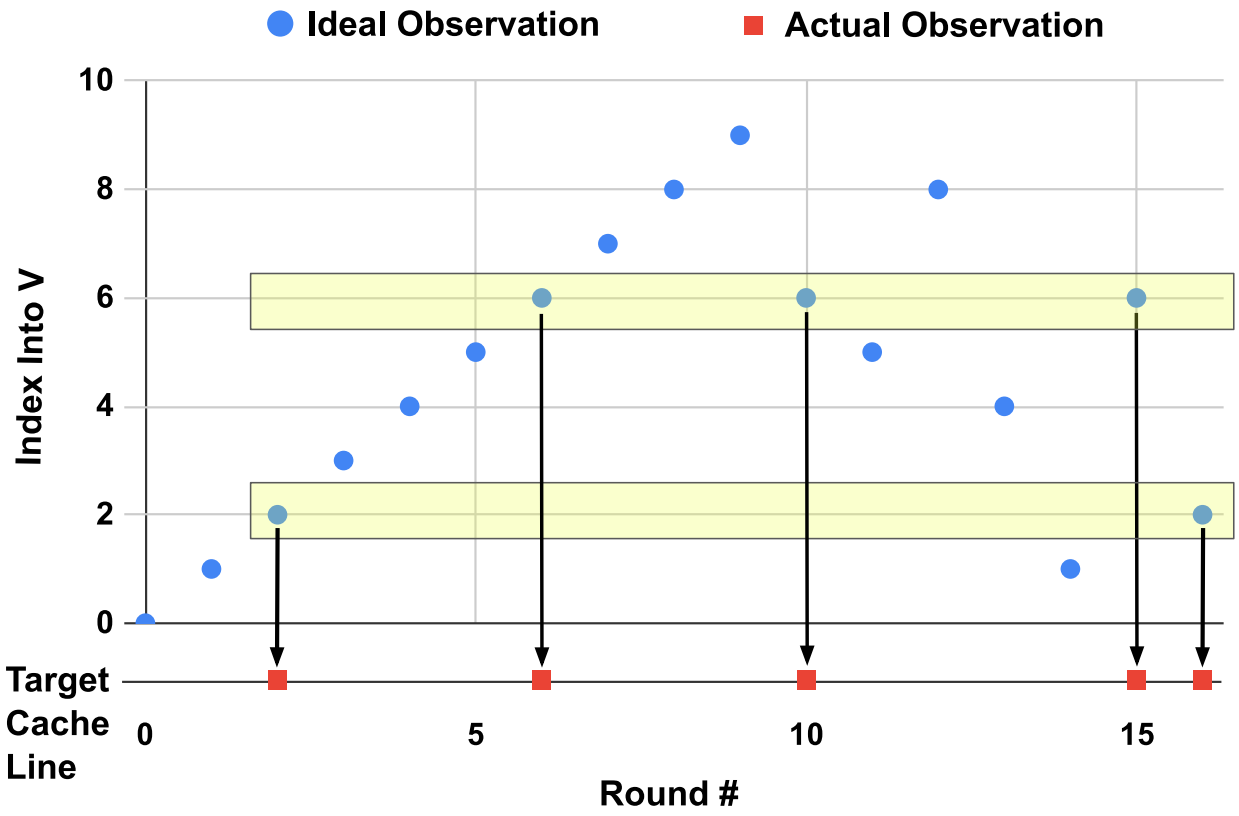


Figure 4.2: **Attacker's View of Memory Accesses.** While an idealized attacker can observe each memory access indicated by the blue dots, a realistic attack can only observe memory accesses into a single cache set (yellow boxes) projected onto a single-dimensional trace, represented by the red dots.

4.4.2 Idealized Side Channel Analysis of Scrypt

In this section we perform a side channel analysis of the *scrypt* hashing algorithm assuming an all powerful attacker that can perfectly observe every single memory access into V.

Memory Layout Mapping. *scrypt*'s leakage stems from the IDMA. As the victim executes the For-loop of the *Initialization Phase*, it accesses each memory location in V sequentially (Line 4), with a call to scryptBlockMix in between each access (Line 5). These sequential accesses result in the diagonal line of hits comprising the left part of Figure 4.2. As the elements of V are accessed sequentially, an attacker that observes the memory accesses in the *Initialization Phase* can learn which elements of V correspond to which memory locations.

Obtaining an Input Dependent Access Pattern. The second half of the accesses in Figure 4.2 comprises the IDMAs. These accesses are governed by the values of j, which in turn depend on

scrypt's secret input P, the password. The attacker learned which memory locations correspond to which indexes and can thus correlate these accesses to the Initialization accesses. This allows the recovery of the values of j at each iteration of the loop in the *Access Phase*. We refer to the sequence of j indexes $(j_0, j_1, \ldots, j_{N-1})$ into V as the *V-Access-Pattern*.

Observing Line 5 of Algorithm 8, the *V-Access-Pattern* is dependent upon the client's canonicalized user name u' and password p. For a specific user name u and password p we denote by $VAP(u, p)$ the access pattern to V resulting from the invocation of Password Leak Detection on u and p.

Leakage Quantification. The precise amount of leakage (in bits) available from $VAP(u, p)$ depends strongly on the concrete parameter choices used by Chrome's Password Leak Detection protocol. First, we note that Chrome sets $N = 4096$ when invoking scryptROMix (Algorithm 10). Thus, the leakage available via a perfect observation of $VAP(u, p)$ is theoretically upper bounded by $log_2(4096^{4096}) = 49152$ bits. However, this theoretical limit is further bounded by the size of the input space into *scrypt*'s scryptROMix function. Analyzing the parameter choices for Algorithm 9, we observe that Chrome sets $p = 1$ and $r = 8$. Thus, the call to the PBKDF2 routine in Line 2 of Algorithm 9 produces a total of $128 \cdot 8$ bytes of output, mapping passwords to a digest space of size 2^{8192}. Finally, given that [258] estimate roughly 23.4–31.2 billion unique credential pairs, we expect each username and password pair (u, p) to create its own distinct $VAP(u, p)$.

Credential Recovery via Dictionary Attacks. As each username and password pair (u, p) creates it own distinct $VAP(u, p)$, an attacker can recover u and p from their $VAP(u, p)$ by mounting a dictionary attack. That is, given a plain-text file F of compromised usernames and passwords, an attacker can pre-compute the dictionary

$$DICT(F) := \{(u^*, p^*, VAP(u^*, p^*)) : \forall (u^*, p^*) \in F\}. \tag{4.1}$$

Next, during the online phase, in case the attacker obtains some *V-Access-pattern* $VAP(u, p)$ corresponding to an attacker-unknown credential (u, p), it is possible to recover (u, p) with high probability by performing a search of $VAP(u, p)$ in F. Finally, we note that this attack violates

the requirement that the Password Leak Detection server acts as an inefficient oracle, as only lookup operations over $DICT(F)$ are used during the online phase, while the pre-computation of $DICT(F)$ from a list of compromised credentials F can be done entirely offline via Eq. (4.1).

4.4.3 The Reality of Cache Attacks

The previous subsection analyzed *scrypt* through the lens of a perfect microarchitectural adversary, capable of completely reconstructing *scrypt*'s memory access patterns. While similar approaches were proposed in prior works [27, 97], in this section we outline the challenges in empirically realizing this theoretical attack. As we show, these challenges result in an extremely limited view into the *V-Access-pattern* for any given *scrypt* execution, necessitating a different approach. In the following subsections, we outline our solutions and demonstrate the first end-to-end attack on *scrypt*.

Challenge 1: Memory Coverage. The theoretical attack described in Section 4.4.2 assumes that the attacker can probe all cache sets in between every iteration of both the For-loops in Algorithm 10. This would be required to ensure that no memory access into V is missed, especially during the IDMA where the value of j cannot be predicted by the attacker ahead of time. In practice however, the loop in the *Access Phase* executes in less time than it takes to Prime+Probe a set, so an attacker must somehow slow the victim in order to have a chance to probe even a *single* cache set at each iteration of the loops. From a signal analysis perspective, this results in the attacker's view of Figure 4.2 being limited to accesses to a single cache set, i.e. the yellow boxes.

Challenge 2: Congruent Cache Lines. There are two yellow boxes, indicating that the attacker views multiple indexes into V, because the array V is large enough to span multiple *congruent cache lines*. Congruent cache lines are lines that map to the same cache set, thereby preventing a Prime+Probe attacker from distinguishing between accesses to addresses that map to congruent cache lines.

As Chrome parameterizes *scrypt* with $N = 4096$, $r = 8$, and $p = 1$, this results in V being an array of 4096 elements with each element 1024 bytes in length, spanning 4MiB total. Next, as the

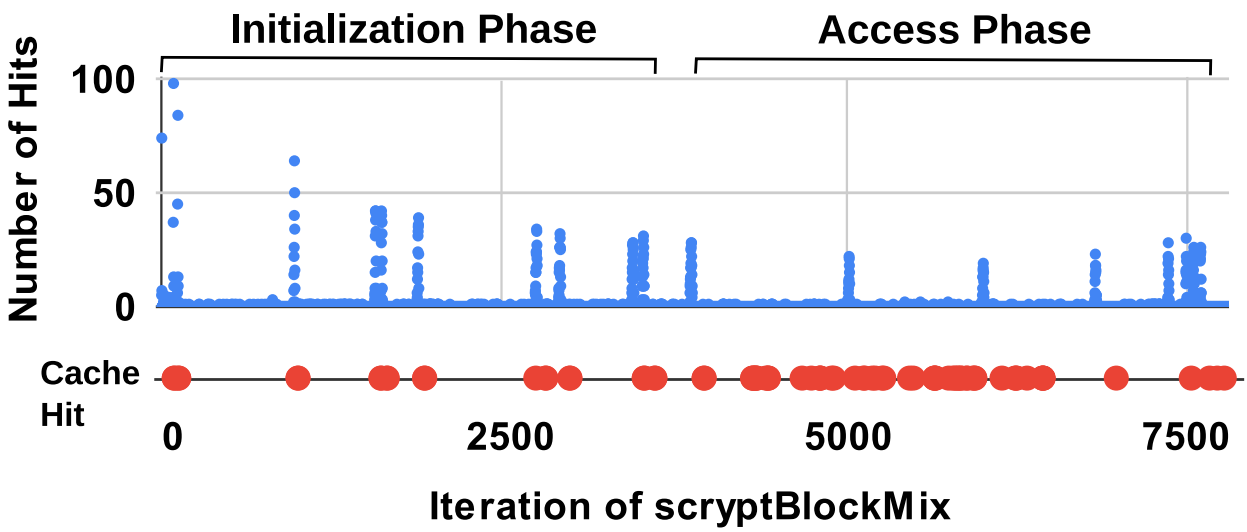


Figure 4.3: . On top is what is obtained from averaging across 150 samples of a custom *scrypt* victim that used the same memory locations for V each time. In reality, we have to use a single-trace, as seen below in red, because Chrome uses different memory locations for V every run.

typical L3 cache on Intel machine uses 0.5 MiB ways, we expect that on average any given element of V will be share the same cache set with $(4096 * 1024)/(0.5 * 1024 * 1024) = 8$ other elements. Thus, rather than observing a single hit during *scrypt*'s *Initialization Phase* followed by a single hit during the *Access Phase*, the attacker should expect to see 8 hits during the *Initialization Phase* and an average of 8 corresponding hits during the *Access Phase*.

Empirically demonstrating this issue, we executed an instrumented version of Chrome's *scrypt* code while monitoring the accesses to a single cache set. The top graph in Figure 4.3 illustrates the access times, where we added the results of 150 *scrypt* traces corresponding to the same cache set. The y-axis measures the number of times a cache-hit was observed, while the x-axis indicates which round it was recorded in. In this example, there are roughly 9 peaks for the *Initialization Phase* followed by another 9 for the *Access Phase*, which is close to what is expected.

Challenge 3: Noise. Further complicating the attack is the presence of noise, which prevents us from perfectly learning which n indices are accessed at which rounds in the *Access Phase*. This can be seen in the Figure 4.3(bottom), which shows a single trace containing 80 hits, though only 16 are expected.

A natural approach to mitigating this issue is averaging the access time across multiple exper-

iments, as was done in the top graph of Figure 4.3. However, this is not possible in the case of attacks on unmodified versions of Chrome because Chrome's implementation of *scrypt* allocates a different set of physical memory locations to store V each time. Thus, we developed techniques to overcome the noise and analyze each experiment in isolation, as opposed to combining the results across multiple experiments.

4.4.4 Attacking Scrypt in Chrome

Having outlined the issues with the theoretical attacks on *scrypt* considered in prior works, we now demonstrate how to attack the *scrypt* implementation used in an unmodified Chrome browser's Password Leak Detection protocol.

Step 1: Observing Control Flow. In order to observe the memory access patterns into the V array during every iteration of both the For-loops in Algorithm 10, we first need to establish a *ticker*, or a marker that indicates the beginning of a new round in either of the For-loops. Since the scryptBlockMix function is called once per iteration, we created our ticker by repeatedly using the Flush+Reload attack on a memory addresses holding the code of this function. We will use this ticker as an indicator of start every iteration of the For-loops in Algorithm 10, allowing us to assign every subsequent side channel observation to its corresponding loop iteration.

Step 2: Performance Degradation. Next, as the iterations of the For-loops in Algorithm 10 are so short in duration, we also had to slow the execution of the scryptBlockMix function using a performance degradation attack [24]. To that aim, we repeatedly used the clflush instruction in order to flush a code region corresponding to the Salsa 20 Core function, which is repeatedly called inside scryptBlockMix.

Step 3: Prime+Probe. With the ticker and performance-degradation established, we choose a random cache set and mount a Prime+Probe attack on it after each occurrence of the ticker. As outlined in Section 4.4.3, the memory layout of V inside the CPU's cache implies that mounting a Prime+Probe attack on a given cache set discloses when an access is performed to any of the roughly 8 congruent elements of V, without the ability to further distinguish between the elements.

4.4.5 Handling Noise

As outlined in Section 4.4.3, a further issue that complicates our attack is the presence of noise, which takes the the form of offsets in the ticker and additions or deletions in the Prime+Probe cache hits. As noted above, we cannot simply average out the noise over multiple traces, since Chrome's *scrypt* implementation ends up using a different set of physical memory locations to store V each time. Thus, rather than combining several measurements into a clean trace, we instead overcome the noise while analyzing each trace in isolation.

Prime+Probe Noise. When we conduct the Prime+Probe attack on a cache set, we see a large amount of false positive noise, where we record cache hits during rounds where there should not be. In addition, we observed a minimal amount of false negative noise, where cache hits are missing.

To overcome this, we implement a scoring system for our dictionary attack, where we assign points to candidate passwords based off how well they "fit" the results from a trace. Given an index j_0 that is accessed during the *Initialization phase*, the attacker can pre-compute at which rounds during the *Access Phase* $V[j]$ will be accessed in case the password candidate is correct. For each of those rounds in the *Access Phase* which the Prime+Probe trace contains a memory access, the candidate password's score is incremented by 1.

We repeat the above approach for each index, $j_0, j_1, ..., j_{n-1}$, for which accesses are detected during *Initialization Phase*, in order to compute the candidate's final score. Finally, after applying this approach on all password candidates in the dictionary, we rank the highest scoring candidates as the most likely passwords.

Ticker Noise. While the above approach is useful for handling noise present in the accesses to V obtained using the Prime+Probe channel, we must adapt this algorithm to also account for noise that is present in our Flush+Reload ticker.

We begin by recalling that we use the ticker to determine at which round the accesses into V are made, during both Initialization and Access phases. Due to both false negatives and false positives, the round corresponding to the memory access is unfortunately rarely correct.

Denoising the Ticker During Initialization. We observe that ticker noise is most damaging

during the *Initialization Phase*. If the ticker is off, then our algorithm ends up assigning points for the trace fitting the wrong *V-Access-Pattern*. Fortunately for the attacker, however, the accesses to V during the *Initialization Phase* occur deterministically, with elements of V accessed sequentially (see Line 4 of Algorithm 10). Thus, the attacker learns some of the low bits of the index due to the elements of V being smaller than one page. In particular, each element of V is 1024 bytes, meaning that 4 fit into each page, and for each j_i accessed during the *Initialization Phase*, the attacker learns $j_i(mod \ \ 4)$. With this optimization, we empirically found it optimal to expanding our scoring algorithm to also consider the indexes that match in the low 2 bits immediately above and below where the hit occurred.

Determining the Transition Point. Next, after denoising the ticker during the *Initialization Phase* of Algorithm 10, we must locate the "halfway point": the point in Line 6 in Algorithm 10, just before the second For-loop starts. This allows us to realign the trace after drifting for 4096 rounds in the *Initialization Phase*. We do so by exploiting the fact that one iteration of the loop in the *Initialization Phase* (Line 3) executes more quickly than one iteration in the *Access Phase* (Line 6) due to the additional code at Lines 7 and 8. Thus, by looking for a point where the time between ticker hits consistently increases, we are able to identify the halfway point, allowing us to identify the ticker's transition to the *Access Phase*.

Denoising the Ticker During Accesses. Ticker noise during the *Access Phase* has a more straightforward solution. When searching for the hits in the *Access Phase* that correspond to the indices found during the *Initialization Phase*, the attacker simply expands her search for any hit within 10 rounds of the expected location. We empirically found 10 to be a good compromise between being too small to overcome the noise, and being so large as to generate too many false positives.

Avoiding Averaging. Overall, we were able to combine the above denoising and scoring techniques into an algorithm to identify candidate passwords based on how well they fit the limited, noisy information the attacker gained on the *V-Access-Pattern*. This improves on prior theoretical attacks on *scrypt* [27, 97], which assumed the attacker has a noiseless and perfect access to the

V-Access-Pattern.

4.4.6 Empirical Evaluation

We now evaluate the effectiveness of our attack on Chrome's Password Leak Detection protocol.

Experimental setup. We conducted our attack against an unmodified Chrome binary running on an Acer Aspire E 15 laptop, equipped with 8GiB of DDR4 memory and an Intel i5-8250U CPU. The i5-8250U features 4 cores and 8 threads and is equipped with a 6MiB 12-way set associative L3 cache. Our machine was running Ubuntu 20.04.3 with Linux kernel version 5.8.0-44.

Attack Scenario. Assuming an attacker with native unprivileged code execution on the target machine, we implemented the attack described in this section using the Mastik toolkit [289]. We ran our attacker against an unmodified version of the Chrome browser, and evaluated our attack against 10 randomly chosen passwords from the "rockyou.txt" dictionary. For each password, the victim browser submitted the username:password pair of ("z", pw) into a website 5 times, where pw was the randomly chosen password from "rockyou.txt". This resulted in our side-channel attacker obtaining 5 traces for each password the victim submitted, where each trace is the result of using side-channels to leak information about the victim's *V-access-pattern*.

A Dictionary Attack. With the collected traces in hand, we applied the approach described in Section 4.4.5 and conducted an offline dictionary attack. We follow the precedent of previous works and benchmark our attack on the "rockyou.txt" password file, which contains 14,341,564 plaintext passwords stolen during the 2009 RockYou data breach.

We computed the *V-access-pattern* for every entry in the entire "rockyou.txt" file and scored them against the 10 sets of traces corresponding to the 10 passwords the victim submitted. We ran this computation on an Intel Xeon server machine, featuring a Platinum 8352Y CPU, 128 cores, and 1.8TB of memory. It took about 8400 core hours of offline computation to complete the dictionary attack, or about 3 days when parallelized across all 128 cores.

The results of our end-to-end attack are displayed in Figure 4.4. The correct password ended up scoring higher than all other passwords in the dictionary 80% of the time. Thus, if the attacker

were to attempt to log in with the candidate passwords in descending order, the attacker would successfully log into the victim's account on the very first try, 80% of the time. In the worst case, the attacker would succeed on the 6th try. Finally, we note that with such a low number of attempts required for success, no reasonable amount of rate limiting on password attempts can prevent our attacker from compromising the target's account.

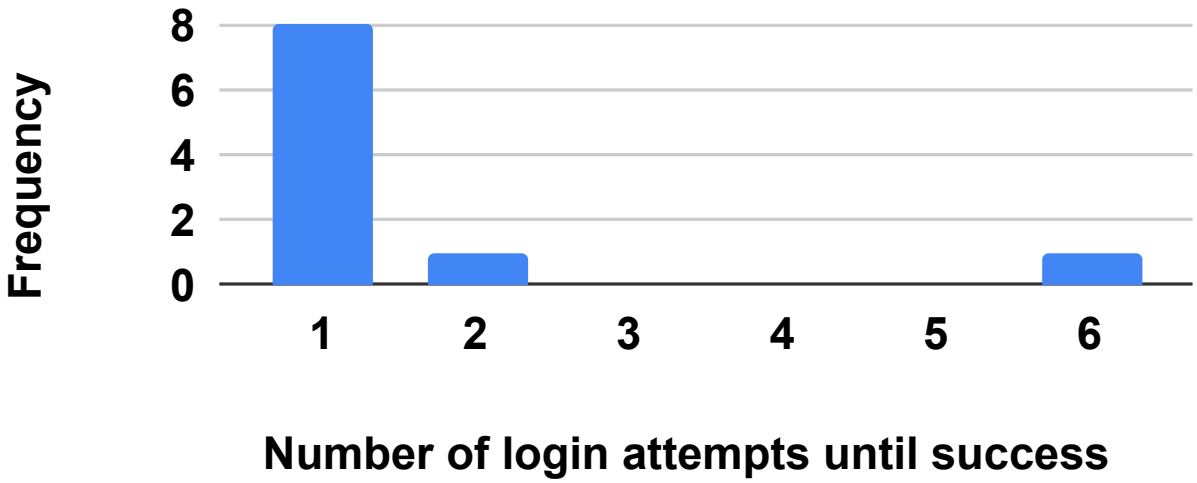


Figure 4.4: **Histogram of the results from our attack on *scrypt* in an unmodified Chrome browser.** 80% of the time, the attacker guesses the correct password on the very first attempt.

4.5 Attacking Hash2Curve

Moving away from attack Password Leak Detection scrypt implementation, in this Section we will examine how Chrome's *hash2curve* usage reveals bits of the user's credentials. Before demonstrating end-to-end attacks from both native code and the Chrome browser on this part of the Password Leak Detection protocol, we now proceed to review Google's *hash2curve* construction and implementation. We used the unmodified Chrome version 106, the latest at the time of writing, for all analyses and experiments in this section.

Algorithm 11. Google's *hash2curve* implementation.

```
1: function hash2curve(m)
2:     p_x ← RandomOracleSHA256(m)
3:     while !OnCurve(p_x) do                    ▷ Input-Dependent Loop
4:         p_x ← RandomOracleSHA256(p_x)
5:     return (p_x, p_y)
```

4.5.1 Hash2Curve Overview

The *hash2curve* algorithm takes an arbitrary length input string and deterministically outputs a point on a specified elliptic curve. Such a primitive is useful for a number of cryptographic applications; in the case of Password Leak Detection, Chrome's *hash2curve* algorithm serves the purpose of mapping the output of *scrypt* to a point on an elliptic curve to prepare it for use with Diffie-Hellman PSI [139]. This can be seen in Line 6 of Algorithm 8.

Chrome's Hash2Curve Implementation. We describe the *hash2curve* algorithm used in Password Leak Detection in Algorithm 11. The variable p_x is first computed by passing the string m as input to the function RandomOracleSHA256, which repeatedly uses SHA-256 and modular addition to approximate a pseudo-random function (PRF). The output p_x is then repeatedly updated by being assigned the value $\mathsf{RandomOracleSHA256}(p_x)$ until p_x is a valid x coordinate of a point $P = (p_x, p_y)$ on the NIST P-256 curve (Line 3). The algorithm then outputs $P = (p_x, p_y)$ (Line 5). About half of the possible values of x correspond to points on the curve, so the while-loop in Line 3 of Algorithm 11 terminates with probability $1/2$ in each iteration.

Side Channel Vulnerability of Hash2Curve. While constant-time *hash2curve* algorithms do exist [274], we note that the *hash2curve* algorithm used by Chrome's Password Leak Detection is inherently not constant-time. More specifically, Algorithm 11 uses a rejection sampling method, repeatedly iterating over candidate p_x values until a suitable value is found. While this design pattern was originally proposed by [55], such an implementation is explicitly discouraged by [133, Appendix A] due to side channel considerations.

A Dictionary Attack on Hash2Curve. An attacker can exploit the rejection sampling design of Algorithm 11 for mounting dictionary attacks similar to those mounted in Section 4.4. Given a

credential dataset D, an attacker can apply the *hash2curve* algorithm to every entry of D, obtaining the corresponding number of iterations taken by Line 3 of Algorithm 11, since it is an input-dependent loop (hereinafter IDL). Next, by using a side-channel attack to obtain the number of iterations taken by the IDL on the target's credentials, the attacker can eliminate candidates of D that do not induce the same number of iterations, thereby reducing the attack's search space.

4.5.2 Native Attack on Hash2Curve

In this section we describe a Flush+Reload based attack on Chrome's *hash2curve* algorithm, executed from unprivileged native code running in the target's machine. Here, we empirically found the strongest results when using Flush+Reload on a string that is touched by Chrome's GetPointByHashingToCurveInternal function, which corresponds to the OnCurve test within the IDL.

Attack Outline. To mount a Flush+Reload attack, we execute an unprivileged attacker process monitoring Chrome's GetPointByHashingToCurveInternal function on the target machine. We then open an exemplary login page and complete the login process with a set of user credentials. This triggers Chrome's Password Leak Detection protocol, allowing us to monitor the exact number of iterations made by the IDL.

Attack Evaluation. In order to measure the accuracy of our Flush+Reload attack, we gathered 5500 pairs of username and password, where every 500 pairs generate the same number of iterations of the IDL between 0 and 10 (inclusive). We then executed the attack on each credential pair, noting the amount of detected iterations compared to the ground truth.

Analyzing the results, our Flush+Reload attack was able to correctly identify the number of loop iterations for 4960 credentials, resulting in a total accuracy rate of 90.18%. Next, in Figure 4.5 we outline the accuracy of our attack as a function of the actual loop iterations performed by the IDL.

Finally, as the IDL exists with probability 1/2 for each loop iteration, we can use the data depicted

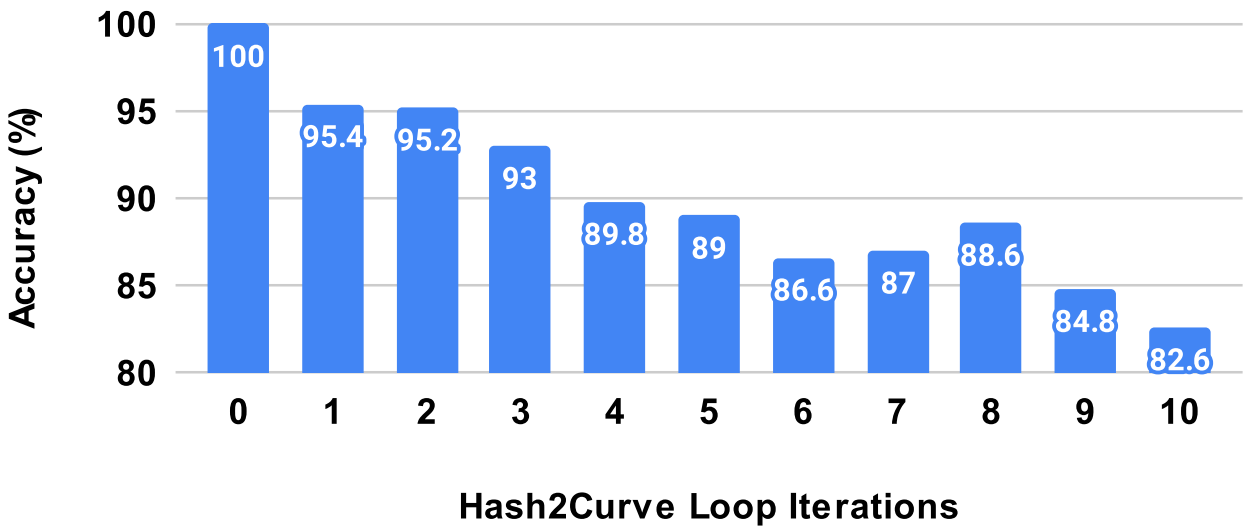


Figure 4.5: **Flush+Reload attack accuracy as a function of the number of *hash2curve* loop iterations.**

in Figure 4.5 to compute our weighted success probability as

$$\mathbb{W}[SR] = 1 * 1/2 + 0.954 * 1/4 + \cdots + 0.826 * 1/2^{11}$$

which is roughly equal to 97%.

A Dictionary Attack. By recovering the number of iterations with 97% accuracy, the attacker is able to conduct a dictionary attack and filter out candidate passwords that don't have the same number of iterations. As the IDL exists with probability 1/2 for each loop iteration, the attacker essentially learns an additional bit about the password with each iteration; this means that in the ideal case where the attacker can perfectly recover the number of *hash2curve* loop iterations, the expected number of bits by which the password's entropy is reduced is:

$$\mathbb{E}[B] = 1 * \frac{1}{2} + 2 * \frac{1}{4} + 3 * \frac{1}{8} = \sum_{i=1}^{\infty} i/2^i = 2.$$

4.5.3 Attacking Hash2Curve Within Chrome

Having established the vulnerability of Chrome's *hash2curve* algorithm to Flush+Reload attacks from native code, in this section we present a browser-based attack on Chrome's *hash2curve* implementation. More specifically, we attack *hash2curve* from within an unmodified Chrome browser, using an attacker webpage that executes malicious JavaScript and WebAssembly code.

This is a weaker assumption on the attacker's capabiltiies than in Section 4.5.2, as it is usually easier for an attacker to trick a victim into visiting the attacker's web page. As such, modern browser versions recognize the danger of browser-based side-channels, and employ a heavily sandboxed environment for code execution compared to a native scenario. Accordingly, browser-based adversaries face additional technical challenges, which we now describe.

Flushing Data in the Cache. The Flush+Reload technique used in Section 4.5.2 requires the clflush instruction and shared memory between attacker and target. However, neither is available in a browser environment.

Instead, we use Prime+Probe to observe the activity of Chrome's *hash2curve* algorithm, using the work of Vila et al. [272] in order to efficiently generate eviction sets.

High-Precision Timing Source. Measuring the cache access patterns of the *hash2curve* algorithm requires the attacker to distinguish cache hits from misses, necessitating a timer with a precision of tens of nanoseconds. However, modern Chrome versions deliberately limit the timer resolution to 5 μs, aiming to foil side channel attacks [278].

We sidestep this issue by using the SharedArrayBuffer JavaScript API, which provides a primitive for a precise counting thread on the order of nanoseconds [242]. While SharedArrayBuffer was previously disabled by Chrome in an attempt to mitigate speculative execution attacks, recent versions of Chrome re-enabled this primitive due to the presence of dedicated Spectre countermeasures [28, 232].

Just-In-Time Compilation. In contrast to attacks that are mounted using native code, which have near-complete control over the attack code executed by the target machine, browser-based adversaries are limited to code emitted by Chrome's JavaScript and WebAssembly execution

engines [34, 256]. This introduces measurement noise, making traces obtained through side-channels unreliable and nondeterministic.

To overcome this issue, we observe that it is possible to introduce a warmup stage into our attack code, causing Chrome to always run its optimizing compiler over our high-level Prime+Probe implementation. Furthermore, we initialize our code in a way that Chrome's optimizing compiler will cache its output [65], allowing us to consistently use Prime+Probe across many attack runs. With both measures in place, we achieve a greater probing frequency compared to naive Prime+Probe implementations, allowing us to reliably monitor the execution of the IDL.

Automatically Selecting the Correct Eviction Set. Having generated eviction sets using Vila et al. [272], we must now determine the eviction set corresponding to the IDL. To that aim, our attacker page renders an attacker-controlled login page inside an iframe, populating it with dummy credentials known to the attacker. This triggers the execution of Chrome's Password Leak Detection protocol, eventually resulting in invocations of the IDL.

For each eviction set generated by Vila et al. [272], we execute a Prime+Probe attack on our dummy credentials using the above procedure, locating the eviction set that recorded the correct amount of cache misses corresponding to the execution of the IDL. With the correct eviction set in hand, we can now mount Prime+Probe attacks on the target's credentials.

Attack Evaluation. Our goal is to determine whether browser-based attacks on Chrome's *hash2curve* algorithm are capable of mirroring the performance of native attacks outlined in Section 4.5.2. With this in mind, we mount a Prime+Probe attack on a targeted username and password pair, using the eviction set found earlier.

Figure 4.6 presents a time series of Prime+Probe attack iterations; the y-axis plots the number of counting thread ticks elapsed while accessing the eviction set in each iteration. We note the 15 spikes of probes that have at least 500 ticks (in red), which corresponds to 15 iterations of the IDL when Chrome's Password Leak Detection is invoked on our credential. We can distinguish the spikes resulting from the target string being accessed from system noise, since the latter only results in access times no longer than 200 ticks. Finally, Figure 4.7 shows how many times out of 30 we

were able to observe the correct number of spikes when running our attack during 1, 5, and 10 login attempts. While we could not distinguish targeted credentials inducing less than 5 IDL iterations, we observe that 5 logins suffice for the attack to succeed in at least half the trials. This results in our browser-based attack with 5 traces reducing the password's entropy by an expected:

$\mathbb{E}[B] = 5 * \frac{19}{30*2^5} + 6 * \frac{19}{30*2^6} + ... + 10 * \frac{20}{30*2^{10}} = 0.24$ bits.

While this may seem like a trivially small amount of leakage in the average case, we contend that only examining the average case and neglecting to account for the danger to passwords with higher numbers of IDL iterations belies the severity of our attack. Instead, we emphasize that our results concretely demonstrate the risk of a significant amount of leakage for a non-trivial number of cases; as shown in Figure 4.7, when there were 10 IDL iterations and 5 login attempts were observed, the attack succeeded in 20 of 30 trials. This means that for these passwords, the attack reduced the entropy of the password by 10 bits, 2 out of 3 times.

Since 1 out of 1024 credentials result in 10 IDL iterations, and the attack succeeded 2 out 3 times, this means 1 out of 1536 passwords will leak 10 bits. Given the scale at which a browser-based attack can be launched, that fact that our attack can leak 10 bits from 1 in 1536 passwords is concerning, even if the average password leaks very little.

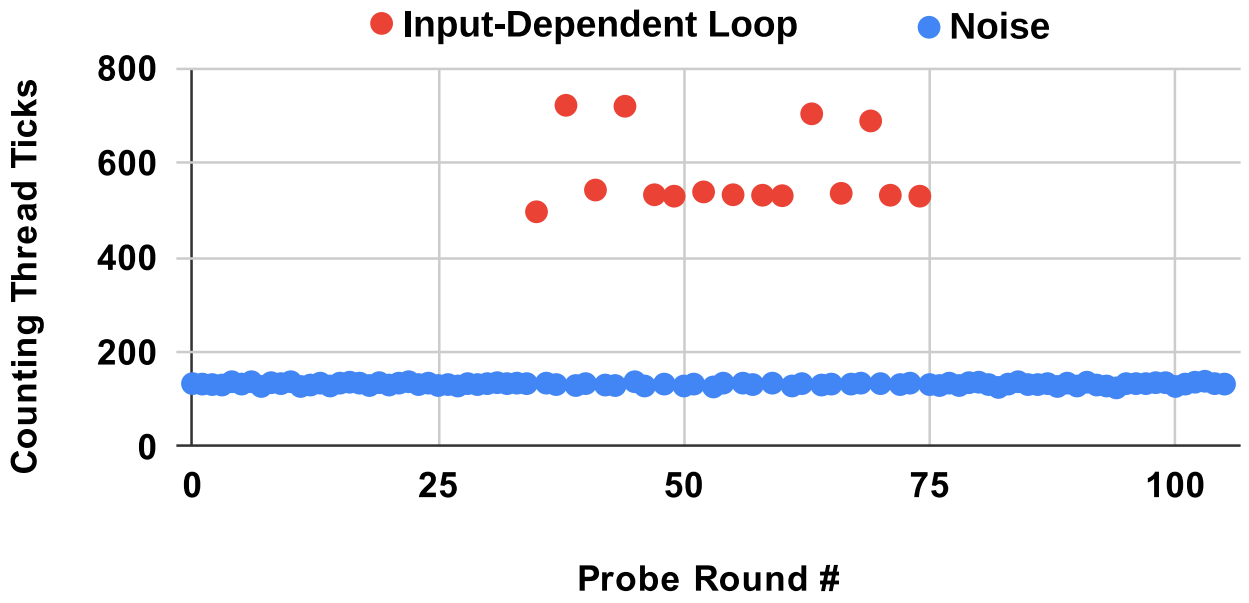


Figure 4.6: **Elapsed counting thread ticks over Prime+Probe iterations.** The 15 spikes correspond to the 15 IDL iterations highlighted in red.

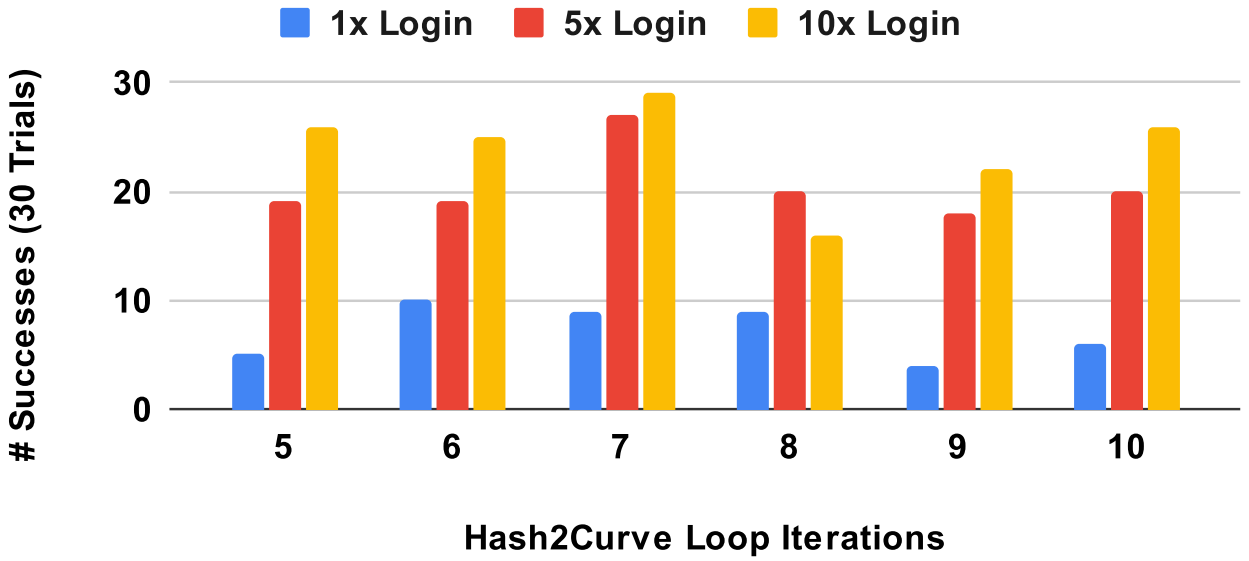


Figure 4.7: **Effect of increasing the login attempts on the number of successes for each number of IDL iterations.**

4.6 Attacking Blinded Hashes

We now examine how Chrome blinds the client's credentials before sending them to the server. In particular, we found that Chrome uses the Binary Extended Euclidian Algorithm (BEEA) to compute the modular inverse of its secret key for blinding, which is susceptible to side-channel attacks.

To investigate the extent to which this compromises Password Leak Detection's security guarantees, we developed a novel cryptanalysis technique that can recover BEEA's inputs after observing only a single, noisy trace obtained via a cache side-channel. We then used our technique to successfully demonstrate the first ever microarchitectural single-trace attack on the BEEA algorithm, improving on prior work that required a controlled execution environment (e.g. SGX [192]) or constraints only present during RSA key generation [21].

Blinding Chrome's Password Leak Detection Protocol. In the Password Leak Detection algorithm, after the client hashes the user's credentials to the point on the curve Q, the client must *blind* the hash before including it in the request to the server. This takes place on Line 7 of Algorithm 8. Specifically, the client blinds the hash by computing Q^a on the elliptic curve, where a is the secret key. This blinding serves the purpose of concealing all bits of the client's hashed credentials from the Password Leak Detection server.

This step is in fact critical, as the Password Leak Detection protocol was designed to preserve privacy in both directions, meaning that the server should learn nothing about the client's credentials. If the server learns the hash of the client's credentials, the server can then launch a dictionary attack to brute force the client's plaintext password.

Computing Modular Inverses using BEEA. To unblind the response received from the server, the client also needs to compute $a^{-1} \bmod p$, where p is the prime modulus used for the NIST P-256 elliptic curve. The value of $a^{-1} \bmod p$ is then used to unblind the response received from the server in order to complete the Diffie-Hellman PSI [139]. Next, to perform the computation of $a^{-1} \bmod p$, Chrome uses the BEEA algorithm, as implemented by BoringSSL's BN_mod_inverse function, which in turn calls BoringSSL's BN_mod_inverse_odd to take advantage of an optimization for

odd moduli.

Threat Model. Similarly to Section 4.4's threat model, we also assume a side-channel adversary that is able to run unprivileged native code on the victim's machine. Moreover, in this section we assume that the attacker has access to the blinded hash, and wants to obtain the value of the unblinded hash. Access to the blinded hash could occur in practice if the server participating in the Password Leak Detection protocol colludes with the attacker; unblinding the hash would allow the server to brute force the client's credentials via a dictionary attack, a scenario Chrome's Password Leak Detection protocol was specifically designed to protect against.

We note that the attacker can access the blinded hash in other ways, besides colluding with the server; if the attacker could somehow compromise the TLS connection (e.g. through a TLS Enterprise Root CA certificate, or a TLS middlebox), the attacker would gain access to the blinded hash.

4.6.1 Prior Attacks on the BEEA Algorithm

The BEEA Algorithm has been studied extensively from a side-channel perspective due to the widespread need for computing modular inversions in cryptographic systems (e.g. RSA, ECDSA). While prior side-channel attacks against BEEA are well known, the specific manner in which Google uses BEEA in Chrome's Password Leak Detection protocol prevents the application of prior attack techniques. More specifically, the theoretical analysis of BEEA done in [13] assumes that the attacker can obtain perfect, noiseless traces of the BEEA's execution, which is not possible with current attack techniques. Furthermore, Chrome generates a new random blinding factor a upon each generation of a request, thereby precluding combining information across traces via either averaging or lattice attacks [22, 102].

Thus, in order to attack BEEA as it is used in Chrome, a side-channel attacker must operate with only a single, noisy trace of the BEEA execution. In prior works, this has only ever been accomplished by either placing the victim inside of an SGX enclave [192] or by exploiting the redundancy and relations between various known parameters when BEEA is used during RSA key

generation [21]. As neither of these scenarios apply to Chrome's Password Leak Detection, in this section we develop a novel noise-tolerant single-trace attack on the BEEA algorithm that enables extraction of modular inverses computed by BEEA.

4.6.2 Attacking BEEA

At a high level, our attack uses cache-attacks to determine the control flow of the BEEA, which in turn allows for the recovery of the inputs to BEEA. In this case, the inputs to BEEA are a, the client's blinding key, and n, the prime modulus for the elliptic curve NIST P-256.

Chrome's BEEA Algorithm. Algorithm 12 is a pseudocode of the BEEA algorithm. Borrowing notation from [13], we use SHIFTS[i] to denote how many times the branch at Line 4 or Line 9 was taken at the ith iteration of the outer while-loop at Line 3 (only one branch or the other will be taken during any given iteration, as one of A and B will be even and the other odd at the beginning of each iteration). We let SUBS[i] denote the outcome of the comparison at Line 14, such that SUBS[i] $= 3$ if branch 3 is taken, and 4 if if branch 4 is taken at the ith iteration. We note that for any iteration i, if SUBS$[i-1] = 3$, then the SHIFTS$[i]$ must all take place in Branch 1 due to the subtraction at Line 16 in Algorithm 12. Likewise, if SUBS$[i-1] = 4$, then the next iteration's $SHIFTS[i]$ take place in Branch 2.

Perfect Trace Requirement. [13] previously showed that if an attacker can perfectly recover the SUBS[] and SHIFTS[] for all iterations, they can reconstruct both inputs to BEEA in polynomial time. The downfall of this analysis of BEEA is that a single error in either SUBS[] or SHIFTS[] completely foils the recovery of the inputs. Moreover, the attacker cannot determine if there were any errors, and thus whether or not the result is correct.

Prior work [192] that utilized the analysis of [13] was able to extract BEEA's inputs via perfect side channel traces, carefully controlling its execution within an SGX enclave. For attacking Chrome, however, the traces obtained via Flush+Flush contain a substantial number of errors over the course of BEEA's roughly 700 branches on its randomized inputs, preventing us from applying the analysis of [13]. Thus, we overcome the perfect trace requirement by presenting the first analysis

Algorithm 12. Binary Extended Euclidian Algorithm: Pseudocode for Chrome's BEEA implementation, which is optimized for the odd modulus n used in NIST P-256. We make the observation that the conditional add, labeled as Branch 5, allows for error correction in a noisy trace.

```
1:  function BN_MOD_INVERSE_ODD(a, n)
2:      A ← n, B ← a, X ← 1, Y ← 0
3:      while B ≠ 0 do
4:          while even(B) do
5:              B ← B/2
6:              if odd(X) then                          ▷ Branch 1
7:                  X ← X + n                           ▷ Branch 5
8:              X ← X/2
9:          while even(A) do
10:             A ← A/2
11:             if odd(Y) then                          ▷ Branch 2
12:                 Y ← Y + n                           ▷ Branch 5
13:             Y ← Y/2
14:         if B ≥ A then
15:             X ← X + Y                               ▷ Branch 3
16:             B ← B − A
17:         else
18:             Y ← Y + X                               ▷ Branch 4
19:             A ← A − B
20:     return Y
```

of BEEA input recovery for the case of noisy traces.

4.6.3 Cryptanalysis of a Noisy Trace

To correct the errors present in a noisy BEEA trace, we draw inspiration from prior works on partial key recovery [130, 132] and develop a branch-and-prune style algorithm for BEEA. At a high level, this involves searching for the correct *key*, where a key is a sequence of branches that could have been taken by the execution of BEEA. Our algorithm repeatedly *branches* towards the most probable keys, as determined by how closely they align with the trace. We then exploit the relationship between different segments of the keys to *prune* key candidates, until the correct key is found.

Probing the BEEA Algorithm. While prior works [13, 21, 102, 192] only probed Branches 1, 2, 3, and 4 via cache attacks, we make the observation that detecting the conditional `add` in Branch 5 can be used to prune potential keys during our search algorithm. This is because the values X and

Algorithm 13. Branch and Prune: Pseudo-code for our branch and prune algorithm that recovers the complete input to BEEA, given a single noisy trace.

```
1:  function BRANCH AND PRUNE(Array Trace)
2:      pq ← PriorityQueue()
3:      pq.push(100, 0, Key([]))
4:      while notEmpty(pq) do
5:          (score, i, curKey) ← pq.pop()
6:          if i == len(T) then
7:              Output(curKey)
8:              Continue
9:          curBranch ← Trace[i]
10:         newKey ← curKey.append(curBranch)
11:         if correctXY(newKey) then
12:             if curBranch == 5 then
13:                 score ← score + 20
14:             pq.push(score, i + 1, newKey)
15:         if LastSub(curKey) == 3 then
16:             newKey ← curKey.append(1)
17:             X ← CalculateX(newKey)
18:             if isOdd(X) then
19:                 Continue
20:             pq.push(score − 20, i, newKey)
21:         else
22:             newKey ← curKey.append(2)
23:             Y ← CalculateY(newKey)
24:             if isOdd(Y) then
25:                 Continue
26:             pq.push(score − 20, i, newKey)
```

Y are completely determined by all the previous branches taken. Furthermore, since X and Y start off initialized to 1 and 0 respectively, and the value n is known to be the prime modulus of NIST P-256, then if we know all the prior branches, we can determine if branch 5 can possibly be taken at the current iteration. We will now cover in detail exactly how our branch-and-prune algorithm works in Algorithm 13.

Algorithm Description. In Line 1, our algorithm takes $Trace$, an array of branches corresponding to the sequence of branches taken by BEEA, as input obtained through a side-channel attack. The branches are simply numbers from 1 through 5, corresponding to the labeled branches in Algorithm 12. We make the assumption that $Trace$ only contains *deletion* errors, and only deletions of 1s and 2s occur; furthermore, 1s and 2s are never deleted when they are immediately preceding

a 5. In Section 4.6.4 we explain why this was the case for when we used cache-attacks to obtain traces on BEEA.

The data structure that we use to process our candidate keys is the priority queue on Line 2, which is sorted by the $score$ of each candidate key. A key's $score$ is a measure of how close to the real key we believe the candidate key to be. In Line 3, we populate pq with with a key with score equal to 100, an iterator i equal to zero, and an array of branches equal to 0. The iterator is used to track how far along the trace the key has progressed through. The key's array of branches represents the sequence of branches taken by the BEEA algorithm. As the algorithm progresses, we will incrementally build up longer keys that get progressively closer to the real key.

Finding New Keys. At each iteration of the loop on Line 4, we process the highest scoring key, corresponding to the key that we currently believe to be the closest to the true key, and push additional keys onto the priority queue with one additional branch added at a time.

To generate these additional keys, our algorithm branches at each iteration of the outer while loop to create 2 additional keys. The first new key is formed by branching towards the $Trace$ by appending the next branch within $Trace$ to the current key, as seen on Lines 9 and 10. This candidate key is then pruned on Line 11 if it fails to satisfy $correctXY()$.

Pruning on X and Y. The $correctXY()$ function inspects the $newKey$'s array of branches. Assuming an execution of BEEA that follows those branches, $correctXY()$ then determines if it is possible for all occurrences of Branch 5 to be at the locations that they are. That is, after every occurence of Branch 1 and Branch 2, it checks to see that X or Y, respectively, at that point are odd if and only if a Branch 5 occurs on the subsequent branch. If not, then $correctXY()$ returns false, and the key is pruned. On the other hand, if the key is not pruned, then $newKey$'s score is incremented by 20 on Line 13, its position in $Trace$ is incremented by 1, and the key is pushed onto pq on Line 14.

We note that in actuality, only the most recent branch and the resulting X and Y need to be checked in this manner. Any inconsistencies between X and Y and the sequence of branches earlier in the key would have resulted in that key already having been pruned.

False Positives. While it is possible for $correctXY()$ to return true for keys that do in fact have errors in them, as the distance from the errors grows, the chance of continual false positives is equal to $1/2^n$, where n is the number of 1 branches and 2 branches since the error. This is because X and Y are modified with each occurrence of branches 1 and 2, effectively randomizing whether or not they are odd at any given point. Furthermore, the subtractions in branches 3 and 4 ensure that errors in X propagate to errors in Y, and vice versa. As n grows, the probability of continuing to follow an incorrect key becomes vanishingly small, and the incorrect key values are pruned back to where the error occurred.

Inserting Branches. After branching towards the $Trace$, the second additional key is found by inserting a potential branch into $curKey$. This is how keys that contain the branches deleted by the noisy trace are discovered and added to pq. Since the $Trace$ obtained in Section 4.6.4 only contains deletions of 1 Branches and 2 Branches, the $LastSub()$ function at Line 15 only needs to determine whether to insert a 1 Branch or a 2 Branch, which depends on whether the most recent branch is 3 or 4 respectively.

After inserting the potential branch at Line 16, Line 17 uses the function $CalculateX()$ to determine the value of X within BEEA after executing the branches in $newKey$. If this X is odd, this would induce a Branch 5 to follow. However, since there are no deletions of Branch 5 or the immediately preceding branch 1s and 2s in $Trace$, inserting an additional branch 5 automatically renders the key incorrect; as such, we prune the key at Line 19 if X is odd. Otherwise, we push the $newKey$ onto the priority queue at Line 20. To prioritize keys that align more closely with $Trace$, we decrement the $newKey$'s score by 20, since for most branches the $Trace$ is correct and does not require an insertion. We leave i untouched because we only inserted an additional branch, and did not progress through $Trace$. Lines 22 through 26 serve the same purpose, only for when $LastSub == 4$.

Termination. Once i iterates through the entirety of the $Trace$, the candidate key is output, along with its score. The algorithm can continue to run indefinitely, continuously outputting more complete keys as it explores them. The higher the score of a key, the more likely it is to be the correct

one. Intuitively, the true key is the one that aligns most closely with $Trace$, with the insertions in the correct places that result in X and Y being odd whenever dictated by the occurrences of branch 5 in the $Trace$.

4.6.4 Implementing the Attack

In this section we describe how we implemented our cache attack to obtain a trace against BEEA, and how our branch-and-prune algorithm recovered its inputs.

Software Setup. Chrome is statically linked against BoringSSL, and as such the Password Leak Detection logic calls BoringSSL's BN_mod_inverse function to compute the modular inverse of the blinding factor. To benchmark our attack, we developed a test harness that calls BoringSSL's BN_mod_inverse, compiled with `gcc` version 9.4.0 using an `-O0` flag. Mirroring Chrome, we call BoringSSL's BN_mod_inverse(a, n) with random 256-bit values of a, where n set as NIST P-256's prime modulus.

This is in contrast to the prior two attacks on *scrypt* and *hash2curve*, where we conducted our attacks against unmodified versions of Chrome. We did this because BEEA sees widespread deployment across numerous commonly implemented cryptosystems, and we believe that our novel cryptanalysis has implications on these as well. Thus, there are broader impacts in analyzing how BEEA leaks using our attack technique. By benchmarking our attack against BoringSSL directly, we demonstrate that our attack applies to a wider variety of BEEA usages (any binary that uses BoringSSL's implementation, such as Chrome), and does not rely on nuances specific to Chrome.

Finally, we run experiments on a laptop featuring a Quad Core Intel i5-8250U CPU, and 4 GB of RAM.

Flush+Flush Probing Locations. In order to generate a trace that attempts to reconstruct the control flow of the the victim's BEEA execution, we used the Flush+Flush attack [118] in order to monitor the 5 branches marked in Algorithm 12. We note that doing so requires a total of 4 Flush+Flush probes, as the same probe can be used to monitor both Branch 1 and Branch 2 since these share the same call to BN_rshift1 on Lines 8 and 13. Similarly, a single probe monitors Branch

5 on both Line 7 and Line 12 as they make the same call to BN_uadd. We use the remaining two probes to detect branches 3 and 4, which in turn allows us to discern between branch 1 and branch 2, as described in Section 4.6.2.

Signal Amplification via Core Assignment. As our i5-8250U processor has 4 physical cores, we run each Flush+Flush probe on a separate core, while having the probe for Branch 5 running on the sibling virtual core to the process executing the BEEA algorithm. As the BEEA process now shares its caches with the probe for branch 5, due to [19] this results in the signal for branch 5 becoming unmistakably strong. This is important, as it virtually eliminates false positives or negatives for branch 5, allowing us to prune keys aggressively by error correcting with the occurrences of Branch 5. Furthermore, this also results in a trace where branch 1s and branch 2s immediately preceding branch 5s are not deleted in the trace, as branch 5 can only ever take place after a 1 or 2.

Gathering Traces. We then allow the BN_mod_inverse function to run while the probing processes monitor the branches. We parse the resulting data from the Flush+Flush probes form the trace, which is the sequence of branches within BEEA observed by the attacker. We find that this results in no insertions or deletions of branches 3, 4, and 5. However, it is common for there to be insertions or deletions in the number of branch 1s and branch 2s in each round of the while-loop in Line 3 of Algorithm 12. This is because a series of 1 branches or 2 branches can execute in very quick succession when branch 5 is not taken inbetween them.

To make sure that the trace only contains deletions, we calibrate our parser to be extremely conservative with adding 1s and 2s to the trace, only adding them to the trace when the signal is extremely clear. This ensures that only deletions, and not additions, appear in the trace.

Attack Results. After collecting a trace with only deletions in branches 1 and 2, we passed the noisy trace to the branch-and-prune algorithm. Within just 34 ms, the algorithm found the correct key, with 18 branches inserted, and with all 703 branches correctly recovered. This key was also the first one output by the program, and after continuing to run the branch-and-prune program for 10 minutes, this key had the largest score, making it clear that it was the correct key. Having recovered all $SUBS[i]$ and $SHIFTS[i]$, we then used the method described by [13] to trivially

recover BEEA's inputs.

Implications for Chrome's Password Leak Detection Protocol. A malicious server that successfully launches this attack can recover the client's secret, a which was used in Line 7 of Algorithm 8 to blind the hash of its credentials as Q^a. After Q^a is sent to the Password Leak Detection server as part of its `request`, the server can easily compute a^{-1} and use it to unblind the client's hash as $(Q^a)^{a^{-1}} = Q$, where Q is the unblinded hash of the client's credentials.

This completely violates the security guarantees of Chrome's Password Leak Detection protocol, as it was designed to allow client's to safely query the Password Leak Detection service without having to place any trust in the server. In this case however, the client is essentially sending a hash digest of their credentials to the server, allowing the server to run offline dictionary attacks using lists of compromised credentials aiming to breach the client's account.

4.7 Mitigations

The defacto standard for mitigating cache side-channel attacks in software is to make use of the constant-time programming paradigm. In this style of programming, the control flow of the program must not depend in any way upon the program's input; moreover, no accessed memory address can depend upon the input [194]; in other words, the execution of the program must be completely oblivious to its input.

Mitigating against the three vulnerabilities described in this paper, however, is not as easy as simply replacing vulnerable components with constant time implementations. This is because Chrome's usage of *scrypt* as a memory-hard hash function poses a difficult problem, with complex trade offs.

Scrypt. Chrome uses *scrypt* as its hash algorithm for Password Leak Detection due to its memory-hardness properties. A memory-hard hash function is one where the cost of evaluating the hash function is primarily dominated by the cost of memory, as opposed to the cost of compute power. While attackers can employ ASICS and FPGAs to gain a computing advantage of up to

100,000x [53] over general purpose computers, the cost of memory remains the same for both general purpose machines and ASICS/FPGAs. This makes it difficult for attackers to compute the memory-hard hash function at a significantly lower cost than honest users, who must compute the hash with general purpose computers. For this exact reason, *scrypt* is an attractive option for hashing passwords, as [26] proved that *scrypt* is maximally memory-hard under the parallel random oracle model.

This memory-hardness property of *scrypt* comes at a price, however. Namely, [25] show that no function can be both maximally memory-hard and input oblivious; as a consequence, *scrypt* is *inherently* vulnerable to cache side-channel attacks, and in order to mitigate our attack, a compromise is required between input obliviousness and memory-hardness.

As such, we recommend replacing *scrypt* with an alternative option, such as one of the side-channel resistant variants of Argon2 [50], the winner of the 2015 Password Hashing Competition. Argon2i is a variant of Argon2 that is completely constant-time; however, it offers the weakest memory-hardness of the Argon2 variants. This may be unappealing for Password Leak Detection, where resistance against parallel GPU cracking attacks is highly desirable.

A compromising solution is Argon2id, which aims to strike a balance between memory-hardness and side-channel resistance. This is accomplished by making the first pass over the input oblivious, while the second half is input dependent, thereby reducing the amount that can be learned by a side-channel attacker. This, however, means that Argon2id is not completely constant-time, and still leaks some amount of information to side-channel attackers. We caution against permitting any side-channel leakage at all; as we demonstrated with our attack against *scrypt*, even an extremely limited view of the victim's memory accesses can potentially lead to a complete breach of security.

Hash-to-Curve. In contrast, protecting the *hash2curve* portion of Password Leak Detection is comparatively simple; it does, however, require a slight change in protocol, due to the current *hash2curve* algorithm's usage of a rejection sampling method, which is inherently non constant-time. Instead, using one of the constant-time hash-to-curve implementations described in [133] is sufficient to mitigate our attack against the *hash2curve* portion of Password Leak Detection.

Modular Inversion. Similarly, the BEEA algorithm used for modular inversion by Chrome is inherently non constant-time; however, there are known alternatives for modular inversion that are indeed constant-time, and exchanging BEEA for one of these does not require any protocol change.

A potential solution is to make use of Fermat's Little Theorem and to compute the inverse of a as $a^{-1} \equiv a^{p-2} (\mathsf{mod\ p})$ where the exponentation is performed in constant-time. We can compute this exponentiation both performantly and in constant-time by taking advantage of the fact that the modulus for Curve P-256 is fixed; by pre-computing an optimally short addition chain for the modulus, we can use the addition chain to exponentiate in constant time, with fewer multiplications than other methods [89].

4.8 Future Work

Other Browsers. Following Chrome's lead, both Microsoft Edge and Mozilla Firefox have implemented their own password leak detection functionality. At the moment, Firefox simply queries the HaveIBeenPwned database; Edge, on the other hand, developed their own novel cryptographic PSI system based off of homomorphic encryption [75, 76]. Investigating their novel cryptosystem's susceptibility to side-channels and other attacks could reveal new insights into how password leak detection systems must consider security and privacy.

Hashing Scheme Tradeoffs. Given our attacks on *scrypt* and Chrome's *hash2curve*, it is natural to wonder if there are any existing hash algorithms would be more suitable with regards to trade offs between performance, memory-hardness, and input-obliviousness. It could be interesting to augment existing hash algorithms, or perhaps even design new ones, that are more desirable for Password Leak Detection.

BEEA Partial Key Recovery. Due to the numerous existing attacks against BEEA, it is easy to imagine how our partial key recovery algorithm can prove useful to that direction of research. However, our key recovery algorithm is specifically tailored to account for the type of noise that we encountered within 0our traces; it is likely possible to expand upon our algorithms capabilities such that it can handle a broader variety of noisy traces.

CHAPTER 5

Pseudorandom Black Swans: Cache Attacks on CTR_DRBG

Modern cryptography requires the ability to securely generate pseudorandom numbers. However, despite decades of work on side-channel attacks, there is little discussion of their application to pseudorandom number generators (PRGs). In this work we set out to address this gap, empirically evaluating the side-channel resistance of common PRG implementations.

We find that hard-learned lessons about side-channel leakage from encryption primitives have not been applied to PRGs, at all levels of abstraction. At the design level, the NIST-recommended CTR_DRBG design does not have forward security if an attacker is able to compromise the state via a side-channel attack. At the primitive level, popular implementations of CTR_DRBG such as OpenSSL's FIPS module and NetBSD's kernel use leaky T-table AES as their underlying block cipher, enabling cache side-channel attacks. Finally, we find that many implementations make parameter choices that enable an attacker to fully exploit the side-channel attack in a realistic scenario and recover secret keys from TLS connections.

We empirically demonstrate our attack in two scenarios. In the first, we carry out an asynchronous cache attack that recovers the private state from vulnerable CTR_DRBG implementations under realistic conditions to recover long-term authentication keys when the attacker is a party in the TLS connection. In the second scenario, we show that an attacker can exploit the high temporal resolution provided by Intel SGX to carry out a *blind* attack to recover CTR_DRBG's state within three AES encryptions, without viewing output, and thus to decrypt passively collected TLS connections from

the victim.

5.1 Introduction

It is a truth universally acknowledged, that a securely implemented cryptographic primitive must be in want of a cryptographically secure pseudorandom number generator [31]. Modern cryptography relies on randomness to prevent an attacker from predicting secret values generated by parties in a cryptographic protocol. Indeed, random values are universally used to ensure security properties for nearly all cryptographic data, including secret keys for confidentiality or integrity, secret keys for public-key encryption, key exchange, or signatures, as well as for protocol nonces to prevent replay attacks.

Thus, a cryptographically secure Pseudorandom Generator (PRG) is one of the fundamental primitives of modern cryptography, both in theory and in practice.

The simplest theoretical PRG construction is an algorithm that expands a smaller seed into a longer output sequence that is computationally indistinguishable from a true sequence of random bits. However, the practical security demands for random number generation are somewhat more complex; in real systems, these pseudorandom number generator constructions are often multi-stage algorithms that collect inputs from environmental entropy sources or hardware into an "entropy pool". The pool is then used to seed a PRG that generates cryptographically secure output. Real world PRGs must also meet additional security guarantees, including recovery from state compromise.

A number of academic works and practical security failures have illustrated the disastrous effects on real-world cryptography from flawed random number generation implementations or designs in the wild. These have ranged from unintentional flaws such as failure to properly seed PRGs [131, 162, 184, 295], to designs prone to implementation mistakes [80], to a suspected intentional back door in the now "deprecated and disgraced" [209] Dual EC DRBG design, which appears to have been repurposed and exploited in the wild [73, 74].

Since their introduction in the seminal works of [41, 210, 214], microarchitectural attacks that exploit contention on internal components to leak information have been used to violate nearly every security guarantee offered by computer systems. Indeed, in recent years there have numerous examples of side-channel attacks with diverse targets and vectors. These range from attacks that extract cryptographic keys from keystroke timing [119, 291] via CPU caches, attacks that exploit transient execution for breaking fundamental OS isolation guarantees [68, 163, 175, 264, 279], and even attacks that exploit limitations in memory hardware to change or read the contents of stored data [66, 160, 167, 168]. Side-channel resistance is among the key security properties demanded of implementations.

Much less is known, however, about the security of PRGs in the presence of side-channel leakage. While backtracking resistance and prediction resistance are stated to be among the main security goals of the designs in NIST's PRG recommendations (NIST SP 800-90A), the standard does not consider the impact of side-channel attacks on these goals. Although some initial evidence [301] already indicates the possibility of exploiting side-channel vulnerabilities in PRG seeding, a systematic exploration of side-channel leakage from PRG implementations has not been performed. Thus, in this paper we set out to explore the following main question:

Are common PRG designs susceptible to microarchitectural side-channel attacks? What are the security implications of such leakage and how can the attacker exploit it?

5.1.1 Our Contribution

Unfortunately, in this paper we give a positive answer to the above questions. CTR_DRBG is the most popular PRG design out of those recommended in NIST SP 800-90A, and is supported by 68% of validated implementations in NIST's Cryptographic Module Validation Program (CMVP). On the first question, we show that CTR_DRBG is vulnerable to state compromise attacks because some popular implementations still use a non-side-channel-resistant implementation of the underlying block cipher. On the second question, we show that several popular CTR_DRBG implementations fail to properly reseed the PRG in many situations, enabling feasible attacks against prediction

resistance. Furthermore, we demonstrate that Intel SGX allows a very strong *blind* state recovery attack in as few as three encryptions, without the attacker having access to PRG output. We demonstrate end-to-end attacks on the CTR_DRBG implementations used by OpenSSL's FIPS module, NetBSD, and FortiOS, allowing an attacker targeting TLS connections to recover session secrets and long-term ECDSA keys used for client authentication, and under SGX, to passively decrypt connections.

The Use of T-Table AES. T-table AES is a performance-oriented AES implementation that uses table lookups to compute the state transitions between individual encryption rounds. Unfortunately, because these lookups are key-dependent, T-table AES has become the canonical example of cache side-channel leakage [56, 201, 210].

While the use of T-table AES for encryption and decryption operations has been greatly reduced in light of the threat posed by side-channels and the availability of AES-NI hardware, similar lessons do not seem to have been learned for the case of random number generation. Remarkably, even after more than a decade of attacks, [41, 63, 121, 197, 210] we show that unprotected and leaky T-tables are still used for encrypting the counter inside CTR_DRBG by the following popular implementations:

- The OpenSSL 1.0.2 FIPS Module uses T-Table AES for CTR_DRBG. We note that use of this library is the *only* way to obtain U.S. government certification for a cryptographic module without submitting to the expensive and time-consuming validation process.
- The NetBSD kernel uses CTR_DRBG with a T-Table AES implementation as the system-wide random number generator.
- The FortiOSv5 network device operating system uses the same vulnerable CTR_DRBG implementation as NetBSD.
- mbedTLS-SGX, a port of the popular mbedTLS cryptography library to SGX [298].
- The nist_rng library [154], which is a library for random number generation used by open source

projects such as libuntu (a C implementation of NTRUEncrypt), the XMHF hypervisor, as well as others.

CTR_DRBG State Recovery. By adapting previous work on AES encryption [201] to the PRG setting, we extend the work of [280] to show how an attacker who observes the cache access patterns of CTR_DRBG-based random number generation can recover the PRG's state using about 2000 bytes of the PRG's output. We then empirically demonstrate how a client that connects to a malicious TLS sever can be coerced to provide enough PRG output that an attacker can recover the PRG state used during the TLS handshake by concurrently observing the PRG's cache access patterns.

Extracting the Client's TLS Authentication Keys. Next, we show that NetBSD's kernel, OpenSSL's FIPS module and FortiOS fail to reseed the PRG with a sufficient amount of entropy. Thus, by using a moderate amount of brute forcing for the client entropy, the attacker can wind forward the client's PRG and recover the ECDSA nonce used by the client to authenticate herself to the malicious TLS server. Finally, using the recovered ECDSA nonce and the signature produced by the client during the TLS handshake, the attacker can recover the client's long term authentication keys. With authentication key in hand, the attacker can impersonate the client in future TLS connections.

State Recovery Without a Malicious TLS Server. The above attack on TLS requires the victim client to connect to a malicious TLS server, allowing the attacker to observe sufficient output generated by the client's CTR_DRBG implementation while simultaneously observing the client's cache access patterns across many AES encryption operations. Tackling this limitation, we perform a a novel differential cryptanalysis attack exploiting side-channel leakage from T-table based CTR_DRBG running inside an SGX enclave. This attack leverages the fact that CTR_DRBG encrypts an incrementing counter. Our technique is capable of extracting the PRG's state from only three AES encryption operations, without requiring the attacker to observe the PRG's output. Thus, we eliminate the need for the TLS client to connect to an attacker-controlled server. We also note that this type of attack might also be applicable to other settings with similar constraints such as

GCM-SIV [120].

Breaking TLS Connections With High-Entropy PRG Reseeding. Finally, we note that any call to CTR_DRBG for random byte generation must use at least three AES encryption operations, producing the cache access information required by our differential cryptanalysis state-recovery technique. As this vector does not require the TLS client to connect to an attacker-controlled server, the attacker can recover the PRG state on *any* request for random bytes, regardless of how the implementation reseeds the PRG. We demonstrate recovery of the premaster secret, master secret, and symmetric encryption keys for any TLS connection made by mbedTLS-SGX (a port of mbedTLS to SGX [298]) to any TLS server. In particular, we are able to passively decrypt the session by observing cache access patterns made by mbedTLS-SGX.

Summary of Contributions. In this work we study the implications of side-channel analysis on random number generation. Our contributions can be summarized as follows.

- We present the first security analysis of CTR_DRBG in the presence of side-channel leakage, showing that the PRG state of many popular implementations can be recovered via cache attacks (Section 5.4).
- We show that PRG reseeding algorithms in popular implementations are sometimes insecure. Combined with the above state recovery attack, we empirically demonstrate an end-to-end attack on TLS that recovers long-term client authentication keys if the TLS client connects to an attacker-controlled TLS server (Section 5.6).
- We present a novel differential cryptanalysis technique that exploits side-channel leakage from CTR_DRBG running inside an SGX enclave to recover the PRG state within three AES encryption operations (Section 5.7.2).
- We demonstrate and end-to-end attack on an enclaved TLS client that is capable of passively decrypting the TLS connections regardless of PRG reseeding (Section 5.7.4).
- Finally, we evaluate CTR_DRBG's popularity by scraping NIST's Cryptographic Module Validation Program database. We show that CTR_DRBG was the most popular design, supported by

68% of the implementations (Section 5.8).

5.1.2 Coordinated Disclosure

We disclosed the vulnerabilities we discovered to the security teams of OpenSSL, Fortinet, and NetBSD in May 2019. OpenSSL responded that these attacks are outside their threat model. Both NetBSD and Fortinet have since shared advisories and remediations for their customers. The DRBG flaw in FortiOS was assigned CVE-2019-15703.

5.2 Background

5.2.1 Pseudorandom Generators

The term "DRBG" does not seem to be widely used outside of the government context, so for the purposes of this paper, we will use the term pseudorandom generator (PRG). We begin by providing basic background regarding pseudorandom generators and their security properties. Informally, a PRG is an algorithm that, given an initial seed, produces a stream of random bits such that an attacker cannot distinguish the produced stream from a truly uniform random bit stream with probability better than some negligible bound.

PRG Definition. Following [91, 280], a *PRG with input* is a triplet of polynomial time deterministic algorithms $\{\mathsf{instantiate}, \mathsf{generate}, \mathsf{reseed}\}$. The PRG is instantiated by calling instantiate on an entropy sample I and a nonce N, and outputs initial state S_0. Next, generate gets as input a state S, a number of bits to output $nbits$, an additional input $addin$, and outputs new state S' and bits $R \in \{0,1\}^{nbits}$. Finally, reseed gets as input a state S, an entropy sample I, an additional input $addin$, and outputs a new state S'.

Random Number Generation. The PRG is instantiated by a single call to instantiate. A user can then repeatedly request up to r random bits through a call to generate, which also outputs a new state for the PRG. Finally, both the user and the generate function can also call update, which updates the state of the PRG to a new state.

PRG Security. [280] define three security properties for a PRG: robustness, backtracking resistance, and prediction resistance. Backtracking resistance is the property that if the generator is compromised at time t_1, an adversary remains unable to distinguish outputs generated prior to t_1 from random. Similarly, prediction resistance ensures that there is some time t_2 after t_1 when no further outputs can be distinguished from random. Robustness incorporates both of these guarantees into a single property.

Next, while the model of [91, 280] includes an attacker that is able to compromise the entropy distribution used for sampling entropy to the PRG, we consider a weaker attacker who is unable to do so.[1]

We instead assume that the PRG correctly receives entropy samples drawn uniformly at random from the entropy space, better matching our real-world scenario.

Finally, as our attack targets the prediction resistance guarantee of CTR_DRBG, we now provide a more formal definition for prediction resistance, from [91].

Prediction Resistance. As mentioned above, prediction resistance models a PRG's ability to recover from state compromise. We begin by modeling an adversary capable of compromising the PRG state by allowing the adversary to execute the following procedures on the PRG.

- **get-output.** Models an attacker's ability to query the PRG for output. Calls $\mathsf{generate}(S, nbits, addin)$ where S is the current state, $nbits$ is the number of bits to output, and $addin$ is known by the attacker, and returns the output R.

- **set-state.** Models an attacker who compromises the state of the PRG. Gets as input an attacker-chosen value S^* and sets the PRG state $S \leftarrow S^*$.

- **next-ror.** Tests an attacker's ability to distinguish output from the PRG from uniformly random output. Sets $R_0 \leftarrow \mathsf{generate}(S, nbits, addin)$ with S as the PRG state, $nbits$ the number of bits in R_0, and $addin$ known by the attacker. It then sets R_1 to a value drawn uniformly at random

[1] We therefore obtain a stronger result as our weaker attacker is able break the PRG despite her inability to corrupt the entropy source.

from the same domain as R_0 and picks a uniform choice bit $b \leftarrow^{\$} \{0,1\}$. The procedure returns R_b to the adversary which outputs a bit b'.

An adversary's advantage, and therefore the security strength of the PRG, is parameterized by the number of calls an adversary makes to the above procedures along with the adversary's probability of successfully guessing the challenge bit in the **next-ror** game. We use the following formal security definition for a PRG:

Definition 9 (PRG with Input Security)**.** A PRG with input $\mathcal{G}$ is called a $(t, q_D, q_R), \delta-$prediction-resistant PRG if for any adversary $\mathcal{A}$ running in time at most t, making at most q_D calls to update with q_R calls to next-ror/get-output, and one call to get-state, which is the last call $\mathcal{A}$ is allowed to make prior to calling next-ror, it holds that

$$\left|\Pr\left[b = b' \mid b' \leftarrow \mathcal{A}_{\mathcal{G}}^{\mathsf{OP}}(q_D, q_R)\right] - 1/2\right| \leq \delta$$

where $\mathsf{OP} = \{\text{next-ror}, \text{set-state}, \text{get-state}, \text{get-output}\}$.

5.2.2 NIST SP 800-90 and Related Standards

NIST Special Publication (SP) 800-90 is entitled "Recommendation for Random Number Generation Using Deterministic Random Bit Generators" and is the de facto standard for algorithms for generating random numbers. The document was first published in 2006 and has undergone three revisions: "800-90 Revised", published in 2007, "800-90 A", published in 2012, and "800-90A Rev. 1", published in 2015. The first three publications contained four pseudorandom number generator designs, while the last publication contained only three. The missing design was the infamous DualEC DRBG, which was removed from the publication after Shumow and Ferguson discovered a design flaw that enabled a backdoor [248] which was later confirmed by Snowden [217]. The three remaining designs in NIST 800-90A Rev. 1 are HMAC_DRBG, HASH_DRBG and CTR_DRBG, which are based on HMAC, hash, and block cipher primitives respectively. For the remainder of this paper, we will refer to the 2015 publication as SP 800-90A.

5.2.3 AES

AES encryptions and decryptions can be decomposed into four operations (AddRoundKey, SubBytes, ShiftRows, and MixColumns). Performance-optimized software implementations usually use a series of lookup tables known as "T-tables" to combine the latter three operations. AES encryptions and decryptions can be decomposed into rounds, which use round keys derived from the secret key to transform the input into a sequence of states. The state at each round is used to index into the T-tables, and the results are XORed with the round key to produce the state for the next round. The final round of AES uses a different T-table from earlier rounds as there is no MixColumns operation in that round. Unfortunately, by observing the memory access patterns to these tables, an attacker can recover the cipher's secret key within only a few encryptions. Indeed, starting from [210], there has been a large body of work on attacking table-based AES implementations [119, 121, 145, 255, 299].

Most modern processors include CPU instructions that perform AES encryptions and decryptions in hardware. In addition to improving performance, these instructions do not rely on table lookups from system memory, thereby mitigating side-channel risks. Although hardware AES is widely implemented in modern desktop processors, many cryptographic libraries still use software-only implementations of AES in a variety of cases.

5.2.4 Cache Attacks

Our work contributes to a long line of cache-based side-channel attacks. These attacks have yielded varied and robust mechanisms [90, 118, 262] for breaking cryptographic schemes using information leakage from cache timings. Popular targets have included digital signature schemes [39, 114] and symmetric ciphers [210, 220, 291], despite the inclusion of countermeasures in popular cryptographic implementation libraries [106, 223]. Recent literature has also begun to examine side-channel vulnerabilities in environments provided by trusted processor enclaves, particularly Intel SGX [61, 172, 190, 264, 283, 286], which are designed to be more secure against even local attackers who are able to run unprivileged code.

Flush+Reload. Flush+Reload is a side-channel attack technique that consists of three steps. In the first step, the attacker *flushes* or evicts a memory location from the cache. The attacker then waits a while, allowing the victim to execute. Finally, in the third step, the attacker *reloads* the monitored memory location and measures the reload time. If the victim has accessed the memory location between the flush and the reload steps, the location will be cached, and the reload will be fast. Otherwise, the memory will not be cached and the reload will be slow. Flush+Reload has been used to attack symmetric [145] and public key [39, 102, 114, 218, 291] cryptography, as well as for non-cryptographic and speculative execution attacks [69, 119, 163, 175, 264, 268, 279, 288].

Prime+Probe. While powerful, Flush+Reload relies on the victim and the attacker accessing the same memory location and is thus typically applied to OS-deduplicated pages in binaries and shared libraries. When shared memory is not available (e.g., for SGX), we use a different cache attack technique called Prime+Probe [210, 262].

A Prime+Probe attack consists of three steps. In the first, the attacker primes the monitored cache lines by making enough memory accesses so that each way (group of cache lines fetched together) of the targeted cache sets is occupied by the attacker's memory value. In the second step, the attacker yields control to the victim process. In the final step, the attacker probes those same cache lines by reading from the corresponding memory locations and measuring their access times. If the victim accessed memory that mapped to the same cache lines, then the attacker will measure larger latencies for probes corresponding to those evicted cache lines.

5.3 CTR_DRBG

CTR_DRBG is a PRG design described in NIST SP 800-90A. It uses the encryption of an incrementing counter under a block cipher to generate outputs. The block cipher may be either 3DES with a 64-bit key or AES with a key of length 128, 192, or 256 bits. The design mixes in additional data at various stages. A derivation function (commonly the same block cipher under a different key) can optionally be used to extract entropy from the additional data. The implementations we

examined all used a derivation function.

Private State and Length Parameters. The private state $\mathcal{S}$ of the PRG is composed of the following:

- A key $K \in \{0,1\}^{keylen}$, with bit length $keylen$ matching that of the underlying cipher.
- A counter $V \in \{0,1\}^{\leq blocklen}$ that is incremented after each call to the block cipher, where $blocklen$ is the output length of the underlying block cipher.
- A reseed counter c that indicates when a reseed is required.

The PRG's nonce space $\mathcal{N}$ is $\{0,1\}^{seedlen}$ and the entropy space is $\{0,1\}^{seedlen}$ where $seedlen = keylen + blocklen$.

PRG Instantiation. CTR_DRBG's instantiate function takes as input an entropy sample I and an arbitrary nonce N chosen by the implementation, of equal length. It computes a temporary value t as the output for the derivation function applied to I and N. It then calls a subroutine update, outlined in Algorithm 14, with inputs $K = V = 0$ and t as the additional input. The initial state $S_0 = (K, V, c)$ consists of the outputs (K, V) from update, and reseed counter $c = 1$.

State Update. Each of CTR_DRBG's functions call a subroutine update, outlined in Algorithm 14, that updates the internal state. The routine's input is a key K, counter V, and additional data $addin$. In Lines 4 to 6 the function increments the counter V and appends the encryption of V under key K to a buffer $temp$. This process is repeated until $temp$ contains $seedlen$ bytes. The resulting buffer is then XORed with $addin$ (Line 7). Finally, in Lines 8 to 9 the function outputs the new key K' as the leftmost $keylen$ bits of the buffer, and new counter value V' as the rightmost $blocklen$ bits of the buffer, where $blocklen$ is the block length of the cipher.

Generating a Random Stream. A user generates output from the PRG by calling the generate function outlined in Algorithm 15. It takes as input the state S, the number of bits requested $nbits$, and a string $addin$, and outputs a string $nbits$ in length and an updated state S'. According to SP 800-90A, the $addin$ parameter "may be a means of providing more entropy for the DRBG internal state". This additional input is allowed to be public or private and may contain secrets if private.

Algorithm 14 Update. The update routine is called by the other routines and passes the current state (and potentially additional input) into the underlying block cipher. It outputs new state $S = (K, V)$ composed of key K and counter V.

1: **function** UPDATE($K, V, addin$)
2: $temp \leftarrow null$
3: **while** $\text{len}(temp) < seedlen$ **do**
4: $V \leftarrow (V + 1) \mod 2^{blocklen}$
5: $output\_block \leftarrow \text{encrypt}(K, V)$
6: $temp \leftarrow temp \| output\_block$
7: $temp \leftarrow temp \oplus addin$
8: $K' \leftarrow \text{leftmost}(temp, keylen)$
9: $V' \leftarrow \text{rightmost}(temp, blocklen)$
10: **return** K', V'

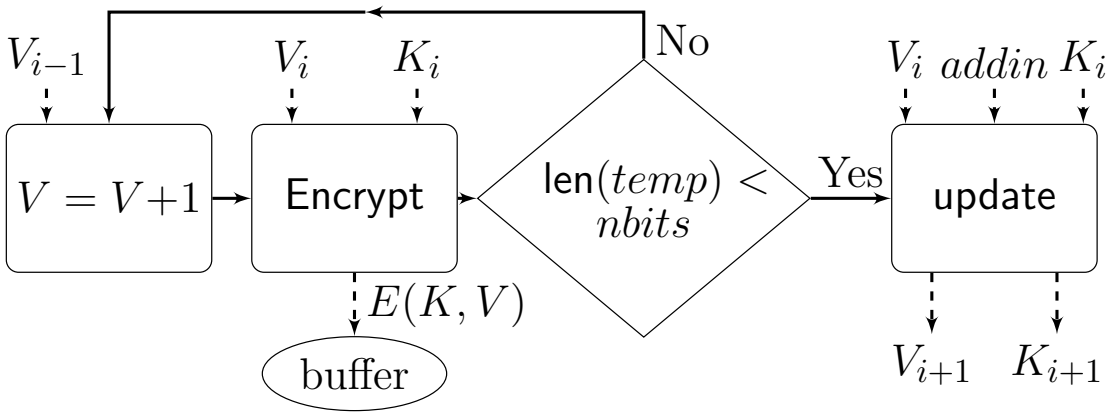


Figure 5.1: **Block Diagram of `CTR_DRBG`.** The central loop of the generate function increments the counter V, encrypts V under K, and adds the output to a buffer $temp$, repeating until $nbits$ have been generated. The function then updates the key and state before returning the contents of the buffer.

The specification notes that "if the additional input is kept secret and has sufficient entropy, the input can provide more assurance when recovering from the compromise of the entropy input, the seed or one or more DRBG internal states". However, the specification does not include requirements for either secrecy or entropy for $addin$.

The generate function first checks if a reseed is needed, and if so, throws an error[2] (Lines 3 to 4).

If the call included additional data $addin$, this data is first whitened by running it through the

[2]While the inclusion of an error message does not strictly adhere to our PRG definition, following [280] we assume inputs are valid and omit consideration of errors from our analysis.

derivation function, and then it is used to update K and V through a call to update (Lines 5 to 7). Otherwise, $addin$ is set to a string of zeros (Line 9). On each iteration of the loop on Lines 11 to 14, the counter V is incremented. V is then encrypted under K and the result is appended to the output buffer. This process is repeated until enough output has been collected. On Line 16 the function calls update with $addin$ to update K and V again before the reseed counter c is incremented (Line 17). The function returns the new key, state, reseed counter, and output.

If the attacker compromises the key K and counter V between Lines 11 to 14 and is able to guess $addin$, she can predict the new key K' and counter V'. She can then predict future PRG outputs as well as future values of K and V. Note that the same symmetric key is used to generate all of the requested output, and the key is only changed at Line 16 after all blocks have been generated. This observation is a crucial element of our attack, since a long output buffer gives the attacker many opportunities to extract K via a side-channel. Indeed, SP 800-90A specifies that at most 65 KB can be requested from the generator in a single call before a key change. This is presumably intended to limit a single state's exposure to an attacker. However, our work demonstrates that state recovery attacks within this limit are still viable.

Reseeding. The reseed function is intended to ensure that high quality entropy is mixed into the state as required. The reseed function takes as input additional input $addin$, an entropy sample I, and a state S that consists of the key K, counter V, and reseed counter c. It calls the update subroutine on a derivation function taken over I and $addin$, which updates K and V. Finally, it resets the reseed counter c to 1 and returns the new key, counter, and reseed counter.

5.3.1 Cryptanalysis of CTR_DRBG

DRBG Security Proofs. Woodage and Shumow [280] note that historical analyses of the security claims in SP 800-90A [67, 134, 156, 238, 247, 293] were limited by simplifying assumptions that were believed to be necessary due to nonstandard elements of the designs. Their analysis evaluated the standard's claims that the designs in the standard are both "backtracking resistant" and "prediction resistant". They provide robustness proofs that include backtracking and prediction

Algorithm 15 Generate. The generate function begins by throwing an error if the reseed counter exceeds the limit, and otherwise updates the state with the optional additional input, produces output by encrypting V under K, then increments V. The encryption and increment steps are repeated until the specified length of output has been produced. The state is then updated again, and the reseed counter is incremented.

function GENERATE($S, nbits, addin$)
  parse (K, V, c) from S
  if $c > reseed\_interval$ **then**
    return reseed_required
  if $addin \neq Null$ **then**
    $addin \leftarrow \mathsf{df}(addin)$
    $(K, V) \leftarrow \mathsf{update}(K, V, addin)$
  else
    $addin \leftarrow 0^{seedlen}$
  $temp \leftarrow Null$
  while $\mathbf{len}(temp) < nbits$ **do**
    $V \leftarrow (V + 1) \mod 2^{blocklen}$
    $output\_block \leftarrow \mathsf{encrypt}(K, V)$
    $temp \leftarrow temp \| output\_block$
  $out \leftarrow \mathsf{leftmost}(temp, nbits)$
  $(K', V') \leftarrow \mathsf{update}(addin, K, V)$
  $c' \leftarrow c + 1$
  return $S = (K', V', c'), out$

resistance for both the HMAC and hash constructions, but were unable to do so for CTR_DRBG and instead identified an attack against the prediction resistance property.

Attacking CTR_DRBG. [42] notes that to obtain prediction resistance after every random bit, the generate process must be called with only a single bit, incurring massive performance costs. Furthermore, SP 800-90A notes that "For large generate requests, CTR_DRBG produces outputs at the same speed as the underlying block cipher algorithm encrypts data". [280] use this observation to propose an attack scenario where large amounts of CTR_DRBG output is buffered, setting the stage for a side-channel attack on the block cipher key. They give the following procedure for recovering output at $t + 1$ from output r_t and key K_t that was compromised at time t:

1. **Counter Recovery From Output.** Attacker computes the state prior to the last update as $V'_t = \mathsf{decrypt}(K_t, r_t)$

2. **Generating S_{t+1}.** The attacker winds the generator forward by computing $K_{t+1}, V_{t+1} =$

$\mathsf{update}(K_t, V_t', addin_t)$

3. **Generating PRG Output r_{t+1}.** This state is now used to compute $r_{t+1} = \mathsf{generate}(K_{t+1}, V_{t+1}, addin_{t+1})$

Overall Attack Complexity. Assuming that the attacker has access to K_t, the complexity of this attack depends only on the difficulty of the attacker guessing $addin_t$ and $addin_{t+1}$. While a naïve attacker might attempt to enumerate the entire space of $2^{seedlen}$ possibilities, we show that in practice implementations use low-entropy or predictable data such as timestamps for this parameter. We observed implementations that required as little as 2^{21} work to find the correct values for both $addin$ values.

We next evaluate the practicality of this attack in the context of cache side-channel attacks on popular CTR_DRBG implementations and evaluate the impact of these attacks on the security of TLS.

5.4 State Recovery Attack

We show that the attack described by [280] is practical by recovering the CTR_DRBG state variables K and V via a cache side-channel attack against the underlying AES implementation. We begin with an overview of the popular implementations we analyze in this section.

5.4.1 Implementation Deep Dives

We examined the CTR_DRBG parameter choices of four implementations representing diverse use cases: the NetBSD operating system, the Fortinet FortiVM virtualized network device, and two versions of the OpenSSL cryptographic library. We identify limitations (if any) on the number of bytes that may be requested in a single call to the PRG, and highlight implementations' use of additional entropy. These parameters determine the viability of the state recovery attack. We also determine how frequently mandatory reseeds occur. If such a reseed incorporates sufficient entropy, an implementation may be able to 'recover' from compromise following the reseed.

FortiOS.

We analyzed FortiOS version 5, the second-most recent major release of Fortinet's network operating system for their hardware and virtual appliances. The operating system is an embedded Linux distribution with proprietary kernel modules that perform device-specific functionality. The software is used both on embedded devices and to operate VMs that perform virtualized network functions.

After reverse-engineering the operating system binaries, we discovered that FortiOSv5 replaces Linux's default implementation of `/dev/urandom` with the nist_rng library [154]. We note that [80] analyzed FortiOSv4 and found that it behaved similarly, replacing the system's default PRG with a FIPS certified design. Both FortiOS v4 and v5 use OpenSSL to provide basic cryptographic functionality, which in turn relies on `/dev/urandom`. While the original OpenSSL will use an AES hardware implementation if it is available, Fortinet's override makes OpenSSL fall back to an unprotected T-table-based AES implementation based on the nist_rng library.

Finally, the FortiOS CTR_DRBG implementation does not use additional entropy on each update and has no explicit reseeding. It returns an error code if more than 99,999 blocks are cumulatively requested from the instantiated DRBG over the course of its lifetime. It therefore lacks meaningful protection against state compromise.

NetBSD. The NetBSD operating system uses CTR_DRBG as the default source of system randomness. The kernel uses the nist_rng library with 128-bit AES as the default underlying cipher. We examined the kernel source code and single-stepped through a running kernel to verify our findings. As in the FortiOS case, the AES implementation is software-based with unprotected T-Table accesses, based on the nist_rng library. However, the OS limits requests to a maximum of 512-bytes from the PRG in a single call, increasing the difficulty of our proposed attack.

On each generate call, the state is updated using additional entropy from `rdtsc`, a high resolution CPU counter. NetBSD uses a 32-bit time-counter as additional entropy for each call to generate. NetBSD schedules an additional reseed after $2^{31} - 2$ calls to the PRG.

OpenSSL FIPS Module. We examined the OpenSSL FIPS module, which supports only

OpenSSL 1.0.2. This implementation is one of a small number of libraries that a manufacturer can use to be FIPS compliant without submitting the entire product for certification [110]. The module uses CTR_DRBG with a user configurable key length. Notably, while OpenSSL 1.0.2 FIPS uses hardware instructions for AES encryption, the CTR_DRBG implementation uses a lower-level interface for AES. Instead of selecting the best implementation available (as the AES interface used for encryption does), the lower-level interface used by CTR_DRBG uses a hand-coded T-Table AES implementation. On each generate call, the state is updated using the time in microseconds, a counter, and the PID. The FIPS module reseeds the PRG after 2^{24} calls to generate.

OpenSSL 1.1.1. The default PRG in OpenSSL 1.1.1, the most recent major release as of this writing, is a CTR_DRBG implementation forked from the OpenSSL FIPS code base. It defaults to 256-bit AES with user-configurable support for 128-bit and 192-bit AES. Unlike version 1.0.2 it *does* default to using hardware instructions for AES, so it is not vulnerable to our side-channel attack.

5.4.2 Side-Channel Attacks on AES-128

T-Table AES is the canonical target for cache side-channel attacks. Starting from [41] many works [119, 121, 145, 210, 299] have demonstrated key extraction from cache access patterns of table-based implementations.

Since CTR_DRBG uses T-Table AES as its underlying cryptographic primitive, we implemented the attack of [201] on the last encryption round of AES in order to extract the AES key from the CTR_DRBG's cache access pattern.

5.5 Cache Attack Details

In this section, we present the details of our state recovery attack. In the synchronous model of [210], an attacker observes the plaintext and is able to probe the cache state immediately before triggering an encryption with an unknown key. The attacker is also able to probe the cache state

immediately after each encryption. Observing the cache access patterns caused by the first round of AES during a few encryption operations is sufficient to recover the key [210].

Attacking the Last Round of AES. Working in the synchronous model of [56, 201, 210] we target the final round of AES, with attacker-observed ciphertext, rather than plaintext.

Implementations commonly use a different T-Table for the final round of encryption, allowing us to measure last round table accesses independently of earlier round accesses. Let q_i be the ith byte within the T-table, c_i be the ith ciphertext byte, and let k_i be the ith byte of the last round key. From the definition of T-table AES we know that $c_i = T[q_i] \oplus k_i$ where T is the final round table. Thus, an attacker who observes c_i and determines q_i by monitoring the cache for accesses can solve this equation for the key byte, yielding $k_i = c_i \oplus T[q_i]$.

Handling Missing Information. While the attack outlined above works when the attacker has perfect visibility over q_i and i, on a real system the attacker does not directly observe q_i. Instead, she identifies a contiguous set of bytes that are fetched into the cache together (a cache line, typically 64 bytes) and thus loses information about some of the least significant bits of q_i. On our test machine, each access corresponded to sixteen different possible values for q_i, as each final T-Table byte is stored four times, in a 4-byte integer, sixteen of which are in each cache line. Further, the attacker does not know i, as she does not know which cache access produced which ciphertext byte. Thus, in order to obtain a candidate key byte k_i, the attacker must somehow *guess* the value of q_i from the table indexes accessed in the last round as well as guess the missing 4 bits from q_i. As we expect about 11 distinct indexes to be accessed in the last round [201], this results in about $11 \cdot 2^4 = 176$ candidate values for each k_i, out of 256 possible candidates.

We notice however, that across many independent encryptions of different plaintexts under the same key, the correct value for every k_i, $i = 0, \cdots, 16$ should *always* appear in the list of candidates. In contrast, we expect incorrect candidates to be uniformly distributed. Thus, if an attacker sees a large number of encryptions, she can combine the information obtained from them to retrieve the

AES key. Let

$$\mathsf{hit}(q, j) = \begin{cases} 1 & \text{if } q\text{-th cache line accessed in } j\text{-th trace} \\ 0 & \text{otherwise} \end{cases}.$$

Following [201], the attacker counts cache hits that could correspond to each possible key byte value k from `0x00` to `0xFF` for each position i and stores the count in a table $\mathcal{S}$:

$$\mathcal{S}[i][k] = \sum_{j=0}^{n} \sum_{q=0}^{\ell} \sum_{b=0 \atop T[2^m \cdot q+b] \oplus c_i = k}^{m} \mathsf{hit}(q, j)$$

with ℓ the number of cache lines, m the number of bytes per cache line, and n the number of traces. As analyzed by [56, 201], the i-th byte of the last round key is then the value of k such that $\mathcal{S}[i][k]$ is maximal.

5.5.1 Obtaining Trace Data

We describe how we mount Flush+Reload against CTR_DRBG. We begin by recalling that the attack of [201] outlined in Section 5.4.2 requires the attacker to gather ciphertexts paired with the corresponding traces of the cache state following the encryption operation that produced that ciphertext.

Matching PRG Output. To recover the AES key, an attacker must match each ciphertext to a trace taken in the interval following the encryption that produced it, but before the subsequent encryption. In the synchronous model of [210] where the attacker triggers encryption operations directly, this matching is trivial. However, in our setting, a request for random bytes initiates a rapid series of encryptions. If the attacker's probes take a long time compared to an encryption operation, the attacker cannot easily interleave probes. This difficulty is exacerbated by the fact that encryptions vary in duration due to other system activity, making the naïve strategy of probing at evenly spaced intervals fail to produce matching traces and ciphertext pairs.

Tickers. In order to use the synchronous setting analysis of [210], we align traces and ciphertexts

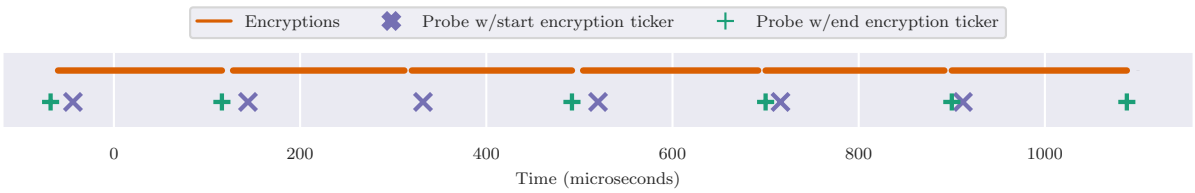


Figure 5.2: **Probe Alignment.** Probes do not perfectly align with the start and end of encryptions. Ideally, the start and end of an encryption probe should follow shortly after the start and end of an encryption. However, fluctuations in encryption duration and ticker timing accuracy cause misalignments. The problem is illustrated at $\approx 380\mu$s, where no end encryption ticker is visible, and at $\approx 900\mu$s where the end encryption ticker appears past the start of the next iteration.

by using what we term "tickers". Tickers are frequent cache probes that measure how long it takes to access cache lines that contain program instructions. A cache hit on a ticker gives the attacker a signal she can use to determine whether to probe the cache lines containing the T-Table used in the last AES encryption round. In our case, we set two tickers. The first ticker queries instructions at the start of the encryption code (as loaded into the process' address space), and the second queries instructions at the end of the encryption code. When either ticker is triggered, we probe the T-Table cache lines, ideally measuring cache state before and after encryptions.

Handling Drift. While tickers provide some signal, as depicted in Figure 5.2, variations in how the probe process is scheduled with respect to the victim process introduce imperfections in the signal provided by the tickers. Therefore, we also use timing heuristics to match traces to corresponding ciphertexts. More specifically, we iterate through the traces we collect, and keep a counter identifying the next ciphertext to be matched to a trace. Then, for each trace, we either match it to the current ciphertext and increment the counter or discard it. We base this decision on the accompanying ticker and timing data.

Our default case is to match the trace and ciphertext only if the ticker indicating a recent end-of-encryption event was triggered for that trace. However, to account for false negatives, the ticker indicating a recent start-of-encryption event is used if the interval between the last matched trace's timestamp and the current trace's timestamp exceeds a threshold we determined empirically. Similarly, if neither ticker was triggered, but the elapsed time is greater than another empirically-determined threshold, we match the trace and ciphertext.

Finally, using a ticker to determine when to start collecting traces may cause the attacker miss some traces belonging to the initial encryptions. We overcome this by running the key recovery algorithm with each possible set of matchings, for a small number of potential initial matches.

Overcoming Prefetching. Modern CPUs attempt to learn a program's cache access pattern and fetch data into caches before this data is actually needed. This data prefetching frustrates cache side-channel attacks against T-Table AES by reducing the extent to which a recorded cache hit corresponds to an actual–rather than predicted–access. If an entire AES T-Table is preemptively fetched into memory, a naïve cache side-channel attack will not succeed because the attacker will record cache hits for every memory line.

We mitigated the effect of the prefetcher by accessing cache lines in an irregular order, using the pointer chasing technique of [210]. This reduces the ability of the prefetcher to predict our cache accesses and therefore prefetch those lines.

Performance Degradation. If the time it takes to probe the cache state is too long relative to the duration of an encryption, an attacker will not be able to generate traces that accurately capture the state of the cache after each encryption. [24] showed that this difficulty could be mitigated by continuously flushing cache lines containing victim program instructions, so that the victim process was significantly slowed down. Flushing cache lines requires the victim to repeatedly fetch code from main memory, increasing access times. On our system, this slowed down the average duration of an encryption from 2 μsec to 32 μsec, giving us a large 34μsec window between successive last AES rounds for cache probing.

Validating Key Candidates. In our setting, plaintexts encrypted within a single call to the CTR_DRBG generate function are sequential integers, providing a simple test to determine the correctness of a recovered key. Given a series of ciphertexts and a candidate key, we validate the key by decrypting the PRG output and checking if the plaintexts form a successive series of integers. The final integer in the sequence is the last counter value before the state is updated at the end of the procedure. Given the recovered key K, counter value V, and a valid guess for $addin$ (if any is used), the subsequent state and output of CTR_DRBG can be computed by executing the update

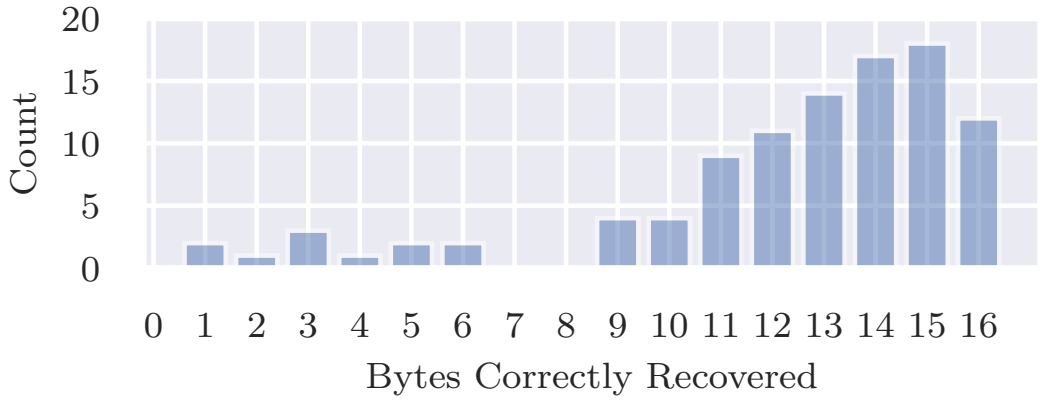


Figure 5.3: **Results Histogram with Prefetcher.** With the prefetcher enabled, our state recovery technique often recovers only a subset of the full 16-byte AES key. We here depict the frequency with which a given number of bytes were recovered, across 100 trials.

subroutine.

5.5.2 Evaluation of State Recovery

Attack Scenario. Our attack scenario is as follows. First, we assume an attacker who can execute unprivileged code on a target machine. Next, a victim process on the same machine uses CTR_DRBG and makes a call to generate, requesting about 2 KB of pseudorandom output. The attacker then uses Flush+Reload to monitor cache accesses during the AES operations inside the CTR_DRBG, and recovers the PRG state using the techniques described above. Our experimental setup instantiates this scenario in a concrete setting.

Targeted Software. We targeted OpenSSL 1.0.2 configured to use the nist_rng library with AES128 as the underlying block cipher for the PRG. Beyond the implementations mentioned in Section 5.4.1, the nist_rng library is used by libuntu (a C implementation of NTRUEncrypt), the XMHF hypervisor, among others. As mentioned before, the nist_rng library uses a leaky T-table based AES implementation and does not support `AES-NI` hardware instructions.

Hardware. We performed our experiments on a desktop equipped with an Intel i7-3770 Quad Core CPU, with 8 GB of RAM and 8 MB last level cache. The machine was running Ubuntu 17.10 (Linux Kernel 4.13.0). To ensure fair comparison, we fixed the initial state of the random

number generator to be the same uniformly sampled state for all the experiments described in this subsection.

Empirical Results. In 100 trials with the prefetcher disabled we were always able to recover the state, with an average false positive rate of 4.58% and false negative rate of 5.01%. As in Figure 5.3, with the prefetcher enabled our attack succeeded in 12.0% of trials with average false positive rate 28.5% and false negative rate 1.94%. State recovery took an average of 19s in both cases, with hardware as above.

5.6 Attacking TLS

In this section, we put our side-channel attack in context and show how recovering the PRG state from CTR_DRBG leads to the attacker being able to compromise long-term TLS authentication keys. We begin with necessary background on TLS and cryptographic primitives.

5.6.1 RSA Background

RSA is a public-key encryption method that can be used as a key exchange method in TLS 1.2 and earlier. RSA is not included as a key exchange mechanism in TLS 1.3.

RSA Cryptosystem. An RSA public key consists of a public encryption exponent e and an encryption modulus N. The private key is the decryption exponent d, which satisfies $d = e^{-1} \bmod \phi(N)$, where $\phi(N) = (p-1)(q-1)$ is the totient function for an RSA modulus $N = pq$.

RSA Padding. An RSA-encrypted key exchange message begins by padding the message using PKCS#1 v1.5 [155] padding as depicted in Figure 5.4. PKCS#1 v1.5 padding is not CCA-secure and has led to numerous cryptographic attacks against RSA in practice [51, 112]. Yet, it remains by far the most common padding method where RSA encryption is still used, including versions of TLS prior to 1.3.

Let m be a message to be encrypted, and $\mathrm{pad}(m)$ be the message with PKCS#1v1.5 padding applied. The encryption m is the value $c = (\mathrm{pad}(m))^e \bmod N$. The padded message $\mathrm{pad}(m)$ can

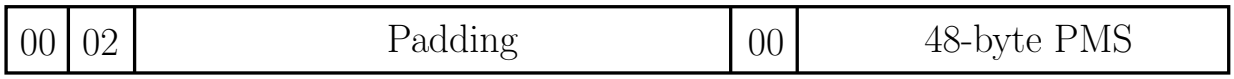


Figure 5.4: **PKCS#1v1.5 RSA encryption padding.** It appends a pseudorandom padding string to the message, together with some fixed bytes. The padding block is filled with $k - 3 - \ell$ non-zero bytes that are generated by a pseudorandom number generator, where k is the byte-length of the modulus and ℓ is the byte-length of the message to be encrypted.

be recovered by the decrypter by computing $\text{pad}(m) = c^d \bmod N$. In normal RSA usage, the decrypter then verifies that the padding is correctly formatted, and strips it off to recover the original message m.

RSA-PSS. RSA-PSS is a probabilistic signature scheme with a formal security proof [188]. The padding scheme is designed to avoid the flaws in PKCS#1 v1.5 padding. The scheme uses a sequence of hashing operations and mask generation functions to generate a padded message from a salt s and the input message m. The salt can in general be a maximum of $\text{len}(m) + hLen$ bytes in length, where $hLen$ is the length of the hash function output. RFC8446 (August 2018) [233] updates TLS 1.2, adding optional support for RSA-PSS signatures [195], but specifies that "the length of the Salt MUST be equal to the length of the [digest] output".

5.6.2 ECDSA

ECDSA is a standardized public key signature algorithm [158]. The global parameters for an ECDSA key pair include a pre-specified elliptic curve C with base point G of order n. The signer's private key is a random integer $1 < d_A < n$ and the public key is $Q = d_A G$.

To sign a message m, the signer generates a random integer nonce $1 < k < n$. The signature is the pair $r = (kG)_{\text{x}} \mod n$ and $s = k^{-1}(H(m) + rd_A) \bmod n$, where x represents the x-coordinate of an elliptic curve point, and $H(m)$ is the hash of the message m using a collision-resistant hash function H. Next, if an attacker learns the value of the nonce k, she can compute the private key d_A from the signature as $d_A = (sk - H(m))r^{-1} \bmod n$. We omit the details of the signature verification procedure, as they are orthogonal to our attacks.

5.6.3 TLS Handshake Protocol

We describe the TLS 1.0, 1.1, and 1.2 handshake protocols necessary for our attack. A TLS handshake begins with a ClientHello message containing a 32-byte nonce along with a list of supported cipher suites. The standard specifies that the nonce should consist of a four-byte timestamp and 28 bytes of raw output from a pseudorandom number generator. The ServerHello message contains a similar nonce and the server's choice of cipher suite. We specialize to the case of RSA key exchange with mutual authentication, an option that is enabled for higher-security deployments, for VPN-over-TLS, and other instances where the server needs assurance of the client's identity. For these cipher suites, the server then sends a Certificate message with its certificate chain, a CertificateRequest message, and a ServerHelloDone message. The client checks the server certificate, generates a 48-byte premaster secret (PMS) and encrypts it to the server's public key from the certificate. The PMS and padding formatting are shown above in Figure 5.4.

The client then sends the RSA-encrypted premaster secret in a ClientKeyExchange message, sends its own certificate in a Certificate message, and a CertificateVerify message containing a signature computed over a transcript of the handshake thus far, that proves it possesses the relevant private key.

Upon receiving the encrypted ClientKeyExchange, the server decrypts the message, verifies that the padding has the correct structure, and then extracts the premaster secret. After the server obtains the PMS, it verifies the client certificate. Both client and server then derive symmetric encryption and authentication keys by applying a key derivation function to the premaster secret and the client and server nonces. Both sides exchange messages to authenticate the handshake, then begin transmitting encrypted traffic.

5.6.4 Finding Randomness in TLS

The state recovery attack described in Section 5.4 required 1996 bytes of output from the random number generator. Thus, for our cache side-channel attack to work at the protocol level, we needed to find places in the handshake where a single random number generator call would request enough

output for an attacker to feasibly carry out state recovery. We evaluated the TLS protocol for potential sources of large or variable length randomness and settled on three possibilities: the ExtendedRandom TLS extension, RSASSA-PSS padding, and RSA PKCS#1 v1.5 padding.

ExtendedRandom TLS Extension. ExtendedRandom is a non-standard extension to TLS that was proposed to the IETF [111] to permit clients to request up to $2^{16} - 1$ bytes of randomness from the server. While our attacks (as well as those of [73, 80]) may have been able to make use of the increased output from the server's generator to recover secret information, there are no known implementations with a functional implementation of this extension [111].

RSA-PSS. We evaluated whether the generation of the random salt for RSA-PSS signatures provided a viable attack vector. Under the PSS specification, for a message of 2^{14} bytes, the maximum salt length allowed is 2016 bytes, or 126 blocks of PRG output, sufficient for our state recovery attack. However, since RFC8446 [233] restricts the salt length when PSS is used in TLS1.2, an attacker in this context cannot observe enough encryptions from calls to the underlying PRG.

PKCS#1 v1.5 Padding in TLS. When a TLS handshake is performed with an RSA cipher suite, the client generates the 32-byte premaster secret and encrypts it to the server's RSA public key, transmitting it in the ClientKeyExchangeMessage. If the malicious sever uses a 16384-bit RSA modulus, the client must generate 2,013 padding bytes, equivalent to 126 blocks of PRG output. This is a sufficient number of blocks for us to mount the state recovery attack. We thus target this mode of TLS.

5.6.5 Targeting TLS Clients

Unlike the attacks in [73, 74, 80], which compromise the server's PRG, we compromise the state of the PRG used by the TLS client, since the client is the party that generates the encrypted key exchange message. However, similar to those works, we use the recovered state to predict future outputs of the PRG. In our case, this allows us to recover the client's long-term authentication key.

Attack Overview. We assume the client connects to a malicious attacker-controlled server supporting TLS 1.0, 1.1, or 1.2 that uses RSA for key exchange, and that the client uses ECDSA

for digital signatures. We also assume that the attacker is capable of running unprivileged code on the client. Next, since the RSA PKCS padding generation procedure requires the client to generate pseudorandom bytes, the attacker can use the cache leakage traces collected during the generation of the PKCS padding to recover the client's PRG state via the method described in Section 5.4. With the client's PRG state successfully recovered, the attacker predicts the subsequent PRG output and thus is able to compute the ECDSA nonce that the client generates in the course of producing the digital signature for the CertificateVerify message. As outlined in Section 5.6.2, an attacker who knows the nonce used to generate an ECDSA signature can trivially recover the long-term private key used for client authentication, even if that key was generated in a secure manner. Recovering the signing key allows the attacker to impersonate the client. This may allow the attacker to access TLS-protected resources that are served only to an authenticated client. Our attack proceeds as follows:

1. **Victim Client Connects to an Attacker-Controlled Server.** A client with an ECDSA certificate is manipulated into visiting a web page with an attacker controlled script. The script initiates TLS handshakes with RSA cipher suites, to an attacker-controlled server. The server transmits an RSA certificate and requests mutual authentication.
2. **Recovering PRG State.** The client's software encrypts the TLS premaster secret to the server's RSA public key, generating PKCS#1v1.5 padding proportional to the size of the certificate. The attacker simultaneously conducts the state recovery attack explored in Section 5.4.
3. **ECDSA Signature Generation.** The client transmits its certificate and generates a random nonce to sign the CertificateVerify message using ECDSA. The client then transmits the signed CertificateVerify message to the server.
4. **Recovering the Client's Nonce.** The attacker conducts an offline search for entropy and additional input parameters used by the PRG to generate the client's ECDSA nonce. The attacker checks the nonce candidates by recomputing the ECDSA signature and validating it against the signature transmitted by the client.
5. **Key Recovery.** Finally, once the attacker successfully recovers the nonce, she is able to compute the client's ECDSA private key and can impersonate the client.

Performing Nonce Recovery. In order to perform Item 4, the state of the client's PRG must be advanced to the point at which ECDSA nonce generation occurs. The attacker can only wind the generator forward, and at each call to the generate and reseed functions the attacker must guess the entropy and additional input parameters. Thus, the attacker must pay close attention to implementation-specific details surrounding the ordering of calls to the PRG.

We illustrate this challenge using OpenSSL 1.0.2, which we use as our baseline implementation for the nonce-recovery attacks described in this and following sections.

5.6.6 Using PKCS#1 v1.5 in OpenSSL 1.0.2 for Nonce Recovery

We begin by describing the steps performed by OpenSSL during the establishment of a TLS connection to generate the random PKCS#1 v1.5 padding and ECDSA nonce. For ease of reference, we label each step of these processes. We then describe our end-to-end attack on OpenSSL 1.0.2.

1. **Initial Padding Generation.** The output of the PRG is fed into an n-byte buffer to be used for PKCS#1 v1.5 padding, where n is the length of padding required (in our case $n = 1996$). The state is updated twice, once before the bytes are generated and once after. State compromise occurs after the first call to update, but prior to the second.
2. **Padding Zero-Fill.** PKCS#1 v1.5 does not allow `0x00` bytes to be present in the random padding, so if there are z `0x00` bytes present in the PKCS output buffer, OpenSSL makes at minimum z more requests for output from the PRG, one for each byte. If any of these additional requests also result in a `0x00` byte, OpenSSL makes repeated requests to the PRG until the output is non-zero. The output from these requests is used to replace the null bytes in the padding to produce a valid non-null padding string under PKCS#1 v1.5. Within each request for random bytes, the PRG state is advanced twice. Both updates use the same underlying additional input.
3. **RAND_seed.** The ECDSA signing routine tries to reseed OpenSSL's RNG via `RAND_seed`. The SHA256 hash of the TLS handshake transcript is used as external entropy.
4. **RAND_add.** A call to `RAND_add` is made as part of `bnrand`, which is used to generate a random integer in a given range. Time in seconds is used as external entropy.

5. **GenNonce.** OpenSSL then generates the ECDSA nonce. Within the call to the CTR_DRBG generate function, the state is updated before the nonce value is finally produced.

Notably, Steps 3 and 4 call functions from the OpenSSL's PRG API, which as discussed in Section 5.6.7 do not always perform the expected function of reseeding or updating CTR_DRBG.

Causing a Large Number of Random Byte Generations. To perform the attack, the attacker must observe side-channel leakage during the generation of a large amount of randomness. Moreover, to recover the PRG's state, the attacker must learn the values of the victim-generated randomness. In our attack scenario, the attacker could cause a victim client to connect to the attack server using a malicious script served by an ad network on a website the user would otherwise normally visit. The attacker's malicious server is configured to support only RSA key exchange, and deliberately serves a 16534-bit RSA certificate, which is the maximum size that OpenSSL will support without throwing an error during the handshake.[3]

Next, while encrypting the premaster secret to the server's 16534-bit RSA public key to generate the ClientKeyExchange message for the TLS handshake, the client generates 1,996 bytes of PKCS#1v1.5 padding output, which, if using CTR_DRBG, gives the server an opportunity to conduct a side-channel attack against 125 AES encryptions. The attack server learns the value of the padding generated by the client by decrypting the padded RSA-encrypted message using its private key. The attacker then recovers the PRG state via the method described in Section 5.4, using the decrypted padding as the ciphertexts.

The Problem of Padding Zero-Fill. As noted above, to comply with the PKCS standard, there must be no `0x00` bytes in this padding. OpenSSL complies by first generating padding of the total length required, and then replacing each null byte with output from further calls to the PRG, each used to replace one zero byte. To encrypt to the malicious server's large certificate, OpenSSL generates 1,996 bytes of output for padding used as per Figure 5.4. In expectation, a ciphertext will have eight such `0x00` bytes that need to be replaced.

Next, for each `0x00` byte in the padding, the generator will have advanced an additional time.

[3]This is due to deliberate, hard-coded limits on message sizes that OpenSSL will accept, in the interest of preventing denial of service attacks [11, 12].

Since the attacker must brute force over the additional entropy added at each step, this increases the search space exponentially in the number of bytes generated in Step 2 to recover the final PRG state.

Bypassing RAND_add. However, as the initial 1996 padding bytes (generated during the initial Padding Generation step) have a uniform distribution over the 256 possible byte values, the probability of the padding not containing `0x00` is $(255/256)^{1996}$. We therefore expect that once every $2470 \approx 2^{11.3}$ TLS handshakes, the padding generated after the Padding 1 step will not require additional calls to CTR_DRBG in the Padding 2 step to produce a valid PKCS#1 v1.5 padding string. Combining this with our success rate for state recovery in Section 5.5.2, an attacker can be expected to recover PRG state once in every 2^{18} handshakes.

Nonce Recovery. With the PRG state recovered, the attacker proceeds to recover the client's ECDSA nonce. Since the nonce is generated in a new call to the PRG, the PRG is reseeded between our state recovery attack and nonce generation. An attacker must therefore obtain the values used during RAND_seed and RAND_add (Steps 3 and 4). The exact strategy of recovering these values is implementation-specific.

5.6.7 Implementation Choices and Nonce Recovery

In this section we describe how implementations use the $addin$ parameter, and how they explicitly reseed the generator. We describe how this impacts our ability to recover the value of $addin$ and entropy used during RAND_seed and RAND_add (Steps 3 and 4) in Section 5.6.6.

FortiOS. FortiOS does not implement `RAND_seed` and `RAND_add`, and instead relies on the nist_drbg library's internal reseed counter. As a result, RAND_seed and RAND_add do not cause a state update, reducing the attack complexity.

Furthermore, as FortiOS does not not use the optional additional input for calls to generate, the PRG can be wound forward without the offline search for additional input.

Custom Parameters for FortiOS. We modify the FortiOS implementation to illustrate that even if it were to improve its reseeding and updating strategies, the implementation can be attacked in the absence of sufficiently high-quality entropy input. To evaluate this, we modified the FortiOS

RAND_METHOD behavior to cause it to reseed during RAND_seed and RAND_add. Moreover, we added support for the additional data parameter, filling it with a microsecond timestamp to emulate OpenSSL FIPS.

OpenSSL FIPS. The OpenSSL 1.0.2 FIPS module also does not reseed the CTR_DRBG during RAND_seed and RAND_add. Instead, these calls add the entropy to a general entropy pool from which the PRG can later be reseeded with a call to reseed in compliance with SP 800-90A. We estimated the amount of entropy added during generate calls to be 12 bits.

OpenSSL 1.1.1. In OpenSSL 1.1.1 (the latest version at the time of writing) the maintainers rewrote much of the random number generation API. Due to the significant changes, this code was professionally audited twice [30, 226], both times finding only minor flaws with the PRG implementation. The implementation gathers additional input from a variety of sources and feeds it into an entropy pool. These include system event timing data, time, thread ids and output from the OS or hardware random number generators. Given this complexity, we did not estimate the entropy added in reseeding.

The ECDSA nonce generation mechanism in OpenSSL 1.1.1 was also improved. The nonce is generated from a hash of the private key, along with the transcript, and PRG output. The inclusion of secret data ensures that even if the PRG is compromised, the nonce cannot be recovered. Together, these measures preclude both state and nonce recovery.

NetBSD. The NetBSD kernel provides a source of random numbers that can be used by a TLS implementation. We consider an implementation that, like FortiOS, chooses to source random numbers for OpenSSL from the system PRG without modification. NetBSD provides additional data in to CTR_DRBG in the form of the least significant 32 bits of the rdtsc cpu counter. If this counter is not available, NetBSD uses the kernel's current time in microseconds, and further falls back to an integer counter if the kernel clock is not yet running. It is not possible for applications to add further entropy as NetBSD does not externally expose the reseed and update functions, and thus we do not model any additional entropy introduced by RAND_seed and RAND_add.

5.6.8 Evaluation

In this section, we empirically evaluate the difficulty of extracting ECDSA signing keys from TLS clients given the different implementation choices described in Section 5.6.7. In order to evaluate the effects of different parameter choices on attack complexity, we reverse-engineered the FortiOS CTR_DRBG implementation and reimplemented it ourselves using the nist_rng library, so that we could easily adjust parameters and hook it into implementations. We modeled attack difficulty against the other implementations by adjusting $addin$ and reseeding behaviors to match the descriptions in Section 5.6.7 of each implementation.

The Victim. For our victim TLS client, we used the sample TLS client code available in the OpenSSL documentation [208], configured to use mutual authentication and the nist_rng library with our choice of modeling parameters. We configured the client to authenticate using an ECDSA certificate with NIST P-256. For the ECDSA nonce, we used the raw PRG output, which matches the behavior of all implementations considered in Section 5.6.7, except OpenSSL 1.1.1.

The Malicious TLS Server. Our malicious server was the default OpenSSL tool, instrumented to dump TLS transcripts and ECDSA signatures to the filesystem, and configured to support only RSA key exchange cipher suites with a 16384-bit RSA certificate, the largest allowed key size as discussed in Section 5.6.6.

PRG State Recovery for Winding Forward. After a TLS connect to the malicious server, we use Flush+Reload to recover the PRG state, as described in Section 5.4. We then brute forced $addin$ and additional entropy to recover the ECDSA nonce, which consists of raw PRG output.

Our ability to wind the generator forward largely depends on the quantity of entropy injected between state recovery and nonce generation. Table 5.1 summarizes the entropy sources and brute force search space for each implementation.

Using Side-Channel Information for Space Reduction. We note that the attacker can use the same cache side-channel used for state recovery to reduce the search space over the additional entropy sources. By placing additional tickers and using timing data acquired during the state recovery process, we narrow down the set of timestamps or CPU counter values that we need to

Target	Sources	Search Space	Reduced Space	CPU Time
OpenSSL FIPS	time, PID counter	2^{24}	2^{21}	30 minutes
NetBSD	rdtsc	2^{32}	2^{21}	30 minutes
FortiOS	none	0	0	N/A
Custom Params	time	2^{48}	2^{43}	200 years

Table 5.1: **Nonce Recovery Search.** We calculated the search space for the attack described in Section 5.6.7. We extrapolated custom parameter timing from smaller timings on our test machine. OpenSSL 1.1.1 is excluded from experimental evaluation due to its non-vulnerable nonce generation mechanism. The full search space corresponds to the search complexity of all possible timestamps of that size, and the reduced space corresponds to a search of one standard deviation from the mean required search, starting from the approximate timing of the encryption operation we gained from our timing attacks, calculated across 100 trials.

search. We empirically evaluate the amount of data that can gained through the instrumentation already in place for conducting state recovery in Table 5.1 as well. We note the entropy brute forcing is highly parallelizable, because after the SSL/TLS handshake has been performed, each element of the search space can be tested independently.

Empirical Results. Our attack succeeded against FortiOS in negligible time (following state recovery) and against OpenSSL FIPS after thirty minutes (2^{21} work) using the hardware setup from Section 5.5.2. For the custom parameters the search space was beyond our computational capabilities, and we terminated our search after approximately one hour of searching. We tabulate our results in Table 5.1. While our experimental results are limited by our CPU's speed of $\approx 2^{22}$ elliptic curve scalar multiplications per hour, [270] achieve a rate of 2^{35} operations per hour using a commodity GPU. We therefore anticipate that using their setup, the custom parameters search (2^{43} work) would be completed within two weeks.

Handling AES-256. To demonstrate key recovery under the constrained set of known ciphertexts available in the TLS setting of Section 5.6, we implemented our attack using AES-128. In Section 5.7, we handle AES-256 in the SGX setting.

5.7 Attacking Full Entropy Implementations

The attack in Section 5.6 relies on both the ability to observe the output of the PRG and brute force the limited entropy of the state update. These are not fundamental requirements, however, as by carrying out a higher-resolution cache attack, we can remove these limitations, and develop a *blind* attack in which the attacker can observe the victim's cache access patterns but not the PRG output. Furthermore, our attack only requires observing two AES encryptions and thus is feasible even when the update entropy is too high to brute force.

However, to achieve this, we require a stronger side-channel adversary, one who can observe the cache accesses during AES encryption at a high temporal resolution. Past research [127, 265, 276] has demonstrated that a side channel adversary with control of the operating system may have access to high resolution cache data when co-located with victim running within an SGX enclave. This setting is congruent with the threat model for SGX enclaves.

We begin with background on SGX, cache attacks on SGX, and the SGX threat model (Section 5.7.1). We then present our novel differential cryptanalysis technique for exploiting side-channel information (Section 5.7.2). Finally, we evaluate our attack on an SGX port of the mbedTLS library (Section 5.7.4).

5.7.1 Secure Enclave Technology

Intel Software Guard Extensions (SGX) [126] is an extension of the x86 instruction set that supports private regions of memory called *enclaves*. The contents of these enclaves cannot be read by any code running outside the enclave, including kernel and hypervisor code. This in theory allows a user-level process to protect its code and data from a highly privileged adversary, such as a malicious operating system or hypervisor.

Cache Attacks on AES Inside SGX. Although SGX protects the enclave from a malicious OS, it renders enclaved code *more* vulnerable to side-channel attacks. Specifically, the cache attack of Section 5.4 can only observe the overall access pattern over an entire encryption. However, when

the victim runs in an SGX enclave, a malicious operating system can obtain much finer temporal resolution. This allows us to observe cache accesses after each of the 16 accesses to the AES T-tables in each of the encryption rounds [127, 265].

Threat Model. Following previous work [61, 190, 264, 286], in this section we assume a root-privileged attacker who controls the entire OS. This is in agreement with Intel SGX's threat model, where an enclave guarantees confidentiality and integrity, even against a malicious OS and hypervisor. Unlike the attack described in Section 5.6, we do not assume that the enclaved TLS client is willing to connect to a malicious attacker-controlled server, or uses imperfect PRG reseeding.

5.7.2 Differential Cryptanalysis of CTR_DRBG in the Presence of side-channel Leakage

We provide the additional details about AES required for the differential attack. AES is a substitution-permutation cipher [49] that operates in a sequence of rounds on a 128-bit internal state S. Each round mixes the state and combines the mixed state with a round key. For a plaintext x, the initial state is $S_0 = x \oplus K_0$. Each consecutive round calculates $S_{j+1} = P(S_j) \oplus K_{j+1}$, where P is the state mixing function and K_j is the key for the j^{th} round. In efficient software implementations, the mixing step is commonly implemented using four T-tables. Each byte of the state selects one entry from a T-table and, since the T-table entries are 32 bits wide, each state bytes affects four consecutive bytes in the mixed state. For example, we can calculate the first four bytes of state S_{j+1} by:

$$S_{j+1,0..3} = T_0[S_{j,0}] \oplus T_1[S_{j,5}] \oplus T_2[S_{j,10}] \oplus T_3[S_{j,15}] \oplus K_{j+1,0..3} \tag{5.1}$$

As before, our cache attack targets accesses to these T-tables. Because we cannot distinguish between entries in the same cache line, the cache leaks only the four most significant bits (MSBs) of each byte of the state in each round. Let $\langle\rangle_U$ denote setting the four least significant bits of each byte to zero, then the leakage on byte k is $L_{j,k} = \langle S_{j,k}\rangle_U$. With a known plaintext x, we can use $L_{j,k}$ to recover the 4 MSBs of every byte of K_0 because $\langle K_{0,k}\rangle_U = \langle x_k\rangle_U \oplus L_{0,k}$.

Unfortunately, in our blind attack we do not know x. Consequently, we cannot learn information on K_0 from the leakage of the first round. Instead, we use the known difference between the plaintexts used in consecutive rounds of AES-CTR to recover the AES state. From the state, we can recover the keys, plaintexts, and ciphertexts. This is in close correspondence to the changes targeted in differential fault attacks [46]. We develop a similar analysis leveraging side-channel leakage, as the basis of our attack.

Notation. We use the following notation:

1. $T_0..T_3$ is the array of 4 AES T-Tables, where $T_i[j]$ is the value in location j of Table i.
2. $\langle x \rangle_U$ denotes the value of x with the lower four bits (nibble) in each byte set to 0.
3. $L_{i,j,k}$ is the value leaked from the cache attack for byte k of round j in trace i. The leaked value is only the 4 MSBs and the lower nibble is always 0.
4. $S_{i,j,k}$ is the real value of the state byte k of round j in trace i. $G_{i,j,k}$ is our current guess for this byte.
5. $R\Delta_{j,k}$ the value of the differential $S_{0,j,k} \oplus S_{1,j,k}$, and $\Delta_{j,k}$ is our current guess for this value.
6. $L\Delta_{j,k} = L_{0,j,k} \oplus L_{1,j,k}$ (lower nibble is always 0).
7. $K_{j,k}$ is the key value of byte k of round j.

Differential Analysis. By analyzing the difference between the state of two encryptions, we can recover state information that is independent of the round keys. In AES-CTR, for two consecutive plaintexts $x0$ and $x1$ we know that $x1 = x0 + 1$, so with probability (255/256) the two plaintexts only differ in the last byte by some value Δ_{ctr}. As the state of round 0 is simply the plaintext XOR

Algorithm 16 Find possible guesses for the last state 0 byte.

```
1: function LastState0Byte(L_{0,0,15}, Δ_{0,15}, LΔ_{1,0..3})
2:     GuessList0 ← Empty
3:     for Nibble ← 0 to 2^4 − 1 do
4:         G_{0,0,15} = L_{0,0,15} ⊕ Nibble
5:         Δ_{1,0..3} = T_3[G_{0,0,15}] ⊕ T_3[G_{0,0,15} ⊕ Δ_{0,15}]
6:         if ⟨Δ_{1,0..3}⟩_U = LΔ_{1,0..3} then
7:             GuessList0.append(G_{0,0,15}, Δ_{1,0..3})
8:     return GuessList0
```

with K_0, the plaintext difference is preserved and $R\Delta_{0,15} = \Delta_{ctr}$. Using Eq. (5.1) we get:

$$\begin{aligned}
S_{0,1,0..3} =& T_0[S_{0,0,0}] \oplus T_1[S_{0,0,5}] \oplus T_2[S_{0,0,10}] \\
& \oplus T_3[S_{0,0,15}] \oplus K_{i+1,0..3} \\
S_{1,1,0..3} =& T_0[S_{1,0,0}] \oplus T_1[S_{1,0,5}] \oplus T_2[S_{1,0,10}] \\
& \oplus T_3[S_{1,0,15}] \oplus K_{i+1,0..3} \\
=& T_0[S_{0,0,0}] \oplus T_1[S_{0,0,5}] \oplus T_2[S_{0,0,10}] \\
& \oplus T_3[S_{0,0,15} \oplus R\Delta_{0,15}] \oplus K_{i+1,0..3} \\
L\Delta_{1,0..3} =& L_{0,1,0..3} \oplus L_{1,1,0..3} = \langle S_{0,1,0..3} \oplus S_{1,1,0..3} \rangle_U \\
=& \langle T_3[S_{0,0,15}] \oplus T_3[S_{0,0,15} \oplus R\Delta_{0,15}] \rangle_U
\end{aligned} \tag{5.2}$$

As $\langle S_{0,0,15} \rangle_U = L_{0,0,15}$ we only need to try the 16 options for the lower four bits until we find a value that satisfies Eq. (5.2) and recover $S_{0,0,15}$ (see Algorithm 16). As $R\Delta_{0,15}$ is unknown, we run Algorithm 16 with each possible value to retrieve the full set of candidates. However, as $R\Delta_{0,15} = x_0 \oplus x_0 + 1$ only eight candidates are possible. The full key and plaintext recovery procedures are described in the full paper [79].

Using Three or More Traces. The above attack requires only two traces to compromise the CTR_DRBG state. However, any request for PRG output causes at least three encryptions, and four when AES-256 is used as the underlying block cipher.

Our attack can be trivially extended to use the extra encryptions to more efficiently eliminate

candidates, which aids in reducing the impact of noisy measurements.

5.7.3 Fine Grained Cache Attack

To generate the required traces, an attacker with OS level privileges (root) monitors cache access through Prime+Probe. The attack obtains fine-grained temporal resolution through a controlled-channel [286] attack. A controlled-channel attack involves disabling the present bit on the enclave's page tables, which by necessity are handled by the OS. By marking the page containing the T-Tables as not-present, the attacker forces an asynchronous enclave exit upon access to the table, thereby transferring control to the attacker controlled OS.

Since all of the T-Tables lie in the same page, the attacker must 'toggle' between the accesses by performing a controlled-channel attack on a page access that occurs in between each T-Table access. We use the page containing the topmost frame of the stack for this, as the mbedTLS implementation must first read the index into the T-Table from the stack before each access.

Unlike the T-Table addresses, however, the location of the stack is randomized by the SGX loader. We overcome this by first using a controlled-channel attack to force an enclave exit upon entrance to the AES function. We then mark all pages in the enclave, except for the thread control structure (TCS), saved state area (SSA), and the pages containing code, as not-present. We then resume execution within the enclave; since the first instruction of the function prologue is `push_rbp`, control immediately returns to our segmentation fault handler. Within the handler, we can determine which page caused the segmentation fault, which in this case will be the page containing the top of the stack.

In this manner, we learn the location of the stack for use in our controlled-channel attack. Then, by forcing enclave exits upon each access to the T-Tables, we use a last-level cache (LCC) Prime+Probe attack to measure each T-Table access separately.

To reduce the amount of noise in the attack, we used Intel's cache allocation technology to partition a single way of the LLC to both the victim and attacking process, and used the `isolcpus` kernel boot parameter to isolate them on a single physical core.

Related Attacks. [235] also demonstrate a blind attack on AES. However, they consider a particularly powerful attacker who is able to generate arbitrary faults in the key schedule. [148] attacked counter mode encryption with an unknown nonce, but required 2^{16} consecutive block encryptions. [236] also showed a blind attack on counter mode encryption targeting the authentication MAC.

5.7.4 Evaluation

The Victim. We performed our experiments on a Lenovo P50 laptop equipped with 16 GB of RAM and an Intel i7-6820HQ CPU clocked at 2.7GHZ with a 8MB L3 cache. The laptop was running Ubuntu 16.04.

Similar to [283], we demonstrate the viability of the differential attack against mbedTLS-SGX [298], an SGX port of the widely-used mbedTLS library. To the best of our knowledge, mbedTLS-SGX is the only library currently available that features a function SGX-based HTTPS client.

Attack Procedure. We demonstrate an end-to-end attack on a connection between the TLS client and `www.cia.gov`, with all of the client's cryptographic operations taking place within the enclave. We first mount a Prime+Probe attack to recover the CTR_DRBG state used to generate the 256 bits of the ECDH ephemeral private key (a total of five AES256 encryptions of an incrementing counter). Using the recovered private key, we were able to calculate the premaster key and subsequently decrypt the HTTPS communication. The details of the side-channel attack are left to Section 5.7.3.

Results. Due to high noise levels in some traces, our attack successfully recovered the enclave's CTR_DRBG state in $\approx 36\%$ of our 1000 trials. The online phase, during which we mount the LCC Prime+Probe, takes less than two seconds. The offline phase in which we recover the state of the PRG and decrypt the TLS stream took negligible time.

After recovering the PRG state, we recovered the TLS symmetric encryption keys and GCM IVs, and subsequently decrypted the HTTPS request.

Attack Complexity. The complexity of the attack is dominated by calculating the set of key candidates. Generating each candidate requires $4 \cdot 2^{16}$ T-Table look-ups for each trace. Eliminating

Design	Certificates
CTR_DRBG	1694 (67.8%)
Hash_DRBG	906 (36.3%)
HMAC_DRBG	922 (37.0%)
Total implementations	2498

Table 5.2: **CMVP-Certified Uses of DRBG Designs.**

candidates by decryption required negligible work.

We tested the number of remaining candidates in each step experimentally; both in the noise free case (using a simulation over 500 random keys) and in the noisy case (1000 SGX attacks). Performing the attack with two traces yields $1.13 \cdot 2^9$ and $1.52 \cdot 2^{11}$ key candidates for the noise free and noisy cases respectively. Running the analysis with three traces immediately yields the single candidate correct in each list in the simulated environment. However, noise in the real-world setting required us to provide an extra fourth trace to narrow the analysis to a single candidate.

5.8 Impact

In order to evaluate the impact of our findings, we scraped a public database of security certificates released under NIST's Cryptographic Module Validation Program (CMVP).

Government Certification. The CMVP allows vendors to certify that their cryptographic modules meet minimal requirements to sell to the United States and Canadian governments.

In order to comply with FIPS 140-2, implementations must use one of the PRGs described in SP 800-90A.

Certification can apply narrowly to a specific product model, or apply to a product line. Most major vendors of network devices and operating systems certify their products.

Database Scraping. We scraped a public facing database of CMVP certifications on May 13, 2019 to assess the potential impact of our findings. Our results are tabulated in Tables 5.2 to 5.3. CTR_DRBG was the most popular design, supported by 67.8% of the implementations in the database. Of 2498 implementations present, 1694 (67.8%) supported CTR_DRBG. Of these 461

Cipher	Certificates
3DES	19 (1%)
AES-128	1163 (69%)
AES-192	598 (35%)
AES-256	1227 (72%)
CTR_DRBG	1694

Table 5.3: **Counter DRBG Certificates.** A majority of the 1694 certified implementations using CTR_DRBG use either AES-128 or AES-256. An implementation may support more than one of these modes.

(25%) exclusively supported AES-128, 1163 (69%) supported AES-128 along with other ciphers, and 1227 (72%) supported AES-256. The CMVP database also permits modules to certify whether prediction resistance is enabled for the DRBG implementation. Of the 1694 total implementations that supported CTR_DRBG, 66 provided no information about prediction resistance, 618 supported use of the DRBG in either mode with the default unspecified, 433 explicitly enabled prediction resistance, and 577 did not support prediction resistance. Among the CTR_DRBG implementations, 85 did not use a derivation function and 1137 did not support an alternate DRBG algorithm.

5.9 Discussion

Limitations. Our results rely on a victim's use of T-Table AES, which has long been known to leak information via side-channels. However, as illustrated in this work, T-Table AES is still used by many modern implementations. In the non-SGX setting, our TLS attack requires code execution on the client, and further succeeds only after thousands of handshakes. This potentially allows for detection of an ongoing attack. While we demonstrate our SGX attack against the only library that provides a working end-to-end example of an HTTPS client, the Intel-supported SGX-SSL cryptographic library [142] (which does not provide support for TLS) uses SGX's hardware-based `RDRAND` PRG and therefore is not vulnerable to a T-Table based attack.

Countermeasures. CTR_DRBG's flaws, both theoretical and practical, suggest that implementations need to take great care when choosing this design. Where FIPS compliance is required,

HASH_DRBG and HMAC_DRBG give better security guarantees [280]. Where CTR_DRBG cannot be replaced, implementers should use AES hardware instructions, limit the quantity of data that can be requested in a single call, reseed frequently, and populate $addin$ with high quality entropy, to provide defense in depth against our attacks. In general, constant-time code should be used for all cryptographic applications, unless hardware support (e.g., AES-NI) is available.

Mismatches Between Theory and Practice. Significant effort has been dedicated to formalizing PRG security properties and designing provably secure constructions. However, theoretical analyses of many of the most commonly-used designs in practice (the Linux RNG [91], CTR_DRBG [280]) have found that these designs do not meet basic security properties such as robustness against state compromise. Unfortunately, implementers are often hesitant to adopt countermeasures without a concrete demonstration of vulnerability.

The Fragility of ECDSA. The fragility of DSA and ECDSA in the face of random number generation and implementation flaws has been repeatedly demonstrated in the literature [62, 290]. It is inevitable that a random number generation failure would compromise a single session or a signature, but DSA/ECDSA are particularly vulnerable to compromise of long-term secrets. Deterministic ECDSA, defined in RFC6979 [222], is the recommended countermeasure.

Future of FIPS. FIPS 140-3 is expected to contain requirements for side-channel mitigations from the inclusion of NIST SP 800-140F, which has yet to be issued and becomes effective in September 2019. FIPS 140-2 CMVP certifications will continue to be issued at least through 2021 [204]. This is a promising step towards widespread deployment of side-channel-resistant cryptography; however, it remains to be seen how improved requirements for certifying modules will feed back into the design and standardization of more secure primitives.

Using `RDRAND` without a PRG. Using the built-in CPU PRG to mitigate concerns with software PRGs is not a panacea. In several SGX ports we have reviewed (including Intel's official port for OpenSSL [142]) the software PRG was replaced with calls to the `RDRAND` instruction. While using the CPU's generator avoids software side-channels, the existence of hard-to-discover bugs in PRGs integrated into CPUs [173, 282] mean this feature is better used as one of many sources of entropy

for a provably secure software PRG.

CHAPTER 6

Hard Drive of Hearing: Disks that Eavesdrop with a Synthesized Microphone

Security conscious individuals may take considerable measures to disable sensors in order to protect their privacy. However, they often overlook the cyberphysical attack surface exposed by devices that were never designed to be sensors in the first place. Our research demonstrates that the mechanical components in magnetic hard disk drives behave as microphones with sufficient precision to extract and parse human speech. These unintentional microphones sense speech with high enough fidelity for the Shazam service to recognize a song recorded through the hard drive. This proof of concept attack sheds light on the possibility of invasion of privacy even in absence of traditional sensors. We also present defense mechanisms, such as the use of ultrasonic aliasing, that can mitigate acoustic eavesdropping by synthesized microphones in hard disk drives.

6.1 Introduction

Magnetic hard disk drives (HDDs) continue to persist in everything from legacy laptops to server racks. Because of their critical role in a wide variety of applications, hard drives make an appealing target for both cyber criminals and nation states alike. Kaspersky describes how an advanced hacking organization, dubbed the "Equation Group," developed malware that reflashes its host machine's hard drive's firmware to gain advanced persistence [10]. Other researchers have shown how even modestly funded adversaries can compromise HDD firmware to create highly stealthy

backdoors for subverting machines [297].

While these incidents have motivated researchers to investigate the threat of hard drive malware from the digital side, little attention has been given to the cyberphysical side-channel attack surface exposed by a hard drive's mechanical components. Due to the complexity of their read/write mechanisms and the granularity at which a hard drive must track its head, hard drives possess certain characteristics that respond to the oscillations in air pressure caused by acoustic waves. This raises the possibility of using a hard drive as an unintentional microphone, thereby allowing attackers to eavesdrop on speech in the vicinity of the drive.

In this paper, we demonstrate how an adversary could leverage HDD firmware resident malware to extract human speech by measuring the offset of the read/write head from the center of the track that it is seeking. Modern hard drives use this offset, known as the Position Error Signal (PES), in a feedback control loop; the microprocessor takes the PES (Figure 6.1) as input for actuating the read/write head by use of a voice-coil motor (VCM) [287].

For both read and write operations, the head can tolerate deviation from the center only on the order of nanometers. Accordingly, PES measurements are taken at a very fine granularity. These extremely precise measurements are sensitive to vibrations caused by the slightest fluctuations in air pressure, such as those induced by human vocalizations.

Extracting speech from the PES, however, is complicated due to a weak signal-to-noise-ratio (SNR). Imperfections in the eccentricity of the platters, thermal drift, and turbulence from the rapid rotation of the disks all contribute to a large quantity of noise in the signal [285]. Through a mixture of digital filtering techniques in both the time domain and the frequency domain, however, we have managed to sufficiently clean the signal such that human speech can be completely reconstructed under certain conditions.

To prove the existence of this acoustic side-channel, we physically probed the PES directly from the hard drive under operation. This is sufficient to explore the possible information leakage available to an attacker with firmware access.

We validate our side-channel attack by performing signal analysis and using Shazam to recognize

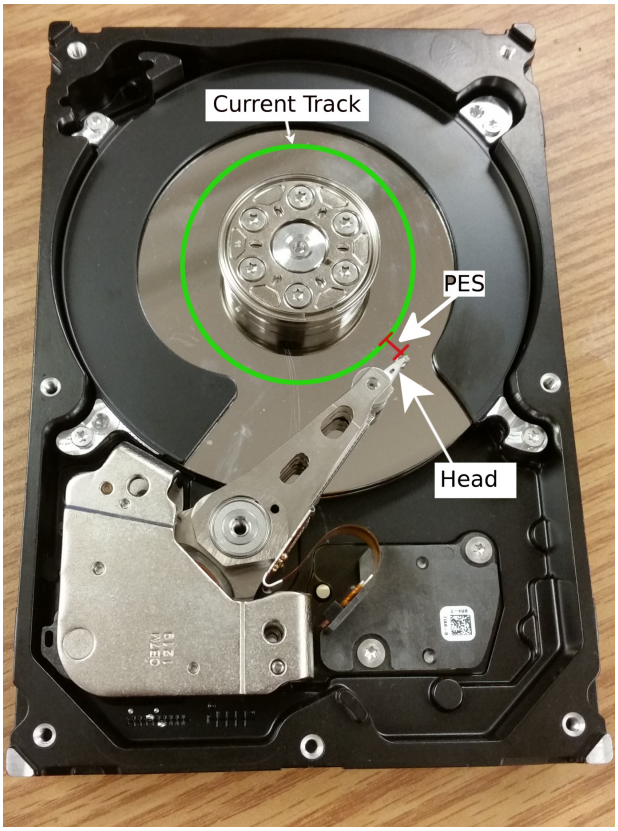


Figure 6.1: **Position Error Signal (PES) diagram.** The PES measures the offset of the read/write head from the center of the track.

songs recorded through the hard drive's PES. Furthermore, we evaluate the intelligibility of the reconstructed speech signal through objective measures of speech quality [5] [6].

Our attack sheds light on the potential for invasion of privacy even in the absence of traditional sensors. Security conscious individuals may make efforts to remove or disable any and all sensors (i.e. placing tape over a laptop's camera, removing the built-in microphone, etc.), but they will often neglect the possibility of how non-sensors can be synthesized into intrusive sensors: in our case, hard disk drives into microphones. Our contributions towards addressing the cyberphysical side channels in HDDs are the following:

- We model how the mechanical components in hard disk drives lend themselves into becoming unintentional sensors.
- A proof of concept attack demonstrates how an attacker can use a hard disk drive as a

microphone to extract human speech. We evaluate our side-channel by (1) performing qualitative signal analysis, (2) using the Shazam service to identify songs recorded through the hard disk drive, and (3) quantitatively analyzing recovered audio through objective measures of speech intelligibility.

- We discuss defenses in both software and hardware, and make suggestions on how manufacturers and end users can mitigate risks with ultrasonic masking and sound dampening.

6.2 Background

Synthesizing a microphone from a hard drive relies on the similarities between its mechanical components and those of a microphone.

6.2.1 Acoustic Waves and Microphones

Human speech is entirely encoded in acoustic waves that propagate as oscillations in air pressure. As such, microphones record audio signals by measuring these small changes in air pressure. They can accomplish this through use of a diaphragm that oscillates back and forth with the fluctuations in air pressure induced by acoustic waves; the microphone then produces as output a voltage in proportion to the distortion of the diaphragm. This analog value, taken as a function of time, then represents the oscillations in air pressure that compose the acoustic wave.

6.2.2 Hard Drive Mechanics

Hard drives read and write from the magnetic platters by making use of a small magnetic slider, called the read/write head, that floats just 5 nm above the surface [302]. This head must follow the center of the track with extreme precision, and in the case of high performance drives, can deviate from the center of the track by no more than 7 nm [9] [83]. As such, HDD's make use of a high precision feedback control loop wherein the offset from the center of the track, called the Position

Error Signal (PES), is fed back to the microprocessor so that it can actuate the read/write arm with a voice coil motor (VCM). As shown in Figure 6.1, the hard drive controller computes the PES by reading out magnetic signals known as "servo bursts" from special sectors on the disk, called servo sectors. Each track contains the same number of servo sectors, and within a given track, the servo sectors are laid out in even intervals [137]. This gives the PES its periodicity, with the frequency proportional to the angular velocity of the disk.

Keeping the read/write head within the allowable margins is a challenging task due to a plethora of noise sources, all of which contribute to disk run-out, which is a measure of how much the slider's rotational path differs from a perfect circle. Disk run-out due to imperfections in the eccentricities of the platters, along with turbulence from the spinning disks, creates white noise that falls out across the spectrum. Furthermore, expansion in aluminum components due to thermal drift can result in alterations in PES 300 to 400 times greater than the width of a track [9]. Finally, acoustically induced vibrations, which are the signals we wish to measure for our attack, also work to push the head off track. Thus, the read/write head assembly approximately functions as a crude diaphragm.

6.3 Eavesdropping

This section describes our assumptions on the attacker, and details how such an adversary might carry out our acoustic eavesdropping attack.

6.3.1 Threat Model

We assume the attacker can reflash the HDD's firmware; this is necessary because the ATA protocol does not expose the PES. An attacker can gain this privilege in one of two primary ways: reflashing it entirely through software, or by intercepting HDDs before they reach the end user.

With reflashing, the attacker can use traditional methods such as binary exploitation, drive-by downloads, or phishing attacks, to compromise the operating system upon which the HDD is attached. Then, the attacker abuses his root privileges to update the firmware over the SATA

connection. This is what Kaspersky says the Equation Group accomplished with its Grayfish trojans [10]. By reverse engineering an off-the-shelf HDD, Zaddach et al. [297] demonstrate how even modestly funded attackers can practically carry out such an operation. We emphasize that even on a device that typically has a microphone installed in it, having root access on the device does not necessarily give the attacker access to a microphone; a privacy-minded user could have disabled the microphone in the BIOS, or even have physically disconnected it. Security conscious individuals have even taken to modifying their devices by adding a switch that only closes the microphone's circuit when switched on [1].

The adversary can also gain firmware access by conducting man-in-the-middle (MITM) attacks against users attempting to download legitimate firmware updates for their hard drives. Even when manufacturers deploy security sensitive downloads with SSL/TLS, they are still potentially unsafe, as the POODLE [193], LOGJAM [15], and DROWN [32] attacks against the SSL and TLS protocols themselves have demonstrated. In a related attack vector, our adversary can employ social engineering and spear phishing techniques in order to direct naive users to attacker controlled websites, where they then proceed to download malicious firmware updates.

The attacker can gain access to the HDD's firmware by intercepting the HDD and planting malware on the device before it ever reaches its destination. For instance, nation states have reportedly intercepted routers [113] and CD-ROMs [107] to plant malware. Furthermore, physical access to HDDs at the factory itself places HDDs at risk of tampering; in 2007, Seagate Maxtor drives manufactured in China shipped with preinstalled malware [4]. In this scenario, even full disk encryption does not mitigate our attack. Since full disk encryption aims to prevent data theft by storing the encryption keys separately from the data on the disk, the hard drive malware would be unable to tamper with user files and subsequently gain access to the microphone. They are, however, still able to record audio and write it to disk, to be recovered at a later time.

In both cases, we make the assumption that no digital signatures are in use. We believe that this is a mild assumption, as only a few of the newer models of HDDs implement this security feature; none of the HDDs that Zaddach et al. [297] reverse engineered signed their firmware updates.

Given that our attack is largely OS independent, we make no assumptions on which operating system is in use. The attacker's goal is simply to reconstruct human speech spoken in proximity to the hard drive.

6.3.2 Speech Exfiltration

There are two primary ways by which the attacker can exfiltrate the data recovered through our attack: (1) through a reverse shell over the internet, and (2) by storing the audio on disk and physically recovering it later.

Reverse Shell Exfiltration. Once the attacker has firmware access, he can leverage the compromised drive to exfiltrate the recorded audio over the internet, thereby leaving no physical trace of the attack. When the hard drive, which acts as the basis of trust, is under the attacker's control, there are a number of ways by which an attacker can accomplish this. One such example: by modifying the `.bashrc` and `/etc/shadow` files on a Linux machine, the malicious drive can trivially establish a reverse shell with root privileges for the remote attacker. Zaddach et al. [297] demonstrated the practicality of such a backdoor by implementing a similarly stealthy covert channel, in which a remote attacker was able to read and write arbitrary blocks to a back-doored hard drive over the internet.

Now that the attacker has privileged arbitrary code execution, it is then a simple matter for the malicious firmware on the hard drive to stream the captured audio over its SATA or SCSI interface to the attached machine, which the remote attacker then extracts over the internet via the reverse shell.

Recording to Disk. If the reverse shell is not an option, perhaps because the hard drive is installed in an air-gapped system, the malicious firmware can instead store the audio on the disk itself. The firmware can accomplish this covertly by writing the captured audio to the System Area, which can contain over 400 MB of unused space [40]. The advantage of using this reserved space is that its blocks are not exposed via any external interface, thus rendering it hidden to anti-virus and forensics tools alike. Kaspersky explains how the Equation Group used the System Area for this

purpose, so as to covertly store data to be recovered at a later time [10].

6.4 How a Hard Drive Hears

Given the physical structure of a hard drive's read/write components, we hypothesize that the PES measurements from the head can be interpreted in the same manner that the values out of a microphone's analog to digital converter (ADC) are. That is, an acoustic wave's oscillations in air pressure will displace the head in the same manner that acoustic waves oscillate a microphone's diaphragm. Thus, the PES readings will directly approximate an acoustic wave's instantaneous amplitude.

Since it is unclear what the exact relation is between this acoustic interference and the PES, we tested this hypothesis using an HDD and measured its PES under various external acoustic inputs. We chose the Seagate Barracuda 7200.12 1TB hard disk as our target because, after examining all of the Seagate F3 drives, it was the only one that exposed the pin by which we extract the PES.

6.4.1 Measuring the Position Error Signal

In our threat model where the adversary has firmware access on the hard drive, the microcontroller reads out and extracts the full 16-bit PES. Patents from Seagate [72] and Western Digital [263] [85] describe how the HDD's main microcontroller is responsible for reading the PES offsets and then computing the required adjustments, thereby demonstrating that the PES is indeed available to the HDD's main microcontroller. We verified this to be the case with the Barracuda 7200.12 by commanding the hard drive controller to output various aggregate statistics on the PES over a specified number of revolutions, and even received ASCII art representations of the PES over the serial diagnostic port, which connects directly to the hard drive controller.

In our case, however, we instead measured the PES by partially reverse engineering the hard drive's debugging interface, which exposes sufficient information for exploring the side-channel leakage available to an attacker with complete firmware access. So as to illustrate some of the

limitations imposed by this approach, we briefly describe the process for pedagogical reasons.

We first attached to the hard drive's serial diagnostic port. The interface allowed us to command the drive to output 8 bits at a time of the 16-bit PES on the "AMUX" pin. While it is likely that we would obtain a higher signal to noise ratio if we had access to all 16 bits, we found that 8 bits was sufficient as a proof of concept. To find this pin, we first probed all exposed pads on the HDD's printed circuit board and observed the output under an oscilloscope. Then, to narrow down our list of candidates for the desired pin, we computed the expected frequency by the following reasoning: if we know that for a particular HDD, n is the number of servo sectors per track, then the read/write head will pass over n servo sectors per revolution, and thus report n evenly spaced samples per period. Given that the platters rotate at f_r revolutions per minute (RPM), we can then compute the frequency f_s as $f_s = f_r \cdot n$. In the case of the Barracuda 7200.12, with parameters $f_r = 7200$ RPM and $n = 288$, we can compute the rate at which our signal is sampled by the following:

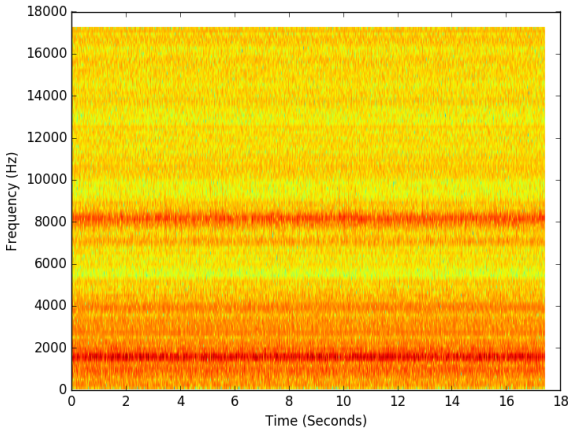
$$\begin{aligned} f_s &= f_r \cdot n \\ &= 120 \text{ Hz} \cdot 288 \\ &= 34,560 \text{ Hz} \end{aligned}$$

As such, we expect to see a square wave of frequency 34,560 Hz on the output of the "AMUX test pin." After probing all exposed pins, we found only one such candidate; we confirmed our guess by toggling the PES output and verified that the output of the pin also toggled on and off. With the AMUX pin identified, we were able to read out 8 bits of the hard drive's real-time PES.

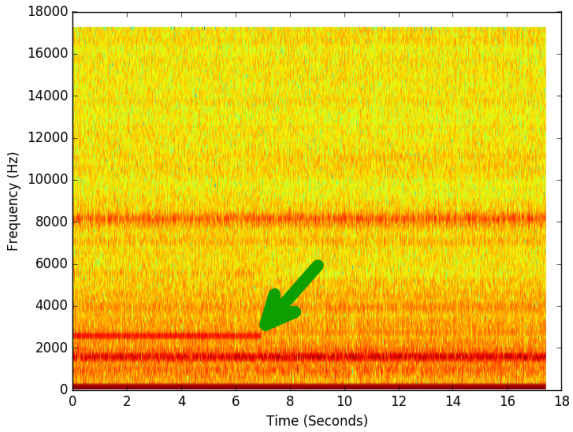
6.4.2 Sampling Rate

Using the frequency f_s from the previous subsection, we find that the sampling rate comes out to 34.56 kHz.

The Nyquist-Shannon sampling theorem states that, given a signal where the highest frequency



(a) **Baseline spectrogram of the position error signal.** The baseline exhibits heavy, persistent noise just above 8 kHz and below 2 kHz.



(b) **Spectrogram of the position error signal while the drive is subjected to a 2.5 kHz tone for the first 7 seconds.** As indicated by the green arrow, it is easy to see when the tone stops.

Figure 6.2: **Comparison of PES spectrograms with and without being subject to an external tone**

component is f, sampling the signal at a rate of $2f$ is sufficient to reconstruct the signal. Thus, the hard drive's sampling rate can perceive signals of up to $\frac{34{,}560}{2}$ Hz$=17.28$ kHz.

This covers almost all of the audible range in humans, which spans from 20 Hz–20 kHz; furthermore, given that the Plain Old Telephone System (POTS) uses a sampling rate of just 8 kHz, our sampling rate is more than sufficient to eavesdrop on human speech. Having an even higher sampling rate than POTS offers the additional advantage of reducing the chances of noise above the Nyquist threshold aliasing in the band of our expected signal. While telephony system designers do not typically expect a significant amount of very high frequency noise to be present, there is no reason to assume the same for a hard drive's PES. In fact, as the spectrogram of the PES in Figure 6.2a shows, a sampling rate of just 8 kHz would have resulted in the dark red line just above 8 kHz aliasing over our signal in the 80–260 kHz band. This is known as "out-of-band" noise.

6.4.3 Sampling Granularity and Resolution

Because of the hard drive's need for extreme precision in actuating the read/write arm, the PES is sampled at both a very fine granularity and a high resolution. Technical specifications from Seagate

show that PES is a 16-bit value, with a granularity equal to just $\frac{1}{2^{12}}$ of the width of one track.

If we desire even more resolution, we can leverage the high sampling rate to oversample and obtain a higher effective resolution [180]. In practice, however, we find no need for the additional resolution.

Due to the manner in which we extract the PES from the hard drive, we can only read out 8 bits per sample. This is problematic; if we extract the low 8 bits of the PES we see substantial clipping under normal operation, which leads to degradation of the signal and the introduction of undesirable harmonics. If we examine only the high 8 bits, minor disturbances in the PES are masked, as is most of our signal. Thus, we must make a trade off between granularity and distortion of the signal. Experimentally, we have found that using the 3rd least significant bit through the 10th least significant bit yields the best compromise, as it is the lowest set of bits such that no clipping occurs. While this effectively reduces our granularity by a factor of four, signal clipping is minimal. Under our threat model, however, the attacker would have access to the full 16-bit signal, and would likely be able to recover a cleaner signal; we leave this possibility to future research.

Given the sampling rate of 34.56 kHz and a 16-bit resolution, SATA 3.0's native transfer rate of 6.0 Gbit/s is more than sufficient for streaming captured audio.

6.4.4 Linearity

In order to function like a microphone, our hard drive blackbox must approximate a linear time invariant (LTI) system. The actuator's attempts to minimize the PES by use of the voice coil motor have the potential to compromise the "sinusoidal fidelity" of the system; that is, an input to the system of a given frequency may not yield an output of equal frequency [252]. To test this property of our hard drive, which is essentially a blackbox, we subjected it to varying frequencies and observed an increase in the corresponding bands in the PES's frequency spectrum. The PES responded strongly to 2.5 kHz, which guided our decision to conduct further tests with this frequency. The spectrogram in Figure 6.2b confirms our assumption of linearity.

	Bare Drive			Enclosed Drive		
	bottom	side	top	side	front	top
Response	13 dB	9 dB	16 dB	10 dB	21 dB	18 dB

Table 6.1: **dB increase in corresponding band while playing 2.5khz tone at 90 dBA from different directions.**

6.4.5 Noise

As can be seen in Figure 6.2, the heaviest bands of noise are concentrated just above 8 kHz and just below 2 kHz. Since these bands don't overlap with the fundamental frequencies of adult speech, this noise can be removed by using linear filtering techniques.

Even after using such filters, however, a substantial amount of noise remains in the signal's frequency band. We speculated that noise due to disk runout is periodic with the rotation of the disk and thus can be reduced with signal subtraction in the time domain. After looking at PES averages for a given servo sector over many revolutions, however, we found that the baseline noise is uniformly distributed and as such requires a different approach. Our non-linear filtering techniques are described in Section 6.5.2.

6.4.6 Directionality and Orientation

To investigate how the hard drive's orientation and the direction of the oncoming waves affected our measurements, we played a 2.5 kHz tone at equal intensities from all five exposed sides while the bare hard drive assumed three different orientations: face up, face down, and on its side. We then repeated the measurements, only with the hard drive housed in the CSE-M35T-1B external HDD enclosure. The high frequency of the 2.5 kHz tone aids in this measurement, due to how acoustic waves diffract. Lower frequencies "bend" around objects more easily; by using a high frequency tone, we minimize this diffraction, and thus isolate the directionality of the tone.

While observing the response in the frequency domain, we found that the hard drive's PES responded very poorly to tones coming from the sides of the hard drive. Given that the PES is a measurement of the read/write head's horizontal, and not vertical, displacement, this is surprising.

It is likely that the weak response is then due to the hard drive's thin profile from the side capturing less energy from the oncoming wave, as the wave vibrates less of the drive's surface area. The PES's response to tones coming from the bottom and top of the hard drive was substantially stronger, as the results in Table 6.1 show.

We did not observe any noticeable differences in the three different orientations of the hard drive. This is unsurprising, given that user manuals for hard drives commonly state that as long as the drive is placed on a flat surface, orientation does not impact the drive's operation.

6.5 Speech Recovery

This section details how an adversary with access to the PES can begin to extract human speech.

6.5.1 Audio Extraction

If HDD components do indeed function sufficiently as a microphone, the PES values will roughly approximate instantaneous air pressure readings; furthermore, the PES sampling rate of 34.56 kHz is more than sufficient to extract speech, given that the Plain Old Telephone System uses a sampling rate of 8 kHz. This allows us to treat the string of PES readings as linear pulse-code modulation values, corresponding to samples of an audio signal. We can simply write the PES values into a WAV file with the appropriate sampling rate, and then use digital signal processing algorithms designed for speech recognition.

6.5.2 Digital Signal Processing

The unprocessed signal taken from the HDD is incredibly noisy, and without further processing, the raw audio is completely unintelligible. However, by exploiting certain spectral properties of human voice and making use of both linear and non-linear filtering algorithms, we demonstrate that the signal can be cleaned up to a sufficient extent to parse human speech.

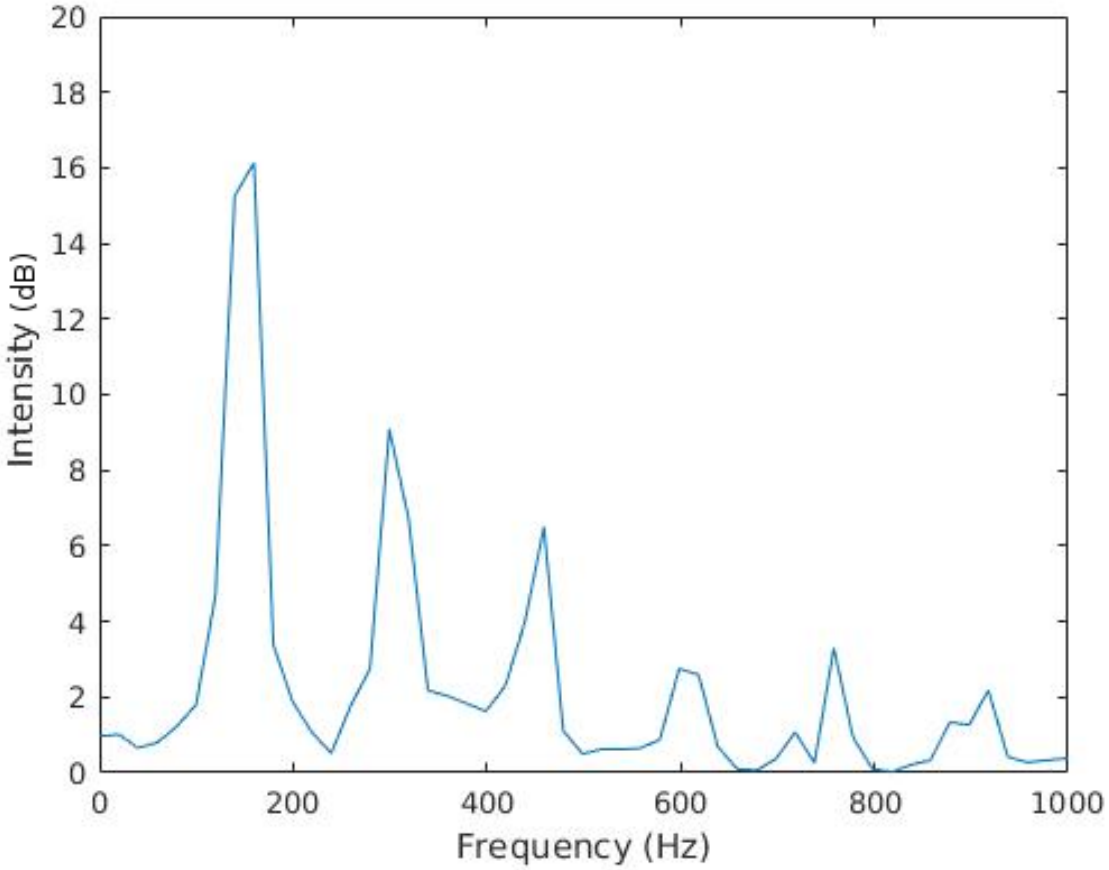


Figure 6.3: **Frequency spectrum of the male speaker's voice, taken over a 50 ms window.** When observing sufficiently short segments, the spectrum of human voice exhibits peaks in smaller sub-bands. A time-variant filter can pass these peaks while rejecting the troughs.

Filtering. The naive, initial approach to any noise reduction problem is to examine the frequency domain and cut out the unwanted frequencies. The spectrogram in Figure 2 reveals heavy, persistent noise in the PES under standard operation, particularly around 8.1 kHz and 1.8 kHz. Since an average male's fundamental frequency lies between the range of 85 Hz to 180 Hz, while a female's lies between 156 Hz and 255 Hz [260] [36], we can easily remove these with a linear bandpass finite-impulse-response filter that passes only frequencies between 80 Hz and 260 Hz. Even after linear filtering, however, there remains a substantial amount of noise that overlaps with our passband, and as such can not be dealt with by a linear filter.

Given that the white noise and our signal overlap in both the time and frequency domains, we

make use of a non-linear time-variant filter. To attenuate the in-band noise, we use a technique known as spectral noise gating. As illustrated in Figure 6.3, while the spectra of human speech and white noise may overlap over the length of a recording, the human speech's energy is largely concentrated in separable bands when observed over a very short frame, i.e. 16 ms. This is in contrast to white noise, which remains spectrally flat when looking at both short and long segments.

Thus, with spectral noise gating we take advantage of the briefly separate spectrums of noise and signal and use a time variant filter that operates over short windows, passing frequencies above the noise floor and rejecting others [252].

6.6 Evaluation

Despite the fact that hard drives were not designed to function as microphones, the mechanics of their internal components allow them to sense acoustic waves to some degree. To understand the limits of what a hard drive can hear, we explore three major questions:

- What are the physical limits of the PES in detecting acoustic waves? (clear information leakage at 75 dBA)
- How difficult is it to automatically recognize structured sound patterns (e.g., music)? (Shazam recognizes at 90 dBA)
- How difficult is it to recover unstructured conversations? (yields speech recordings recognizable to human ear at 85 dBA)

Our evaluation answers these questions by performing signal analysis on the input and output to the hard drive; by using objective measures of speech quality for spoken phrases; by testing how well the Shazam service can recognize music recovered from the PES; and by analyzing the feasibility of advanced techniques such as acoustic arrays of hard drives.

6.6.1 Experimental Setup

In all of our experiments, the hard drive lies enclosed in the CSE-M35T-1B SuperMicro HDD enclosure, which comes attached with a San Ace 92 9GV0912P1H03 8500 RPM 42 Watt fan [7], mimicking the typical usage of an external hard drive or server rack. In this section, in addition to the baseline testing, we present results in which we drive the fan separately at max power so as to simulate an exaggeratedly loud internal fan that may be present in certain desktops or datacenters; furthermore, we conduct experiments while continuously writing a large file to the HDD.

When playing our audio samples at 75 dBA, which is comparable to a loud conversation, we are able to recover muffled recordings; however, in order to yield a large signal to noise ratio (SNR) for the purpose of demonstrating our proof of concept attack, our audio samples are played at a volume of 85 dBA. While this is louder than what can be expected in most practical scenarios, we aim only to demonstrate the presence of such a side-channel, and expect that an attacker using state of the art filtering and voice recognition algorithms can substantially amplify the channel's strength.

In our setup shown in Figure 6.4, we are careful to physically separate the hard drive from the speaker so as to eliminate any possible mechanical coupling. When a speaker projects a tone, the rapid oscillation of its diaphragm vibrates the speaker itself, and transfers this energy into the platform upon which it rests. This energy can then potentially transfer into the hard drive, and thus vibrate the read/write head. We accomplish this separation by suspending the speaker from a ruler, which is mounted on a clamp attached to a separate table. This assures that the transmission of our audio is solely the result of acoustic waves, and not from the vibration of a shared surface.

The samples are recordings of Harvard sentences [140], which are specifically designed to feature phonemes at the same frequency that they naturally appear in spoken English. The female sample is from list 1, while the male sample is from list 57. The specific audio samples are taken from the Open Speech Repository [6].

All tests were conducted using the Seagate Barracuda 7200.12 1 terabyte hard disk, due to the fact that it was the only drive that has the "AMUX pin" exposed on the top side of the printed circuit board. We leave the question of whether or not certain drives yield better results to future work.

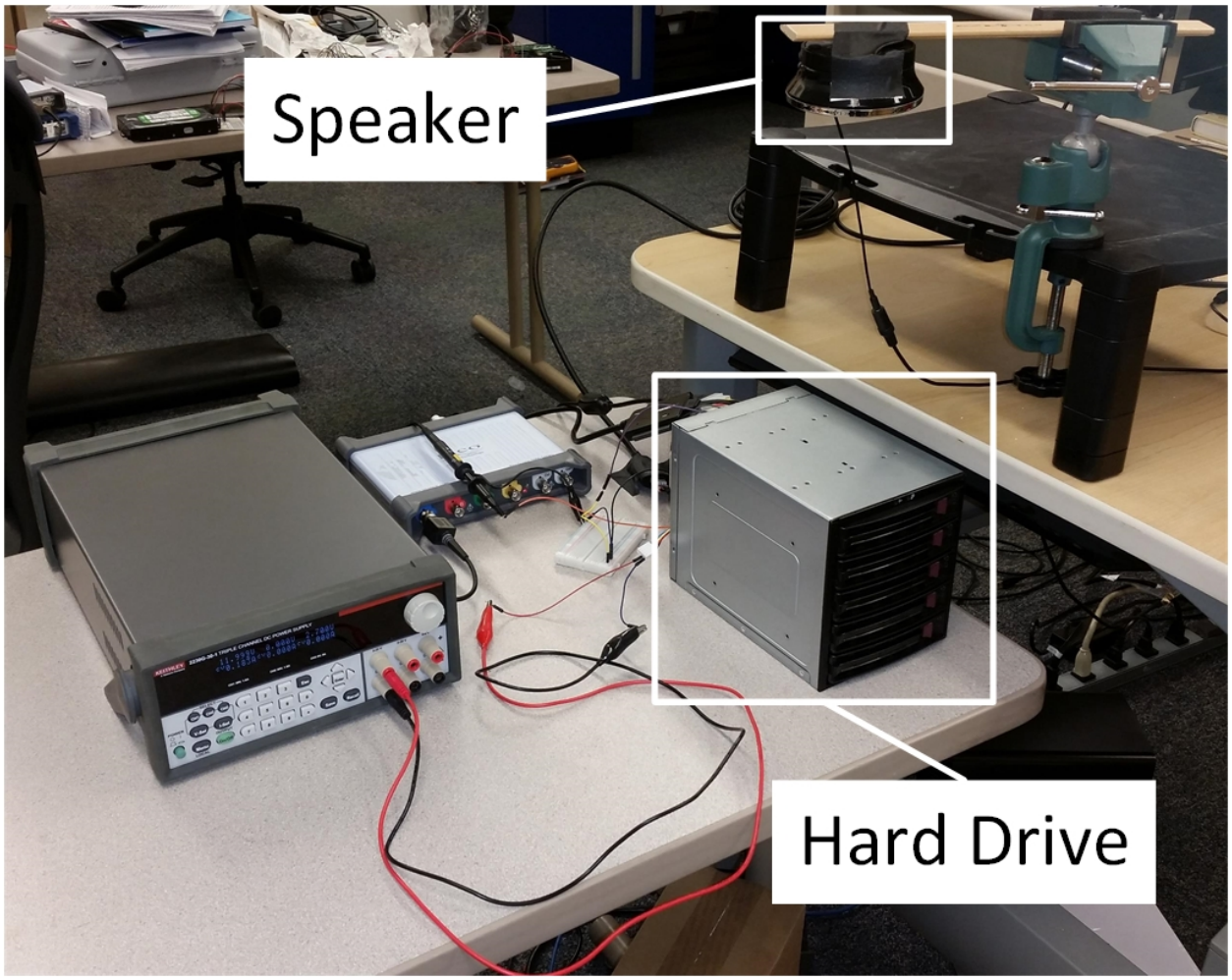


Figure 6.4: **The speaker is positioned 10 inches directly above the HDD enclosure. It is suspended from a ruler that is attached to a neighboring, but not connected desk, so as to eliminate any mechanical coupling between the hard drive and the speaker.**

6.6.2 Signal Analysis

A simple side-by-side comparison of the time domains of the original signal and the one extracted by the hard drive presents clear evidence of information leakage. In Figure 6.5, both signals are from the male sample; the signal on the top is the filtered and processed audio extracted from the hard drive, while the bottom is of the original audio sample. The speech, as annotated, was taken from list 57 of the Harvard sentences [140]:

Paint the sockets in the wall dull green. The child crawled into the dense grass. Bribes fail where

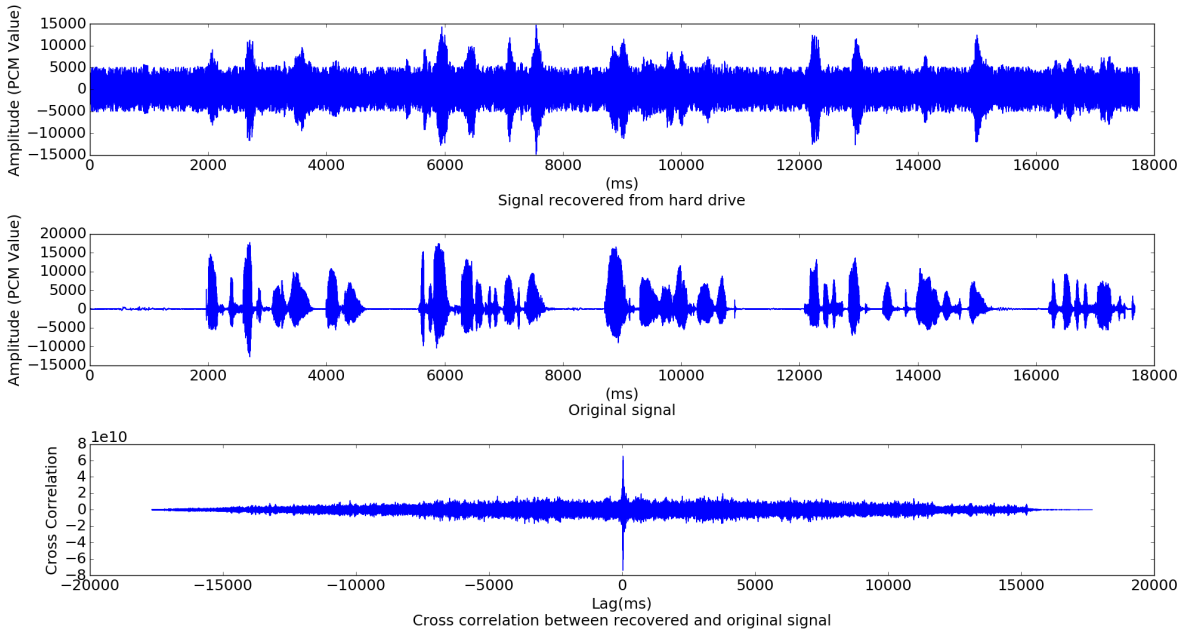


Figure 6.5: **Time domain comparison of the original audio and the recovered audio after it has been cleaned through digital signal processing techniques.** The recording is of the first two Harvard sentences from list 57, spoken by a male. The bottom graph plots the cross correlation between the two signals, with a sharp spike at lag=0 seconds. This indicates a strong correlation between the two signals.

honest men work. Trample the spark, else the flames will spread.

The spikes in amplitude seen in the recovered signal clearly align with corresponding spikes in the original. To quantify the similarity between the input and the output to the hard drive, we also computed the discrete cross correlation between the two time series, as shown in Figure 6.5. The cross correlation, as defined by the following formula:

$$(f \star g)[n] \stackrel{\text{def}}{=} \sum_{m=-\infty}^{\infty} f^*[m]\, g[m+n].$$

is a measure of the dot-product of the two signals as a function of time displacement, also known as lag. As such, the sliding dot product will hit its maximal value when the peaks align with peaks, and troughs with troughs. The large spike when the lag is equal to zero indicates a strong correlation between the unshifted original and recovered audio. This is to be expected, as the two audio samples

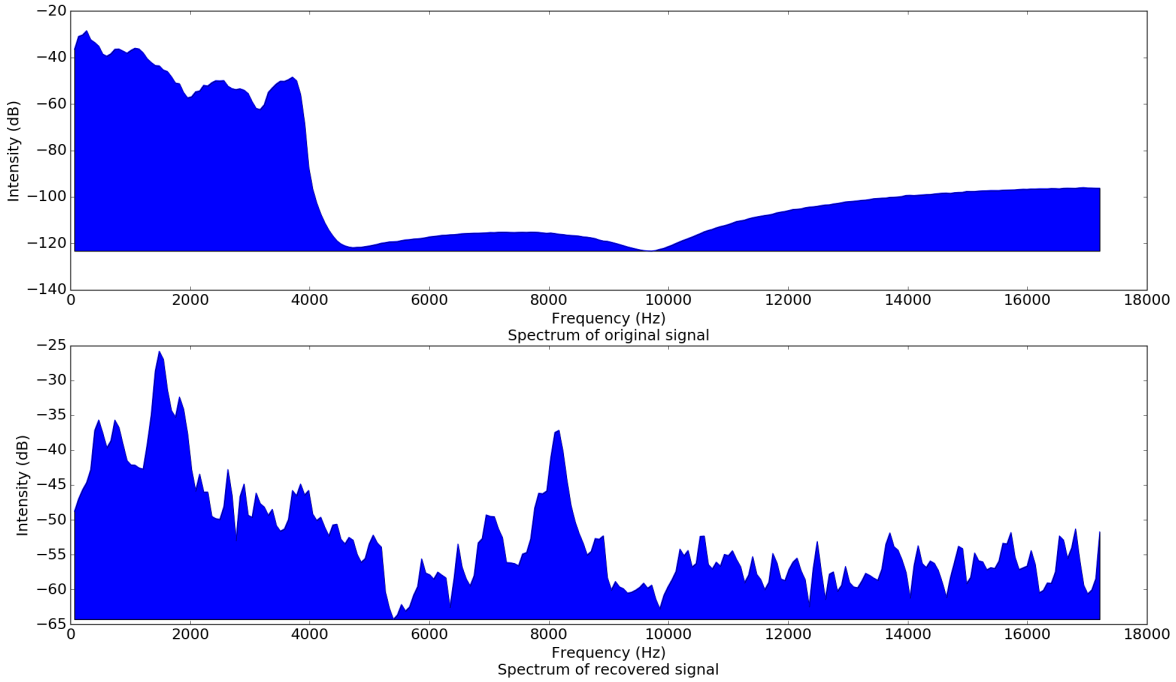


Figure 6.6: **Frequency spectra of the original signal, on top, and of the recovered signal, on the bottom.** Note the treble heavy response.

are already time aligned. Furthermore, this result agrees with our assessment of the hard drive microphone's linear properties. Billing's book on nonlinear system identification [47] states that cross correlations between the input and output of nonlinear systems can yield false positives, even in the presence of strong correlations.

We can observe the response of the hard drive in Figure 6.6, which displays the power density spectrums of both the source signal and of the recovered audio through the hard drive. The lack of any energy beyond 4 kHz is due to the fact that the source audio is a WAV file that was recorded at a sampling rate of 8 kHz. Thus, the presence of such noise in the recovered audio is a combination of noise from the hard drive itself and artifacts of the spectral noise gating process. Such artifacts can arise from discrimination errors, wherein bands within the signal fall too close to the noise floor and are subsequently filtered out.

Also of note is the severe attenuation of the lower frequencies compared to the higher bands. This treble heavy response is likely a consequence of vibrational resonance. A system's natural

frequency is the frequency at which the system vibrates when the vibrating force is removed, and resonance occurs when the input force oscillates at a frequency that divides the natural frequency. At these resonant frequencies, input to the system results in particularly high amplitude responses.

In Dutta's dissertation [92] on hard disk performance in the presence of noise, he utilized finite element analysis to show that the hard drive components that affect the read/write head respond most to frequencies in the 2–8 kHz range, corresponding to their natural frequencies. We then conclude that resonance is largely responsible for the hard drive's treble heavy response.

Previous research has demonstrated that a substantial amount of information can be recovered simply by observing very coarse measurements of human speech [281] and using pattern recognition and knowledge of the spoken language's structure. In our case, however, we are actually able to listen to the processed audio and successfully parse the recorded speech.

We qualitatively found that placing the drive in the enclosure actually amplified the audio signal. This somewhat counterintuitive result can be potentially attributed to the observation that the larger size of the chassis presents a greater surface area to oncoming waves, thereby enabling it to absorb more energy and then transmit it to the drive within. It is also possible that the tighter, confined space within the enclosure causes the waves to reflect and superpose, thereby increasing the amplitude.

Additionally, while driving the fan at maximum power to simulate a noisy environment, we found that the intelligibility of the signal degraded only slightly. By observing the frequency spectrum of the PES while the fan was active, we found that the noise due to the fan results in very narrow peaks at 200 Hz and its harmonics; as such, despite the fan being quite loud, the periodic nature of the noise is easily filtered. Likewise, recording the PES while writing a large file to the HDD resulted in very little loss of intelligibility. The only noticeable degradation was a moderately periodic clipping noise, which we attribute to the read/write head seeking between tracks.

We subjectively found that it is actually the male voice that is more intelligible. Given the treble heavy response of the hard drive, this is surprising; we attribute this to the heavy band of noise just below 2 kHz present in the baseline. Even though virtually no adult female's fundamental frequencies will fall in that range, a female voice's harmonics overlap that range to a much stronger

extent. As those harmonics are filtered out along with the noise, the recorded audio loses enough intelligibility such that it is actually rendered less intelligible than that of the male. This hypothesis is in agreement with the results seen in Figure 6.6, wherein a large dip in the 2 kHz band can clearly be seen.

Sample audio recordings can be found at, https://www.dropbox.com/sh/q5du7yzcoenq5u6/AACsr-cDTRy7xxKBIWAfv_UUa?dl=0. We back up our qualitative claims with objective measurements.

6.6.3 Signal to Noise Ratio

The first manner in which we quantitatively characterized the effectiveness of our side-channel was by making use of the Laboratory for the Recognition and Organization of Speech and Audio's SNReval Objective measures of speech quality/SNR [5]. These measures are MATLAB script implementations of commonly used measurements compiled from academic research, and are designed for characterizing distorted speech. Following is a brief description of the measures:

1. **NIST STNR:** NIST Signal to Noise Ratio. Estimates the signal to noise ratio as

$$\frac{peak\_speech\_power}{mean\_noise\_power}$$

   where power is computed as the variance of a given signal

2. **PESQ MOS:** Perceptual Evaluation of Speech Quality Mean Opinion Scores. Estimates intelligibility of speech, and is recommended by the International Telecommunication Union's Telecommunication Standardization Sector [138].

Between these two measures, PESQ is most closely correlated with the intelligibility of human speech [124]. The values obtained from computing these measures over the recovered audio obtained through the hard drive are displayed in Table 6.2; for reference, we also computed the

	Baseline		Under Write Workload		Fan at Maximum Power	
	NIST STNR	PESQ MOS	NIST STNR	PESQ MOS	NIST STNR	PESQ MOS
Male Speaker	8.0 dB	1.7 dB	11.2 dB	1.7	7.8 dB	1.6
Female Speaker	6.2 dB	1.7 dB	7.8 dB	1.5	3.5 dB	1.7

Table 6.2: **SNReval measurements while using the hard drive in the enclosure to record audio.**

	NIST STNR	PESQ MOS
Male Speaker	11.8 dB	1.8
Female Speaker	12.8 dB	2.1

Table 6.3: **SNReval measurements when using a microphone to record audio.**

same values over recordings obtained through an actual microphone, as displayed in Table 6.3, and through the bare drive, while it was not housed in an enclosure in Table 6.4.

As expected, both the female and male recordings degrade to some degree when recorded through the hard drive as opposed to a real microphone. While the SNR measurement exhibits a large drop, PESQ drops by a lesser extent. This is significant because the potential of the side channel is most closely aligned with the intelligibility of the recovered audio, and not the presence of objectionable noise. This also supports our previous qualitative claims, as the PESQ MOS values don't vary much in both the setup where the fan is driven at max power and the setup where a large file is continuously written. This minimal loss in fidelity enables the attacker to write the recorded audio to disk, to be retrieved at a later time if extraction over the internet is not possible.

6.6.4 Shazam Recognition

We further validate our side-channel by using the mobile application Shazam to identify songs played at the hard drive. By doing so, we demonstrate the possibility of using the hard drive as a microphone to match audio patterns.

Shazam operates by extracting and storing the most robust features of over 8 million songs and storing them in a database [275]. They accomplish this by computing spectrograms of the songs, and then reducing the songs to a series of the spectrograms' peaks. These "spectral fingerprints" serve

	NIST STNR	PESQ MOS
Male Speaker	5.5 dB	1.4
Female Speaker	1.5 dB	1.9

Table 6.4: **SNReval measurements while using the bare hard drive to record audio.**

as references against which the audio sample in question is compared. Thus, Shazam's recognition algorithm amounts to identifying which frequencies are present in the audio.

To test the hard drive's limitations at recording complex audio, we played Iron Maiden's song, "The Trooper", and used the hard drive to record it. We found that when played at 90 dBA, Shazam was able to correctly identify the song from the recording, despite the audio sounding like completely unintelligible noise to a listener's ear. While both powering the fan at maximum power and writing continuously to the HDD, the threshold increased to 94 dBA.

Notably, the requisite amplitude is substantially higher than what we played the recordings of human speech at. There are a few explanations for why such a high sound pressure level is required. The primary reason is that we are unable to use the DSP techniques described previously to filter the dirty, recovered audio in any effective manner. In fact, at the threshold volumes, Shazam is unable to recognize songs from the filtered audio. This is likely a result of the manner in which Shazam matches spectral fingerprints against its database. Since a song's spectrum is much wider than that of conversational speech, linear filtering is again ineffective for removing wide-band white noise. While spectral noise gating proved effective in the case of human speakers, it is not helpful in this case, as most of a song's energy is not concentrated in human voice. Thus, the momentary spectral separation of noise from the signal, as shown in Figure 6.3 does not arise, and the signal lies too close to the noise floor. This results in discrimination problems, wherein bands containing the signal are misclassified as noise and are subsequently removed; this destroys the spectral fingerprint, and renders Shazam incapable of recognizing the songs.

Thus, we attribute the necessity of high volumes to Shazam's classification algorithms rendering our signal processing useless. However, our results demonstrate that a hard drive can approximate a microphone closely enough to capture very complex waveforms.

6.6.5 Potential Improvements

In this section we discuss potential situations and algorithms that can aid in the recovery process.

6.6.5.1 Multiple Hard Drives

We consider the situation wherein the attacker has access to more than a single hard drive in close proximity. This can arise when a single machine, either a desktop or laptop, is using a combination of internal and external hard drives. Other possibilities include a conference room with multiple hard drives, or even a server room.

The presence of N hard drives opens up the possibility of using signal averaging to strengthen the signal in comparison to the noise. With signal averaging, we simply construct the cleaned signal by computing the average over the N corresponding measurements from the hard drives. Intuitively, this works because the average of the common signal is simply the signal itself, while the average of white noise will tend to its mean, by the Law of Large Numbers. This technique yields an improvement in the SNR by a factor of N [129].

One complication to this technique is the possible difference in phase between the wave as it hits the different hard drives. Signal averaging relies on the signal samples being time aligned, which may not be the case. Given the speed of sound at 343 m/s, a 500 Hz wave will have a wavelength of

$$\frac{343 \text{ m/s}}{500 \text{ Hz}} = 0.68 \text{ m}$$

meaning that in the conference room setting, the signals from the hard drives can easily be more than an entire wavelength out of phase. To remedy this, we can again use cross correlation to time-align the samples.

One situation that is likely to benefit from signal averaging is that where the attacker controls a multitude of hard drives within a data center. Given the large number of receivers, a linear improvement in the SNR will yield a very substantial increase in the intelligibility of the extracted speech.

A reasonable concern would be one with regards to the loud volume of background acoustical noise in data centers. Counter intuitively, however, we claim that this noise actually acts only to improve our attack. Acoustic noise within a data center reaches volumes of up to 80 dBa [189], and results in people raising their voices to communicate. This loud noise, however, largely originates from very cyclic processes: namely, fans and electronics hum. As such, we can use linear band-stop filtering techniques to remove this noise. Verbally communicating humans, however, don't typically make use of such noise reduction techniques, and as a result resort to shouting and otherwise raising their voices. This strengthens the signal our side-channel attempts to extract, and thus makes our attack even more practical in the data center setting.

In most multiple hard drive settings, it is unlikely that any form of adaptive noise cancellation can be employed, as the closed nature of a hard drive's spinning platters sufficiently isolates them so as to diminish any correlations in noise. When the noise present in multiple receivers bears no correlation, adaptive noise canceling is rendered useless. It may be possible, however, that hard drives stacked upon one another exhibit correlations in their noise due to mechanical coupling, thus leaving open the possibility of using adaptive noise cancellation. We leave this question to future work.

Moreover, in the presence of multiple hard drives, we can effectively increase the sampling rate. This is because N hard drives will take N times as many samples of the same signal as a single drive. The algorithms discussed in [185] demonstrate how to account for time and gain mismatches to allow us to leverage multiple hard drives that may not be equally spaced or oriented with respect to the signal source. By oversampling at a rate of

$$N \cdot f_n,$$

where f_n is the nyquist frequency, we gain $\log_2(N)$ additional bits of resolution [3]. With a higher sampling rate, we can increase our sampling resolution, and thus further improve the hard drive's sensitivity as a microphone. This could potentially work to allow the hard drive to pick up softer

sounds at a greater range.

6.6.5.2 Repetition

By building off the same principles that take advantage of multiple sensors, we can also leverage multiple repetitions of a given utterance to improve upon our attack. In typical conversations, certain words may be repeated more than once, resulting in a very similar acoustic wave being picked up by a single hard drive. This essentially provides additional samples of the same signal, which can then be averaged to improve the SNR. In order to find repetitions of certain utterances, we can make use of what is known as auto-correlation. This is simply the cross correlation of a signal with itself, and similar utterances will result in the largest peaks within the auto-correlation. The lag associated with these peaks is then the offset at which we are likely to find repeated utterances.

6.7 Defenses

Eavesdropping through the use of our acoustic side-channel leverages the very same mechanisms that allow high density, high performance hard drives to operate. Thus, defenses must take into consideration the trade off between security and performance, and cannot ignore the strict constraints that a hard drive operates under. In this section we discuss the range of defensive measures one can take to mitigate this attack against privacy; we both examine how retroactive defenses can secure already deployed hard drives and how manufacturers can take steps towards securing future hard drives.

6.7.1 Ultrasonic Masking

Since our techniques for improving our side-channel rely on increasing the SNR, an obvious approach for mitigating the attack is to work to reduce the SNR. We can accomplish this by either increasing the noise or decreasing the strength of the signal.

To accomplish the former, we propose the idea of ultrasonic acoustic masking. Simply using

a white noise generator to mask over the frequencies wherein human speech is contained has the undesirable side effect of being noticeable and annoying to nearby humans. By leveraging the phenomenon known as aliasing, however, we can make use of an acoustic mask that is imperceptible to humans.

Aliasing is a phenomenon where a signal matches the amplitude of a different signal at each point that it is sampled, despite being of a different frequency. This occurs when a sinusoid of frequency f is sampled at a sub-Nyquist frequency f_s, and results in the signal becoming indistinguishable from

$$|f - \ell \cdot f_s|, \ell \in \mathbb{Z}.$$

Then, we can leverage this effect to create a sound mask that lies above the hard drive's sampling rate, such that it aliases and masks over the same frequency bands as human voice. Figure 6.7 illustrates how the acoustic mask sits just above the sampling rate of the hard drive, yet is aliased so that it ends up in the baseband spectrum, where it masks human voice. In the case of our Seagate Barracuda 7200.12 1 terabyte hard disk, the sampling rate of 34.56 kHz is well above the 20 kHz ultrasonic threshold.

While this mitigation does have the benefit of being undetectable by humans, designers must be careful to find an appropriate threshold for the intensity of the sound mask. The National Counterintelligence and Security Center gives guidelines for acoustic masking and states that the intensity of the mask must exceed the level of the conversations [200]. However, given the treble heavy response of hard drives, the high frequency tones generated in defense have the potential to disturb the normal operations of the hard drive; if the added noise results in the head being completely unable to stay on track, the hard drive is then rendered useless. If the ultrasonic mask lacks sufficient power, however, it will not be effective in mitigating the side-channel. We leave the exploration of this trade off to future research.

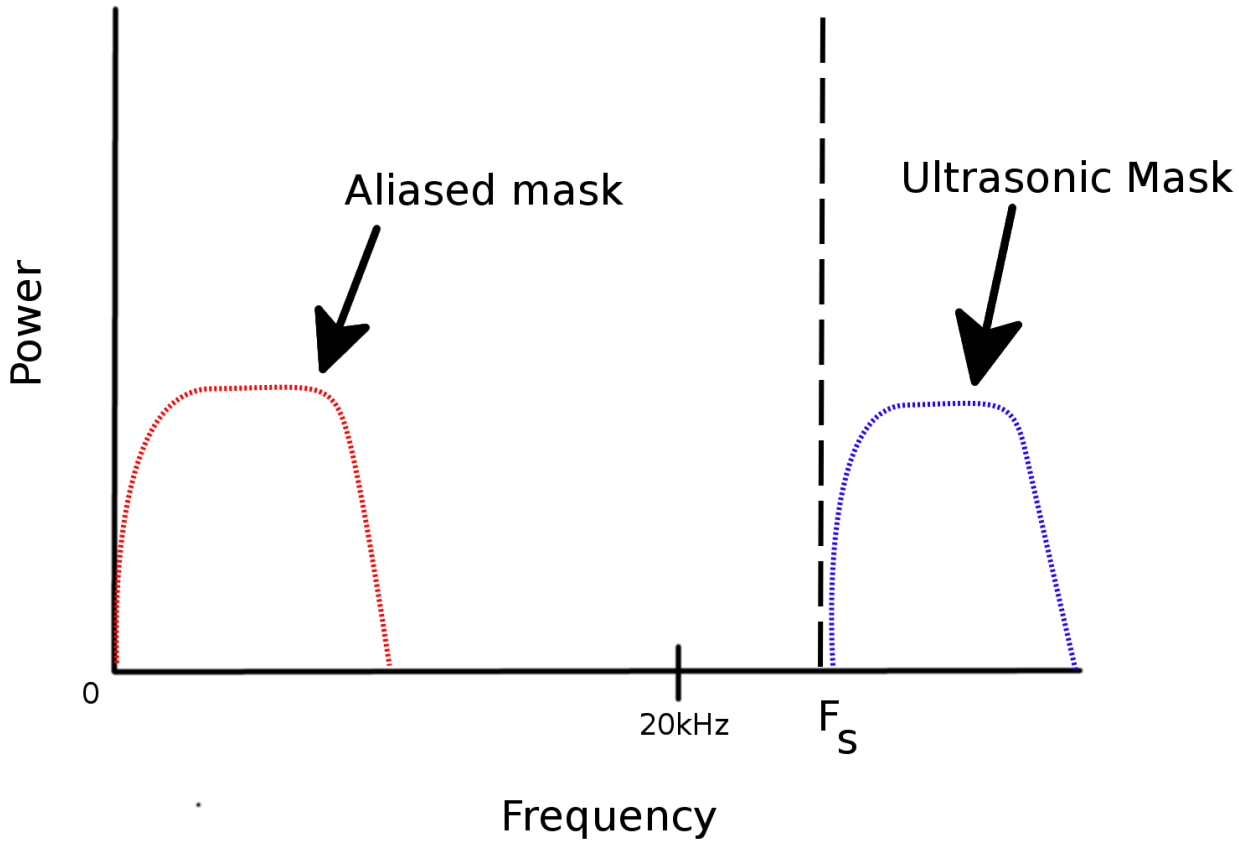


Figure 6.7: **Ultrasonic Mask as a Defense.** The ultrasonic mask generates white noise in the region just above the hard drive's sampling rate f_s. Due to the insufficient sampling rate, the mask is aliased over the region just above 0 Hz.

6.7.2 Securing Future Disks

We have thus far discussed only solutions that can be deployed retroactively to protect systems already in use. We now discuss potential mitigations and steps manufacturers can take towards protecting future hard drives during the development cycle.

The primary challenge with this is that any attempts to mitigate the side-channel at the hard drive level must also be mindful of impairing the performance of the hard drive. As such, preventative measures such as reducing the sampling rate or resolution are not practical due to the performance of the hard drive's reliance on exactly these qualities. Thus, mitigations are limited to decreasing the read/write head's susceptibility to acoustic interference.

One approach to doing this is to simply build the hard drive with more effective acoustic

dampening built into the sides. As we can see from the hard drive's dampened response to acoustic waves approaching from the side, this approach can substantially attenuate external noise. A potential drawback of this approach is that the acoustic insulation may also act to insulate heat, and thereby impair the hard drive's performance.

A similar defense involves increasing the resonant frequencies of the read/write head assembly such that it will have a weak response to human vocalizations. This can be accomplished by increasing the rigidity of the arm.

6.8 Related Work

The research that most closely resembles our own is Michalevsky et al.'s work on extracting speech from a smartphone's gyroscope [185]. While both their work and our own leverage acoustically induced perturbations in sensitive mechanical components to eavesdrop on proximal human speech, our work differs in the primary challenges and limitations. Whereas Michalevsky et al.'s primary obstacle was the low (100 Hz) sampling rate of the gyroscope, the sampling rate of our hard drive was sufficient for perceiving most of the audible range. On the other hand, we had to overcome the presence of a large amount of noise present in spinning disks. Another considerable difference exists in the experimental setup; they conducted their experiments by playing audio through speakers sharing a common surface with the phone. In contrast, we mechanically decoupled the hard drive from the speaker so as to isolate the acoustic signal.

One recent work that has explored the topic of turning hard drives into microphones is Ortega's presentation at EkoParty 2017[1]. His work proposes extracting speech by measuring hard drive write latencies from user space. This side-channel differs from our own in matters of timing precision and the requisite audio volume. In order for acoustic waves to have any impact on read/write latencies, the head must be pushed far off track; Sandahl et al. [240] demonstrate in their study that amplitudes upwards of 110 dB are required to begin affecting latencies. Additionally, timing delays introduced

[1] https://www.youtube.com/watch?time_continue=1&v=ntw32kYDryM

by the operating system's multiplexing of the hard drive will add unpredictable skew to latency measurements, whereas the even spacing of servo sectors yields a PES with equally spaced delays between samples.

Roy et al. [237] demonstrated how to jam microphones with inaudible sounds. Their work differs from our own defense in that they leveraged non-linearities specific to microphones, while our ultrasonic mask relies on aliasing due to insufficient sampling rates.

Previous work has demonstrated that subjecting hard drives to extremely loud tones can result in degradation of performance. Nickerson et al. [202] swept hard drives from 1 to 16 kHz and identified which tones resulted in severe reductions in throughput. Dutta's thesis [92] confirmed these results, demonstrated that the most sensitive frequencies corresponded to the resonant frequencies of the HDD's internal components, and used finite element analysis to gain deeper insight into how acoustic waves interact with the head stack assembly. Bolton et al. [52] further examined acoustically induced throughput loss, and designed an attenuator controller for reducing the impact of acoustic interference on HDDs.

Sensor Side Channels. Information leakage through unintended means occurs through mediums known as side channels. Various researchers have demonstrated examples of how sensors can leak information that they were never designed to measure. Marquardt et al. showed how to leverage an accelerometer's readings to recover keystrokes from a nearby keyboard [182]. Biedermann et al. [45] used the magnetometer on a smartphone to deduce the activities of a hard drive, due to the magnetic fields produced by the read/write head. Michalevsky et al. demonstrated how to geolocate a smartphone by measuring its power consumption [186]. Owusu et al. showed how attackers can recover passwords input to a smartphone's touch screen by observing accelerometer readings [211]. Guri et al. utilized speakers' near identical circuitry to approximate microphones for the purpose of eavesdropping [124]. In contrast to these works, our own acoustic side channel extracts information through a device that was never intended to function as an acoustic sensor in the first place.

Hard Drive Security. There exists a substantial body of work investigating the implications of HDD malware. Zaddach et al. demonstrated the ease with which even modestly funded attackers

can reverse engineer a HDD's malware and reflash it to implement an incredibly stealthy back door into a system [297]. In a line of research that accomplished the opposite goal of our own, Guri et al. demonstrated how to use the acoustic emanations given off by the movement of a HDD's head to establish a covert channel [123]. Guri et al. again used a hard drive for a covert channel by modulating arbitrary bits over the hard drive's flashing LED [125].

6.9 Discussion

Although solid state drives are increasingly encroaching on hard drives' portion of the market share, many legacy systems, desktops and laptops alike, still rely upon spinning disks. In fact, even in 2017 worldwide sales for hard drives in PCs nearly doubled that of solid state drives [2] [8]. As such, a large number of computer users are presently at risk from this side-channel attack. Furthermore, as hard drive technology continues to advance, and bit density and rotational frequencies increase, the need for extremely high precision feedback control loops for positioning the read/write head will become even more necessary. The implication is that hard drives will become even more well suited to functioning as microphones, making the need for a solution all the more urgent.

Firmware Security. Our research demonstrates yet another risk that hard drive malware presents. As such, we view our proof of concept acoustic side channel as a call to action for hard drive manufacturers to adopt simple defensive measures that have already been proven effective in other domains (i.e. web and mobile security).

Simply cryptographically signing firmware updates is the single most effective way to prevent the spread of hard drive firmware malware. Though previous research has demonstrated that determined attackers can bypass digital signatures in some cases via side channels, timing attacks, or mathematical weaknesses [54] [251], making use of signatures significantly increases the effort required on the behalf of attackers. Moreover, when distributing updates for hard drives that have no support for verifying digitally signed firmware, manufacturers should adopt TLS to prevent MITM attacks.

While they were unable to accurately measure just how prevalent the use of digital signatures is, Zaddach et al. [297] subjectively found that few do in practice. They also found that hard drives only verify signatures at load time, and are thus still vulnerable to run time injections. Furthermore, should any such vulnerability exist, it would be easily exploitable as none of the drives they examined took measures to mitigate classical binary exploits, such as address space layout randomization (ASLR) or data execution prevention (DEP).

Future Directions. A hard drive is able to function as a microphone due to its ability to detect vibrations at an extremely high resolution. This opens up the possibility for other vibrationally induced side channels. For example, it may be possible to extract keystrokes from nearby keyboards or mice; by leveraging the sensitive data input to the keyboard, an attacker could then create a virus that spreads between laptops that share common surfaces, much the same way that a biological virus spreads.

In a *write* side channel, an attacker may be able to use the hard drive's voice coil motor to drive the read/write head the same way it would drive a speaker's diaphragm. Thus, the hard drive operates as a speaker, and can potentially inject arbitrary voice commands into nearby voice control systems such as Siri, Google Now, Alexa, and others.

Additionally, our work sheds light on a broader area of research that has been historically neglected. While substantial effort has been invested into exploring the limitations of what side channels are available to sensors, few have considered the threat surface that is exposed by devices that were never designed to be sensors in the first place. Beyond hard drives, a printer must also stabilize its head, and thus may offer a similar side channel.

Furthermore, this cyberphysical attack surface is only expected to grow as the Internet of Things intrudes further into our personal lives. As such, we believe this line of research will open up a rich set of research directions to pursue that will only become more relevant with time.

6.10 Conclusion

Our work demonstrates the threat posed by an overlooked attack vector; that is, the potential for non-sensing devices to infringe upon privacy through a cyberphysical side channel. In particular, we have leveraged a hard drive's capability to act as an unintentional microphone to extract and parse human speech. Despite only having access to a subset of the full position error signal, we were still able to validate the side-channel through use of the Shazam service and by exploring the hard drive's properties, both qualitatively and quantitatively.

We then examined the challenges of defending both systems that are already deployed and those yet to be, and explored the fundamental trade offs between security and performance. As a consequence of our work, we recommend that security and privacy sensitive systems should adopt solid state drives.

CHAPTER 7

Conclusion

In this final chapter, we systemize the findings contained in this thesis and distill them into their key contributions, and discuss their implications for defending systems against these attacks. Then, we build upon those lessons and lay out future directions that will build upon this body of work.

Broadly, this thesis aims to contribute to our understanding of both the Rowhammer vulnerability, and the susceptibility of cryptographic algorithms to side-channels.

7.1 Rowhammer

In Chapter 2, we explored the pervasive memory vulnerability known as Rowhammer, and showed how it can directly compromise memory confidentiality. In Chapter 3, we continued to examine the Rowhammer bug, and showed how it can enable practical failure boosting attacks on the FrodoKEM post-quantum key encapsulation mechanism. In both works, we found that there are common prerequisites for conducting Rowhammer attacks that generalize well across the entire spectrum of Rowhammer attacks.

Memory Massaging. The first requirement to conduct a Rowhammer attack is *adjacency*; that is, the attacker requires some way to acquire memory in rows near the victim's memory in order to induce bit flips via Rowhammer. While it is possible for this to occur due to random chance, in order for the attack to succeed with some reasonable probability, the attacker requires a method to *massage* memory into an exploitable configuration.

In Chapter 2 we developed "Frame Feng Shui" to accomplish this massaging via an attack on Linux's deterministic memory allocator. We then used Frame Feng Shui in both the end-to-end attack on OpenSSH and our attack in Chapter 3. Without a memory massaging primitive, neither of these attacks would have been possible in practice.

Timing. Another requirement for performing a Rowhammer attack is that the attacker requires the exploitable layout to remain in memory for sufficient time to conduct the attack. This means that after massaging memory into the desired configuration, the attacker must somehow prevent the victim from ruining the configuration in any way.

In Chapter 3, we accomplished this by performing the first performance degradation attack against the victim's usage of the Shake-128 hash function. We found that shake-128 contains a tight inner loop on a short section of code; by rapidly flushing the cache-line containing that code, we successfully slowed down FrodoKEM's key generation for a long enough time to flip the targeted bits. More generally, the techniques we developed aim to maximize the length of time during which the victim's memory is in a vulnerable configuration.

7.1.1 Defenses

Given these common preconditions that are required to perform Rowhammer based attacks in practice, they make natural choke points points at which we can build defenses.

Preventing Memory Massaging. If attackers cannot massage memory into exploitable configurations, no bit flips can be obtained. As such, memory allocators should be designed in a way that stymies massaging attempts. This can be accomplished by replacing Linux's deterministic memory allocator with a randomized one, where the attacker cannot possibly predict or determine where in memory their pages lie. Another approach is to design is to design the memory-allocator to serve memory requests in a way that actively prevents exploitable memory configurations from ever occurring. Both of these approaches are attractive options as this level of defense occurs at the OS level, and requires no cooperation from application developers. Moreover, given that such defenses would be entirely based in software, these defenses are backwards compatible with computers that

have already been deployed.

Minimizing the Hammering Window. Given how critical the timing during which sensitive data appears in memory is to a Rowhammer attacker, as a principle, system designers that are concerned about Rowhammer should aim to minimize the time during which sensitive information is stored in memory. How exactly this looks in practice is highly context dependent. To use Chapter 2 as an example, OpenSSH could wipe the RSA keys from memory immediately after using them. To prevent our performance degradation attack in Chapter 3, FrodoKEM could use the hardware accelerated AES-NI instructions, which are much harder to slow down, to avoid the use of shake-128.

7.2 Crypto Side-Channels

For the second half of this thesis, we demonstrated how cache side-channel attackers can compromise both Google's Password Leak Detection service (Chapter 4) and the CTR_DRBG pseudorandom number generator (Chapter 5). In both chapters, our results concretely demonstrated the need for input-oblivious algorithms when computing on data that is in any way sensitive, and that even highly limited leakage can result in catastrophic breaches. Throughout both works, we uncovered common lessons that can be applied to inform the process of designing cryptographic algorithms going forward.

Performance Degradation. The first is strongly related to the the previous discussion on Rowhammer preconditions. That is, we found that cache-side channel attacks on cryptographic algorithms are highly constrained on how quickly they can sample the cache-state, relative to the speed of the victim. If the attacker cannot sample at least once per event in the victim, then the cache-attack cannot recover perfect traces.

In this thesis we presented techniques to overcome this limitation. Primarily, we demonstrated novel performance-degradation attacks to slow the victim sufficiently to generate adequate traces for end-to-end attacks. By rapidly flushing common code lines found in AES, Scrypt, and BEEA

code, we managed to successfully slow victims down to practically conduct cache-attacks. Thus, application developers can harden their code against cache side-channels by limiting the opportunities for performance-degradation (e.g. by using hardware acceleration or by disabling hyper-threading).

SGX as a Defense. Another hard-learned lesson we discovered was that TEEs cannot be viewed as panacea's for all desired security goals. Counterintuitively, in Chapter 5 we actually showed that placing the victim encryption algorithm in an SGX enclave and using the SGX threat model actually resulted in increasing the victim's susceptibility to side-channel attacks. Thus, if system designers are looking to augment the security of their system with a TEE enclave, they need to thoroughly consider what security guarantees are claimed by each individual TEE technology.

7.3 Future Work

With regards to both Rowhammer and cryptography side-channels, we uncovered new classes of attacks, forcing security researchers to rethink their assumptions about adversarial capabilities. The common theme with these side-channels was that the computer systems failed when my colleagues and I utilized a hitherto unknown vector to compromise the system; this is because systems cannot defend against threat vectors that have not yet even been conceived of. As such, we must design the future of computing devices with an adequate understanding of adversarial capabilities and what classes of attacks devices need to be resilient against.

As my prior work has demonstrated, I am interested in uncovering hidden threats from novel attack vectors, and subsequently securing machines against these threats in a principled manner.

Continuing along the same general directions as my previous work, my future work has three primary thrusts:

7.3.1 Memory Security

Memory systems that are secure against both integrity and confidentiality attacks are a fundamental necessity for building secure computer systems. Since the discovery of how to remotely compromise

memory integrity via Rowhammer in 2014, the security community has grappled with what this means for systems security. Numerous publications from academia and industry both have proposed mitigations for the problem, with just as many publications demonstrating why these mitigations are insufficient. Even more concerningly, new attacks against memory, beyond Rowhammer, have begun emerging [179, 277]. Accordingly, much work remains to be done in order to systematically understand memory attacks and find effective, practical solutions.

Moreover, we are currently at a pivotal point in the development of DRAM technology, with the widespread adoption of DDR5 memory in the near future and completed DDR6 design specifications expected by 2026. As such, to secure the future of DRAM it is critically important to conduct research towards further understanding how attackers can use Rowhammer to induce bit flips on DDR5; in particular, I intend to investigate the effectiveness of DDR5's on-DIMM mitigations: namely, on-DIMM ECC and Refresh Management (RFM).

While my prior work on RAMBleed [168] demonstrated how Rowhammer can directly violate confidentiality, even when ECC memory corrects every bit flip, it is not clear to what extent ECC protects DDR5. This is due to the fact that DDR5 memory performs ECC on the DIMM, and not at the level of the memory controller, presenting an interesting problem on reverse engineering the DDR5 ECC mechanism and analyzing its resilience against memory attacks.

Another feature of DDR5 that deserves further scrutiny is the new Refresh Management(RFM) command that was introduced with the DDR5 standard as a Rowhammer countermeasure. RFM aims to facilitate cooperation between the memory controller and the DIMM, hopefully resulting in a better compromise over the tradeoffs between memory-controller based mitigations and DRAM based mitigations. Unfortunately, DRAM vendors all implement their own RFM based mitigations in closed source hardware, and as a result it is unclear whether or not RFM actually improves over DDR4's targeted row refresh (TRR) in mitigating Rowhammer. I will conduct a deeper investigation into this approach to model RFM's limitations, and subsequently provide guidance in how to design improved countermeasures that could either be retrofitted to RFM on DDR5 or incorporated into the design of DDR6.

Looking beyond Rowhammer, I also plan to investigate other memory attacks, on DRAM and newer, emerging memory technologies alike. The recent discovery of the RowPress bug [179], which can result in bit flips in DRAM in a different manner from Rowhammer, provides evidence that this topic is still under-explored. It is essential that we understand RowPress and how it interacts with Rowhammer, so that we can build defenses that are effective not only against RowPress, but also against the combined threat of RowPress and Rowhammer in conjunction.

7.3.2 Speculative Attacks.

Ever since Spectre was published in 2018, numerous publications have demonstrated how various parts of a processor's speculative engine can be abused to leak sensitive data. With each new generation of CPUs since then, old vulnerabilities have been fixed, while new vulnerabilities have appeared with new optimizations. As CPUs continue to evolve, I will continue modeling the threat surface exposed by these optimizations and subsequently using the insights obtained through our evaluation to help inform succeeding generations of defenses, both in software and hardware.

With Cacheout [269] and SGAxe [267], we uncovered and exploited new microarchitectural components used by Intel as a performance optimization:namely, micro-architectural buffers. Beyond these buffers, though there are additional components of Intel's speculative engine that have not yet been investigated. This is just one example of an under researched microarchitectural component; as more speculative optimizations are deployed in the future, there will only be more speculative vulnerabilities waiting to be uncovered.

While most of both my own research and the wider academic community's focus has been on Intel's CPUs, new emerging machines have been neglected. These under-studied architectures are rich with opportunities for high impact research. One such example is Google's Titan chip, which aims to establish a hardware-based root of trust on Google's machines. These next generation architectures present new challenges to security researchers. Have lessons from older architectures (e.g. Intel and AMD) been learned and applied? What classes of currently known speculative attacks do they fail under, and what new speculative attacks are they susceptible to?

7.3.3 Secure Cryptographic Systems.

Moving beyond more traditional encryption and digital signature schemes, in recent years we have seen wider development and deployment of more advanced cryptographic primitives, such as Zero-knowledge proofs, private set intersection (PSI), and fully homomorphic encryption (FHE). Due to their sophistication, however, many of them have been designed without consideration for side-channel resilience. Unfortunately, some major software companies have already deployed untested implementations to millions of consumers.

My work [169] on studying Chrome's Password Leak Detection protocol is merely one example where we empirically demonstrated the lack of side-channel resilience in a novel cryptosystem. Following Chrome's lead, the Safari browser also incorporated PSI into their own password monitor. As PSI continues to make its transition from theory to practice, there is a clear need to conduct further analysis on how these systems fare when they are deployed on real-world, imperfect hardware. As such, I am interested in investigating how to design such PSI primitives that remain private, even when an adversary has side-channel capabilities.

Fully homomorphic encryption, which is what Microsoft based Edge's password monitor on, presents another interesting problem for side-channel security. Due to the malleable nature of ciphertexts in a homomorphic encryption scheme, an adversary can trivially maul ciphertexts, which may potentially revive prior techniques used to amplify side-channel leakage. This raises other interesting questions: beyond the ramifications for password monitors in browsers, what is the implication for the design of future protocols utilizing homomorphic encryption? How can we design side-channel secure fully homomorphic encryption that is still practically efficient?

As more and more cryptographic systems make their transition from theory into practice, there is a clear need to continue investigating their susceptibility to side-channels. Given the reality that these cryptosystems will be deployed on leaky hardware for the foreseeable future, I will continue my work towards hardening these systems in a way that preserves an acceptable level of performance.

BIBLIOGRAPHY

[1] Cell phone privacy modification. http://stahlke.org/dan/phonemute.

[2] HDD still dominate the storage wars. https://datastorageasean.com/daily-news/hdd-still-dominate-storage-wars.

[3] Improving ADC resolution by oversampling and averaging. https://www.silabs.com/documents/public/application-notes/an118.pdf.

[4] Maxtor basics personal storage 3200. http://web.archive.org/web/20080411051058/http://www.seagate.com/www/en-us/support/downloads/personal_storage/ps3200-sw.

[5] Objective measures of speech quality/snr. http://labrosa.ee.columbia.edu/projects/snreval/.

[6] Open speech repository. http://www.voiptroubleshooter.com/open_speech/american.html.

[7] San Ace 92 Datasheet. http://www.farnell.com/datasheets/1878751.pdf?_ga=2.178000229.1377951826.1522458792-2128987597.1522171362.

[8] Shipments of hard and solid state disk (HDD/SSD) drives worldwide from 2015 to 2021 (in millions). https://www.statista.com/statistics/285474/hdds-and-ssds-in-pcs-global-shipments-2012-2017/.

[9] Exceeding capacity, speed and performance expectations. Technical report, Seagate, 2011. https://www.seagate.com/files/staticfiles/docs/pdf/whitepaper/seagate-acutrac-TP624.1-1110US.pdf.

[10] Equation group: Questions and answers. Technical report, Kaspersky, 2015. https://securelist.com/files/2015/02/Equation_group_questions_and_answers.pdf.

[11] [openssl.org #4063] re: Client hello longer than 2^{14} bytes are rejected. mta.openssl.org/pipermail/openssl-dev/2015-September/002860.html, 2015.

[12] Openssl software failure for RSA 16K modulus. mta.openssl.org/pipermail/openssl-users/2016-July/004056.html, 2016.

[13] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. In *IMA International Conference on Cryptography and Coding*, pages 185–203, 2007.

[14] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *CT-RSA*, pages 225–242, 2007.

[15] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, et al. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *Conference on Computer and Communications Security (CCS)*, pages 5–17. ACM, 2015.

[16] Misiker Tadesse Aga, Zelalem Birhanu Aweke, and Todd Austin. When good protections go bad: Exploiting anti-dos measures to accelerate rowhammer attacks. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 8–13. IEEE, 2017.

[17] Ayush Agarwal, Sioli O'Connell, Jason Kim, Shaked Yehezkel, Daniel Genkin, Eyal Ronen, and Yuval Yarom. Spook.js: Attacking chrome strict site isolation via speculative execution. In *IEEE SP*, 2022.

[18] National Security Agency. Frequently asked questions: Quantum computing and post-quantum cryptography, Aug 2021.

[19] Alejandro Cabrera Aldaya and Billy Bob Brumley. HyperDegrade: From GHz to MHz effective CPU frequencies. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2801–2818, Boston, MA, August 2022. USENIX Association.

[20] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *IEEE SP*, pages 870–887. IEEE, 2019.

[21] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. Cache-timing attacks on RSA key generation. *TCHES*, 2019(4):213–242, 2019.

[22] Alejandro Cabrera Aldaya, Alejandro J Cabrera Sarmiento, and Santiago Sánchez-Solano. SPA vulnerabilities of the binary extended Euclidean algorithm. *Journal of Cryptographic Engineering*, 7(4):273–285, 2017.

[23] Erdem Alkim, Joppe W. Bos, Léo Ducas, Patrick Longa, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Chris Peikert, Ananth Raghunathan, Douglas Stebila, Karen Easterbrook, and LaMacchia Brian. Frodokem: Practical quantum-secure key encapsulation from generic lattices, Apr 2022.

[24] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ACSAC '16, page 422–435, New York, NY, USA, 2016. Association for Computing Machinery.

[25] Joël Alwen and Jeremiah Blocki. Efficiently computing data-independent memory-hard functions. In *CRYPTO*, pages 241–271, 2016.

[26] Joël Alwen, Binyi Chen, Krzysztof Pietrzak, Leonid Reyzin, and Stefano Tessaro. Scrypt is maximally memory-hard. In *EUROCRYPT*, pages 33–62, 2017.

[27] Mark Matthew Anderson. Attacking scrypt via cache timing side-channel. https://crypto.stanford.edu/cs359c/17sp/projects/MarkAnderson.pdf, 2017.

[28] Jake Archibald and Eiji Kitamura. SharedArrayBuffer updates in Android Chrome 88 and desktop Chrome 92. https://developer.chrome.com/blog/enabling-shared-array-buffer/, 2021.

[29] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando G. S. L. Brandao, David A. Buell, Brian Burkett, Yu Chen, Zijun Chen, Ben Chiaro, Roberto Collins, William Courtney, Andrew Dunsworth, Edward Farhi, Brooks Foxen, Austin Fowler, Craig Gidney, Marissa Giustina, Rob Graff, Keith Guerin, Steve Habegger, Matthew P. Harrigan, Michael J. Hartmann, Alan Ho, Markus Hoffmann, Trent Huang, Travis S. Humble, Sergei V. Isakov, Evan Jeffrey, Zhang Jiang, Dvir Kafri, Kostyantyn Kechedzhi, Julian Kelly, Paul V. Klimov, Sergey Knysh, Alexander Korotkov, Fedor Kostritsa, David Landhuis, Mike Lindmark, Erik Lucero, Dmitry Lyakh, Salvatore Mandrà, Jarrod R. McClean, Matthew McEwen, Anthony Megrant, Xiao Mi, Kristel Michielsen, Masoud Mohseni, Josh Mutus, Ofer Naaman, Matthew Neeley, Charles Neill, Murphy Yuezhen Niu, Eric Ostby, Andre Petukhov, John C. Platt, Chris Quintana, Eleanor G. Rieffel, Pedram Roushan, Nicholas C. Rubin, Daniel Sank, Kevin J. Satzinger, Vadim Smelyanskiy, Kevin J. Sung, Matthew D. Trevithick, Amit Vainsencher, Benjamin Villalonga, Theodore White, Z. Jamie Yao, Ping Yeh, Adam Zalcman, Hartmut Neven, and John M. Martinis. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

[30] Jean-Philippe Aumasson and Antony Vennard. Audit of OpenSSL's randomness generation. ostif.org/wp-content/uploads/2018/09/opensslrng-audit-report.pdf, 2018.

[31] Jane Austen. *Pride and Prejudice*. 1813.

[32] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J Alex Halderman, Viktor Dukhovni, et al. DROWN: Breaking TLS Using SSLv2. In *USENIX Security Symposium*, pages 689–706, 2016.

[33] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. ANVIL: Software-based protection against next-generation Rowhammer attacks. *ACM SIGPLAN Notices*, 51(4):743–755, 2016.

[34] Clemens Backes. Liftoff: a new baseline compiler for webassembly in v8. https://v8.dev/blog/liftoff, 2018.

[35] Kuljit Bains, John Halbert, Christopher Mozak, Theodore Schoenborn, and Zvika Greenfield. Row hammer refresh command. US Patent Application 2014/0006703A1, 2014.

[36] Ronald J Baken and Robert F Orlikoff. *Clinical Measurement of Speech and Voice*. Cengage Learning, 2000.

[37] Aurélie Bauer, Henri Gilbert, Guénaël Renault, and Mélissa Rossi. Assessment of the key-reuse resilience of newhope. In *Topics in Cryptology–CT-RSA 2019: The Cryptographers' Track at the RSA Conference 2019, San Francisco, CA, USA, March 4–8, 2019, Proceedings*, pages 272–292. Springer, 2019.

[38] Johannes Bauer, Michael Gruhn, and Felix C Freiling. Lest we forget: Cold-boot attacks on scrambled DDR3 memory. *Digital Investigation*, 16:S65–S74, 2016.

[39] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "Ooh aah... just a little bit" : A small amount of side channel can go a long way. In *CHES*, 2014.

[40] Ariel Berkman. Hiding data in hard-drive's service areas. Technical report, Recover Information Technoloies LTD, 2013. https://dl.packetstormsecurity.net/papers/general/SA-cover.pdf.

[41] Daniel J. Bernstein. Cache-timing attacks on AES, 2005.

[42] Daniel J. Bernstein. Fast-key-erasure random-number generators, 2017.

[43] Daniel J Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding right into disaster: Left-to-right sliding windows leak. In *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, pages 555–576, 2017.

[44] Sarani Bhattacharya and Debdeep Mukhopadhyay. Curious case of Rowhammer: Flipping secret exponent bits using timing analysis. In *CHES*, 2016.

[45] Sebastian Biedermann, Stefan Katzenbeisser, and Jakub Szefer. Hard drive side-channel attacks using smartphone magnetic field sensors. In *Financial Cryptography and Data Security: 19th International Conference*, 2015.

[46] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *CRYPTO*, pages 513–525, 1997.

[47] Stephen A Billings. *Nonlinear system identification: NARMAX methods in the time, frequency, and spatio-temporal domains*. John Wiley & Sons, 2013.

[48] Nina Bindel and John M Schanck. Decryption failure is more likely after success. In *International Conference on Post-Quantum Cryptography*, pages 206–225. Springer, 2020.

[49] Alex Biryukov. *Substitution–Permutation (SP) Network*, pages 1268–1268. Springer US, Boston, MA, 2011.

[50] Alex Biryukov, Daniel Dinu, Dmitry Khovratovich, and Simon Josefsson. Argon2 memory-hard function for password hashing and proof-of-work applications. RFC 9106, September 2021.

[51] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard pkcs# 1. In *CRYPTO*, 1998.

[52] Connor Bolton, Sara Rampazzi, Chaohao Li, Andrew Kwong, Wenyuan Xu, and Kevin Fu. Blue Note: How intentional acoustic interference damages availability and integrity in hard disk drives and operating systems. In *Proceedings of the 39th Annual IEEE Symposium on Security and Privacy*, May 2018.

[53] Dan Boneh, Henry Corrigan-Gibbs, and Stuart Schechter. Balloon hashing: A memory-hard function providing provable protection against sequential attacks. In *ASIACRYPT*, pages 220–248, 2016.

[54] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In *EUROCRYPT*, 1997.

[55] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In *ASIACRYPT*, pages 514–532, 2001.

[56] Joseph Bonneau. Robust final-round cache-trace attacks against AES. IACR ePrint archive 2006/374, 2006.

[57] Jonathan Bootle, Claire Delaplace, Thomas Espitau, Pierre-Alain Fouque, and Mehdi Tibouchi. Lwe without modular reduction and improved side-channel attacks against bliss. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 494–524. Springer, 2018.

[58] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: Memory deduplication as an advanced exploitation vector. In *IEEE SP*, 2016.

[59] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

[60] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. CAn't touch this: Software-only mitigation against rowhammer attacks targeting kernel memory. In *USENIX Security*, pages 117–130, 2017.

[61] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *WOOT*, 2017.

[62] Joachim Breitner and Nadia Heninger. Biased nonce sense: Lattice attacks against weak ECDSA signatures in cryptocurrencies. In *FC*, 2019.

[63] Samira Briongos, Pedro Malagón, Juan-Mariano de Goyeneche, and Jose Moya. Cache misses and the recovery of the full AES 256 key. *Applied Sciences*, (5), 2019.

[64] Shawn T Brown, Paola Buitrago, Edward Hanna, Sergiu Sanielevici, Robin Scibek, and Nicholas A Nystrom. Bridges-2: A platform for rapidly-evolving and data intensive research. In *Practice and Experience in Advanced Research Computing*, pages 1–4. 2021.

[65] Bill Budge. Code caching for WebAssembly developers. https://v8.dev/blog/wasm-code-caching, 2019.

[66] Yu Cai, Saugata Ghose, Yixin Luo, Ken Mai, Onur Mutlu, and Erich F. Haratsch. Vulnerabilities in MLC NAND flash memory programming: Experimental analysis, exploits, and mitigation techniques. In *HPCA*, pages 49–60, 2017.

[67] Matthew J Campagna. Security bounds for the NIST codebook-based deterministic random bit generator. 2006.

[68] 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 *28th USENIX Security Symposium (USENIX Security 19)*, pages 249–266, Santa Clara, CA, August 2019. USENIX Association.

[69] 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*, 2019.

[70] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. Kaslr: Break it, fix it, repeat. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 481–493, 2020.

[71] Anirban Chakraborty, Sarani Bhattacharya, Sayandeep Saha, and Debdeep Mukhopadhyay. Explframe: Exploiting page frame cache for fault analysis of block ciphers. *arXiv*, 1905.12974v3, 2020.

[72] Dar-Der Chang and Richard Wong. Hardware PES calculator. US Patent 6130798, 2000.

[73] Stephen Checkoway, Jacob Maskiewicz, Christina Garman, Joshua Fried, Shaanan Cohney, Matthew Green, Nadia Heninger, Ralf-Philipp Weinmann, Eric Rescorla, and Hovav Shacham. A systematic analysis of the Juniper dual EC incident. In *CCS*, 2016.

[74] Stephen Checkoway, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J Bernstein, Jake Maskiewicz, Hovav Shacham, and Matthew Fredrikson. On the practical exploitability of dual EC in TLS implementations. In *USENIX Security*, 2014.

[75] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. Labeled PSI from fully homomorphic encryption with malicious security. In *CCS*, pages 1223–1237, 2018.

[76] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *CCS*, pages 1243–1255, 2017.

[77] Yueqiang Cheng, Zhi Zhang, and Surya Nepal. Still hammerable and exploitable: on the effectiveness of software-only physical kernel isolation. *arXiv*, 1802.07060, 2018.

[78] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, 49:1162–1174, 2016.

[79] Shaanan Cohney, Andrew Kwong, Shahar Paz, Daniel Genkin, Nadia Heninger, Eyal Ronen, and Yuval Yarom. Pseudorandom black swans: Cache attacks on CTR_DRBG. In *IEEE SP*, pages 1241–1258, 2020.

[80] Shaanan N. Cohney, Matthew D. Green, and Nadia Heninger. Practical state recovery attacks against legacy RNG implementations. In *CCS*, 2018.

[81] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting correcting codes: On the effectiveness of ECC memory against Rowhammer attacks. In *IEEE SP*, 2019.

[82] Dana Dachman-Soled, Léo Ducas, Huijing Gong, and Mélissa Rossi. Lwe with side information: attacks and concrete security estimation. In *Annual International Cryptology Conference*, pages 329–358. Springer, 2020.

[83] Albert Dayes and John Treder. Drive performance-TMR.

[84] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. SMASH: Synchronized Many-sided Rowhammer Attacks from JavaScript. In *USENIX Security*, August 2021. Pwnie Nomination for the Most Underhyped Research.

[85] Abhishek Dhanda, Toshiki Hirano, Tetsuo Semba, and Satoshi Yamamoto. Magnetic Recording Disk Drive With Position Error Signal (PES) Blocks in The Data Tracks for Compensation of Track Misregistration. US Patent 9412403, 2016.

[86] Guillaume Didier and Clémentine Maurice. Calibration done right: Noiseless Flush+Flush attacks. In *DIMVA*, pages 278–298, 2021.

[87] Jintai Ding, Saed Alsayigh, R V Saraswathy, Scott Fluhrer, and Xiaodong Lin. Leakage of signal function with reused keys in rlwe key exchange. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–6, 2017.

[88] Jintai Ding, Scott R. Fluhrer, and Saraswathy RV. Complete attack on RLWE key exchange with reused keys, without signal leakage. In Willy Susilo and Guomin Yang, editors, *ACISP 18: 23rd Australasian Conference on Information Security and Privacy*, volume 10946 of *Lecture Notes in Computer Science*, pages 467–486, Wollongong, NSW, Australia, July 11–13, 2018. Springer, Heidelberg, Germany.

[89] Yuanchao Ding, Hua Guo, Yewei Guan, Hutao Song, Xiyong Zhang, and Jianwei Liu. Some new methods to generate short addition chains. *TCHES*, 2023:270–285, 03 2023.

[90] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean M. Tullsen. Prime+Abort: A timer-free high-precision L3 cache attack using Intel TSX. In *USENIX Security*, 2017.

[91] Yevgeniy Dodis, David Pointcheval, Sylvain Ruhault, Damien Vergniaud, and Daniel Wichs. Security analysis of pseudo-random number generators with input. In *CCS*, 2013.

[92] Trinoy Dutta. Performance of hard disk drives in high noise environments. Master's thesis, Michigan Technological University, 2017.

[93] Jan-Pieter D'Anvers, Qian Guo, Thomas Johansson, Alexander Nilsson, Frederik Vercauteren, and Ingrid Verbauwhede. Decryption failure attacks on ind-cca secure lattice-based schemes. In *Public-Key Cryptography–PKC 2019: 22nd IACR International Conference on Practice and Theory of Public-Key Cryptography, Beijing, China, April 14-17, 2019, Proceedings, Part II 22*, pages 565–598. Springer, 2019.

[94] Jan-Pieter D'Anvers, Mélissa Rossi, and Fernando Virdia. (one) failure is not an option: bootstrapping the search for failures in lattice-based encryption schemes. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 3–33. Springer, 2020.

[95] Michael Fahr Jr, Hunter Kippen, Andrew Kwong, Thinh Dang, Jacob Lichtinger, Dana Dachman-Soled, Daniel Genkin, Alexander Nelson, Ray Perlner, Arkady Yerukhimovich, and Daniel Apon. When Frodo flips: End-to-end key recovery on FrodoKEM via Rowhammer. In *CCS*, pages 979–993, 2022.

[96] Scott Fluhrer. Cryptanalysis of ring-lwe based key exchange with key share reuse. Cryptology ePrint Archive, Report 2016/085, 2016. https://eprint.iacr.org/2016/085.

[97] Christian Forler, Stefan Lucks, and Jakob Wenzel. Memory-demanding password scrambling. In *ASIACRYPT*, pages 289–305, 2014.

[98] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand pwning unit: Accelerating microarchitectural attacks with the GPU. In *IEEE SP*, pages 195–210, 2018.

[99] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the Many Sides of Target Row Refresh. In *S&P*, May 2020. Best Paper Award, Pwnie Award for the Most Innovative Research, Honorable Mention in IEEE MICRO Top Picks.

[100] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *Advances in Cryptology – CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 537–554, Santa Barbara, CA, USA, August 15–19, 1999. Springer, Heidelberg, Germany.

[101] Bundesamt für Sicherheit in der Informationstechnik. Bsi tr-02102-1: "cryptographic mechanisms: Recommendations and key lengths" version: 2022-1, Jan 2022.

[102] Cesar Pereida García and Billy Bob Brumley. Constant-time callees with variable-time callers. In *USENIX Security*, pages 83–98, 2017.

[103] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time protection: the missing OS abstraction. In *EuroSys*, 2019.

[104] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptographic Engineering*, 8(1):1–27, 2018.

[105] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. In *ACNS*, pages 83–102, 2018.

[106] GnuPG Project. Gnupg. www.gnupg.org, 2019.

[107] Dan Goodin. How "omnipotent" hackers tied to NSA hid for 14 years-and were found at last. https://arstechnica.com/information-technology/2015/02/how-omnipotent-hackers-tied-to-the-nsa-hid-for-14-years-and-were-found-at-last/, 2015.

[108] Mel Gorman. *Understanding the Linux virtual memory manager*. Prentice Hall, 2004.

[109] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. Aslr on the line: Practical cache attacks on the mmu. In *NDSS*, volume 17, page 26, 2017.

[110] Matthew D. Green. Twitter thread on openssl. https://twitter.com/matthew_d_green/status/1115013260783255558?s=12.

[111] Matthew D. Green. The strange story of "extended random". blog.cryptographyengineering.com/2017/12/19/the-strange-story-of-extended-random/, 2017.

[112] Matthew D. Green. Wonk post: chosen ciphertext security in public-key encryption (part 2). blog.cryptographyengineering.com/2018/07/20/wonk-post-chosen-ciphertext-security-in-public-key-encryption-part-2/, 2018.

[113] Glenn Greenwald. *No Place to Hide: Edward Snowden, the NSA, and the U.S. Surveillance State*. Picador, 2014.

[114] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, Gauss, and reload – a cache attack on the BLISS lattice-based signature scheme. In *CCS*, 2016.

[115] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another flip in the wall of Rowhammer defenses. In *IEEE SP*, pages 245–261, 2018.

[116] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 368–379, 2016.

[117] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in JavaScript. In *DIMVA*, pages 300–321, 2016.

[118] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: a fast and stealthy cache attack. In *DIMVA*, pages 279–299, 2016.

[119] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security*, pages 897–912, 2015.

[120] Shay Gueron and Yehuda Lindell. GCM-SIV. In *CCS*, 2015.

[121] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games - bringing access-based cache attacks on AES to practice. In *IEEE S&P*, 2011.

[122] Yanan Guo, Andrew Zigerelli, Youtao Zhang, and Jun Yang. Adversarial prefetch: New cross-core cache side channel attacks. In *IEEE SP*, pages 1458–1473, 2022.

[123] Mordechai Guri, Yosef Solewicz, Andrey Daidakulov, and Yuval Elovici. Acoustic data exfiltration from speakerless air-gapped computers via covert hard-drive noise ('diskfiltration'). In *22nd European Symposium on Research in Computer Security (ESORICS)*, 2017.

[124] Mordechai Guri, Yosef Solewicz, Andrey Daidakulov, and Yuval Elovici. SPEAKE(a)R: Turn speakers to microphones for fun and profit. In *11th USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[125] Mordechai Guri, Boris Zadov, Eran Atias, and Yuval Elovici. LED-it-GO: Leaking (a lot of) Data from Air-Gapped Computers via the (small) Hard Drive LED. In *DIMVA 2017. Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference*, 2017.

[126] Matthew H. Intel SGX for dummies (intel SGX design objectives), 2013.

[127] Marcus Hähnel, Weidong Cui, Marcus Peinado, and Tu Dresden. High-resolution side channels for untrusted operating systems. In *USENIX ATC*, 2017.

[128] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest we remember: cold-boot attacks on encryption keys. In *USENIX Security*, 2008.

[129] Umer Hassan and Muhammad Sabieh Anwar. Reducing noise by repetition: introduction to signal averaging. *European Journal of Physics*, 31(3):453, 2010.

[130] Wilko Henecka, Alexander May, and Alexander Meurer. Correcting errors in RSA private keys. In *CRYPTO*, pages 351–369, 2010.

[131] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *USENIX Security*, 2012.

[132] Nadia Heninger and Hovav Shacham. Reconstructing RSA private keys from random key bits. In *CRYPTO*, pages 1–17, 2009.

[133] A Hernandez, Sam Scott, Nick Sullivan, Riad Wahby, and Christopher A Wood. Hashing to elliptic curves. https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-12, 2022.

[134] Shoichi Hirose. Security analysis of DRBG using HMAC in NIST SP 800-90. In *WISA*, 2009.

[135] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017: 15th Theory of Cryptography Conference, Part I*, volume 10677 of *Lecture Notes in Computer Science*, pages 341–371, Baltimore, MD, USA, November 12–15, 2017. Springer, Heidelberg, Germany.

[136] James Howe, Ayesha Khalid, Marco Martinoli, Francesco Regazzoni, and Elisabeth Oswald. Fault attack countermeasures for error samplers in lattice-based cryptography. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2019.

[137] Shan Hu and Barmeshwar Vikramaditya. Servo control techniques for track following on hard disk drive spin stand testers. In *2014 American Control Conference*, 2014.

[138] Yi Hu and Philipos C Loizou. Evaluation of objective quality measures for speech enhancement. *IEEE Transactions on Audio, Speech, and Language Processing*, 16(1):229–238, 2008.

[139] Bernardo A Huberman, Matt Franklin, and Tad Hogg. Enhancing privacy and trust in electronic communities. In *ACM conference on Electronic commerce*, pages 78–86, 1999.

[140] IEEE. Harvard sentences. http://www.cs.columbia.edu/~hgs/audio/harvard.html.

[141] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In *CHES*, pages 368–388, 2016.

[142] Intel. Intel software guard extensions SSL. github.com/intel/intel-sgx-ssl, 2017.

[143] Intel Corporation. 6th generation Intel processor datasheet for S-Platforms, 2015.

[144] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. MASCAT: preventing microarchitectural attacks before distribution. In *CODASPY*, pages 377–388, 2018.

[145] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, cross-VM attack on AES. In *RAID*, 2014.

[146] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: Speculative load hazards boost Rowhammer and cache attacks. In *USENIX Security*, pages 621–637, 2019.

[147] Saad Islam, Koksal Mus, Richa Singh, Patrick Schaumont, and Berk Sunar. Signature correction attack on dilithium signature scheme. *CoRR*, abs/2203.00637, 2022.

[148] Josh Jaffe. A First-Order DPA Attack Against AES in Counter Mode with Unknown Initial Counter. In *CHES*, 2007.

[149] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking down the processor via Rowhammer attack. In *SysTEX*, page 5, 2017.

[150] Patrick Jattke, Victor van der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. Blacksmith: Scalable Rowhammering in the Frequency Domain. In *S&P*, May 2022.

[151] JEDEC Solid State Technology Association. JEDEC. Standard No. 79-3F. DDR3 SDRAM Specification, 2012.

[152] JEDEC Solid State Technology Association. Low power double data rate 4, 2017.

[153] JEDEC Solid State Technology Association. Low power double data rate 4. http://www.jedec.org/standards-documents/docs/jesd209-4b, 2017.

[154] Henric Jungheim. henric.org/random/#nistrng, 2019.

[155] Burt Kaliski. PKCS #1: RSA encryption version 1.5. RFC 2313, 1998.

[156] Wilson Kan. Analysis of underlying assumptions in NIST DRBGs. 2007.

[157] Naghmeh Karimi, Arun Karthik Kanuparthi, Xueyang Wang, Ozgur Sinanoglu, and Ramesh Karri. MAGIC: Malicious aging in circuits/cores. *ACM (TACO)*, 12(1):5, 2015.

[158] Cameron F. Kerry, Acting Secretary, and Charles Romine Director. FIPS pub 186-4: Digital signature standard (DSS), 2013.

[159] Dae-Hyun Kim, Prashant J Nair, and Moinuddin K Qureshi. Architectural support for mitigating row hammering in DRAM memories. *IEEE Computer Architecture Letters*, 14(1):9–12, 2015.

[160] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372, 2014.

[161] Daniel Kirkwood, Bradley C. Lackey, John McVey, Mark Motley, Jerome A. Solinas, and David Tuller. Failure is not an option: Standardization issues for post-quantum key agreement, 2015. https://csrc.nist.gov/csrc/media/events/workshop-on-cybersecurity-in-a-post-quantum-world/documents/presentations/session7-motley-mark.pdf.

[162] Alex Klyubin. Some SecureRandom thoughts. android-developers.googleblog.com/2013/08/some-securerandom-thoughts.html, 2013.

[163] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE SP*, 2019.

[164] Paul C; Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, pages 104–113, 1996.

[165] Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. {Half-Double}: Hammering from the next row over. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3807–3824, 2022.

[166] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. {ZebRAM}: Comprehensive and compatible software protection against rowhammer attacks. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 697–710, 2018.

[167] Anil Kurmus, Nikolas Ioannou, Nikolaos Papandreou, and Thomas P. Parnell. From random block corruption to privilege escalation: A filesystem attack vector for Rowhammer-like attacks. In *WOOT*, 2017.

[168] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. Rambleed: Reading bits in memory without accessing them. In *41st IEEE Symposium on Security and Privacy (S&P)*, 2020.

[169] Andrew Kwong, Walter Wang, Jason Kim, Jonathan Berger, Daniel Genkin, Eyal Ronen, Hovav Shacham, Riad Wahby, and Yuval Yarom. Rambleed: Reading bits in memory without accessing them. In *41st IEEE Symposium on Security and Privacy (S&P)*, 2020.

[170] Andrew Kwong, Wenyuan Xu, and Kevin Fu. Hard drive of hearing: Disks that eavesdrop with a synthesized microphone. In *2019 IEEE symposium on security and privacy (S&P)*, 2019.

[171] Mark Lanteigne. How Rowhammer could be used to exploit weaknesses in computer hardware. http://www.thirdio.com/rowhammer.pdf, 2016.

[172] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security*, 2016.

[173] Hin-Tak Leung. Redhat bug 1150286 - rdrand instruction fails after resume on AMD CPU. bugzilla.kernel.org/show_bug.cgi?id=85911, 2019.

[174] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for LWE-based encryption. In Aggelos Kiayias, editor, *Topics in Cryptology – CT-RSA 2011*, volume 6558 of *Lecture Notes in Computer Science*, pages 319–339, San Francisco, CA, USA, February 14–18, 2011. Springer, Heidelberg, Germany.

[175] 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*, 2018.

[176] Moritz Lipp, Michael Schwarz, Lukas Raab, Lukas Lamster, Misiker Tadesse Aga, Clémentine Maurice, and Daniel Gruss. Nethammer: Inducing rowhammer faults through network requests. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 710–719. IEEE, 2020.

[177] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, 2015.

[178] Xiaoxuan Lou, Fan Zhang, Zheng Leong Chua, Zhenkai Liang, Yueqiang Cheng, and Yajin Zhou. Understanding Rowhammer attacks through the lens of a unified reference framework. *arXiv*, 1901.03538, 2019.

[179] Haocong Luo, Ataberk Olgun, Abdullah Giray Yağlıkçı, Yahya Can Tuğrul, Steve Rhyner, Meryem Banu Cavlak, Joël Lindegger, Mohammad Sadrosadati, and Onur Mutlu. Rowpress: Amplifying read disturbance in modern dram chips. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, 2023.

[180] Jayanth Murthy Madapura. Achieving higher ADC resolution using oversampling. http://ww1.microchip.com/downloads/en/AppNotes/Achieving%20Higher%20ADC%20Resolution%20Using%20Oversampling%2001152A.pdf, 2008.

[181] Michele Marazzi, Patrick Jattke, Flavien Solt, and Kaveh Razavi. ProTRR: Principled yet Optimal In-DRAM Target Row Refresh. In *S&P*, May 2022. Patent pending, ETH Spark Award Nomination.

[182] Philip Marquardt, Arunabh Verma, Henry Carter, and Patrick Traynor. (sp) iPhone: decoding vibrations from nearby keyboards using mobile phone accelerometers. In *Proceedings of the 18th ACM conference on Computer and Communications Security*, 2011.

[183] Daniele Micciancio and Oded Regev. Lattice-based cryptography. In *Post-quantum cryptography*, pages 147–191. Springer, 2009.

[184] Kai Michaelis, Christopher Meyer, and Jörg Schwenk. Randomly failed! The state of randomness in current Java implementations. In *CT-RSA*, 2013.

[185] Yan Michalevsky, Dan Boneh, and Gabi Nakibly. Gyrophone: Recognizing speech from gyroscope signals. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[186] Yan Michalevsky, Aaron Schulman, Gunaa Arumugam Veerapandian, Dan Boneh, and Gabi Nakibly. Powerspy: Location tracking using mobile device power analysis. In *USENIX Security Symposium*, 2015.

[187] Microsoft. Cache and memory manager improvements. https://docs.microsoft.com/en-us/windows-server/administration/performance-tuning/subsystem/cache-memory-management/improvements-in-windows-server, April 2017.

[188] Phillip Rogaway Mihir Bellare. PSS: Provably secure encoding method for digital signatures, 1998.

[189] Dubravko Miljković. Noise within a data center. In *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2016 39th International Convention on*, pages 1145–1150. IEEE, 2016.

[190] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In *CHES*, 2017.

[191] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural data leakage via automated attack synthesis. In *USENIX Security*, 2020.

[192] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. CopyCat: Controlled instruction-level attacks on enclaves. In *USENIX Security*, pages 469–486, 2020.

[193] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE bites: exploiting the SSL 3.0 fallback. 2014.

[194] David Molnar, Matt Piotrowski, David Schultz, and Davi Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *ICISC*, pages 156–168, 2006.

[195] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch. PKCS #1: RSA cryptography specifications version 2.2. RFC 8017, 2016.

[196] Praveen Mosalikanti, Chris Mozak, and Nasser A. Kurd. High performance DDR architecture in Intel Core processors using 32nm CMOS high-K metal-gate process. In *VLSI-DAT*, pages 154–157, 2011.

[197] Keaton Mowery, Sriram Keelveedhi, and Hovav Shacham. Are AES x86 cache timing attacks still feasible? In *CCSW*, 2012.

[198] Koksal Mus, Saad Islam, and Berk Sunar. QuantumHammer: A practical hybrid attack on the LUOV signature scheme. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020: 27th Conference on Computer and Communications Security*, pages 1071–1084, Virtual Event, USA, November 9–13, 2020. ACM Press.

[199] Onur Mutlu and Jeremie S. Kim. Rowhammer: A retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(8):1555–1571, 2020.

[200] NCSC. Technical specifications for construction and management of sensitive compartmented information facilities. https://www.dni.gov/files/NCSC/documents/Regulations/Technical-Specifications-SCIF-Construction.pdf.

[201] Michael Neve and Jean-Pierre Seifert. Advances on Access-Driven Cache Attacks on AES. In *Selected Areas in Cryptography*, 2007.

[202] Matthew L Nickerson, Kent Green, and Nitin Pai. Tonal noise sensitivity in hard drives. In *Proceedings of Meetings on Acoustics 166ASA*, volume 20, page 040006. ASA, 2013.

[203] NIST. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf, Dec 2016.

[204] NIST. Announcing issuance of federal information processing standard (FIPS) 140-3, security requirements for cryptographic modules. 2019.

[205] NIST. Post-quantum cryptography - round 3 submissions. https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions, Apr 2022.

[206] NIST. Post-quantum cryptography standardization. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization, Apr 2022.

[207] Kento Oonishi and Noboru Kunihiro. Attacking noisy secret CRT-RSA exponents in binary method. In *ICISC*, pages 37–54, 2018.

[208] OpenSSL. SSL/TLS Client. wiki.openssl.org/index.php/SSL/TLS_Client, 2018.

[209] OpenSSL Software Foundation. *User Guide for the OpenSSL FIPS Object Module v2.0*, 2013.

[210] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and counter-measures: The case of AES. In *CT-RSA*, 2006.

[211] Emmanuel Owusu, Jun Han, Sauvik Das, Adrian Perrig, and Joy Zhang. Accessory: password inference using accelerometers on smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, 2012.

[212] Kenneth G Paterson, Antigoni Polychroniadou, and Dale L Sibborn. A coding-theoretic approach to recovering noisy RSA keys. In *ASIACRYPT*, pages 386–403, 2012.

[213] Mathias Payer. HexPADS: a platform to detect "stealth" attacks. In *ESSoS*, pages 138–154, 2016.

[214] Colin Percival. Cache missing for fun and profit. In *BSDCan 2005*, 2005.

[215] Colin Percival. Stronger key derivation via sequential memory-hard functions, 2009.

[216] Colin Percival. The scrypt password-based key derivation function. RFC 7914, August 2016.

[217] Nicole Perlroth. Government announces steps to restore confidence on encryption standards. bits.blogs.nytimes.com/2013/09/10/government-announces-steps-to-restore-confidence-on-encryption-standards, 2013.

[218] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. To bliss-b or not to be: Attacking strongswan's implementation of post-quantum signatures. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1843–1855, 2017.

[219] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: exploiting DRAM addressing for cross-CPU attacks. In *USENIX Security*, pages 565–581, 2016.

[220] Rishabh Poddar, Amit Datta, and Chester Rebeiro. A cache trace attack on CAMELLIA. In *InfoSecHiComNet*, 2011.

[221] Ruth Pordes, Don Petravick, Bill Kramer, Doug Olson, Miron Livny, Alain Roy, Paul Avery, Kent Blackburn, Torre Wenaus, Frank Würthwein, et al. The open science grid. In *Journal of Physics: Conference Series*, volume 78, page 012057. IOP Publishing, 2007.

[222] T. Pornin. Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). RFC 6979, 2013.

[223] The OpenSSL Project. OpenSSL: The open source toolkit for SSL/TLS. www.openssl.org, 2003.

[224] Rui Qiao and Mark Seaborn. A new approach for Rowhammer attacks. In *HOST*, pages 161–166, 2016.

[225] Yue Qin, Chi Cheng, Xiaohan Zhang, Yanbin Pan, Lei Hu, and Jintai Ding. A systematic approach and analysis of key mismatch attacks on lattice-based nist candidate kems. In *ASIACRYPT 2021, Tibouchi and H. Wang (Eds.)*, pages 92–121, 2021.

[226] Quarkslab SAS. Openssl security assessment. ostif.org/wp-content/uploads/2019/01/18-04-720-REP_v1.2.pdf, 2019.

[227] Prasanna Ravi, Mahabir Prasad Jhanwar, James Howe, Anupam Chattopadhyay, and Shivam Bhasin. Side-channel assisted existential forgery attack on dilithium-a nist pqc candidate. *Cryptology ePrint Archive*, 2018.

[228] Prasanna Ravi, Mahabir Prasad Jhanwar, James Howe, Anupam Chattopadhyay, and Shivam Bhasin. Exploiting determinism in lattice-based signatures: practical fault attacks on pqm4 implementations of nist candidates. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 427–440, 2019.

[229] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *USENIX Security*, pages 1–18, 2016.

[230] Red Hat. *Red Hat Enterprise Linux 7 - Virtualization Tuning and Optimization Guide*, 2017.

[231] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th Annual ACM Symposium on Theory of Computing*, pages 84–93, Baltimore, MA, USA, May 22–24, 2005. ACM Press.

[232] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site isolation: Process separation for web sites within the browser. In *USENIX Security*, pages 1661–1678, 2019.

[233] E. Rescorla. The transport layer security (TLS) protocol version 1.3. RFC 8446, 2018.

[234] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *CACM*, 21(2):120–126, 1978.

[235] Thomas Roche, Victor Lomné, and Karim Khalfallah. Combined fault and side-channel attack on protected implementations of AES. In *CARDIS*, 2011.

[236] Eyal Ronen, Adi Shamir, Achi Or Weingarten, and Colin Oflynn. IoT goes nuclear: Creating a Zigbee chain reaction. In *IEEE S&P*, 2018.

[237] Nirupam Roy, Haitham Hassanieh, and Romit Roy Choudhury. Backdoor: Making microphones hear inaudible sounds. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 2–14. ACM, 2017.

[238] Sylvain Ruhault. SoK: Security models for pseudo-random number generators. *FSE*, 2017.

[239] Tsunekazu Saito, Keita Xagawa, and Takashi Yamakawa. Tightly-secure key-encapsulation mechanism in the quantum random oracle model. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 520–551, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Heidelberg, Germany.

[240] Derek Sandahl, Alan Elder, and Andrew Barnard. The impact of sound on computer hard disk drives and risk mitigation measures. Technical report, Tyco, Michigan Technical University, 2015. https://www.ansul.com/en/us/DocMedia/T-2016367.PDF.

[241] Michael Schwarz. *DRAMA: Exploiting DRAM Buffers for Fun and Profit*. PhD thesis, Graz University of Technology, 2016.

[242] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript. In *FC*, pages 247–267, 2017.

[243] Mark Seaborn and Thomas Dullien. Exploiting the DRAM Rowhammer bug to gain kernel privileges. https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html, 2015.

[244] Johanna Sepulveda, Andreas Zankl, and Oliver Mischke. Cache attacks and countermeasures for ntruencrypt on mpsocs: post-quantum resistance for the iot. In *2017 30th IEEE International System-on-Chip Conference (SOCC)*, pages 120–125. IEEE, 2017.

[245] Igor Sfiligoi, Daniel C Bradley, Burt Holzman, Parag Mhashilkar, Sanjay Padhi, and Frank Wurthwein. The pilot way to grid resources using glideinwms. In *2009 WRI World congress on computer science and information engineering*, volume 2, pages 428–432. IEEE, 2009.

[246] Adi Shamir. Method and apparatus for protecting public key schemes from timing and fault attacks. US Patent 5,991,415A, 1999.

[247] Thomas Shrimpton and R. Seth Terashima. Salvaging weak security bounds for blockcipher-based constructions. In *ASIACRYPT*, 2016.

[248] Dan Shumow and Niels Ferguson. On the possibility of a back door in the NIST sp800-90 dual EC PRNG. In *CRYPTO*, 2007.

[249] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. Prime+Probe 1, JavaScript 0: Overcoming browser-based side-channel defenses. In *USENIX Security*, pages 2863–2880, 2021.

[250] Kirill A. Shutemov. Pagemap: Do not leak physical addresses to non-privileged userspace. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=ab676b7d6fbf4b294bf198fb27ade5b0e865c7ce, March 2015. Retrieved on November 10, 2015.

[251] Evgeny Sidorov. Breaking the Rabin-Williams digital signature system implementation in the crypto++ library. *IACR Cryptology ePrint Archive*, 2015:368, 2015.

[252] Steven W Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Pub, 1997.

[253] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy*, pages 574–588. IEEE, 2013.

[254] Alexander Sotirov. Heap feng shui in JavaScript. In *BlackHat Europe*, 2007.

[255] Raphael Spreitzer and Thomas Plos. Cache-access pattern attack on disaligned AES T-Tables. In *Constructive Side-Channel Analysis and Secure Design*, 2013.

[256] Leszek Swirski. Sparkplug — a non-optimizing javascript compiler. https://v8.dev/blog/sparkplug, 2021.

[257] Andrei Tatar, Radhesh Krishnan, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer attacks over the network and defenses. In *USENIX ATC*, 2018.

[258] Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, Sarvar Patel, Dan Boneh, et al. Protecting accounts from credential stuffing with password breach alerting. In *USENIX Security*, pages 1556–1571, 2019.

[259] Mehdi Tibouchi and Alexandre Wallet. One bit is all it takes: a devastating timing attack on bliss's non-constant time sign flips. *Journal of Mathematical Cryptology*, 15(1):131–142, 2021.

[260] Ingo R Titze and Daniel W Martin. *Principles of Voice Production*. Prentice Hall, 1998.

[261] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G Shin. Spechammer: Combining spectre and rowhammer for new speculative attacks. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 681–698. IEEE, 2022.

[262] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, (1), 2010.

[263] Tetsuo Ueda, Kiyoshi Satoh, Hiroyuki Ono, and Toshiaki Wada. Disk drive and write control method for a disk drive. US Patent 6111714, 1998.

[264] 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*, pages 991–1008, 2018.

[265] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step. In *SysTEX*, 2017.

[266] 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*, pages 1675–1689, 2016.

[267] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. SGAxe: How SGX fails in practice. https://sgaxe.com/, 2020.

[268] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Rogue in-flight data load. In *IEEE SP*, 2019.

[269] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking data on Intel CPUs via cache evictions. In *IEEE SP*, May 2021.

[270] Mathy Vanhoef and Eyal Ronen. Dragonblood: A security analysis of wpa3's sae handshake. *eprint*, 2019.

[271] Saru Vig, Siew Kei Lam, Sarani Bhattacharya, and Debdeep Mukhopadhyay. Rapid detection of Rowhammer attacks using dynamic skewed hash tree. In *HASP@ISCA*, pages 7:1–7:8, 2018.

[272] Pepe Vila, Boris Köpf, and José F Morales. Theory and practice of finding eviction sets. In *IEEE SP*, pages 39–54, 2019.

[273] Ricardo Villanueva-Polanco. Cold boot attacks on bliss. In *International Conference on Cryptology and Information Security in Latin America*, pages 40–61. Springer, 2019.

[274] Riad S. Wahby and Dan Boneh. Fast and simple constant-time hashing to the BLS12-381 elliptic curve. *TCHES*, 2019(4):154–179, 2019.

[275] Avery Wang. An industrial-strength audio search algorithm. In *Proceedings of the 4th International Conference on Music Information Retrieval*, 2003.

[276] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *CCS*, 2017.

[277] Zixuan Wang, Mohammadkazem Taram, Daniel Moghimi, Steven Swanson, Dean Tullsen, and Jishen Zhao. Nvleak: Off-chip side-channel attacks via non-volatile memory systems.

[278] Yoav Weiss and Eiji Kitamura. Aligning timers with cross origin isolation restrictions. https://developer.chrome.com/blog/cross-origin-isolated-hr-timers/, 2021.

[279] 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, 2018.

[280] Joanne Woodage and Dan Shumow. An Analysis of the NIST SP 800-90A Standard. In *EUROCRYPT*, 2019.

[281] Charles V Wright, Lucas Ballard, Fabian Monrose, and Gerald M Masson. Language identification of encrypted VOIP traffic: Alejandra y Roberto or Alice and Bob? In *USENIX Security Symposium*, pages 43–54, 2007.

[282] Wtdrog. Systemd Issue #11810 - Can't suspend again after suspending one time. github.com/systemd/systemd/issues/11810, 2019.

[283] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. Stacco: Differentially analyzing side-channel traces for detecting SSL/TLS vulnerabilities in secure enclaves. In *CCS*, 2017.

[284] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops: Cross-VM row hammer attacks and privilege escalation. In *USENIX Security*, 2016.

[285] Shaomin Xiong and David Bogy. Position error signal generation in hard disk drives based on a field programmable gate array (fpga). In *Microsystem Technologies*, volume 19, 2013.

[286] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE SP*, pages 640–656, 2015.

[287] Takashi Yamaguchi, Yoshio Soyama, Haruhiko Hosokawa, Katushiro Tsuneta, and Hiromu Hirai. Improvement of settling response of disk drive head positiong servo using mode switching control with initial value compensation. In *IEEE Transactions on Magnetics*. IEEE, 1996.

[288] Mengjia Yan, Christopher Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. arxiv:1808.04761, 2018.

[289] Yuval Yarom. Mastik: A micro-architectural side-channel toolkit, 2016.

[290] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA nonces using the Flush+Reload cache side-channel attack. IACR ePrint archive 2014/140, 2014.

[291] Yuval Yarom and Katrina Falkner. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, 2014.

[292] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.

[293] Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. Verified correctness and security of mbedTLS HMAC-DRBG. In *CCS*, 2017.

[294] 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 *IEEE SP*, pages 79–93, 2009.

[295] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. When private keys are public. In *IMC*, 2009.

[296] Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Reetuparna Das, and Todd Austin. Cold boot attacks are still hot: Security analysis of memory scramblers in modern processors. In *HPCA*, pages 313–324, 2017.

[297] Jonas Zaddach, Anil Kurmus, Davide Balzarotti, Erik-Oliver Blass, Aurelien Francillon, Travis Goodspeed, Moitrayee Gupta, and Ioannis Koltsidas. Implementation and implications of a stealth hard-drive backdoor. In *Proceedings of the 29th annual computer security applications conference*. ACM, 2013.

[298] Fan Zhang. mbedtls-SGX. `github.com/bl4ck5un/mbedtls-SGX`, 2018.

[299] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y. Thomas Hou. TruSpy: Cache side-channel information leakage from the secure world on ARM devices. *ePrint*, 2016.

[300] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. Cloudradar: A real-time side-channel attack detection system in clouds. In *RAID*, pages 118–140, 2016.

[301] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In *CCS*, 2014.

[302] Jian-Gang Zhu. New heights for hard disk drives. *Materials Today*, 6(7):22–31, 2003.