Jolt: Recovering TLS Signing Keys via Rowhammer Faults

Koksal Mus
Worcester Polytechnic Institute
kmus@wpi.edu

Yarkın Doröz
Worcester Polytechnic Institute
ydoroz@wpi.edu

M. Caner Tol
Worcester Polytechnic Institute
mtol@wpi.edu

Kristi Rahman
Worcester Polytechnic Institute
krahman@wpi.edu

Berk Sunar
Worcester Polytechnic Institute
sunar@wpi.edu

***Abstract*—Digital Signature Schemes such as DSA, ECDSA, and RSA are widely deployed to protect the integrity of security protocols such as TLS, SSH, and IPSec. In TLS, for instance, RSA and (EC)DSA are used to sign the state of the agreed upon protocol parameters during the handshake phase. Naturally, RSA and (EC)DSA implementations have become the target of numerous attacks, including powerful side-channel attacks. Hence, cryptographic libraries were patched repeatedly over the years.**

Here we introduce Jolt, a novel attack targeting signature scheme implementations. Our attack exploits faulty signatures gained by injecting faults during signature generation. By using the signature verification primitive, we correct faulty signatures and, in the process deduce bits of the secret signing key. Compared to recent attacks that exploit single bit biases in the nonce that require 2^{45} signatures, our attack requires less than a thousand faulty signatures for a 256-bit (EC)DSA. The performance improvement is due to the fact that our attack targets the secret signing key, which does not change across signing sessions. We show that the proposed attack also works on Schnorr and RSA signatures with minor modifications.

We demonstrate the viability of Jolt by running experiments targeting TLS handshakes in common cryptographic libraries such as `WolfSSL`, `OpenSSL`, `Microsoft SymCrypt`, `LibreSSL`, and `Amazon s2n`. On our target platform, the online phase takes less than 2 hours to recover 192 bits of a 256-bit ECDSA key, which is sufficient for full key recovery. We note that while RSA signatures are protected in popular cryptographic libraries, `OpenSSL` remains vulnerable to double fault injection. We have also reviewed their FIPS hardened versions which is slightly less efficient but still vulnerable to our attack. We found that (EC)DSA signatures remain largely unprotected against software-only faults, posing a threat to real-life deployments such as TLS, and potentially other security protocols such as SSH and IPSec. This highlights the need for a thorough review and implementation of faults checking in security protocol implementations.

1. Introduction

RSA and Digital Signature Algorithms (EC)DSA have been serving for over two decades as the backbone of our security network. Naturally, it has become the target of numerous attacks, especially ones exploiting imperfections in its implementation. While developers have come a long way in protecting against side-channel attacks, the development and vulnerability testing methodologies are far from perfect. For instance, [1] surveyed 44 developers of 27 prominent open-source cryptographic libraries and found that, while most developers are aware of timing attacks and of their potentially dramatic consequences, they choose to prioritize other issues over the perceived huge investment of time and resources currently needed to make their code resistant to such attacks. Quite naturally, vulnerabilities, including leakages in RSA and ECDSA implementations, are far from being resolved. While most leakages have been fixed, cryptanalytical techniques for recovery have advanced to the point where extremely subtle leakages as small as a bit have been exploited to yield full key recovery [2].

Curse of EC(DSA) Nonces. A well-known and often exploited weakness is that of the nonce values used during the signing process [3], [4], [5], [6], [7], [8], [9]. Nonces are required to be uniform and free from any bias. Otherwise, a number of techniques can be used to efficiently recover the secret key from signature samples. Hence, developers need to make sure their random number generators are free of any biases and their implementations of sensitive functions are free from leakage. In particular, software-only attacks, such as the ones exploiting signature generation timing [6] have repeatedly shown to be effective in recovering signing keys. Most (EC)DSA implementations, therefore, have deployed countermeasures to achieve constant-time execution. Despite these efforts, (EC)DSA implementations are still being haunted by subtle leakages.

Fault Injection Attacks. Beyond passive attacks, signature schemes have been a target of active tampering, such as in fault injection attacks. Early on, Boneh et al. [10] introduced so-called *Bellcore* attacks, showing how glitches introduced during CRT-based exponentiation can be used to recover RSA keys from faulty signatures. Early practical attacks targeted smartcards. For instance, [11] uses glitching to zero out part of the nonce in DSA and recover the key using a lattice attack, while [12] used faults to target information flow and retrieve parts of the ephemeral ECDSA key. Ravi et al. [13] experimented with electromagnetic fault injection on the post-quantum signature scheme and NIST finalist Crystals-Dilithium. With the proliferation of attacks targeting the nonces and imperfect randomization, deterministic versions of DSA and schemes such as EdDSA emerged with the promise of built-in resilience. Ambrose et al. [14] question the merits of deterministic schemes by introducing differential fault attacks against them and propose various low-cost countermeasures to protect real-life deployments.

Scanning for Natural Faults. Most relevant to our study is the Technical Report Weimer published in 2015 [15]. Weimer demonstrated that active scans performed in TLS traffic over months can reveal errors in RSA signatures. The faults in several hundred faulty signatures led to the compromise of private keys. The faults were traced to devices by several vendors. Most significantly, Weimer linked the source of the faults to the failure of several TLS libraries to implement fault-checking mechanisms in the signing code. The work demonstrates the fragility of RSA signature padding and the possible threats that come with generating signatures at a massive scale on imperfect hardware. More recently, Sullivan et al. [16] similarly scanned their university networks' logs for faulty signatures generated in TLS 1.2 and were indeed able to recover RSA keys, including those of a Alexa top-10 site, several browser-trusted wildcard certificates for organizations that used a popular VPN product, etc. This work demonstrated that, despite the addition of RSA signature fault checks in TLS libraries, the threat still persists.

Software Faults via Rowhammer. The emergence of Rowhammer gave a realistic tool for an attacker to inject software faults without direct physical access. In [17], for instance, Razavi et al. demonstrated an end-to-end attack breaking `OpenSSH` public-key authentication and GPG signature forgery. In their attack, Rowhammer is used to inject bit flips into the RSA public key modulus until factorization and thereby secret key recovery is achieved. Weissman et al. [18] demonstrated a Bellcore attack on a CRT-based RSA implementation in `WolfSSL`, to recover secret keys on an FPGA enabled cloud platform. Further, in [19] Poddebniak et al. analyze the security of deterministic ECDSA and EdDSA signature schemes and find that the elimination of randomization enables new kinds of fault attacks. The authors demonstrate a successful Rowhammer attack on EdDSA, but also show that EdDSA, as used in the TLS, SSH and IPSec protocols, remains immune to their attack.

Signature Correction Attack. More recently, Rowhammer was used by Mus et al. [20] to target the NIST Round 2 competitor, signature scheme LUOV. The attack works by injecting faults via Rowhammer and then tracing the fault to the faulty signature output, recovering a fraction of the internal secret bits in the process. This process continues until a breakdown in the system of multi-variate equations is reached, enabling full key recovery. A more enhanced version, called the Signature Correction Attack, was developed by Islam et al. [21] to target Crystal-Dilithium, the finalist of the NIST post-quantum signature competition. The attack again uses Rowhammer to inject faults during signing, and subsequently corrects faulty signatures to deduce key bits of Crystals-Dilithium. This shows that post-quantum schemes are also vulnerable to fault injection attacks.

In this work, we demonstrate that signature schemes in common cryptographic libraries are still vulnerable to software-only fault injection attacks. Specifically, we employ Rowhammer and **Signature Correction** adapted to work with (EC)DSA, Schnorr and RSA signatures, to achieve full key recovery. The main advantage of Jolt is that implementations commonly implement countermeasures against Bellcore and Nonce based attacks. Thus, most (EC)DSA and some RSA signature implementations remain vulnerable to Jolt.

Our Contribution

We present a fault injection attack on popular signature schemes:

- Our technique works by co-locating with the victim and injecting faults into the victim memory space, e.g., using Rowhammer. The faulty signatures are post-processed using a novel technique called Jolt, an ElGammal style-specific Signature Correction Attack, which yields the signing key. The attack applies to ElGammal style signature schemes such as (EC)DSA, Schnorr and RSA signatures.
- We analyzed popular cryptographic libraries such as `OpenSSL, WolfSSL, LibreSSL, Microsoft SymCrypt, Amazon s2n` and determined that their TLS 1.2 and 1.3 implementations are vulnerable to the proposed attack. We have also reviewed their FIPS hardened versions which is slightly less efficient but still vulnerable to our attack. Further in `Microsoft Symcrypt`, only the first signature is checked, leaving further signatures unprotected.
- We demonstrate full ECDSA signing key recovery in `OpenSSL`. Specifically, with faults injected in less than 2 hours of the online phase, we recover 192 bits using 515 signatures, which are then post-processed with 2^{32} complexity to yield the full 256-bit ECDSA signing key. This is the first end-to-end demonstration of a signature correction attack in a real-world setting in TLS server-client communication.
- In addition to existing DDR3 Rowhammer vulnerability profiles, 4 new DDR4 DRAM chips from various vendors and 1 new DDR3 DRAM chip are profiled.
- Further, we demonstrate that despite extensive fault checks implemented in `OpenSSL`, RSA signatures are still vulnerable to Signature Correction Attacks, resulting in the gradual exposure of bits of the signing key.

- Finally, while we demonstrate the viability of our attack on TLS, other security protocols that rely on signatures might also be potentially vulnerable.

1.1. Outline of the Paper

The rest of the paper is organized as follows. In Section 2, we summarize known attacks exploiting leakages and briefly explain Rowhammer fault injection and proposed countermeasures in the literature. In Section 3, we provide the threat model for Jolt. In Section 4, we explain the Jolt Signature Correction Attack on secret key and nonce leakage on ECDSA. In Section 5, we show how Jolt works on Schnorr and RSA signatures. In Section 7, we present the results of our experiments performing Jolt on TLS Handshakes. Additionally, we present a vulnerability analysis of popular cryptographic libraries, a summary of possible countermeasures and finish with the conclusions.

2. Background

2.1. Partial Leakage Attacks on ECDSA

The ECDSA primitives are briefly summarized in the Appendix. In this section, we summarize techniques for exploiting partial leakages from (EC)DSA.

2.1.1. Exploiting Leakages from the nonce k. A standard assumption in cryptographic schemes is that nonces are collected from uniform sources. In practice, however, this is difficult to achieve, e.g., because there might be inherent biases in the random number generation mechanisms, or some bits might be revealed either due to implementation bugs or through side-channel leakages.

It is well known that leakage of even a small number of nonce bits may be exploited to recover the key. Such leakage can be used to formulate the problem of recovering the key and solve a version of the *hidden number problem (HNP)* introduced by Boneh and Venkatesan [3]. In practice, there is always a trade-off between the size of a leakage and the number of collected signatures needed to solve the HNP.

One can reduce HNP to Closest Vector Problem (CVP) and solve the lattice problem depending on the key size and the number of leaked nonce bits. For example, Lattice-based techniques [4], [5], [6], [7], [8], [9] allow efficient key recovery for small signature sizes, such as 2 bit leakage for 160-bit signatures and at least 4 bit leakage for 256-bit signatures with few (100s) signatures. On the other hand, lattice based techniques do not work for less data leakage [22], [2]. For less than 2-bit leakage, even extremely subtle leakages have been exploited to yield full key recovery for less than 192-bit ECDSA signatures. [2], [22], [23].

De Mulder et al. [24] revisited Bleichenbacher's idea of using Fast Fourier Transformation (FFT) [25] by deploying BKZ-based approach to reduce the complexity of collecting signatures for collision search. They succeeded in recovering the full secret key by using 5 MSB nonce bits of 4000 signatures. Aranha et al. [22] improved the idea by using a GLV/GLS decomposition technique to reduce the need of nonce leakage to a single bit with a signature budget of 2^{33} for 160-bit ECDSA.

In [2], Aranha et al. exploit small timing differences in the Montgomery ladder implementation to guess the MSB of nonces in ECDSA with high probability. They succeed in reducing the need for nonce leakage and the number of signatures required for key recovery. That said, still an impractically high number of signatures (with nonce leakage) are needed for breaking ECDSA for more than 192-bits.

2.1.2. Exploiting Leakages from the Secret Key d. Partial leakage attacks on ECDSA concentrate on the leakage from biased nonces. De Michelli et al. [26] note that there are no known attacks capable of exploiting scattered partial leakages from the secret key of ECDSA. We will explain our method for overcoming this problem in Section 4.3.2.

2.2. Fault Injection via Rowhammer

As the complexity of the programs and computational power of processors increase, memory systems became the limiting factor in terms of both capacity and latency. This trend forced modern DRAM technology to become more compact. As DRAMs get denser, they also become more prone to disturbance errors [27]. Therefore, DRAM rows need to be periodically refreshed due to charge leakage in the capacitors. If one memory row is accessed repeatedly, it might cause interference with the neighboring rows due to the dense memory structure, resulting in faster leakage. If the refresh rate cannot keep up, bit flips may occur. This phenomenon, which is known as the *Rowhammer vulnerability*, was first introduced by Kim *et al.* in 2014 [27]. Using Rowhammer, an attacker is able to inject bit flips into the victim memory, even if the attacker resides in a process logically isolated from the victim process.

Modern CPUs and memory systems isolate processes by two distinct levels of address translations. The Memory Management Unit (MMU) is responsible for translating virtual addresses to physical addresses and managing page allocations. The `pagemap` file in Linux systems stores virtual to physical address translations and requires admin privileges to read. The physical address is then further translated into channels, ranks, banks and columns inside the DRAM by the memory controller. To successfully flip bits via Rowhammer, an attacker needs to bypass these translation barriers and find physically neighbor rows in DRAM. The DRAM address translation is not publicly disclosed for modern CPUs. However, Pessl *et al.* [28] showed that the translation is deterministic and can be reverse engineered. After finding physically neighboring rows, an attacker can repeatedly access their own rows and cause bit flips in the neighboring rows.

Further improvements of Rowhammer and demonstrations in practical attacks followed. Seaborn *et al.* [29] improved Rowhammer by hammering in two rows that sandwich the victim row, resulting in faster leakage in the victim cells. Gruss *et al.* [30] achieved root access with opcode flipping in `sudo` binaries. Gruss *et al.* [31] and Ridder *et al.* [32] have shown that Rowhammer can even be applied through JavaScript in browsers remotely. Tatar *et al.* [33] and Lip *et al.* [34] have shown that Rowhammer can be made to work remotely over a network. Further, Rowhammer has been shown to work in cloud environments [35], [36] and heterogeneous FPGA-CPU platforms [18]. In addition to that, Veen *et al.* [37] have shown that Rowhammer attack can be implemented on commodity mobile platforms. Kwong *et al.* introduced RamBleed [38], an attack that exploits the data dependency of Rowhammer to recover bit values directly from victim memory. Rowhammer has also been used to target cryptographic libraries, e.g., Mus et al.[20] and Islam et al.[21] used Rowhammer to inject faults into two post-quantum signature schemes, LUOV and Dilithium, to achieve key recovery. Rowhammer has also been used to inject faults into machine learning models to cause misclassification [39].

Numerous techniques have been proposed to detect [40], [41], [42], [43], [44], [45], [46], [47] and mitigate [31], [37], [48] Rowhammer. Gruss *et al.* [30] have shown that all of these countermeasures are ineffective. A number of countermeasures require modifications to the hardware, bootloader or a BIOS update [48], [49], [50], [27], [51], [52]. In practice, such techniques are rarely implemented. Cojocar *et al.* [53] in 2019 reverse engineered the ECC memories showing that ECC countermeasure is not secure either. Another hardware countermeasure, Target Row Refresh (TRR) on DDR4 chips, has also been bypassed [54], [55] using n-sided and non-uniform Rowhammer patterns. Ridder *et al.* [32] applied this work to attack TRR enabled DDR4 memory from JavaScript and claimed that more than 80% of the DRAM chips in the market are still vulnerable to the Rowhammer attack.

3. Threat Model

Our model is similar to those in other Rowhammer attacks [27], [30], [56], [35], [36] where the attacker needs co-location in the same system as the victim. Specifically our threat model assumes

- The adversary runs a process with read write access to DRAM devices shared with the victim. The adversary has the ability to run a TLS client and connect to the server run by the victim.
- The victim and the adversary may, but do not have to, run on the same kernel. The kernel(s) are considered secure and free of any software vulnerabilities with process isolation in place to logically separate the adversary from the victim. In fact, the adversary and the victim may run on completely separate machines as long as the DRAM is shared.
- The adversary runs isolated as a common user process with no root privilege hence has no access to services that expose mapping between virtual and physical memory addresses.
- The shared DRAM devices are vulnerable to Rowhammer-induced bit flips. While certainty is not required, higher probability flips will yield faster attacks.

Note that the co-location can range from trivial to subtle, i.e. victim and attacker processes may run on the same host, or in virtual machines hosted on a shared server, or even in separate CPUs sharing a memory subsystem. Microarchitectural attacks commonly assume some form of co-location. Meltdown, for instance, requires execution on the same processor, Spectre assumes a shared Branch Prediction Unit, while cross-hyperthread MDS attacks require co-location with the victim on the same core.

4. Jolt: Signature Correction Attack

The Jolt attack proposes novel algorithms targeting ElGammal style signature such as (EC)DSA, Schnorr, and RSA signatures. The Signature Correction Attack is a framework to use fault injection, to recover secrets by correcting the faulty signatures, as demonstrated on LUOV [20] and Dilithium [21]. We introduce Jolt, a specialized Signature Correction Attack which specifies where to inject the faults, how to correct the faulty signatures, and how to recover the remaining bits on ElGammal style signatures.

These signature schemes are commonly used to secure TLS handshakes, SSH and IPSec protocols. Jolt exploits the fact that most cryptographic implementations of (EC)DSA do not verify the signature after a signing operation. Hence, before the signing operation Jolt injects faults into the secret signing key, collects the resulting faulty signatures and using a novel post-processing technique recovers bits of secret key (or nonce). Note that here, for simplicity, we demonstrate the details of the attack on ECDSA. Note also that, in Section 7 we are going to use Jolt to exploit TLS Handshake and recover full signing key d.

Similar to other Signature Correction based attacks [20], [21], Jolt works in three distinct phases:

1) **Offline Memory Profiling Phase:** The attacker studies the memory system to identify vulnerable (flippy) locations by allocating memory pages and running hammering experiments. This is the most time consuming part of the attack. However, the victim does not have to be actively running (or even present) on the machine during this phase.
2) **Online Faulting Phase:** In this phase, the attacker and victim are co-located on the same memory subsystem. While the victim is performing ECDSA signing operations, the attacker injects faults in the previously studied (but since released) memory pages. Any resulting faulty signatures are collected by the attacker from the network traffic.
3) **Offline Post-processing Phase:** This is where the Signature Correction Algorithm is used on each faulty signature; first, the error pattern is recovered and the corresponding secret key bit is deduced from the error pattern. If the number of faulty signatures is not sufficient

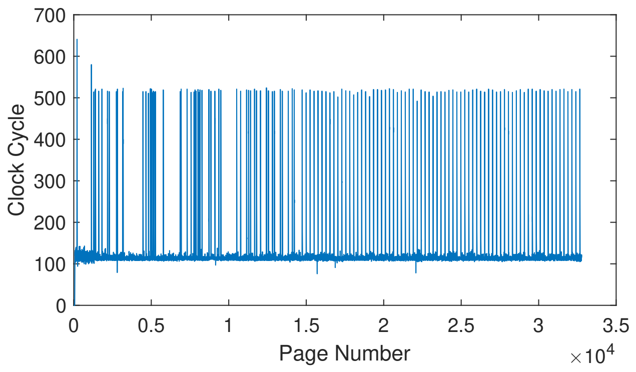

Figure 1: The physical addresses of pages are contiguous when the peaks found by SPOILER are equally distant.

to fully recover the remaining key bits using exhaustive search, we run a modified version of Shanks' algorithm.

In what follows we explain these steps in greater detail.

4.1. Offline Memory Profiling Phase

To inject faults via Rowhammer into ECDSA secrets, e.g. the secret signing key d or the nonce k, the attacker needs to identify the flippable locations in the physical memory and collect addresses that will fall into the same bank and in adjacent hammering rows.

4.1.1. Finding Contiguous Memory. With the aim of allocating the attacker rows near the victim rows, we first find physically contiguous memory addresses. For finding contiguous physical memory, one of the methods is using *huge pages* which guarantees consecutive virtual addresses to be contiguous in physical address space as well. However, it requires a special system configuration. Since we do not assume any special configuration or privilege, we use SPOILER vulnerability [57] in Intel CPUs, which allows us to detect 8MB of contiguous chunks of physical addresses from the user space. Figure 1 shows the equidistant peaks, which correspond to pages that belong to the same 8MB memory chunk. We refer readers to [57] for details of SPOILER implementation.

4.1.2. Finding Neighbor Rows. Following the allocation of the contiguous memory chunk, we find the pages that are mapped into the same DRAM bank. For this, we leverage row conflict side-channel [28] which gives us the pages mapping to the same bank. As we know, accessing two addresses from two different banks causes low memory latency compared to accessing them from the same bank. The reason behind this is that accessing two addresses from the same bank will cause one row to be written back to its original position before another row can be loaded into the row buffer. The measurements from the experiment are shown in Figure 2, where we have set a threshold value of 380 clock cycles to extract the addresses that are located in the same DRAM bank.

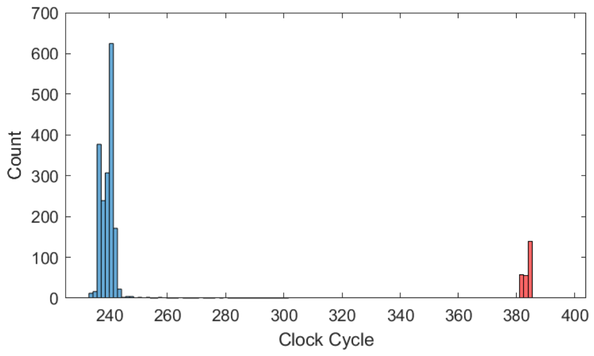

Figure 2: Histogram of access times to the pages in the buffer. The pages that are physically located on the same DRAM bank take longer to access due to row conflict. We set the threshold to 380 clock cycle for distinguishing the pages on the same DRAM bank.

4.1.3. Finding Flippy Locations. After successfully allocating rows in the same DRAM bank within a contiguous physical memory, we mounted a practical Rowhammer attack. For systems with DDR3 DRAM, we implement a double-sided Rowhammer attack where each victim row is surrounded by an aggressor row from top and bottom. For DDR4 systems, double-sided Rowhammer does not work due to the TRR mitigation. Therefore, we implement n-sided Rowhammer attack where multiple victim rows are targeted in an alternating aggressor and victim row placement. Then, we identify the rows with bit flips by repeatedly accessing (hammering) the aggressor rows and checking the value changes in the victim row.

4.2. Online Phase: Injecting Faults into ECDSA

Using Rowhammer, we can (probabilistically) target a number of variables, e.g., the secret signing key d, the nonce k before or after its inverse is computed, or its inverse k^{-1} or the signature s[1]. For the attack to be successful, however, we need the fault to be

- **Traceable** to an exhaustible number of possible fault patterns in the faulty signature; and
- **Masking a secret** to permit us to deduce the secret.

Traceability ensures that we can recover the error; however, for the error pattern to be useful, it should apply to a secret. We identify two targets in ECDSA, i.e. the nonce k (before inversion) and the signing key d either of which can be chosen in this stage for Rowhammer fault injection.

4.2.1. Mapping the Secret to Flippy Locations. To be able to flip bits in the ECDSA key, we need to manipulate the memory so that the key is located in one of the victim rows that we identified in Section 4.1.3. After unmapping the flippy locations, the victim generates/loads the private

1. Note that another approach would be to bypass or modify computational steps targeting code/instructions instead of data in memory. However, such attacks are beyond the scope of this work

key. Since the Linux Buddy Allocator allocates recently unmapped physical pages on the page cache frame in first-in-last-out order [58], the private key is mapped into the flippy row. Alternative methods that can be used for mapping the victim are spraying [29], [35], [56], waylaying [59], [38], [60] or grooming [37].

In what follows, we consider concrete attack targets and discuss the error recovery mechanisms in the offline phase.

4.3. Offline Phase: Post-processing

4.3.1. Signature Correction with Faulty d. Another option is to inject the fault to the private key d after key generation/during signing. $\bar{d} = d + \Delta_d$ where Δ_d is the injected fault in d. Since d is used only in signature generation s, r stays the same and faulty s is $\bar{s} = k^{-1}(H(M) + \bar{d}r) \bmod n$. Therefore, the faulty signature is generated as $(r, \bar{s})$. It is obvious that signature is going to fail during verification. The details of Signature Generation and Verification steps are given in Algorithms 1 and 2.

Algorithm 1 ECDSA Signing with a faulty private key

```
1: Input: A message $M \in \{0,1\}^*$, private key $\bar{d} = d + \Delta_d$,
2: Output: Faulty signature $(r, \bar{s})$
3: Choose a nonce/ephemeral key $k \in Z_n^*$.
4: Compute the curve point $R = kP$, and compute the $x$
   coordinate $R_x = r = (kP)_x$.
5: Compute $\bar{s} = k^{-1}(H(M) + \bar{d}r) \bmod n$.
6: return Signature pair $(r, \bar{s})$.
```

Algorithm 2 ECDSA faulty signature verification attempt

```
1: Input: Signature $(r, \bar{s})$, a message $M \in \{0,1\}^*$, public
   key $Q \in E$,
2: Output:
3: Compute $H(M)$.
4: Compute $\bar{w} = (\bar{s})^{-1} = k(H(M) + \bar{d}r)^{-1} \bmod n$.
5: Compute $\bar{u}_1 = H(M)\bar{w} \bmod n$.
6: Compute $\bar{u}_2 = r\bar{w} \bmod n$.
7: Compute $\bar{R} = \bar{u}_1 P + \bar{u}_2 Q \in E$.
8: Take $\bar{r} = (\bar{R})_x$.
9: If $\bar{r} == r$: verify, Else reject.
10: return Reject since $\bar{r} \neq r$.
```

Claim 1. **An observation on Correctness**
The difference introduced by the fault can be captured as $\bar{R} + \Delta_d \bar{u}_2 P = R$.

Proof Sketch. Using the public key $Q = dP$ and $\bar{d} = d + \Delta_d$. Also $\bar{w} = \bar{s}^{-1} = k(H(M) + \bar{d}r)^{-1}$. Expand the LHS as follows

$$\begin{aligned}\bar{R} + \Delta_d \bar{u}_2 P &= \bar{u}_1 P + \bar{u}_2 Q + \Delta_d \bar{u}_2 P \\ &= k(H(M) + \bar{d}r)^{-1}(H(M) + dr + \Delta_d r)P \\ &= kP = R\end{aligned}$$

Finding Δ_d. We can efficiently find Δ_d from the Claim 1. Note that, $\bar{R}$ and $\bar{u}_2$ are generated for verification, P is publicly known. Since r is known, we can find R by inserting into underlying elliptic curve E. All that remains is to use $\bar{R} + \Delta_d \bar{u}_2 P = R$ to check all possible error patterns Δ_d. For m-bit private key d, we need $2m$ trials to find 1-bit fault of the injected Δ_d. The doubling is due to the fact that we need to try both positive (0 to 1 faults) and negative error patterns (1 to 0 faults). Further, we need $2^2\binom{m}{2}$ trials to find a 2-bit fault and recover 2 bits of Δ_d. Once the fault pattern Δ_d is found, the log of its magnitude gives us the location of the fault, and its sign gives us the value of the bit of d, e.g. negative is decoded as a logic 1 bit and positive into a logic 0 bit value.

4.3.2. Recovering the remaining bits of d. While Rowhammer is a great tool, it allows recovery of only a limited number of key bits due to the unflippable memory cells and repeating bit flips. Even after recovering most of the bits using Rowhammer, it takes significant time to exhaust all the remaining bits. Assuming $t < m$ bits are recovered out of $m = \log(n)$. The DLP problem with the information gained from bits of the secret key d, which we call *hints*, can be formalized as follows.

Definition 1 (Generalized DLP with Hints). Given a finite cyclic group G of order n, a generator α of G, an element $\beta \in G$, and $t < \log(n)$ randomly selected bits of d, find the integer d, $0 \leq d \leq n-1$, such that $\alpha^d = \beta$.

Note that, hints are randomly distributed over d. Unfortunately, efficiently recovering d with more than few chunks of unknown bits is an open problem [26].

The most obvious but least viable approach is to run an exhaustive search attack on the remaining $w = m - t$ bits on the public key $Q = dP$. We gain exponentially with increasing t, however, ideally, we would like to take advantage of DLP algorithms with square-root complexity.

We overcome this open problem by modifying Shanks' Algorithm as follows.

Modified Baby Step-Giant Step Algorithm. We can adapt the Shanks' Algorithm (see Section 2 in Appendix) to take advantage of the t known bits of d recovered by the Signature Correction Algorithm. For this, we first represent d as the sum of two m-bit numbers as the known and unknown parts of d, $d^{(k)}$ and d', respectively, i.e., $d = d^{(k)} + d'$. The known part $d^{(k)}$ consists the recovered t bits of d, in which known bits (either 0 or 1) are in place whereas the rest of the bits (unknown bits) are 0. The unknown part d' consists of the unknown $w = m - t$ bits of d, $d' = (d'_{m-1}, \ldots, d'_0)_2$ in which known t bits are 0.

$$Q = dP = (d' + d^{(k)})P \quad \Rightarrow \quad Q' = Q - d^{(k)}P = d'P$$

Hence our DLP problem becomes to find d' such that $Q' = d'P$ where d' has t zeros and w unknown bits in binary representation.

We optimize Shanks' algorithm to solve the size-reduced DLP problem as follows. First note that, we can skip the known bits in the algorithm by eliminating (neither computing nor storing) known bits and using related values from the Lookup Table. Every bit we have recovered will reduce the computation and storage needs by half. We start by eliminating known bits from the equation then continue pre-computing some points on the Elliptic curve using the bits of the unknown part. Then we create a Lookup table for the first w_1 unknown least significant bits of d' (Baby Steps). Finally, we search for a point on the Elliptic curve by $w_2 = w - w_1$ most significant part of the unknown bits that matches a point in the Lookup table (Giant Steps). When we find a match, we can deduce all the unknown bits of d, namely d', recovering $d = d' + d^{(k)}$. Note that, to keep track of the bits in the algorithm, we represent d' in two different forms: as m bits and as w bits (only the unknown bits) as follows:

- m-bit representation is $d' = (d'_{m-1}, \cdots, d'_0)_2$ where $d'_i = 0$ if d'_i is known, else, $d'_i = d'_{w_j}$.
- w-bit representation is $d' = (d'_{w-1}, \cdots, d'_0)_2$.

The proposed Modified Baby Step-Giant Step Algorithm is shown in Algorithm 3. We can summarize Baby Step-Giant Step Algorithm in four main steps:

1) **Eliminating Recovered Bits:** In this step, we reduce the size of the DLP problem by eliminating known bits from the equation $Q = dP$. Size reduced problem is $Q' = d'P$ in which d' is still an m-bit number in which t known bits are set to zero.
2) **Precomputation Table:** For every unknown bit in d', a base point is pre-computed as a power of 2, i.e., $A_j = 2^i P$ in which i is the index for m-bit representation and j is the index for w-bit representation. In other words, we change the representation of d' from m-bit representation to w-bit representation. Store the w-bit index of the bit j, the m-bit index of the bit a_j and the related base point $A_j = 2^i P$ in a Precomputation Table.
3) **Baby Step:** This step is similar to the Baby-Steps in Shanks' Algorithm. We store only the Baby-Step points. In other words, we find the potential curve points and its first w_1 unknown bits of m-bit representation and store them in Lookup table. Every point can be computed as $P_k = \Sigma_{j=0}^{w_1-1} k_j A_j$ and m-bit coefficient is $b_k = \Sigma_{j=0}^{w_1-1} k_j 2^{a_j}$ for the least significant w_1 unknown bits $k = (k_{w_1-1}, \cdots, k_0)_2$ and stored in a Lookup Table as (k, b_k, P_k).
4) **Giant Step:** This step is similar to the Giant-Steps in Shanks' Algorithm. In the search phase, we go through 2^{w_2} possible curve points for w_2 unknown most significant bits of d'. Every point can be computed by $B_i = \Sigma_{j=0}^{w_2-1} i_j A_{j+w_1}$ for $i = (i_{w_2-1}, \cdots, i_0)_2$. After every computation, we check if $Q' - B_i$ is equal to any of stored Lookup table points P_k. Unknown bits of d,d', are $(i)_2 || (k)_2$ equivalently $d' = c_i + b_k$ for the matching $Q' - B_i = P_k$. Return $d = d' + d^{(k)}$. Note that, known bits in d' and unknown bits in d^k are set to zero. Therefore, there is no carry bit in the summation of these numbers.

Finally, we note that Shanks' algorithm is flexible, e.g. storage usually imposes a more significant restriction and

Algorithm 3 Modified Baby-Step Giant-Step DLP Solver

```
    Input: Cyclic group $E$ with generator $|P| = n$, and public key $Q = dP$,
    $m$-bit scalar number $d$ in which certain $t$ bits are known.
    Output: Unknown $w = m - t$ bits of $d$, $d'$.
    Initialization
 1: Represent $d$ as $d = d^{(k)} + d'$
 2: Compute $Q' = Q - d^{(k)}P$  // $Q' = d'P$
 3: Choose $w_1, w_2$ s.t. $w = w_1 + w_2$ for mem/cycle budgets $2^{w_1}, 2^{w_2}$
    Precomputation
 4: $j = 0$, $A = P$
 5: for $i = 0$ to $m - 1$ do
 6:     if $d'_i$ is unknown then
 7:         $A_j = A$  // $A_j = 2^i P$
 8:         $a_j = i$  // index of $m$-bit representation
 9:         Store $(j, a_j, A_j)$ in Precomputation Table
10:         $j = j + 1$
11:     end if
12:     $A = 2A$
13: end for
    Baby Step Computations
14: for $k = 0, 1, \ldots, 2^{w_1} - 1$ do
15:     $P_k = 0$, $b_k = 0$
16:     for $j = 0$ to $w_1 - 1$ do
17:         if $k_j = 1$ then // $k = (k_{w_1-1}, \cdots, k_0)_2$
18:             $P_k = P_k + A_j$
19:             $b_k = b_k + 2^{a_j}$  // baby step indices
20:         end if
21:         Store triplet $(k, b_k, P_k)$ in Lookup Table $\mathbf{T}$.
22:     end for
23: end for
24: Sort Lookup Table $\mathbf{T}$ w.r.t. $P_k$ column.
    Giant Step Computations
25: Set $w_2 = w - w_1$
26: for $i = 0, 1, \ldots, 2^{w_2} - 1$ do
27:     $B_i = 0$, $c_i = 0$
28:     for $j = 0$ to $w_2 - 1$ do
29:         if $i_j = 1$ then // $i = (i_{w_2-1}, \cdots, i_0)_2$
30:             $B_i = B_i + A_{j+w_1}$
31:             $c_i = c_i + 2^{a_{j+w_1}}$  // giant step indices
32:         end if
33:     end for
34:     if $Q' - B_i$ matches any $P_k$ in $\mathbf{T}$ then
35:         $d' = (c_i + b_k)_2$
36:         Return $d = d' + d^{(k)}$
37:     end if
38: end for
```

computation. One can change the algorithm to precompute a smaller table with $j = 0, \ldots, w_1 < w$ values and then search in a larger space $i = 0, \ldots, w_2 > w_1$ values with the condition $w_1 + w_2 = w$.

Complexity. The modified Shanks' algorithm iterates works in three phases:

- **Precomputation:** In this phase we scale the base point P with all powers of 2 and stores the ones for the unknown locations. Since at most only m iterations and storage are needed, this part is negligible in complexity.
- **Baby step computations:** For Modified Baby Step we need to compute

$$\mathcal{A}(w_1) = \Sigma_{i=1}^{w_1} (i-1) \binom{w_1}{i}$$

point additions in E. Note that $\binom{w_1}{i}$ is the number of k values of weight i, each of which require $i-1$ additions to

assemble kP together from the precomputed A_j values[2] For $w_1 = 32, 35, 40$ we obtain $\mathcal{A}(w_1) = 2^{35.9}, 2^{39.0}$, and $2^{44.2}$, respectively. In table $\mathbf{T}$, we store 2^{w_1} points along with the indices.
- **Giant step computations:** The Modified Giant Step does not need to store any data but still needs to compute $\mathcal{A}(w_2)$ point addition operations for B_i, and 2^{w_2} point additions to compute $Q' - B_i$ for the comparisons.

All in all assuming uniform distribution of recovered bits, and an even split, i.e. $w_1 = w_2 = (m-t)/2$, the storage and time complexities of the modified Baby-Step Giant Step Algorithm are in the order of $O(\sqrt{2^{m-t}})$.

4.3.3. Signature Correction with Faulty nonce k. We consider the scenario that a fault is injected into k after kP is computed, i.e., $\bar{k} = k + \Delta_k$ where Δ_k is the injected fault in k. Note that, $\Delta_k = 2^{i-1}$ for the bit flip in the i^{th} bit of k for 1-bit flip. If there are more than 1-bit flips in k then Δ_k can be written as $\Sigma_{i \in I} 2^{i-1}$ where I is the set of bit flip positions in k. Since the fault is injected into k after r is generated, s is faulty $\bar{s} = \bar{k}^{-1}(H(M) + dr) \bmod n$ whereas r is not faulty. Here $H(.)$ represents a cryptographic hash function such as SHA-256. Therefore, the faulty signature is $(r, \bar{s})$. In Algorithm 4 and 5, you can find the details of Signature Generation and Verification steps for a faulty nonce.

Claim 2. **An observation on Correctness**
The difference introduced by the fault can be captured as $\bar{R} = R + \Delta_k P$

Proof Sketch.

$$\bar{R} = \bar{u}_1 P + \bar{u}_2 Q = \bar{k}(H(M) + dr)(H(M) + dr)^{-1} P \\ = \bar{k}P = R + \Delta_k P.$$

Finding Δ_k We can find Δ_k by correcting the faulty signature and then checking a limited number of possible error patterns. Thanks to the Claim 2, it is enough to subtract $\Delta_k P$ from $\bar{R}$ and check if $(\Delta_k P - \bar{R})_x = r$ or not. Note that, $\bar{R}$ is calculated for verification, P is publicly known and r is known since it is a part of the signature. For an n-bit nonce, we need $2n$ trials to recover a 1-bit fault, $2^2\binom{n}{2}$ trials to check for a 2-bit fault patterns. Once the fault pattern Δ_k is found, the magnitude of Δ_k gives us the location of the fault, and the sign gives us the value of the bit of d at this location. For example, from $\Delta_k = -2^3$, we can deduce that $d_3 = 1$. Since the direction of change is negative, the original bit value must have been a logic 1 in this position.

2. The alternative (more costly) approach would be to compute kP for all possible baby step index values without removing the recovered locations, by repeatedly adding P to itself and storing the ones with indices that match the ones at the recovered bits. This would require about $2^{w_1+t/2}$ additions.

Computing the d from recovered bits of k. An alternative approach is to exploit the k bits recovered by Signature Correction using existing techniques such as lattice reduction [4], [5], [6], [7], [8], [9] or more powerful FFT based techniques [25], [24], [22], [23], [2]. Lattice reduction techniques allow very efficient key recovery (with only a few hundred signatures) but require 2 bit leakages for 160-bit signatures and at least 4 bit leakages for 256-bit signatures.

Due to restrictions imposed by our fault injection mechanism, we are only able to recover a single bit from k per faulty signature[3]. Hence, we are left with only Bleichenbacher's FFT based approach [25] which was shown recently [2] to recover the secret key d with bias less than one bit in k per signature (r, s): $s = k^{-1}(e + dr) \pmod{n}$. The **less than** here means that the attack works with recovered bits that are only probabilistically correct. This is not a problem in our attack, where we recover one exact bit per nonce from each faulty signature. Aranha et al. [22] employed the FFT approach to attack 160-bit ECDSA where k is 1-bit biased. They can succeed in retrieving the secret key with 2^{33} signatures in about 2^{37} time and with 2^{33} memory complexity. To recover a 256-bit ECDSA key d, they estimate that HNP can be solved with 2^{52} signatures with 1-bit leakage. A more recent work [2] takes advantage of leakage in the Montgomery ladder implementation of ECDSA for FFT based key recovery with fewer signatures. They estimate that 2^{20} and 2^{45} signatures, for 160-bit and 256-bit curves, respectively, will be required in the best scenario with 1 bit nonce leakage.

As it relates to our proposed attack, the FFT based approach may be used to exploit the 1-bit leakage from the nonces, however, at a significant cost of 2^{45} faulty signatures. The lower security setting, i.e., for 160-bit curves with 2^{20} signatures appears to be practical. Finally, we want to note that, exploiting leakage from k appears more difficult compared to exploiting leakage from d as it requires significantly more signatures. However, it is also much more likely for a memory device to be vulnerable to single bit flips than many, as required by the former approach, see Section 7.2. Hence, while much more costly, the FFT based approach might be the only option for a successful attack in certain hardware configurations.

Simultaneously exploiting k and d bits. An interesting open question is how one might pool together the leakages recovered via Signature Correction to simultaneously exploit recovered bits of d and k to improve over the attacks discussed earlier. We are not aware of any such techniques in the literature.

5. Applying Jolt to Other Signature Schemes

5.1. DSA Signatures

While our description of the Signature Correction Algorithm assumed ECDSA (and used additive notation), the

3. Double flips are rare and even when they occur are likely non-contiguous, hence not useful for exploitation using existing techniques

Algorithm 4 ECDSA Signing with a faulty nonce

```
1: Input: A message $M \in \{0,1\}^*$, private key $d$,
2: Output: $(r, \bar{s})$ signature.
3: Choose a nonce/ephemeral key $k \in Z_n^*$.
4: Compute the curve point $R = kP$, and compute the $x$
   coordinate $r = (kP)_x$.
5: Inject a fault in $k$, $\bar{k} = k + \Delta_k$.
6: Compute $\bar{s} = (\bar{k})^{-1}(H(M) + dr) \bmod n$.
7: return Signature pair $(r, \bar{s})$.
```

Algorithm 5 ECDSA faulty signature verification attempt

```
1: Input: Signature $(r, \bar{s})$, a message $M \in \{0,1\}^*$, public
   key $Q \in E$,
2: Output: Reject.
3: Compute $H(M)$.
4: Compute $\bar{w} = (\bar{s})^{-1} = \bar{k}(H(M) + dr)^{-1} \bmod n$.
5: Compute $\bar{u}_1 = H(M)\bar{w} \bmod n$.
6: Compute $\bar{u}_2 = r\bar{w} = r\bar{k}(H(M) + dr)^{-1} \bmod n$.
7: Compute $\bar{R} = \bar{u}_1 P + \bar{u}_2 Q$.
8: Take $\bar{r} = (\bar{R})_x$
9: If $\bar{r} == r$: verify, Else reject.
10: return Reject since $\bar{r} \neq r$.
```

attack applies in an identical manner to DSA by only changing the group setting to a multiplicative one. To save space, we refrain from repeating the description.

5.2. Schnorr Signatures

Schnorr signatures are defined using a group G, of prime order q, with generator α, in which the discrete log problem is assumed to be hard. We also use a cryptographic hash function $H : \{0,1\}^* \to \mathbb{Z}_q$. For key generation we choose a private signing key $d \in \mathbb{Z}_q^*$. The public verification key is $\beta = \alpha^d$. To sign a message M:

- Choose random nonce $k \in \mathbb{Z}_q^*$.
- Compute $r = \alpha^k$.
- Compute $e = H(r||M)$, where $||$ denotes concatenation and r is represented as a bit string.
- Compute $s = k - de$.
- The signature is the pair (s, e).

Verification

- Compute $r_v = \alpha^s \beta^e$.
- Compute $e_v = H(r_v||M)$.
- If $e_v = e$ then the signature is verified.

If we inject a fault into d before signing, i.e., $\bar{d} = d + \Delta_d$ then the fault propagates into the signature $\bar{s} = k - \bar{d}e$ and verification fails since $\bar{r}_v = \alpha^{\bar{s}}\beta^e$ and thus, $\bar{e}_v = H(\bar{r}_v||M)$ and $\bar{e}_v \neq e$. This immediately points to an error correction strategy that will reveal Δ_d.

Claim 3. The difference introduced by the fault can be captured as $\bar{r}_v = r_v \alpha^{-\Delta_d e}$.

This means upon receiving the faulty signature $(\bar{s}, e)$ we can simply try fixing $\bar{r}_v$ by multiplying it with $\alpha^{\Delta_d e}$ for all possible Δ_d until the hash result e_v matches the received e during verification. As before, the error pattern allows us to deduce the corresponding bit of the signing key d.

5.3. RSA Signatures

The Signature Correction approach indeed does apply on RSA signature with little modification. Very briefly remember, in textbook RSA, a public key consists of integers (N, e), where $N = pq$ is a public modulus obtained as the product of two large primes p and q, and e represents the public exponent used in signature verification. In majority of TLS sessions we have $e = 2^{16} + 1$. The private key d is the secret signing exponent obtained as $d = e^{-1} \bmod \phi(N)$ where $\phi(N) = (p-1)(q-1)$. A textbook RSA signature on a message M is defined as $s = M^d \bmod N$. To verify the signature we check if $M = s^e \bmod N$ holds.

In practice, depending on the specifics of the scheme, i.e. PKCS#1v1.5, PKCS#1v2.1 or RSA-PSS signatures, the message is processed using a combination of hashing, padding and possible randomization via a nonce and then fed into the textbook version as message M described above.

Jolt is agnostic to the specifics, hence, we will assume the input M represents the preprocessed message. Assume during signing we inject a fault into the signing key $\bar{d} = d + \Delta_d$ thereby obtaining a faulty signature $\bar{s} = M^{\bar{d}} \bmod N$. The Signature Correction step would simply try for possible $\Delta_d = \pm 2^i$, where i ranges over the bits of N, until $\bar{s}M^{-\Delta_d} \bmod N$ hits the correct signature which we will know when it verifies using the public key, i.e. $(\bar{s}M^{-\Delta_d})^e = M \bmod N$.

5.3.1. Recovering the full RSA signing key. Using Signature Correction, we can recover one bit of the signing key d per faulty signature. This means as in the ECDSA case, we will obtain a collection of scattered bits of d. While there are numerous techniques to amplify partial leakages from p and q or $d_p = d \bmod p - 1$ or equivalently $d_q = d \bmod q - 1$, it remains an open problem to exploit a limited number of scattered leakages from d to recover the full key [26]. This means we need to recover a significant fraction of the bits of d, leaving only an exhaustible number of bits unknown.

6. Attack Strategies

Depending on the partial leakages recovered from (EC)DSA or RSA, one may choose the most optimal attack strategy as visualized in Figure 3. Red boxes show attacks that exploit nonce leakages and provide the number of signatures required for 256-bit ECDSA. The green boxes highlight the results of this work. For simplicity, we assume a conservative limit of 50 bits for tractable space/cycles.

7. Experiment Results

In this section we describe our experiments on TLS. We first describe Jolt experiments followed by an analysis of the handshake process through which faulty signature are collected in selected crypto libraries.

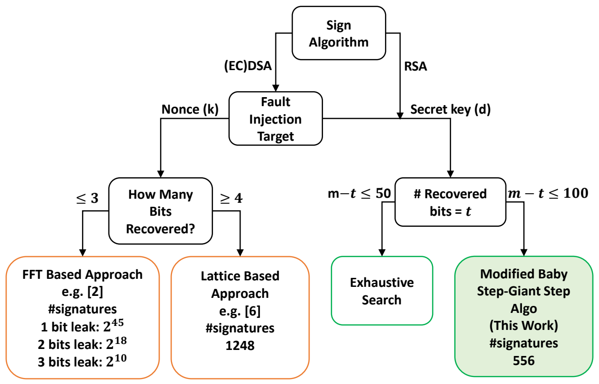
Sign Algorithm
(EC)DSA
RSA
Nonce (k)
Fault Injection Target
Secret key (d)
≤ 3
How Many Bits Recovered?
≥ 4
m−t ≤ 50
# Recovered bits = t
m − t ≤ 100
FFT Based Approach e.g. [2] #signatures 1 bit leak: 2^45 2 bits leak: 2^18 3 bits leak: 2^10
Lattice Based Approach e.g. [6] #signatures 1248
Exhaustive Search
Modified Baby Step-Giant Step Algo (This Work) #signatures 556

Figure 3: Summary of nonce/secret key leakage attacks on m-bit (EC)DSA and RSA signatures.

7.1. Experiment Setup

We evaluate 19 DRAM chips from the two most popular DRAM generations, DDR3 and DDR4. The raw memory profiles of 14 DDR3 DRAM chips are taken from [61]. The remaining 5 memory profiles are generated from our DRAM chips. DDR4 experiments are done on a Ubuntu 20.04.1 LTS system with Intel(R) Core(TM) i9-9900K CPU @3.60GHz and 4 DDR4 DRAM chips produced by Corsair, GSkill and Crucial. ECDSA fault injection experiments are done on a Ubuntu 16.04.7 LTS system with Samsung DDR3 DRAM chip with serial number M378B5773DH0 and Intel(R) Core(TM) i7-4770 CPU @3.4GHz. The row refresh interval for all experiments is set to the default value of 64 ms.

For the ECDSA experiments, we use `prime256v1`, a.k.a. the `NIST P-256` elliptic curve, where d is 256 bits. However, our attack applies to any other elliptic curve. We attack the OpenSSL v3.0.4 implementation, which is the latest stable version of OpenSSL, as this work is done.

The attacker can initiate the TLS connection request remotely as long as the IP Address of the TLS Server is known. Alternatively, the attacker can initiate the connection using the localhost IP as long as they are co-located in the same server. As explained in Section 3, the attacker needs to share with the server the same memory subsystem. To simplify the experiment setup, we use a single DRAM chip in the system to ensure both the server and the attacker use the same memory device.

7.2. Jolt Experiments

Offline Memory Profiling Phase is dependent only on the DRAM chip. Hence, we evaluate the Offline Memory Profiling Phase of our attack on 15 DDR3 and 4 DDR4 DRAM chips. Specifically, we take 128MB memory profiles from each DDR3 DRAM chip and 256MB from each DDR4 DRAM chip and count the number of bit flips that land on a 256-bit area in a 4KB page. In our setup, profiling 128MB on DDR3 takes 95 minutes and profiling 256MB on DDR4 takes 480 minutes. We observe that, the page offset of the 256-bit ECDSA key d is fixed in a 4KB memory page when it is generated in a new session since ASLR does not randomize the last 12 bits of addresses in heap memory. However, it changes between different library versions. Therefore, to analyze the effect of page offset, we sweep the page offset of d from `0x000` to `0xFFF` and take the average and standard deviation of bit flips. Table 1 shows the number of bit locations that can be flipped in d. We observe that the number of bit flips in d is highly dependent on the DRAM chip and less on the page offset. Although the number of possible bit flips is low in some DRAM chips, it is possible to profile another 128/256MB memory buffer and increase the number of recovered bits. For instance, in a 4GB memory with a memory buffer size of 256MB, we have a chance to increase the number of bits at most 16 fold. The upper bound is due to the overlapping locations. A more significant limitation is library crashes caused by excessive flipping in unintended memory locations. The last column in Table 1 marks the chips as vulnerable where the leakage in bits of d can be reliably amplified to enable full key recovery by collecting bits from multiple memory buffers.

	Brand	Serial Number	Size [GB]	# Flips in d / profile	Vuln?
DDR3	Corsair	CMD16GX3M2A1600C9	16	232 ± 7	✓
	Corsair	CML16GX3M2C1600C9	16	47 ± 7	✓
	Corsair	CML8GX3M2A1600C9W	8	7 ± 3	✗
	Corsair	CMY8GX3M2C1600C9R	8	245 ± 5	✓
	Crucial	BLS2C4G3D1609ES2LX0CEU	8	4 ± 2	✗
	Geil	GPB38GB1866C9DC	8	55 ± 7	✓
	Goodram	GR1333D364L9/8GDC	8	6 ± 3	✗
	GSkill	F3-14900CL8D-8GBXM	8	231 ± 8	✓
	GSkill	F3-14900CL9D-8GBSR	8	53 ± 8	✓
	Hynix	HMT351U6CFR8C-H9	8	253 ± 1	✓
	V7	V73T8GNAJKI	8	37 ± 6	✓
	PNY	MD8GK2D31600NHS-Z	6	37 ± 6	✓
	Integral	IN3T4GNZBIX	4	203 ± 12	✓
	Samsung	M378B5173QH0	4	17 ± 4	✗
	Samsung	M378B5773DH0	2	196 ± 8	✓
DDR4	Corsair	CMU64GX4M4C3200C16	64	255 ± 1	✓
	Corsair	CMK32GX4M2B3200C16	32	1 ± 1	✗
	GSkill	F4-3600C16D-16GVKC	16	196 ± 10	✓
	Crucial	CT8G4DFD824A.C16FF	8	2 ± 2	✗

TABLE 1: Number of possible bit flips in d calculated on 14 different DDR3 chips per profile (128 or 256MBs). In our setup, it takes 95 minutes to profile a 128 MB on DDR3 and 480 minutes to profile 256MB on DDR4 chips.

We demonstrated the attack over a network by targeting a TLS server. As the client, each time we initiate a connection request to the server, by injecting faults into the server memory, we receive the faulty signature. We implemented the end-to-end attack on OpenSSL ECDSA implementation in Samsung M378B5773DH0-2GB and flipped bits in the secret key.

In the **Online Phase**, we successfully flipped bits of the ECDSA key and collected faulty signatures. Although the shorter private key size compared to the previous work [20], [21] requires fewer bits to recover, it also introduces new

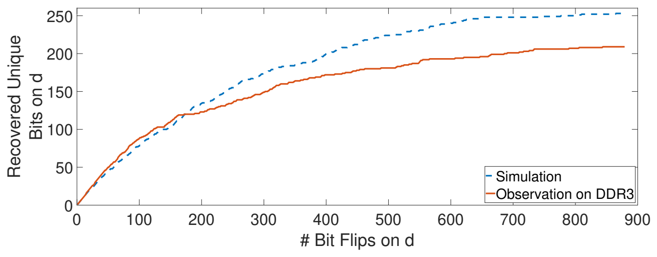

Figure 4: Recovered number of bits on an ECDSA private key d. Dashed line represents the simulated bit flips when the fault distribution is assumed uniformly random. Solid line represents the observed number bit flips on d on DDR3 DRAM.

challenges. Since the key size is 256-bit, which is much smaller than a page size 32,768-bit, most of the bits that can be flipped land outside of the key. In our experiments, bit flips that land outside of the key either do not affect the signature generation (a valid signature is generated) or the signing operation fails. Specifically, we attempted the online phase 29,918 times in 170 mins and it did not crash the operating system. In 22,845 trials, the library was not affected at all, i.e. it created a signature and verified successfully. In 6,192 trials, the library was affected, i.e. a flip was injected, but the attack failed due to library crashes. In 881 trials, we have seen flips on the secret key, d, and were able to recover a bit of d from a faulty signature using the Signature Correction.

In the first 117 mins of online phase, we collected 515 signatures. We have seen 1 time triple bit flips, 39 times double bit flips and 475 times single bit flips. We did not observe more than 3 bit flips for d.

In **Offline Post-processing Phase**, we recovered 192 unique bits $(\log_2(d) = 256)$ via Signature Correction as explained in Section 4.3.1 using 515 faulty signatures. The remaining bits are recovered using the modified Shanks' Algorithm with 2^{32} space and time complexity (See Algorithm 3.). Figure 4 shows, unique bits recovered on DDR3 is lower than the simulation with the assumption of random uniform distribution. We claim that the reason for a larger overlap rate in the bit flips is caused by the Buddy Memory Allocation system of the Linux kernel. The progression of recovered bit location are illustrated in Figure 5.

7.3. TLS Handshake Experiments

We can successfully initiate the Jolt attack in most commonly used TLS versions 1.2 and 1.3. Although there are a few key differences between the two, the main difference between the versions is the number of round trips (fewer in TLS 1.3) to settle the shared key between communicating parties. Our attack scenario is not affected by these differences. Therefore, from now on, we will only focus on TLS 1.3 and explain our attack.

Our attack targets the handshake protocol between a client and a server to recover the bits of the secret key of

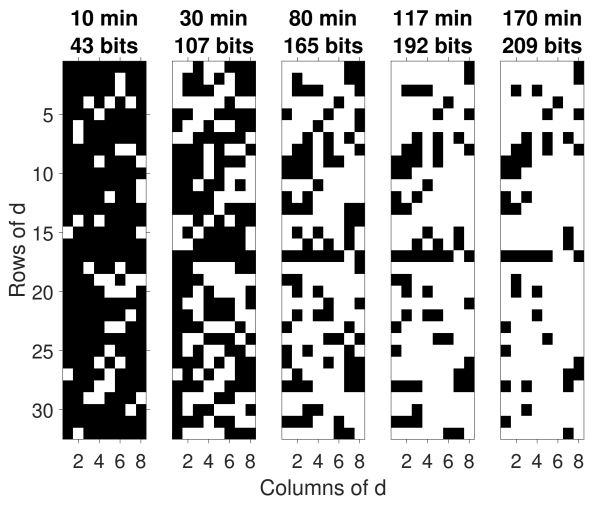

Figure 5: The progression of unique bit flip locations on an ECDSA private key d. White and black cells represent flipped and not flipped bits respectively. After running the online phase of Rowhammer for 117 minutes, 192 unique bits are recovered out of 256 bits.

the server. In TLS, a handshake protocol is initiated to set up a shared key between a client and a server. In our attack scenario (Figure 6), client is the attacker and server is the victim. The attacker also has access to the same system that the server runs on. This is needed in order to inject faults into the secret key. Below are the steps of our attack during a TLS handshake:

- The first step of the TLS handshake is initiated by a client (attacker) by sending the following message: `(ClientHello, KeyShare)`. The attacker does not perform any modifications on the message contents.
- Once the server (victim) receives the message from the client, it creates and sends a response message: `(ServerHello, KeyShare, VerifyCert, Finished)`. Here the attacker targets `VerifyCert` message, which contains the signature of the entire handshake. The attacker injects fault into the secret key of the certificate. Therefore, any signature that is created during the `VerifyCert` message step, becomes a faulty signature, and the message `VerifyCert′` becomes faulty. After the response message (along with the faulty signature) is created, the server forwards it to the client.
- Once the client receives the response `(ServerHello, KeyShare, VerifyCert′, Finished)`, it can recover the bits of the secret key from the faulty signature in `VerifyCert′`.

The success of this attack is tied to two key steps:

- The attacker needs to inject the fault into the secret key before the server initiates the signature creation in `VerifyCert` step.
- The client (attacker) needs to successfully receive the faulty signature from the server.

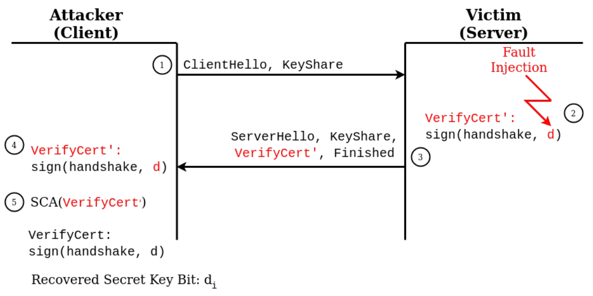

Figure 6: Fault injection during TLS 1.3 handshake.

7.4. Jolt Vulnerabilities in Crypto Libraries

We analyzed a number of prominent crypto libraries to see how their implementations are handling faults and whether they are vulnerable to the proposed Signature Correction Algorithm. The targeted libraries are widely used in cloud service systems. Specifically, Symcrypt is deployed on Azure and S2N deployed on AWS. Furthermore, OpenSSL and LibreSSL are also integrated into and used in S2N, so we included them on our analysis. Another library WolfSSL is also included in the analysis since it is the most common lightweight library that supports embedded systems. As for other libraries, we encountered incompatibility issues and opted instead to concentrate on end-to-end attacks including reception of faulty signatures through TLS. The results of our analysis are summarized in Table 2. We can group the targeted libraries into two categories. Ones that only implement the lower level crypto primitives, i.e. `Microsoft SymCrypt`, and others such as `OpenSSL`, `LibreSSL`, `WolfSSL` and `Amazon s2n`, that in addition implement a TLS stack.

In each of the libraries summarized below we,

- First, we manually reviewed the lower level signature primitives to determine if there are any countermeasures implemented to prevent fault injection or suppression of faulty signatures.
- We then injected faults into the secret signing key to determine if a faulty signature was indeed generated, which was then subsequently corrected using our Signature Correction Algorithm.
- Finally, in the libraries where TLS is supported, we have run a client-server setup, where we injected faults on the server during signature generation as part of the TLS handshake, and checked whether the faulty signature was received on the client side.
- We repeated the experiments on regular releases as well as FIPS hardened versions of the cryptographic libraries whenever available.

OpenSSL 3.0.4. OpenSSL software is one of the most mature and full-featured cryptographic libraries. OpenSSL supports TLS and also comes with a built-in module that supports FIPS certification. We have verified that OpenSSL ECDSA is vulnerable to fault injection, i.e., when faults are injected on a TLS server to the secret signing key,

```
// SERVER SIDE: Fault injection
    ...
// Enable for *simulated* faults on secret key
RandomBitFlip(eckey->priv_key);

if (eckey->meth->sign != NULL)
    return eckey->meth->sign(type, dgst, dlen,
    sig, siglen, kinv, r, eckey);
    ...

// CLIENT SIDE: Bit recovery algorithm
    ...
BN_mod_mul(u1, m, u2, order, ctx);
BN_mod_mul(u2, sig->r, u2, order, ctx);

for(test_add=0; test_add<2; test_add++)
    for(i =0; i<256; i++){
        //mult with 2**i
        powmul(u3, u2, i, order, ctx);
        if(test_add)
            BN_mod_add(u1,u1,u3,order,ctx);
        else
            BN_mod_sub(u1,u1,u3,order,ctx);

        ...
        /* if the signature is correct
        u1 is equal to sig->r */
        if((BN_ucmp(u1, sig->r))
            return i;
    }
```

Figure 7: OpenSSL code snippet for simulated fault injection on server side and signature correction on the client side

faulty signatures are generated. The faulty signatures go unnoticed by the server, and are then relayed to the TLS client. By running the Signature Correction Algortihm on the faulty signatures, we were able to recover secret key bits. In Figure 7, we provide a code snippet for simulated faults on the server side and for the Signature Correction Algorithm on the client side. Even when FIPS mode is activated, the OpenSSL TLS server still sends (and the client still receives) the faulty signatures allowing the attack to proceed.

Besides ECDSA in the TLS handshake, we detected a vulnerability that applies to handshakes with RSA. The RSA library first performs a signing operation which uses the CRT reduced versions of the secret key and applies the CRT method to speed up exponentiation. After that, it verifies the signature. If the signature does not pass, it uses the non-CRT method to recompute the signature from scratch. However, after the second signature is computed, there is no check implemented. Hence, it is possible to force the server to send faulty signatures by injecting faults into both the secret key and the CRT encoded versions of the secret key. With this approach an attacker can bypass the signature verification and still recover a faulty signature.

wolfSSL 5.3.1. wolfSSL is a lightweight SSL/TLS library designed for use in IoT, embedded, and RTOS environments. wolfSSL implements low level cryptographic primitives as well as TLS 1.3 and a FIPS ready version to support the FIPS certification of products that use wolfSSL. We have

Library/Version	Detected?	Signature leaked?	Standalone	Signature check?
`wolfSSL` 5.3.1	No	transmitted by server	Yes	None
`OpenSSL` 3.0.4	No	transmitted by server	Yes	None
`OpenSSL-FIPS` 2.0.8	Yes	transmitted by server	Yes	PK-SK pair is checked with fix message
`LibreSSL` 3.5.3	No	transmitted by server	forked from OpenSSL	None
`Amazon s2n` 1.3.18	No	transmitted by server	uses OpenSSL for crypto layer	None
`MS SymCrypt` 102.0	No	only first signature is detected	Yes	PK-SK pair is checked with fix message (for first signature only)

TABLE 2: Summary of results in cryptographic libraries analyzed for Jolt vulnerability

verified the faulty injection attack on ECDSA works on the TLS server side during the handshake, where the faulty signatures are received on the client side and corrected to yield secret signing bits.

Microsoft SymCrypt 102.0. After the addition of asymmetric schemes in Windows 10 (1703 release), SymCrypt has been the primary crypto library for all algorithms in Windows. Besides providing safety and assurance, a goal of SymCrypt is to support FIPS 140-2 certification. SymCrypt does not implement TLS. However, a SymCrypt-OpenSSL module allows SymCrypt primitives to be used to secure TLS connections.

In our SymCrypt experiments, we have first injected faults into the secret signing key obtaining faulty signatures, which we used to recover key bits via the Signature Correction Algorithm. When FIPS mode is enabled, whenever a new ECDSA key is generated or used for the first time, it is verified throwing an error when a fault is injected. Nevertheless, if faults are injected after the first signature is generated, then the fault goes unnoticed and our attack works. Hence, by implementing a short time delay before injection starts, faulty signatures can be collected from SymCrypt as well.

LibreSSL 3.5.3. LibreSSL is a TLS/crypto stack originally forked from OpenSSL. It supports TLS, however, does not support FIPS. We have verified that, LibreSSL TLS is vulnerable. Fault injection into the secret key during ECDSA signature generation does generate faulty signatures, which are then relayed to a client. On the client side, we were able to run the Signature Correction Algorithm on the faulty signature and recover secret key bits.

Amazon S2N-TLS 1.3.18. S2N-TLS is an implementation of TLS/SSL protocols designed by Amazon for use in AWS. Their library is structured in a way to make use of other cryptographic libraries as an underlying framework. They currently support OpenSSL, LibreSSL, BoringSSL, and the Apple Common Crypto libraries. We performed our tests the OpenSSL library since that comes as default from Amazon. In this setting, we simulated the attack slightly differently since it is built on top of another cryptographic library. We know that OpenSSL is vulnerable to our attack. Therefore, we focused on checking if the signatures were still sent by the server even if they were faulty. Right after the cryptographic library performed the signing operation, we injected fault into the signatures. The server still sends the faulty signature to the client without performing any checks. Hence, we can still receive and recover the secret bits on the client side.

8. Responsible Disclosure

We have disclosed our findings to the security teams of OpenSSL, WolfSSL, LibreSSL, Microsoft, and AWS and received confirmation of disclosure receipt. Amazon-s2n (CVE-2022-42962), WolfSSL (CVE-2022-42961), LibreSSL (CVE-2022-42963) verified the vulnerability and shared candidate patches that incorporate ”Verify after Sign” to our team. After verification the patches were released to the public. The OpenSSL and Microsoft Symcrypt teams validated the vulnerabilities, but opted against issuing patches, stating that Rowhammer is currently not in their threat model.

9. Countermeasures

In this section, we mostly focus on software-based defense, taking into account that architectural solutions, while more powerful, incur substantial overhead and require significant reengineering. We provide a brief overview of countermeasures to inhibit Rowhammer and mitigate Jolt.

9.1. Rowhammer Countermeasures

Rowhammer has become a powerful tool for attacking a broad class of DRAM chips and has therefore commanded the attention of device manufacturers and researchers. Here, we only summarize the most prominent countermeasures.

Increasing DRAM Refresh rate. One of the standard Rowhammer countermeasures are to take away the root cause by increasing the refresh rate, i.e., reduce the refresh window below 64 ms. This will significantly reduce the chances of flips at the cost of increasing the power consumption and decreasing performance.

Using ECC. Early on, use of error-correction code (ECC) which is readily available in many chipsets, was recommended [27]. Although ECC can correct single bit flips and detect double bit flips, [62] showed that it is possible to bypass ECC by locating and flipping three or more bit locations within a word using timing side-channels. Therefore, while it helps in practice, ECC is not a strong countermeasure.

Target Row Refresh. Recently Target Row Refresh (TRR), which utilizes mitigative *distance-1* aggressor row refresh, has proved to be working well against detecting and mitigating *distance-1* Rowhammer attacks. However, quite recently Halfdouble [63] attack was circumventing TRR. Halfdouble has shown that by hammering *distance-2* aggressor rows, one might exploit the TRR mitigative refreshes of *distance-1* aggressor rows to the effect of increasing the likelihood of getting bit flips in the victim rows. The authors recommend also refreshing *distance-2* aggressor rows as a potential defense mechanism and leave it as an open question on how wide of a refresh window needs to be implemented around the victim row. Furthermore, tackling the contiguous memory allocation through the underlying system allocator, will make the system more resilient to the Halfdouble attack.

9.2. Signature Correction Attack Countermeasures

Verify after Sign. Using this verify after sign, approach a sender can detect the existence of an adversary injecting faults via Rowhammer. The advantage of the verification algorithm is that it is approximately three times faster than the double signing procedure. Moreover, a verification step is trivial to implement in existing libraries. That said, the downside of verify after sign is that, there is a chance that the checking mechanism itself will become a victim of the attack by using instruction skips via opcode flipping [59] in which one bypass the checking step.

Redundant Signing. A simple countermeasure is to sign multiple times with the same values (with each copy stored in different memory locations) and then check if the results match. The adversary would have to inject the same fault multiple times into the same positions in all signing operations to succeed. In addition to the overhead, as before the check step might become a target to instruction skipping attacks. Other attacks such as RAMBleed [38] targeting a single computation might still be able to bypass this simple protection scheme.

Masking Sensitive Values. There is a wealth of literature on protecting secrets in hardware against leakages and fault injections. A powerful such technique is *masking* [64] where sensitive internal secrets are randomly split into multiple parts, e.g. additively in $\mathbb{Z}_n$: $d = d_1 + d_2 \bmod n$ for ECDSA (or w.r.t. any other operation that fits the computation domain), and then the desired function is computed over separate randomized paths and aggregated at the end. The advantage is that the shares change in every iteration making the exploitation of partial leakages much harder. On the downside, masking requires randomization. As long as the adversaries' probing capabilities are limited. Otherwise one can use higher order masking (more shares) to improve the resilience of the scheme. In our setting, masking will provide protection as long as the initial mask randomization step is secured. If the fault is injected before this initial step, the attack will work exactly as before.

10. Discussion

In this work, we demonstrated how software fault injection attacks can compromise TLS signing keys. We stress that we used the TLS protocol as a target to demonstrate our attack. However, digital signatures are tightly integrated into our computing infrastructure in numerous products, and therefore, other security protocols that use signatures are likely vulnerable to the presented attack vector as well. This work underlines the significance of a comprehensive review of security software in their usage scenarios under clearly defined threat models.

11. Conclusion

In this work, we introduced a novel fault injection attack on popular signature schemes such as (EC)DSA, Schnorr and RSA signature schemes. For fault injection we utilized Rowhammer, a software only approach, which makes our attack relevant to cloud servers and other platforms with shared memory subsystems. We demonstrated the power of the attack on TLS handshakes secured using RSA and ECDSA signatures. For this, we analyzed five popular cryptographic libraries: `OpenSSL`, `LibreSSL`, `wolfSSL`, `Amazon s2n` and `Microsoft Symcrypt` and found that all are vulnerable to ECDSA key recovery. Specifically, when used in TLS handshakes, a malicious client can recover the signing key with as little as a few hundred faulty signatures on platforms vulnerable to Rowhammer. The problem is exacerbated by either missing or insufficient signature checks and worse incorrect handling of detected faults. For instance, in `Microsoft Symcrypt` only the first signature is checked, leaving further signatures unprotected. We recommend a thorough review of security libraries, the implementation of universal signature checks, and correct failure handling.

Acknowledgments

We thank our anonymous reviewers for their insightful comments and for improving the quality of this paper. This work is supported by the National Science Foundation, under grant CNS-2026913.

References

[1] J. Jancar, M. Fourné, D. D. A. Braga, M. Sabt, P. Schwabe, G. Barthe, P. Fouque, and Y. Acar, "“they're not that hard to mitigate”: What cryptographic library developers think about timing attacks," in *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2022, pp. 755–772. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46214.2022.00044

[2] D. F. Aranha, F. R. Novaes, A. Takahashi, M. Tibouchi, and Y. Yarom, "Ladderleak: Breaking ecdsa with less than one bit of nonce leakage," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2020, p. 225–242. [Online]. Available: https://doi.org/10.1145/3372297.3417268

[3] D. Boneh and R. Venkatesan, "Hardness of computing the most significant bits of secret keys in diffie-hellman and related schemes," vol. 1109, 08 1996, pp. 129–142.

[4] M. Liu and P. Q. Nguyen, "Solving bdd by enumeration: An update," in *Topics in Cryptology – CT-RSA 2013*, E. Dawson, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 293–309.

[5] P. Q. Nguyen and I. E. Shparlinski, "The insecurity of the digital signature algorithm with partially known nonces," *Journal of Cryptology*, vol. 15, pp. 151–176, 2000.

[6] D. Moghimi, B. Sunar, T. Eisenbarth, and N. Heninger, "TPM-FAIL: TPM meets Timing and Lattice Attacks," in *29th USENIX Security Symposium (USENIX Security 20)*. Boston, MA: USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi

[7] K. Ryan, "Return of the hidden number problem.: A widespread and novel key extraction attack on ecdsa and dsa," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2019, no. 1, p. 146–168, Nov. 2018. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/7337

[8] ——, "Hardware-backed heist: Extracting ecdsa keys from qualcomm's trustzone," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 181–194. [Online]. Available: https://doi.org/10.1145/3319535.3354197

[9] S. Weiser, D. Schrammel, L. Bodner, and R. Spreitzer, "Big numbers - big troubles: Systematically analyzing nonce leakage in (ec)dsa implementations," in *USENIX Security Symposium*, 2020.

[10] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of eliminating errors in cryptographic computations," *Journal of Cryptology*, vol. 14, pp. 101–119, 2015.

[11] D. Naccache, P. Q. Nguyên, M. Tunstall, and C. Whelan, "Experimenting with faults, lattices and the dsa," in *Public Key Cryptography - PKC 2005*, S. Vaudenay, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 16–28.

[12] J.-M. Schmidt and M. Medwed, "A fault attack on ecdsa," *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pp. 93–99, 2009.

[13] P. Ravi, M. P. Jhanwar, J. Howe, A. Chattopadhyay, and S. Bhasin, "Side-channel assisted existential forgery attack on Dilithium-A NIST PQC candidate." *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 821, 2018.

[14] C. Ambrose, J. W. Bos, B. Fay, M. Joye, M. Lochter, and B. Murray, "Differential attacks on deterministic signatures," in *Topics in Cryptology – CT-RSA 2018*, N. P. Smart, Ed. Cham: Springer International Publishing, 2018, pp. 339–353.

[15] F. Weimer, "Factoring rsa keys with tls perfect forward secrecy." Red Hat Technical Report, 2015. [Online]. Available: https://www.redhat.com/en/blog/factoring-rsa-keys-tls-perfect-forward-secrecy

[16] G. A. Sullivan, J. Sippe, N. Heninger, and E. Wustrow, "Open to a fault: On the passive compromise of TLS keys via transient errors," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 233–250. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/sullivan

[17] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip feng shui: Hammering a needle in the software stack," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 1–18. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/razavi

[18] Z. Weissman, T. Tiemann, D. Moghimi, E. Custodio, T. Eisenbarth, and B. Sunar, "Jackhammer: Efficient rowhammer on heterogeneous fpga-cpu platforms," *arXiv preprint arXiv:1912.11523*, 2019.

[19] D. Poddebniak, J. Somorovsky, S. Schinzel, M. Lochter, and P. Rösler, "Attacking deterministic signature schemes using fault attacks," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018, pp. 338–352.

[20] S. Islam, K. Mus, and B. Sunar, "Quantumhammer: A practical hybrid attack on the LUOV signature scheme," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1071–1084.

[21] I. Saad, K. Mus, R. Singh, P. Schaumont, and B. Sunar, "A signature correction attack on the post-quantum scheme dilithium," in *Proceedings of the IEEE European Workshop on Security and & Privacy*, 2022.

[22] D. F. Aranha, P. Fouque, B. Gérard, J. Kammerer, M. Tibouchi, and J. Zapalowicz, "GLV/GLS decomposition, power analysis, and attacks on ECDSA signatures with single-bit nonce bias," in *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I*, ser. Lecture Notes in Computer Science, P. Sarkar and T. Iwata, Eds., vol. 8873. Springer, 2014, pp. 262–281. [Online]. Available: https://doi.org/10.1007/978-3-662-45611-8_14

[23] A. Takahashi, M. Tibouchi, and M. Abe, "New bleichenbacher records: Fault attacks on qdsa signatures," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2018, no. 3, p. 331–371, Aug. 2018. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/7278

[24] E. De Mulder, M. Hutter, M. E. Marson, and P. Pearson, "Using bleichenbacher"s solution to the hidden number problem to attack nonce leaks in 384-bit ecdsa," in *Cryptographic Hardware and Embedded Systems - CHES 2013*, G. Bertoni and J.-S. Coron, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 435–452.

[25] D. Bleichenbacher, "On the generation of one-time keys in dl signature schemes," in *Presentation at IEEE P1363 working group meeting*, 2000, p. 81.

[26] G. D. Micheli and N. Heninger, "Recovering cryptographic keys from partial information, by example," *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 1506, 2020.

[27] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.

[28] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for Cross-CPU attacks," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 565–581.

[29] M. Seaborn and T. Dullien, "Exploiting the dram rowhammer bug to gain kernel privileges," *Black Hat*, vol. 15, p. 71, 2015.

[30] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another flip in the wall of rowhammer defenses," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 245–261.

[31] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer. js: A remote software-induced fault attack in javascript," in *International conference on detection of intrusions and malware, and vulnerability assessment*. Springer, 2016, pp. 300–321.

[32] F. de Ridder, P. Frigo, E. Vannacci, H. Bos, C. Giuffrida, and K. Razavi, "SMASH: Synchronized many-sided rowhammer attacks from JavaScript," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1001–1018.

[33] A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi, "Throwhammer: Rowhammer attacks over the network and defenses," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 213–226.

[34] M. Lipp, M. Schwarz, L. Raab, L. Lamster, M. T. Aga, C. Maurice, and D. Gruss, "Nethammer: Inducing rowhammer faults through network requests," in *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2020, pp. 710–719.

[35] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One bit flips, one cloud flops:{Cross-VM} row hammer attacks and privilege escalation," in *25th USENIX security symposium (USENIX Security 16)*, 2016, pp. 19–35.

[36] L. Cojocar, J. Kim, M. Patel, L. Tsai, S. Saroiu, A. Wolman, and O. Mutlu, "Are we susceptible to rowhammer? an end-to-end methodology for cloud providers," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 712–728.

[37] V. Van Der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic rowhammer attacks on mobile platforms," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 1675–1689.

[38] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "RAMBleed: Reading bits in memory without accessing them," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 695–711.

[39] M. C. Tol, S. Islam, A. J. Adiletta, B. Sunar, and Z. Zhang, "Toward realistic backdoor injection attacks on dnns using rowhammer," 2021. [Online]. Available: https://arxiv.org/abs/2110.07683

[40] G. Irazoqui, T. Eisenbarth, and B. Sunar, "MASCAT: Stopping microarchitectural attacks before execution." *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 1196, 2016.

[41] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Applied Soft Computing*, vol. 49, pp. 1162–1174, 2016.

[42] T. Zhang, Y. Zhang, and R. B. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016, pp. 118–140.

[43] N. Herath and A. Fogh, "These are not your grand Daddys cpu performance counters–cpu hardware performance counters for security," *Black Hat Briefings*, 2015.

[44] M. Payer, "HexPADS: a platform to detect "stealth" attacks," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2016, pp. 138–154.

[45] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+ Flush: a fast and stealthy cache attack," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 279–299.

[46] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "ANVIL: Software-based protection against next-generation rowhammer attacks," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 743–755, 2016.

[47] J. Corbet, Defending against Rowhammer in the kernel, Oct. 2016, https://lwn.net/Articles/704920/.

[48] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "CAn't touch this: Software-only mitigation against rowhammer attacks targeting kernel memory," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 117–130.

[49] J. S. S. T. Association, *Low Power Double Data Rate 4 (LPDDR4)*, Jan. 2020, https://www.jedec.org/standards-documents/docs/jesd209-4b.

[50] D.-H. Kim, P. J. Nair, and M. K. Qureshi, "Architectural support for mitigating row hammering in DRAM memories," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 9–12, 2014.

[51] M. Ghasempour, M. Lujan, and J. Garside, "Armor: A run-time memory hot-row detector," 2015.

[52] IBM, "IBM Chipkill Memory: Advanced ECC memory for the IBM Netfinity 7000 M10," 2019, http://ps-2.kev009.com/pccbbs/pc_servers/chipkilf.pdf.

[53] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting correcting codes: On the effectiveness of ECC memory against rowhammer attacks," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 55–71.

[54] P. Frigo, E. Vannacc, H. Hassan, V. Van Der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "TRRespass: Exploiting the many sides of target row refresh," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 747–762.

[55] P. Jattke, V. van der Veen, P. Frigo, S. Gunter, and K. Razavi, "Blacksmith: Scalable rowhammering in the frequency domain," in *2022 IEEE Symposium on Security and Privacy (SP)*, vol. 1, 2022.

[56] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in javascript," *CoRR*, vol. abs/1507.06955, 2015. [Online]. Available: http://arxiv.org/abs/1507.06955

[57] S. Islam, A. Moghimi, I. Bruhns, M. Krebbel, B. Gulmezoglu, T. Eisenbarth, and B. Sunar, "SPOILER: Speculative load hazards boost rowhammer and cache attacks," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 621–637.

[58] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

[59] D. Gruss, M. Lipp, M. Schwarz, and D. Genkin, "J. ju nger, s. o'connell, w. schoechl, and y. yarom,"another flip in the wall of rowhammer defenses,"," *arXiv preprint arXiv:1710.00551*, 2017.

[60] L. Xu, R. Yu, L. Wang, and W. Liu, "Memway: in-memorywaylaying acceleration for practical rowhammer attacks against binaries," *Tsinghua Science and Technology*, vol. 24, no. 5, pp. 535–545, 2019.

[61] A. Tatar, C. Giuffrida, H. Bos, and K. Razavi, "Defeating software mitigations against rowhammer: A surgical precision hammer," in *Research in Attacks, Intrusions, and Defenses*, M. Bailey, T. Holz, M. Stamatogiannakis, and S. Ioannidis, Eds. Cham: Springer International Publishing, 2018, pp. 47–66.

[62] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 55–71.

[63] A. Kogler, J. Juffinger, S. Qazi, Y. Kim, M. Lipp, N. Boichat, E. Shiu, M. Nissler, and D. Gruss, "Half-double: Hammering from the next row over," in *31st USENIX Security Symposium: USENIX Security'22*, 2022.

[64] M. Rivain and E. Prouff, "Provably secure higher-order masking of aes," in *Cryptographic Hardware and Embedded Systems, CHES 2010*, S. Mangard and F.-X. Standaert, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 413–427.

[65] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ecdsa)," *International journal of information security*, vol. 1, no. 1, pp. 36–63, 2001.

[66] C. F. Kerry and P. D. Gallagher, "Digital signature standard (dss)," *FIPS PUB*, pp. 186–4, 2013.

Appendix

Elliptic Curve Digital Signatures

The Elliptic Curve Digital Signature Algorithm (ECDSA) [65] is an elliptic curve variant of the Digital Signature Algorithm (DSA) [66] in which the prime subgroup in DSA is replaced by a group of points on an elliptic curve over a finite field. The ECDSA key generation process starts with the selection of an elliptic curve, specified by the curve parameters and the base field F_q over which the curve is defined, and a base point $P \in E$ of cryptographically large order n in the group operation. Cryptographic security of ECDSA is based on Discrete Logarithm Problem (DLP) over finite additive cyclic groups.

Algorithm 6 ECDSA Key Generation

```
1: Input: $E$: an Elliptic Curve, $P \in E$: a base point on $E$ of order $n$
2: Output: $Q \in E$: Public Key, $d \in Z_n^*$: Private Key
3: Randomly choose a private key $d \in Z_n^*$.
4: Compute the curve point $Q = dP \in E$.
5: return integer $d$: The Private key, and $Q \in E$: the Public key.
```

Algorithm 7 ECDSA Signing

```
1: Input: A message $M \in \{0,1\}^*$, private key $d$,
2: Output:
3: Choose a nonce/ephemeral key $k \in Z_n^*$.
4: Compute the curve point $kP$, and compute the $x$ coordinate $r = (kP)_x$.
5: Compute $s = k^{-1}(H(M) + dr) \bmod n$, where $H(.)$ represents a cryptographic hash function such as SHA-256.
6: return Signature pair $(r, s)$.
```

Algorithm 8 ECDSA Verification

```
1: Input: Signature $(r', s')$, a message $M' \in \{0,1\}^*$, public key $Q \in E$,
2: Output:
3: Compute $H(M')$ where $H(.)$ is the same cryptographic hash function in signing.
4: Compute $w = (s')^{-1} \bmod n$.
5: Compute $u_1 = H(M')w \bmod n$.
6: Compute $u_2 = r'w \bmod n$.
7: Compute $R' = u_1P + u_2Q \in E$
8: Check if $x$ coordinate of $R'$, $r' = (R')_x$ is equal to $r$ or not. If $r' = r$ verify, if not then reject.
9: return Verify or reject.
```

Baby-Step Giant-Step Algorithm

One of the earliest meet in the middle algorithms for computing discrete logarithms in finite cyclic groups is the Baby-Step Giant-Step Algorithm by Shanks. A version adapted to the additive Elliptic Curve Group setting is given in Algorithm 9. The algorithm is searching for a collision between a computed value y parameterized by the index i against the searched item jP parameterized by j. A collision is found when $y = jP$. In each iteration of the i loop we are updating $y = y + u = y - wP$. Thus after i iterations $y = Q - iwP$. Since $y = jP$, it holds that $Q - iwP = jP$, and hence $Q = (iw + j)P$ and $d = iw + j$. In other words when the algorithm terminates i and j hold the lower and higher half of the bits of d, respectively. The Algorithm succeeds in finding the discrete logarithm with time and space complexity of $O(\sqrt{n})$. Note that, in practice, in the search loop the majority of the time is spent checking table entries, although this step can also be speed up with hash lookups. Moveover, we can make different allocations for Baby-Step and Giant Step parts, i.e., reach different trade-off points between storage and computation, by unevenly dividing n into two parts.

Algorithm 9 Baby-Step Giant-Step DLP Algorithm

```
1: Input: A cyclic group $E$ of order $n$, having a generator $P$ and an element $Q$
2: Output: A value $d$ satisfying $Q = dP$.
3: Set $w = \lceil \log \sqrt{n} \rceil$
4: for $j = 0, 1, \ldots, 2^w - 1$ do
5:     Compute $jP$ and store the pair $(j, jP)$ in a table.
6: end for
7: Compute $u = -2^wP$
8: Set $y = Q$
9: for $i = 0, 1, \ldots, 2^w - 1$ do
10:     if $y$ matches any $(jP)$ in the table then
11:         Return $i2^w + j$.
12:     end if
13:     Compute $y = y + u$
14: end for
```