Nonce@Once: A Single-Trace EM Side Channel Attack on Several Constant-Time Elliptic Curve Implementations in Mobile Platforms

Monjur Alam
Georgia Tech
md.monjur@gatech.edu

Baki Yilmaz
Georgia Tech
b.berkayyilmaz@gatech.edu

Frank Werner
Georgia Tech
fwerner6@gatech.edu

Niels Samwel
Radboud University
nsamwel@cs.ru.nl

Alenka Zajic
Georgia Tech
alenka.zajic@ece.gatech.edu

Daniel Genkin
University of Michigan
genkin@umich.edu

Yuval Yarom
University of Adelaide
and Data61
yval@cs.adelaide.edu.au

Milos Prvulovic
Georgia Tech
milos@cc.gatech.edu

2021 IEEE European Symposium on Security and Privacy (EuroS&P) | 978-1-6654-1491-3/21/$31.00 ©2021 IEEE | DOI: 10.1109/EUROSP51992.2021.00041

***Abstract*—We present the first side-channel attack on full-fledged smartphones that recovers the elliptic curve secret scalar from the electromagnetic signal that corresponds to a single scalar-by-point multiplication in current versions of Libgcrypt, OpenSSL, HACL* and curve25519-donna. To avoid leaking information via side channels, these implementations follow the recommendations of RFC 7748 and use a constant-time conditional swap operation. Our attack targets signal differences created by systematic changes in operand values during this conditional swap operation.**

We deploy the attack, using low-cost equipment (<$800), against two Android-based mobile phones and against a Linux-based IoT development board. We repeat the attack 100 times, each time with a different scalar, on each device. In all of the implementations considered in this work, our attack successfully recovers the full secret key within seconds.

To mitigate the attack we suggest randomizing the exclusive-or mask in the conditional swap operation. We show that this countermeasure is effective in preventing this and similar attacks.

1. Introduction

Since their introduction in Van Eck and Laborato [73] and Kocher [40], side-channel attacks have proven to be a severe threat to the security of cryptographic implementations. Dating back to the mid-90's, there is a large body of work focusing on mounting physical side-channel attacks on embedded devices (e.g., microcontrollers, FPGAs, and smart cards) using numerous channels, such as high-bandwidth power analysis [20, 35, 41, 48, 49, 70, 80], electromagnetic emanations (EM) analysis at clock-scale frequencies [1, 24, 52, 63], and fault injection attacks [14, 37]. More recently, the research community had started exploring the vulnerability of complex devices, such as desktops, laptops, server computers, phones, and tablets to physical side-channel attacks. While the complexity and GHz-scale speed of such devices has so far prevented high-bandwidth physical attacks, low-bandwidth attacks running well below the CPU's clock speed, were demonstrated using several channels, such as electromagnetic emanations [3, 18, 28–30, 83], power analysis [26, 30], acoustics [27], and even temperature fluctuations [15].

PCs and mobile devices are also vulnerable to side-channel attacks stemming from contention on microarchitectural CPU resources [25]. These, so-called microarchitectural side channels, include cache- and timing-based attacks on cryptographic implementations [12, 16, 17, 40, 46, 59, 60, 79], ASLR derandomization [32, 34], attacks on mobile platforms [44], as well as attacks from JavaScript [31, 58, 69]. In 2018, microarchitectural transient execution attacks were used to leak data across security domains, with devastating security consequences [19, 39, 45, 71, 72, 74, 75, 77].

Recognizing the danger that microarchitectural information leakage poses, the cryptographic community attempted to protect code running on complex devices against non-speculative side channels by establishing the notion of *constant-time code*. At a high level, the aim is to avoid secret-dependent control flow and memory access patterns, thereby decoupling the program's secrets from leakage observable through physical and microarchitectural channels. Perhaps the most well-known example of such a design is Curve25519 [13], which uses the highly-regular Montgomery-ladder algorithm [50] and has a secret-independent memory access pattern.

However, constant-time coding offers little protection against high-bandwidth physical attacks. Demonstrating this, [9, 53, 54] showed high-bandwidth physical attacks on constant-time implementations of elliptic curve cryptography, by using high-bandwidth oscilloscopes to sample leakage from microcontroller-level targets at rates exceeding the device's clock-speeds. For PCs and mobile devices however, physical attacks have only been shown at bandwidths much lower than their GHz-scale clock speeds, with results being limited to extracting keys from naive and non-constant-time cryptographic implementations by exploiting high-level key-dependent branches [27–30].

To address concerns of microarchitectural side-channel attacks, RFC 7748 [42] now recommends constant-time coding practices. However, the impact of such countermeasures on low-bandwidth physical leakage is not clear. Thus, in this paper we set out to investigate the interaction between the regularity introduced by microarchitectural side-channel countermeasures and the feasibility of low bandwidth physical side channels on complex devices.

More specifically, we ask the following two main questions.

DOI 10.1109/EuroSP51992.2021.00041

(i) How vulnerable are constant-time elliptic curve implementations running on complex devices to simple power and EM analysis, conducted well below the device's operating speed? (ii) Can countermeasures for microarchitectural attacks have adverse affects on the resilience of complex devices to low-bandwidth physical side-channel attacks?

1.1. Our Contribution

Answering the first question, in this paper we show that, unfortunately, constant-time elliptic curve implementations running on full-fledged mobile platforms can be vulnerable to simple, low bandwidth, EM analysis. More specifically, we evaluate four implementations of elliptic curve digital signatures (Libgcrypt, OpenSSL, HACL* and Curve25519-donna), all of which use practices of constant-time coding for microarchitectural side-channel protection and completely avoid any key-dependent branches and memory accesses. We demonstrate that the high regularity employed by constant-time elliptic curve implementations fails to offer adequate protection even against coarse and low bandwidth physical side channels. In particular, we show how using only a single EM trace, an attacker can extract keys from mobile devices running constant-time elliptic curve cryptography within seconds.

Tackling the second question, we show that the constant-time conditional swap operation, recommended in RFC 7748 as a countermeasure for microarchitectural attacks, ironically amplifies EM side-channel leakage. This allows an EM attacker to use attacks at a lower bandwidth, making cheap and non-invasive EM attacks on mobile devices feasible. Overall, to the best of our knowledge, this is the first work that demonstrates key extraction from a mobile device running a highly regular constant-time implementation of elliptic curve cryptography, where the bandwidth required by the attack is significantly below the device's native execution speed.

Attack Setting. We target mobile phones, which are equipped with 1.1–1.4 GHz CPU and run full-fledged Android. We do not modify the phones' hardware in any way, and in particular leave the phones' cases closed with all internal EM shielding intact. Moreover, we do not apply any modifications that might help with side-channel analysis to the software stack. This includes running an unmodified Android operating system, with all CPU cores enabled, and with power management settings and background applications in their factory defaults. We also avoid the use of triggers for side-channel acquisition and signal location, leaving all side-channel countermeasures in the targeted software in place. Finally, the phones' WiFi is enabled and connected to our university network.

We do assume, however, that the attacker has a brief and non-invasive access to the target device and can thus place an inexpensive EM probe within a few centimeters of the device. Prior work has shown that such probes can be easily hidden in public locations, such as under a table in a coffee shop or inside the charging surface at a public phone charging station [30]. As such access is inherently limited, we constrain ourselves to short attack windows and small and easily concealable attack equipment. For equipment, we avoid expensive and bulky top-of-the-line spectrum analyzers used in prior works [3]. Instead, we use a cheap and small software defined radio (SDR) device (around $800), limiting our bandwidth to 40 MHz, which is well below the target devices' clock speeds. We further avoid hardware-based cycle-accurate artificial triggering, running the SDR in continuous sampling mode. Moreover, unlike prior works, which require thousands of traces of emanations from the target device [30], we perform key recovery using only one trace.

Attack Overview. The attack consists of two main phases. In the profiling phase, the attacker collects a small number (up to 10) of EM traces from the device, while using known scalars. The attacker uses these traces to train classifiers that first identify the regions corresponding to the scalar multiplication operation within each trace and then identify the condition of each constant-time swap operation executed during the scalar multiplications.

In the attack phase, the attacker places a probe next to the target device and continuously monitors its EM emanations, passively waiting for the device to perform a cryptographic signing operation. Once a single EM trace corresponding to such an operation is captured, our attack automatically identifies the location of the scalar-by-point multiplication operation and analyzes the condition of each constant-time swap operation. Finally, using these conditions, the attacker recovers the nonce used during the signing operation and combines it with the (public) message and signature to recover the target's secret signing key.

Leveraging Constant-Time Swap. To achieve constant-time coding, the swap operation proposed by RFC 7748 uses a bit mask to swap two machine words, thereby avoiding secret-dependent control flow. The value of this mask is drawn from a random distribution if the swap takes place and is all-zero if the swap does not take place. Adopting this methodology, newer designs of cryptographic implementations tend to process the secret key one-bit-at-a-time, using the constant-time swap operation to swap two internal values based on the key. Unfortunately, the power consumption of mobile devices typically correlates with the Hamming weight of processed values. Hence, the difference between all-zero and random masks causes a small bias in the CPU's power consumption and with it in the device's electromagnetic emanations. Due to the relatively low bandwidth of the attack, we cannot distinguish between an all-zero and a random-looking value in a single instruction. However, the repeated swapping of multiple machine words, required for swapping two internal values, amplifies the leakage via physical side channels allowing us to distinguish the values. Thus, ironically, using constant-time swap operation, while protecting against microarchitectural side-channel attacks, renders newer cryptographic implementations more vulnerable to low bandwidth physical side-channel analysis compared to older generation algorithms that process several key bits at once.

We note here that a previous work [54] has demonstrated an attack on the constant-time swap operation using invasive high bandwidth attack techniques on 8-bit microcontrollers and under well-controlled lab conditions. More specifically, Nascimento et al. use an 8-bit ATMega328P development board running at 7 MHz and executing a bare-metal (i.e., no OS) elliptic curve scalar-by-point multiplication [54, Appendix A]. They measure the microcontroller's power consumption with a high-bandwidth

500 MHz oscilloscope while artificially increasing leakage by modifying the electronic circuit, introducing a 50 Ohm resistor into the microcontroller's ground path. As the microcontroller in Nascimento et al. requires two cycles per load [54, Section 3], this results in a high-bandwidth signal consisting of about 136 samples for each load operation. With such precise observations, Nascimento et al. [54] can recover a scalar by observing minute Hamming weight variations in individual instructions (thereby avoiding the use of amplification). In contrast, our attack is non-invasive, requires no hardware modifications, and uses 40 MHz bandwidth to attack a 1.4 GHz mobile phone (e.g., about 0.03 samples per cycle). We thus leverage the EM signal amplification provided by the constant-time swap operation in order to recover the swap condition, resulting in key recovery using only a single leakage trace. Finally, unlike Nascimento et al., who execute a bare-metal version of the scalar-by-point multiplication with artificial cycle-accurate triggering, we target deployed cryptographic libraries running on multi-core full-fledged Android mobile phones without any triggering while overcoming the noise generated by the operating system and other Linux background activities.

Countermeasure. As a final contribution, we develop a countermeasure for the constant-time swap operation, which prevents this systematic operand bias with only a negligible impact on performance, increasing the execution time of the ECDSA signing operation by less than 0.1%. Our technique ensures a pseudo-random Hamming weight for the masks used in the swap operations by XORing the mask with a pseudo-random value prior to swapping words and then XORing the possibly swapped words with the same pseudo-random value. XORing with the pseudo-random value removes the bias in the original mask. We have empirically confirmed that this countermeasure effectively mitigates our attack.

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

- We identify a weakness in a common implementations of constant-time code running on mobile devices which amplifies the EM leakage allowing detection using inexpensive equipment (Section 3).
- We evaluate the problem with four leading implementations of elliptic curves on three high-end devices, showing that the technique allows complete secret recovery from a single EM trace (Section 4).
- We design a countermeasure that masks the EM signal and show how to implement a portable version that resists compiler optimizations (Section 5).

Responsible Disclosure. Following the practice of responsible disclosure, we reported our findings to the developers of the cryptographic libraries discussed in this work and are working with them on implementing countermeasures.

1.2. Targeted Software and Hardware

We show attacks against four different software implementations of ECC:

- The EdDSA signing routine in Libgcrypt 1.8.4, operating over the Ed25519 curve.
- The ECDSA signing routine in OpenSSL 1.1.1, operating over the secp256k1 curve.
- The scalar-by-point multiplication routine in the Donna implementation of Curve25519.
- The EdDSA signing routine in HACL* operating over the Ed25519 curve.

All software is compiled with gcc 4.6.3 for OLinuXino and with gcc 5.4.0 for Android devices, using the default configuration, makefile, optimization level, and side-channel countermeasures.

We empirically demonstrate our attack using two ARM-based Android mobile phones, a ZTE ZFIVE and Alcatel Ideal, with both phones use Qualcomm CPUs and are currently sold on Amazon.[1] We further demonstrate some of the attacks on an ARM-based IoT prototyping board, A13-OLinuXino, running Debian Linux and equipped with a CPU made by Allwinner. The electromagnetic signals are acquired using an Ettus B200 mini SDR, connected to a custom-built shielded probe implemented as a three-layer 20 mm diameter circular PCB (printed circuit board), with top and bottom layers acting as shield and the middle layer containing a circular feed line. See Figure 1 for a picture of our setup. Note that the probe is positioned several millimeters above the device's chassis, without touching it.

2. Background

2.1. Related Work

Side-channel attacks [40, 41] have been used to defeat numerous cryptographic implementations. EM side channels in particular, in addition to being used to extract cryptographic secrets from smart cards and other relatively simple devices [1, 24, 63], have also been used to detect malware [55, 67] and for performance profiling [66]. A significant body of recent work investigates how to systematically create [83] and quantify [81, 82, 84] EM emanation caused by instruction-level differences in program execution.

Attacking Elliptic Curve (ECC) Cryptography. Among prior side-channel attacks on cryptographic implementations using elliptic curves, many [22, 23] target relatively simple devices, but some recent attacks target more sophisticated devices, such as PCs [29] and mobile phones [10, 30]. However, all these prior attacks target either naive or NAF-based implementations of elliptic curve scalar-by-point multiplication, which has since been replaced by newer implementations that masks secret-dependent control flow by using the constant-time swap operation.

Cache-based side-channel attacks [5, 11, 62, 78] rely on secret-dependent differences in the sequence of memory locations accessed during EC scalar-by-point multiplication. Such attacks are mitigated by using constant-time conditional-swap operation, which exhibits the same sequence of accesses regardless of the value of the secret nonce.

Physical Attacks on non ECC implementations on Complex Devices. Physical side-channel attacks on complex devices have also been used to attack non ECC primitives, primarily modular exponentiation, which is the key

1. https://www.amazon.com/TracFone-ZTE-ZFive2-Prepaid-Smartphone/dp/B07CQ858Y3/ and https://www.amazon.com/GoPhone-Alcatel-Ideal-Memory-Prepaid/dp/B01JGYLV4O/

Figure 1. Experimental setup for capturing EM emanations from ZTE ZFIVE (left) and Alcatel Ideal (right) phones. In each setup, our custom probe (the flat, beige, circular object at the end of the silver-colored cable) is positioned close to but not touching the phone. The cable is held in position by a mechanical arm, which is visible in the photo on the right. An Ettus B200-mini SDR (white box) digitizes the signal and sends it through a USB cable to a personal computer (not shown) for analysis.

primitive in RSA and ElGamal. Recently, several physical side-channel attacks on these implementations have been published. These include chosen-message attacks [26, 27], and an attack that exploits signal differences created by fine-grain (only a few instructions) non-constant-time key-dependent deviations in control-flow [38]. Additionally, the One&Done attack [3] recovers bits of the secret exponents from a signal that corresponds to windowed modular exponentiation during RSA decryption in OpenSSL using a state-of-the-art analog chain of a Keysight spectrum analyzer. Finally, attacks have also been demonstrated on symmetric cryptography running on GHz-scale System-on-Chips made by ARM [8, 47] and Intel [65], albeit using highly invasive techniques, precise probe placement, and requiring case intrusion.

Physical Attacks on Embedded Hardware running Constant-Time Elliptic Curve Implementations. As mentioned in Section 1, several works have considered mounting high-bandwidth physical attacks on constant-time cryptography running on embedded hardware.

Batina et al. [9] proposes a chosen input attack on several constant-time elliptic curve implementations, where a single target trace is used during the key extraction phase. Specifically, they use a high-bandwidth oscilloscope and an accurate trigger to monitor the power consumption of an ATmega 163 based smart card while injecting a chosen input P into the computation of the scalar-by-point multiplication operation. They then show that an attacker can recover the card's secret key from a single target trace, assuming that the profiling traces corresponding to P are available. Finally, to generate the profiling traces, Batina et al. [9] assumes the attacker can collect additional traces after the target trace is collected, using inputs $P'_1, \cdots, P'_{256}$, i.e., one input for every key bit.

Dugardin et al. [21] extends [9] by also accounting for the carry bit propagation from the finite field operations. While this requires more profiling traces corresponding to the chosen input P, they show that these additional traces can be used to detect and correct errors in recovered secret key bits. Finally, Dugardin et al. [21] demonstrate their attack on mbedTLS, a popular cryptographic library running on a 168 MHz STM32F4 microcontroller, sampled at 1 GHz using a lab-grade oscilloscope.

Roelofs et al. [64] identifies a limitation of the technique of Batina et al. [9]. In many cases, e.g., in ECDSA, the attacker does not have control over the inputs sent to the scalar-by-point multiplication operation without extensive modifications. Exploiting the fact that signals sampled at bandwidths comparable to the device's execution speed allow the attacker to obtain information about the individual field operations performed during scalar-by-point multiplication, [64] proposes using ECDSA's verification step, which must take external inputs, to generate the profiling traces on the same device without hardware modifications. Here, the target platform is an XMEGA 8-bit controller running at 32 MHz, sampled via a ChipWhisperer Lite board at 25 MHz using a cycle accurate trigger, and taking about 60 s to complete a single scalar-by-point multiplication operation.

Recently, machine learning has been used to recover the secret key of an elliptic curve signature scheme [76], again using a 168 MHz STM32F4 microcontroller sampled at 1 GHz using a lab-grade oscilloscope via a cycle-accurate trigger. Here, the authors use a convolutional neural network to profile and attack a function inside WolfSSL that reads precomputed values from a lookup table. Using roughly 500 profiling traces, the authors can classify the single target trace with 100% accuracy, thereby extracting the key.

Finally, Nascimento et al. [54] show an attack on an 8-bit ATMega328P microcontroller mounted on a development board, running a bare-metal OS-less scalar-by-point

multiplication at 7.3 MHz, and using several options for implementing the constant-time swap. To acquire the signal, they modify the board, introducing a 50 Ohm resistor on the microcontroller ground plane. Nascimento et al. [54] then sample the resulting signal using a 500 MHz oscilloscope, obtaining a $500/7.3 \cdot 2 = 136$ samples for each 8-bit memory load operation (which lasts two cycles on the ATMega328P). Finally, [54] also modify the target application to generate a cycle-accurate artificial trigger (see Appendix A and Section 3 of [54]). Under such controlled conditions, Nascimento et al. [54] experiment with collecting ten or more training and testing sets, and achieve an accuracy of around 90%. They show that increasing the number of traces used for training and for testing does not improve attack results. This feasibility result was further extended in Nascimento and Chmielewski [53], showing a non-profiled single trace attack on Curve25519's scalar by point multiplication from the μNaCL library implemented on an STM32F4 Cortex-M4 core running at 168 MHz and sampled at 2.5s GHz. While the proposed attack can recover ∼90% of the key bits, it likewise requires modification to the code that adds a cycle-accurate trigger and ~ 15 samples per cycle.

Compared to their attack, we attack full-fledged Android mobile phones running multi-core CPUs at speeds of 1.1–1.4 GHz. As our attack scenario assumes that the attacker only has a short and non-intrusive access to the target device. We do not use high quality signal analyzers or oscilloscopes. Additionally, we do not modify the phones' hardware in any way, leaving the cases closed, with all internal circuity and shielding intact. Instead, we passively acquire the phones' EM signal using a software defined radio at a bandwidth of 40 MHz, extracting the secret nonce from a single trace containing about $1400/40 = 0.03$ samples per load operation (compared to 136 samples in [54] and 14 in [53]).

In terms of software, we attack the full digital signature routine in multiple deployed cryptographic libraries while leaving all of the side-channel countermeasures in the libraries enabled. This contrasts with Nascimento et al. [54] and Nascimento and Chmielewski [53], which only attacked a prototype scalar-by-point multiplication. Likewise, we do not modify any of the phones' software, keeping it in its default state, with all power management and Android background activities running. The phones' WiFi is enabled and is connected to our university network with default Android background traffic. Furthermore, to ensure a realistic attack model, we do not use any triggering. Instead, we rely on signal processing techniques to locate the scalar-by-point multiplication operation and the constant-time swaps inside it, as well as to accurately extract the swap condition from the leakage signal.

2.2. Overview of ECDSA

The Elliptic Curve Digital Signature Algorithm (ECDSA) is a public key signing scheme based on the discrete logarithm assumption over elliptic curves.

Key Generation. First, the participating parties agree on a choice of curve parameters: the elliptic curve field and equation, and a base point G on the curve, which generates a cyclic group. Let n be the order of G (i.e., $nG = \mathcal{O}$, where $\mathcal{O}$ is the group's neutral element, also known as the point at infinity). Key generation consists of selecting a random scalar d and computing $Q = dG$ where dG is the scalar-by-point multiplication of d and G, using additive group notation. The secret signing key is d and the public verification key is Q.

Message Signing and Signature Verification. To sign a message m, the signer computes the cryptographic hash of m and truncates the hash digest to the bit-length of n. We denote this value as z. Next, the signer generates a random nonce $1 \leq k \leq n-1$ and computes $(x, y) = kG$, $r = x \bmod n$ and $s = k^{-1}(z + r \cdot d)$. If r or s are zero, the process is repeated with a new random k. Finally, the digital signature corresponding to m is then set to $\sigma = (r, s)$.

To verify σ, the verifying party computes z from m as described above, computes $w = s^{-1} \bmod n$, $u_1 = zw \bmod n$, $u_2 = rw$, $(x, y) = u_1 G + u_2 Q$ and finally checks that $x \equiv r \bmod n$, rejecting the signature otherwise.

Nonce Secrecy. The security of ECDSA relies on the attacker not having information about the value of the nonce k. If the attacker learns the value of an nonce k corresponding to some signature $\sigma = (r, s)$ on a message m, she can trivially recover the secret key by computing $d = (s \cdot k - z)/r \bmod n$ where z is the truncated-hashing of m as described above. Furthermore, even if partial information about the nonce k is revealed, it may be possible to recover the secret key [56].

2.3. Scalar By Point Multiplication

Dating back to Coron [20], elliptic curve based schemes have been the target of numerous attacks [2, 6, 11, 33, 56, 62]. While exceptions do exist [61], most attacks (including the attack described in this work) target the scalar-by-point multiplication routine. As such, it is critical that this routine be implemented in a side-channel safe way, avoiding leaking information about k to the attacker. Older implementations of elliptic curve cryptography used the naive double and (sometimes) add algorithm or the more performant sliding or fixed window methods. However, both these methods leak information through nonce-dependent control flow and memory access patterns. Hence, the current recommendation is to use implementations based on the Montgomery powering ladder, which we now describe.

Algorithm 1 Montgomery ladder multiplication.

Input: A positive scalar $k = k_{n-1} \cdots k_0$ and an elliptic curve point P.
Output: $[k]P$.

```
1: procedure MONTGOMERY_LADDER(k, P)
2:   R_0 ← O                          ▷ O is the neutral element
3:   R_1 ← P
4:   for i ← n − 1 to 0 do
5:     Q_0, Q_1 ← CT_SWAP(R_0, R_1, k_i)      ▷ swap if k_i = 1
6:     S_0 = 2Q_0
7:     S_1 = Q_0 + Q_1
8:     R_0, R_1 ← CT_SWAP(S_0, S_1, k_i)      ▷ swap if k_i = 1
9:   return R_0
```

The Montgomery Powering Ladder. Algorithm 1 is a pseudocode of the Montgomery Ladder scalar-by-point multiplication operation [51]. At a high level, the Montgomery ladder processes scalar bits one at a time,

from the most significant to the least significant. For each bit, the algorithm performs the Montgomery ladder step, which constitutes a single elliptic curve double and a single elliptic curve add operation (Lines 6 and 7) thereby exhibiting key-independent control flow and memory access pattern. The key material itself (k) is handled during a single swapping routine (CT_SWAP, Lines 5 an 8), which *must* be implemented in a side-channel safe way to avoid leaking the values of bits of k.

The Montgomery ladder algorithm can be further optimized in two ways. First, certain elliptic curves (known as Montgomery curves) have a unified addition formula which allows computing the double and add operations using one formula, thereby replacing Lines 6 and 7 with a single ladder step. Secondly, one swap operation can be removed by merging swaps across consecutive iterations. With this optimization, Line 8 is eliminated, and Line 5 is modified to swap R_0 and R_1 when $k_i \oplus k_{i+1} = 1$.

Algorithm 2 Double-and-always-add multiplication.

Input: A positive scalar $k = k_{n-1} \cdots k_0$ and an elliptic curve point P.
Output: $[k]P$.

1: **procedure** DOUBLE_ALWAYS_ADD(k, P)
2: $R \leftarrow \mathcal{O}$ ▷ $\mathcal{O}$ is the neutral element
3: **for** $i \leftarrow n-1$ **to** 0 **do**
4: $R \leftarrow 2R$ ▷ always double R
5: $T \leftarrow R + P$
6: $R, T \leftarrow$ CT_SWAP(R, T, k_i) ▷ swap if $k_i = 1$
7: **return** R_0

The Double and Always Add Algorithm. Some implementations use an alternative method called double-and-always-add (Algorithm 2). As in the Montgomery ladder case, the key bits are processed in one main loop which performs a single double and a single add operation in each iteration (Lines 4 and 5). As the result of the add operation is only needed when the current key bit is 1, a conditional swap is used to update R when $k_i = 1$ (Line 6).

2.4. The Conditional Swap Operation

Algorithm 3 Conditional Swap.

Input: Two arrays a, b of size n machine words and an integer $cond \in \{0, 1\}$.
Output: Swap a and b if $c = 1$ and leave a, b as is if $c = 0$.

1: **procedure** CT_SWAP($a, b, cond$)
▷ when $cond$ is 0, set mask to all-zeros
▷ when $cond$ is 1, set mask to all-ones
2: $mask \leftarrow 0 - cond$
3: **for** $i \leftarrow 0$ **to** n **do**
4: $\delta \leftarrow (a[i] \oplus b[i])\&mask$
5: $a[i] \leftarrow a[i] \oplus \delta$
6: $b[i] \leftarrow b[i] \oplus \delta$

As explained above, implementing the conditional swap operation in a side-channel safe way is critical for avoiding side-channel leakage. In particular, it is imperative that the attacker cannot leverage side-channel information to learn whether the swap operation has indeed occurred. To help developers safely implement this operation, RFC 7748 [42, Section 5.1] recommends using the pseudocode presented in Algorithm 3. For each machine word i, the algorithm uses an exclusive-or (XOR) to compute bitwise difference between $a[i]$ and $b[i]$. It then applies a bit mask to this difference, such that the difference is either kept as-is or zeroed out, depending on the condition $cond$. This masked difference is then applied (via XOR operations) to the two words $a[i]$ and $b[i]$. The end result is that when $cond = 1$ the values of a and b are swapped, but otherwise they remain unchanged. The key property of this conditional swap is that the sequences of instructions and of memory accesses performed are independent of the (secret) value of $cond$. This prevents cache-based and many analog-signal side-channel attacks from recovering the secret key by obtaining information about the value of the swap's condition.

3. The Nonce@Once Attack

3.1. Attack Overview

In this section we describe our attack on the constant-time swap operation as standardized in RFC 7748 [42] and presented in Algorithm 3. Recall from Section 2.4 that the constant-time swap operation in Algorithm 3 hides the value of the condition using a secret independent control flow and memory access pattern. The key observation we make in this paper is that the value of δ used in the XOR instructions in Lines 5 and 6 of Algorithm 3 highly depends on the value of $cond$.

More specifically, for practical purposes, we can consider a and b to be random. When $cond = 1$, the value of $mask$ is all-ones, resulting in δ being the exclusive-or of random values. Hence, with high probability, it consists of a similar number of 0-valued and 1-valued bits. Conversely, when $cond = 0$, $mask$ is all-zero, and so are the bits of δ. Next, we note that the value of δ is used during the main loop of Line 3. Thus, if $cond = 1$ about half of the bits in every word of a and b are toggled, whereas no such toggling occurs when $cond = 0$, as it implies $\delta = 0$.

Hamming-weight Leakage. In the Hamming-weight model, which seems to apply to the devices we use, the leakage correlates with the Hamming weight (number of bits with a value 1) of values or of changes in values (also known as Hamming distance). Thus, the difference in the values of δ for true and false conditions results in leakage under the Hamming-weight model. However, for high-speed processors this difference may be too small to be detectable without extremely high-end equipment.

Leakage Amplification. In our scenario the condition-dependent bias occurs for each word of a and b. As the phones we attack use a 32-bit architecture while every elliptic curve point contains two coordinates of 256 bits each, the bias repeats 16 times for each point. Furthermore, in addition to point coordinates, implementations also store several (up to four) machine words as implementation-specific metadata and context, which are also swapped using Algorithm 3. Overall, the XOR leakage is multiplied by a factor of about $20 \cdot 2 = 40$ for both points, accentuating the difference between the $cond = 0$ and the $cond = 1$ cases, so it can be easily observed.

A Physical Side Channel Attack. Our attack records the unintentional electromagnetic emanations that the target device radiates during a *single* ECDSA signing operation. We then analyze this signal to identify snippets

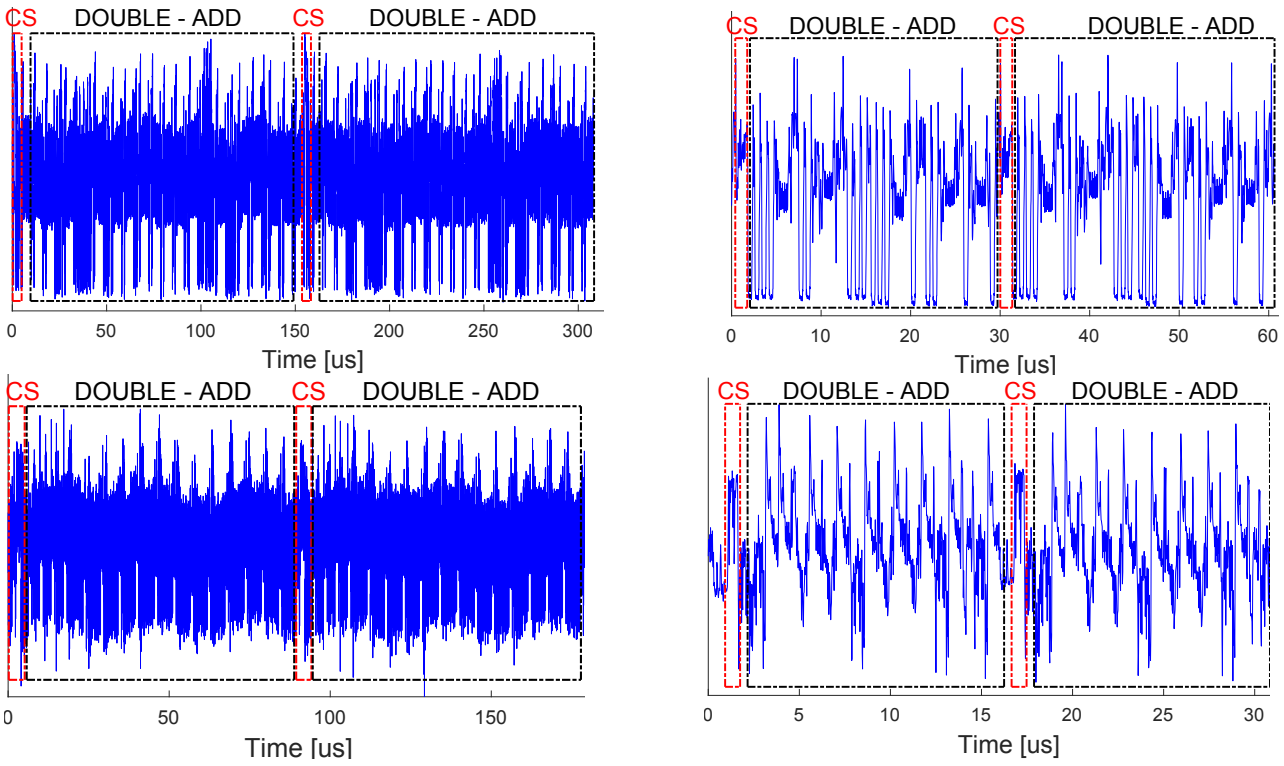

Figure 2. Signal examples covering two scalar-by-point loop iterations each for Libgcrypt, OpenSSL, HACL*, and curve25519-donna (left to right, top to bottom) recorded on the OLinuXino board. The signal for a conditional swap (CS) is indicated by dashed red rectangles, whereas the signal for the point arithmetic (double and add) is indicated by dashed black rectangles. While the signals differ between implementations, the iterations within each implementation are very similar to each other.

corresponding to the swap operations. We then analyze each such snippet and recover the value of $cond$. As both Algorithm 2 and Algorithm 1 use the nonce bits as condition values, our attack can recover almost all of the nonce bits from a single ECDSA leakage trace. As noted in Section 2, we can then use these nonce bits to mathematically recover the secret signing key.

3.2. EM Signal Acquisition and Signature Location

Setup. We use an Ettus B200-mini software defined radio (SDR) connected to a custom electromagnetic probe to capture the target's electromagnetic emanations. The SDR is set to capture a 40 MHz-wide frequency band around the CPU frequency of each target (1.4 GHz for ZTE ZFIVE, 1.1 GHz for Alcatel Ideal, and 1 GHz for A13-OLinuXino). The recorded signal is then digitally demodulated and up-sampled before passing it through the custom signal analysis that implements our attack.

Probe Positioning. For each target considered in this work, we manually located a position which exhibited the strongest electromagnetic leakage and positioned the probe at this point. In all experiments, the probe does not make electrical contact with the device's chassis. Our attack is effective at a range of about 20 millimeters away from the device. While this range does imply physical proximity to the target, we note that this is sufficient for many realistic attack scenarios, such as placing the probe underneath a tablecloth or inside a desk surface [30]. Finally, as we did not modify the phone's chassis, the leakage signal had to penetrate the phone's PCB shielding, battery, and outer case to get to our electromagnetic probe.

Triggering. The first step in our signal analysis approach is to identify the part of the signal that corresponds to the overall scalar-by-point multiplication. Past work achieves this by While past works achieved this by changing the source code of the target library to generate a trigger signal, such modifications are outside our attack model because the attacker is not assumed to be capable of installing malicious software on the target device. Instead, we sample continuously, and resort to signal analysis to locate the trace snippet corresponding to the scalar-by-point multiplication.

Locating the Signal. Because we do not use artificial triggers to identify the signals corresponding to the scalar-by-point multiplication operation, we need to scan the captured signal and recognize the patterns that correspond to it. Moreover, as the implementations we consider use unrelated and significantly different code bases (including in terms of count, order, or type of instructions), the signals they produce vary widely, as Figure 2 shows. However, we observe that most of the execution time of scalar-by-point multiplications is spent on repeated point addition and point doubling operations (Lines 6 and 7 in Algorithm 1 and Lines 4 and 5 in Algorithm 2). These point operations are designed to eliminate variation in their execution time, control flow, and data access patterns. Consequently, as Figure 2 shows, loop iterations within each implementation display very little variations in their EM leakage patterns.

To identify the scalar-by-point multiplication operation, we exploit this repetition of similar patterns. More specifically, in the profiling phase of the attack, we collect a

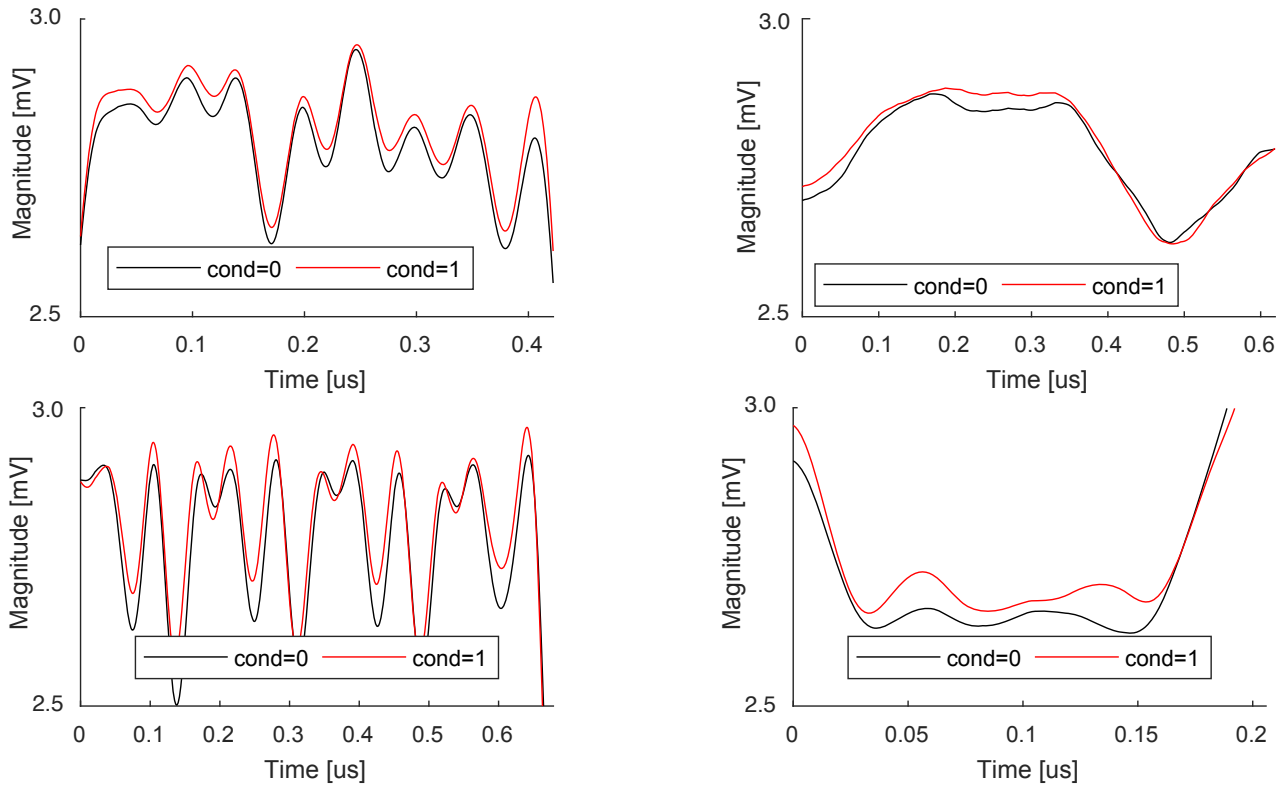

Figure 3. Signal snippets corresponding to the conditional swap operation in Libgcrypt, OpenSSL, HACL*, and curve25519-donna (left to right, top to bottom) when the swap condition is true (black) and when the swap condition is false (red). The recordings where done using the Olimex target and up-sampled for visual clarity.

training trace that includes scalar-by-point operations using randomly generated nonces. We then average the samples of loop iterations in the training set, creating a template for a single loop iteration. In the attack phase we use a moving correlation between the template and the signal, and identify a segment of the signal that contains the expected number of matches to the template. In our experiments we found that the loop iterations are sufficiently long and have enough prominent signal features to reliably locate the scalar-by-point multiplication in the collected trace using a training trace of the signal corresponding to just one scalar-by-point multiplication operation.

3.3. Identifying Conditional Swap Signals

Our attack aims at extracting the condition argument of the conditional swap operations used during the scalar-by-point multiplication. After identifying the signal segment that corresponds to the scalar-by-point multiplication operation, we turn our focus to identifying the snippets corresponding to the individual conditional swap operations. Here, we again rely on the signal similarity between the individual iterations of the scalar-by-point multiplication loop, using the template matching from Section 3.2 to identify individual loop iterations. We then identify the signal segments that correspond to the conditional swap, as these are located between two longer sequences of point doubling and adding.

In all four implementations considered in this work (Libgcrypt, OpenSSL, HACL*, and curve25519-donna), the signal that corresponds to the conditional swap (shown as CS in Figure 2) is much shorter than the signal that corresponding to the elliptic curve double and add operations (shown as DOUBLE-ADD in Figure 2). Furthermore, the signal corresponding to the double and add operations has a number of sharply defined spikes that can help precisely locating and aligning it. In contrast, there are fewer features in snippets that correspond to the conditional swap (CS), as these features are less sharply-defined. Thus to reliably obtain signal snippets for each conditional swap, we leverage the constant-time design of Algorithms 1 and 2. More specifically, we first use correlation with the template of the loop iteration to locate the start of the double and add operations. We then use the end of one double-add snippet and the beginning of the next double-add snippet as reference points in the signal's timeline to extract conditional-swap snippets at fixed timing offsets relative to double-add reference points.

3.4. Recovering the Value of the Swap Condition from a Snippet

After identifying the signal snippets corresponding to conditional swaps, we need to analyze the signal and to classify the snippets based to the value of the condition that has been used. As we describe in Section 2, the conditional swap constructs a mask, which is either all-zeros or all-ones depending on the swap condition, which it then applies to an exclusive-or-based swap for each machine word in the internal representation of two elliptic curve points. Next, the internal representation of an elliptic curve point includes the big number representation of each coordinate (stored as an array of several machine words) as well as several machine words that contain point metadata.

Overall, the swap operation is responsible for conditionally swapping several tens of machine words (about 50 in the implementations we consider).

Leakage Amplification. Notice that when the swap condition is false, there are about 80 exclusive-or operations whose one operand is always zero, and there are 80 machine words that are written to memory without actually changing their values (Lines 4 and 5 in Algorithm 3). However, when the swap condition is true, the operands in those 80 exclusive-or operations have about the same number zeros and ones, and the values written to those 80 machine words result in toggling about half of the bits in those words. This repetition amplifies the difference in the signal corresponding to the conditional swap operation, resulting in a noticeable bias in the captured signal. Indeed, Figure 3 shows signal snippets that correspond to the relevant part of the conditional swap operation in the four implementations considered in this work. In each case, different values of the swap condition produce clearly observable differences in the signal, presumably due to the $\times 80$ amplification factor created by the code of the conditional swap.

Analyzing the Signal of the Swap Operation. To recover the swap condition from the signal snippet, we first perform a training phase with a known scalar. During this phase, for each value of the swap condition we use a k-means clustering algorithm, with a Euclidean distance metric, to form c clusters of snippets collected during training. After removing anomalous clusters that contain only one-snippet, we take the centroid of each remaining cluster as a reference signal. During the attack phase, done on other traces with an unknown key, we compute the distance between the new snippet and each of the reference signals (up to c cluster centroids for false and for true swap conditions), and use the swap value of the closest reference signal as the recovered value of the new snippet. We empirically find that when c is above an implementation-specific threshold, ranging between 5 and 8, the swap conditions are recovered with high accuracy (practically error-free in our experiments). We further find that, increasing c beyond the threshold does not degrade accuracy and it remains practically perfect. Increasing c primarily increases the number of anomalous clusters that are removed, without significantly impacting the number of clusters that are actually used for swap-condition recovery. Since the analysis is not very sensitive to the value of c, instead of optimizing c for each combination of device and cryptographic implementation, in our experiments we simply use $c = 10$ for all attacks.

3.5. Nonce and Key Recovery

Using the signal analysis described above, we obtain the reconstructed values for the swap conditions of each swap-related signal snippet. Applying this method to the signal corresponding to the entire scalar-by-point multiplication operation allows us to directly recover the sequence of swap conditions, and thereby recovering the scalar k.

However, while our approach is capable of recovering most of the bits of k, some bits are inadvertently missing due to signal noise, algorithmic corner cases, and measurement errors. We use a brute-force approach to recover the missing bits. We generate candidates that match the known bits of k. For each such candidate k_c, we use the known values of z, r, and s to compute a candidate private key d_c, using the equation $d_c = (s \cdot k_c - z)/r \bmod n$ as described in Section 2.2. We then use d_c to compute a public key candidate $Q_c = d_c G$, where G is the elliptic curve group generator, and compare it with the actual public key Q used to produce the signature (r, s). If the public keys match, the analysis completes successfully. Otherwise, we repeat the procedure with the next candidate. Because we know the positions of the missing bits, each missing bit roughly doubles the search space. Fortunately, the typical number of missing bits is small, and the search is tractable.

In our experimental evaluation, we tested our signal analysis on 100 ECDSA signing operations, each with a different private key. We found that our signal analysis approach failed to recover at most 15 key bits, thus requiring a brute-force test of up to 32768 candidates for each signing operation. Full key recovery using the above procedure was completed for each of the 100 signing operations. Overall, no signature required more than a few CPU minutes to recover the correct private key, using only a single core on a moderately powerful laptop system (a Macbook Pro with an 2.7 GHz Intel Core i5 processor).

4. Experimental Evaluation

In this section we describe our measurement setup and the results of recovering private keys from Libgcrypt, OpenSSL, HACL* and curve25519-donna during ECDSA signature computation on three different devices.

4.1. Experimental Setup

Targeted Hardware. We run the targeted applications on two Android mobile phones, ZTE ZFIVE LTE2 [85] and Alcatel IDeal [4], and on an Olimex A13-OLinuXino IoT development board [57]. The ZTE ZFIVE has a quad-core 1.4 GHz Qualcomm Snapdragon processor, while the Alcatel Ideal has a quad-core 1.1 GHz Qualcomm Snapdragon processor. The A13-OLinuXino is a single-board computer with an ARM Cortex A8 processor [7] made by Allwinner with Debian Linux.

As mentioned above, we aim for a realistic scenario and avoid modifications that assist with side-channel analysis. In particular, the devices are running unmodified Android / Linux operating systems, with all CPU cores enabled, and with Android's background applications and services running in their factory defaults. Finally, the phones' WiFi is enabled and connected to the university network.

Targeted Software. We attack Libgcrypt 1.8.4 and OpenSSL 1.1.1a. For HACL* and curve25519-donna implementations, we use the latest version directly from their respective git repositories. All four cryptographic implementations use constant-time code for scalar-by-point multiplication, relying on conditional swap operations to avoid control flow that depends on the bits of the scalar. Unless stated otherwise, we use Curve25519, whose maximum order n is a 253-bit value, implying a 253-bit nonce value. We note, however, that the choice of curve has no significant impact on our attack. Any ECDSA implementation that processes the nonce bits using the constant-time operation outlined above is likely to be similarly vulnerable. Finally, we use randomly generated

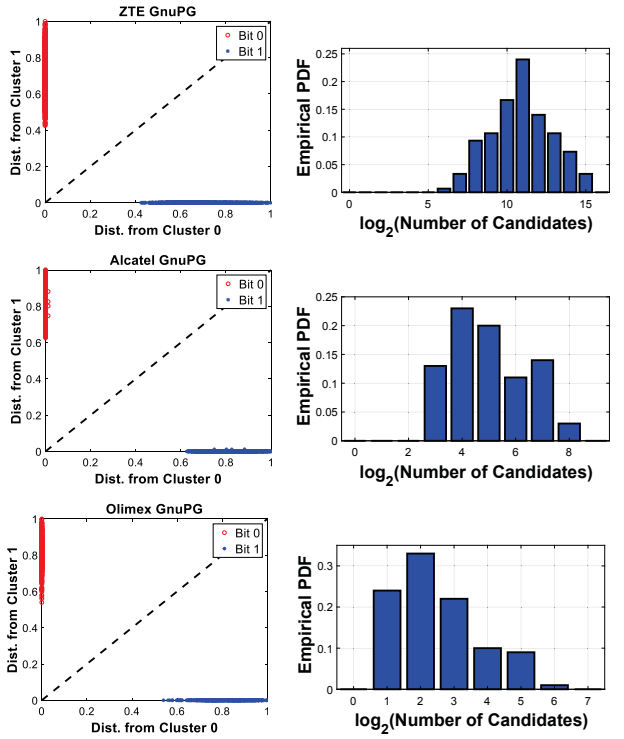

Figure 4. Recovery of swap condition (left) and number of key candidates in full-key recovery (right) for Libgcrypt on each of the target devices (ZTE phone, Alcatel phone and Olimex board, from top to bottom).

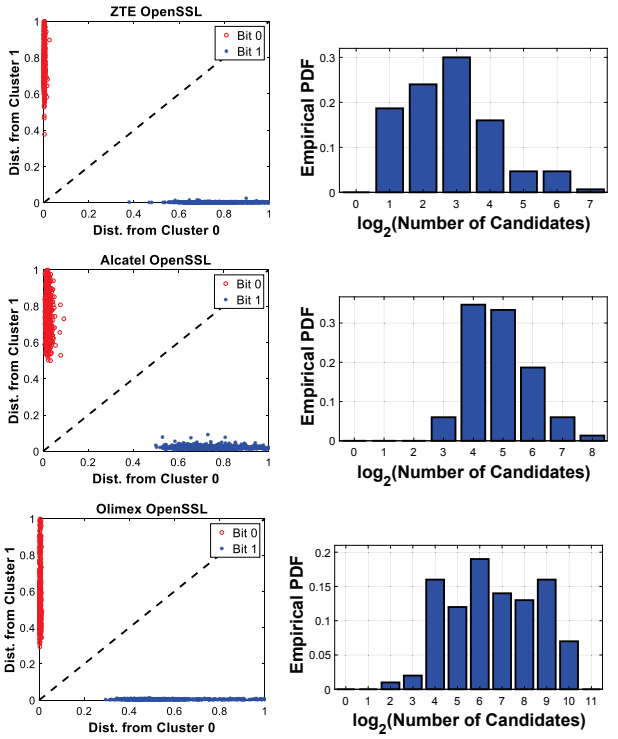

Figure 5. Recovery of swap condition (left) and number of key candidates in full-key recovery (right) for OpenSSL on each of the target devices (ZTE phone, Alcatel phone and Olimex board, from top to bottom).

nonces for both the profiling and for the key extraction phases.

Data Acquisition Setup. Figure 1 shows our setup for capturing the EM emanations from our target devices. It consists of a small custom-made magnetic probe to receive the EM signals, an Ettus B200-mini software defined radio (SDR) to digitize the EM signal in the desired frequency band, and a personal computer to process the digitized signals. As shown in Figure 1, the probe is placed in close physical proximity to the target device, but without touching it and without opening its enclosure. A mechanical arm is used to hold the probe in the desired position during the experiments. The probe is connected to the SDR which digitizes the signal and sends it, through a USB cable, to a personal computer (not shown in Figure 1) where the signal analysis and ECDSA key recovery is implemented in MATLAB. Note that MATLAB is used mainly for convenience, and that signal analysis and key recovery would likely be significantly faster if they were implemented in the Field Programmable Gate Array (FPGA) that is available within the B-200mini SDR itself.

4.2. Attack Results

Our experimental results are based on repeating the attack 100 times for each of the four target implementations on each of the three devices. In each attack, we first use the target's API in order to randomly generate an ECDSA private key and a message. Then, we initiate signal collection while signing the message with the ECDSA key. The signal from this single ECDSA signing operation is then analyzed to recover the nonce k and the ECDSA private key d. Finally, the recovered d is compared to the ground-truth private key to determine if the attack was successful.

Attacking Libgcrypt. Our first set of experimental results consists of attacking a single instance of Libgcrypt's ECDSA signing operation, repeating this attack 100 times on each of the three target devices considered in this work. All 300 of these attack instances successfully recovered the ECDSA private signing key. Figure 4 shows further details on the results of our attack. For each device, we show the clustering of the signal snippets according to their distance from the closest 0-value cluster and from the closest 1-value cluster. We also show the histogram for the size of the search space for brute forcing the nonce k that were considered before the value of the ECDSA key was recovered. We can observe that, for all three devices, the signal snippets that correspond to two possible values of the swap condition are well-separated, resulting in correct recovery of the swap condition for all snippets that were identified. We also observe that the number of key candidates to consider during full-key recovery is highest on the ZTE phone, where the number of key candidates is typically 2^{11}, with a maximum of 2^{16}. For the Alcatel phone and the OLinuXino development board full key recovery requires significantly fewer candidates.

Overall, we observed large signal quality variations between the devices we tested. First, on the two Android-based devices interrupts occur significantly more often than on the Debian-based OLinuXino. Additionally, we found that both phones have interferences due to bursts of activity on other cores, occasionally preventing identification of swap-related signal snippet (especially on the ZTE phone). Finally, on the ZTE phone the signal for Libgcrypt's point

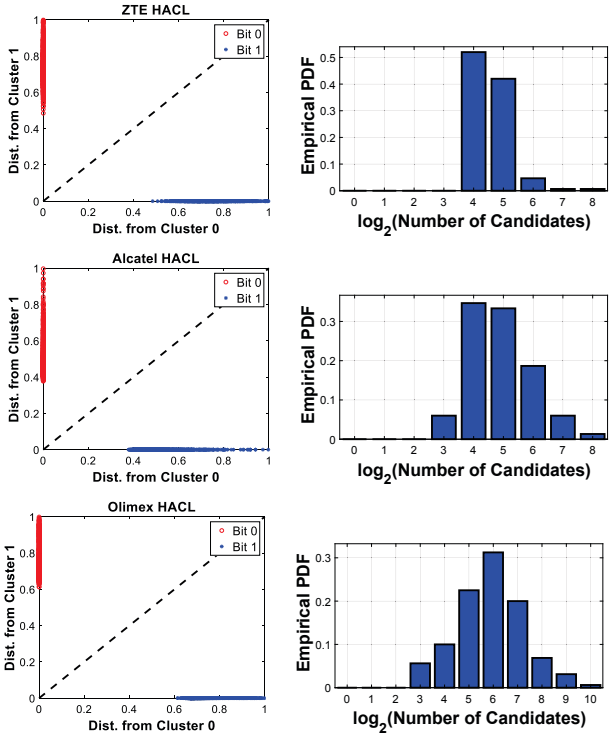

Figure 6. Recovery of swap condition (left) and number of key candidates in full-key recovery (right) for HACL* on each of the target devices.

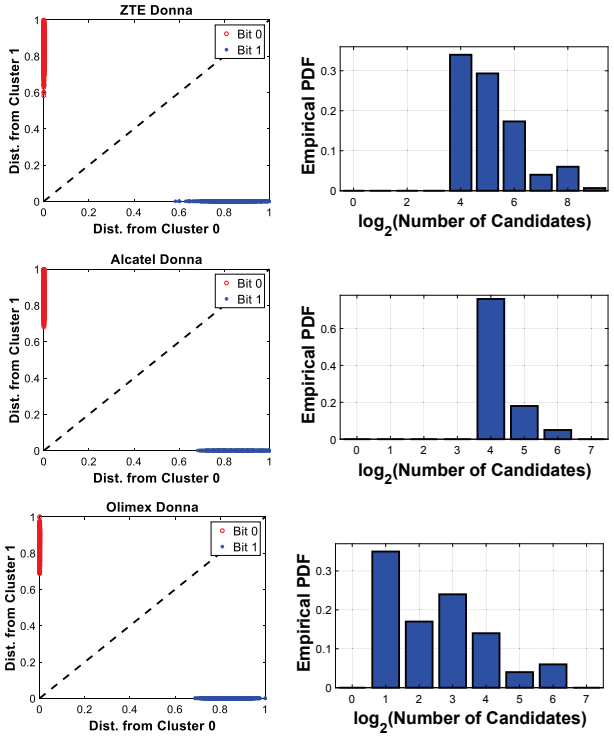

Figure 7. Recovery of swap condition (left) and number of key candidates in full-key recovery (right) for curve25519-donna on each of the target devices.

multiplication is unusually "choppy", resulting in sporadic failures to identify the swap snippets even in the absence of interrupts and other interferences.

Attacking OpenSSL. Our second set of experiments targets OpenSSL, repeating the attack 100 times for each device. Figure 5 shows the clustering of signal snippets, and the number of candidate values for the nonce, for this set of experiments. As in Libgcrypt, the snippets that correspond to the two swap conditions are clearly separable, and all 300 instances of the attack are successful in recovering the key. However, compared to Libgcrypt, the identification of snippets in OpenSSL performs significantly better on the ZTE phone, with snippets missing only for interrupt and interference related reasons. Overall, the analysis time for an attack for OpenSSL was less than 2 seconds in all attack instances.

Attacking HACL* and Curve25519-donna. Finally, our experiments for HACL* (Figure 6) and curve25519-donna (Figure 7) show results that are mostly similar to those for OpenSSL, with success on all 300 attack instances for HACL* and all 300 attack instances for curve25519-donna, with each attack instance using less than 2 seconds of analysis time.

4.3. Other Elliptic Curves and Primitives

We performed a few additional attacks where we used other EC curves in Libgcrypt and OpenSSL, as well as attacks on Libgcrypt's Elliptic Curve Diffie-Hellman (ECDH) implementation. As the scalar-by-point multiplication in all these instances still uses the same conditional swap code, our attack performs similarly to the Curve25519 case.

5. Mitigation

As explained in Section 3.1, the main cause enabling our attack is that the conditional swap operation computes a value δ (Line 4 in Algorithm 3) which, depending on the value of a nonce bit, is either random looking or zero. Algorithm 3 xors δ with tens of machine words that make up the internal representation of an elliptic curve point. While we cannot recover the value of the current nonce bit from a single computation of Line 4 of Algorithm 3, the repeated xor of δ with tens of machine words amplifies the Hamming-weight leakage of the value of δ, which depends on a secret bit.

Avoiding Leakage Amplification. The main idea behind our proposed mitigation is to modify the constant-time swap operation to avoid repeated use of values with a Hamming weight that directly reveals the current nonce bit. Algorithm 4 below is an example of such a modification. After computing δ (Line 5), we mask it using the random word r generated in Line 3, thereby decoupling its Hamming weight from the current nonce bit. We then apply this masked δ to the machine words that make up the elliptic curve points a and b, arranging the different xor operations in a way that also decouples the Hamming weight of intermediate values from the value of the current key bit (Lines 7 and 8).

Empirical Evaluation. To empirically evaluate this countermeasure, we manually checked the signal snippets corresponding to the conditional swap operation and found they no longer exhibit condition-dependent differences. Next, Figure 8 shows the results of applying the clustering algorithm from Section 3.4 on these signal snippets. As can be seen, after applying this countermeasure the clusters

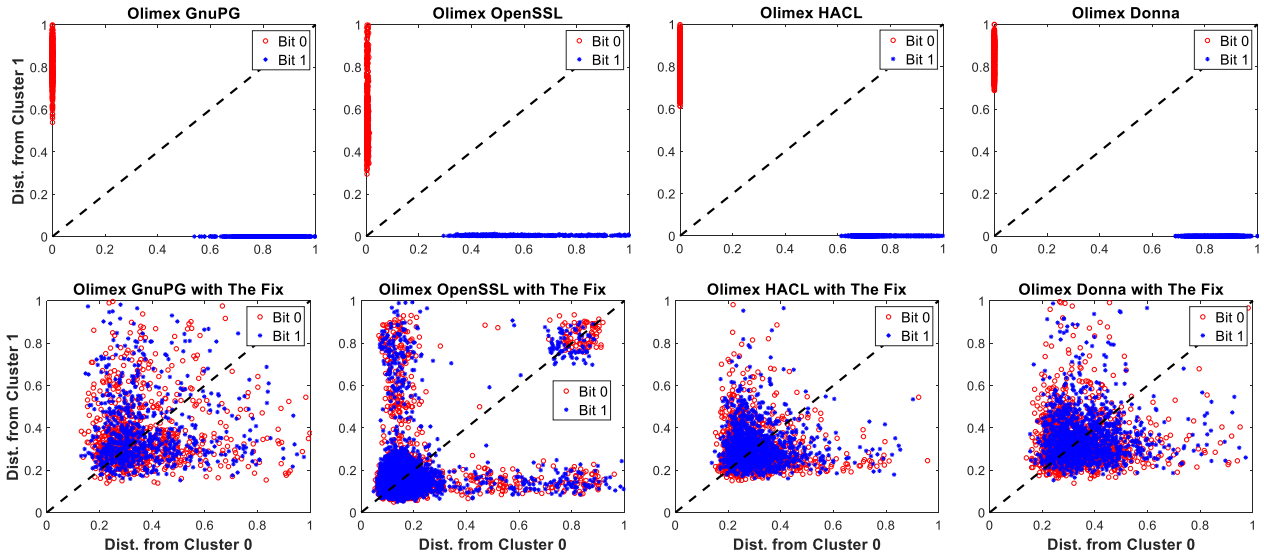

Figure 8. Correlation-based clustering of signal snippets without (top) and with (bottom) the proposed mitigation. With mitigation applied (bottom), signal snippets for $cond = 0$ and $cond = 1$ can no longer be separated.

that correspond to the two values of the swap condition are no longer separable. Similar results hold even for a larger amount of clusters (e.g., $c = 100, 1000$). Finally, we further verified that when employing this countermeasure, the accuracy of detecting the swap condition using our attack, is statistically equivalent to a random guess.

Algorithm 4 Modified Conditional Swap.

/small

Input: Two arrays a, b of size n machine words and an integer $cond \in \{0, 1\}$.

Output: Swap a and b if $c = 1$ and leave a, b as is if $c = 0$.

1: **procedure** CT_SWAP($a, b, cond$)
▷ when $cond$ is 0, set mask to all-zeros
▷ when $cond$ is 1, set mask to all-ones
2: $mask \leftarrow 0 - cond$
3: $r \leftarrow random\_word()$
4: **for** $i \leftarrow 0$ **to** n **do**
5: $\delta \leftarrow (a[i] \oplus b[i])\&mask$
6: $\delta \leftarrow \delta \oplus r$
7: $a[i] \leftarrow (a[i] \oplus \delta) \oplus r$
8: $b[i] \leftarrow (b[i] \oplus \delta) \oplus r$

Dangerous Compiler Optimizations. Interestingly, while implementing the countermeasure outlined in Algorithm 4 in OpenSSL, we found that the EM signal from the resulting binary is very similar to the EM signal from the unmodified OpenSSL code, rendering the modified version vulnerable to our key extraction attack. Further investigation revealed that when used with high optimization levels ("-O2" and above), the machine code the compiler produces does not contain the countermeasure. We experimented with three versions of the gcc compiler (4.6.3, 5.4.0, and 7.3.0), and also checked the results when compiling for an x86-64 platform.

It appears that in its effort to improve the performance of the code, the compiler is effectively undoing our mitigation. Intuitively, the compiler finds that, after the value of δ is computed at Line 5 of Algorithm 4, the expression for the new value of $a[i]$ can be written as $a[i] = a[i] \oplus \delta \oplus r \oplus r$. Next, using the properties of the bitwise xor operator, the compiler rewrites the above expression as $a[i] = a[i] \oplus \delta$. After applying the same optimization to the value of $b[i]$ (Line 8), the compiler further realizes that the value of r is not needed, eliminating Lines 2 and 6 altogether. This effectively reverts the changes between Algorithm 4 and Algorithm 3.

Overcoming Compiler Optimizations. While the constant-time swap operation could be written from scratch in assembly, the result might be complicated and hard to verify due to the complex structure of the elliptic curve point representation. Instead, we used gcc's extended inline assembly syntax to tell the compiler that the value of the variable holding r might change in memory between lines 6 and 7 of Algorithm 4. This forces gcc to treat the values of r in Lines 6 and 7 as if they were different, preventing the optimization. We confirmed that the fix works and the code behaves as required regardless of the level of optimization. Figure 9 below shows the final implementation of the constant-time swap routine (in C), including the inline assembly syntax for avoiding the values of `rand` being optimized out (Lines 12–17).

Performance Overheads. Empirically evaluating the overhead of the additional XOR operations introduced by our countermeasure, we have benchmarked the ECDSA signing operation, with the constant-time swap operation presented in Figure 9. We find that our approach increases the duration of the ECDSA signing operation by less than 0.1%. While this demonstrates the efficiency of our countermeasure, we note that this low overhead is not surprising, as ECDSA is dominated by the mathematical operations over elliptic curves, which are orders of magnitude longer than XORs.

Swapping Pointers. As an alternative to the above suggestion, Nascimento et al. [54] propose swapping the pointers of a and b in Algorithm 3 as opposed to their values. While this reduces the use of δ to only a single

machine cycle, which our attack cannot recover, we note that this solution is not easily applicable to existing libraries such as OpenSSL. This is since the API of the swap operation does not get as input a pointer to the `struct` holding the number, but instead gets the `struct` itself. Thus, unless a library-wide modification of the constant-time swap API is performed, multiple uses of the leaky δ are required to swap the `struct`'s elements, potentially producing detectable side channel leakage.

Moreover, swapping pointers introduces secret-dependent memory access patterns as it makes the addresses holding the swap's outputs (as opposed to the contents) depend on the swap condition. This defeats the purpose of RFC 7748, as the resulting implementation is now vulnerable to cache-based attacks, such as Prime+Probe [46], or to attacks that monitor the address bus [36, 43].

```
1  EC_point_swap_cond(a,b,cond){
2    // When cond is 0, set mask to all-zeros
3    // When cond is 1, set mask to all-ones
4    mask=0-cond;
5    // Random value for mitigation
6    rand=random_word();
7    // For each machine word in the
8    // EC point representation
9    for(i=0;i<nwords;i++){
10     delta = (a->w[i] ^ b->w[i]) & mask;
11     delta = delta ^ rand;
12     asm volatile(
13          // No actual assembler code
14         ``;''
15         // But specify that rand is changed
16         : ``+r'' (rand)
17         : :);
18     a->w[i] = (a->w[i] ^ delta) ^ rand;
19     b->w[i] = (b->w[i] ^ delta) ^ rand;
20   }
21 }
```

Figure 9. Conditional `swap` with our mitigation. The value of `delta` is masked to avoid systematic `cond`-dependent differences in exclusive-or operands.

6. Conclusions

We present the first side-channel attack on full-fledged smartphones that recovers the secret scalar from the electromagnetic signal that corresponds to a single signing operation in current versions of Libgcrypt, OpenSSL, HACL* and curve25519-donna. Specifically, these implementations use a constant-time conditional swap to avoid creating control flow and memory access pattern differences that correlate with bits of the secret scalar. Rather than relying on different control-flow or memory access patterns, our attack uses the signal differences created by systematic differences in operand values during the conditional swap operation itself to recover each bit of the secret.

We deploy the attack, using low-cost equipment (<$800), against two Android-based mobile phones and against a Linux-based IoT development board. We repeat the attack 100 times, each with a different scalar, on each device. Our attacks successfully recover the full secret key within seconds for all of the implementations considered in this work.

Our proposed countermeasure, which randomizes the exclusive-or mask in the conditional swap operation, is effective in preventing this and similar attacks, and has been submitted to the developers of the targeted libraries before the public disclosure of the results presented in this paper.

7. Limitations and Future Work

Attack Range. The attack presented in this paper requires that the EM probe to be positioned very closely to the phones case, at a range of about 20 millimeters. While this still allows for realistic threat models, such as hiding the EM probe inside a desk or under a table cloth, other works attacking non-constant-time code running on laptop computers have achieved a range of about a meter [28]. We acknowledge this limitation and leave the task of enhancing the attack range while maintaining its capability of analyzing constant-time code to future work.

Similarly, all currently presented non-invasive physical attacks on laptops and phones target public key primitives, which are relatively slow compared to their symmetric counterparts (e.g., AES). While invasive attacks on hardware AES engines on Intel CPUs exist [65], these are invasive and require the EM probe to touch the target's main board. Thus, the task of developing non-invasive attacks on constant time symmetric primitives running on non-embedded devices is an important future work.

Cross Device Training. The attacks in this paper use the same physical device for the training and the key extraction phases. This is a limitation because in realistic scenarios the attacker is unlikely to be able to perform profiling runs on the target's phone. We do note, however, that the leakage signal is mostly affected by the probe's exact positioning, as opposed to manufacturing differences between identical devices. In particular, removing the probe from the device and putting it back caused more differences than changes induced by two identical devices. While prior works already suggests that attacks similar to ours can be generalized to work across different devices [3, 68], we leave the task of mounting the attack descried in this paper with cross-device training to future work.

Attacking High-End Targets. Finally, the attacks presented in this paper are performed on commercially available mobile phones. While these do run GHz-scale multi-core CPUs, the leakage of constant-time code on more powerful hardware such as high-end phones and tablets or personal computers still remains unexplored. Previous works targeting laptop computers [26–29] all attacked highly non-constant time implementations, further amplifying their leakage using an adversarially chosen ciphertext. As such implementations have now been mostly replaced with modern constant-time code, we leave it to future work to develop attacks capable of targeting constant time code running on high-power devices.

Acknowledgments

This research was supported by the Air Force Office of Scientific Research (AFOSR) under award number FA9550-20-1-0425; the Australian Research Council project numbers DE200101577 and DP210102670, DARPA ARFL contracts FA8750-19-C-0531 and HR001120C0087; DARPA LADS contract FA8650-16-C-7620; National Science Foundation (NSF) grants CCF-1563991 and CNS-1954712;

Office of Naval Research (ONR) contracts N00014-17-1-254 and N00014-19-1-2287; the Research Center for Cyber Security at Tel-Aviv University established by the State of Israel; and gifts from Intel and AMD. The opinions and recommendations expressed in this work are those of the authors, and do not neccessarily reflect the views of any of the supporting organizations.

References

[1] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi, "The EM side-channel(s)," in *CHES*, 2002, pp. 29–45.

[2] T. Akishita and T. Takagi, "Zero-value point attacks on elliptic curve cryptosystem," in *International Conference on Information Security (ISC)*, 2003, pp. 218–233.

[3] M. Alam, H. A. Khan, M. Dey, N. Sinha, R. L. Callan, A. G. Zajic, and M. Prvulovic, "One&done: A single-decryption EM-based attack on OpenSSL's constant-time blinded RSA," in *USENIX Security*, 2018, pp. 585–602.

[4] Alcatel, "Alcatel ideal specifications," http://www.phonescoop.com/phones/phone.php?p=5097, Feb 24, 2016.

[5] T. Allan, B. B. Brumley, K. E. Falkner, J. van de Pol, and Y. Yarom, "Amplifying side channels through performance degradation," in *ACSAC*, 2016, pp. 422–435.

[6] 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 *CCS*, 2020, pp. 225–242.

[7] ARM, "ARM Cortex A8 processor manual," https://www.arm.com/products/processors/cortex-a/cortex-a8.php, accessed April 3, 2016.

[8] J. Balasch, B. Gierlichs, O. Reparaz, and I. Verbauwhede, "DPA, bitslicing and masking at 1 GHz," in *CHES*, 2015, pp. 599–619.

[9] L. Batina, Ł. Chmielewski, L. Papachristodoulou, P. Schwabe, and M. Tunstall, "Online template attacks," in *Indocrypt*, 2014, pp. 21–36.

[10] P. Belgarric, P. Fouque, G. Macario-Rat, and M. Tibouchi, "Side-channel analysis of Weierstrass and Koblitz curve ECDSA on Android smartphones," in *CT-RSA*, 2016, pp. 236–252.

[11] N. Benger, J. van de Pol, N. P. Smart, and Y. Yarom, ""Ooh aah... just a little bit" : A small amount of side channel can go a long way," in *CHES*, 2014, pp. 75–92.

[12] D. J. Bernstein, "Cache-timing attacks on AES," 2005, http://cr.yp.to/papers.html#cachetiming.

[13] D. J. Bernstein, "Curve25519: New Diffie-Hellman speed records," in *PKC*, 2006, pp. 207–228.

[14] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of eliminating errors in cryptographic computations," *J. Cryptology*, vol. 14, no. 2, pp. 101–119, 2001.

[15] J. Brouchier, T. Kean, C. Marsh, and D. Naccache, "Temperature attacks," *IEEE SP*, vol. 7, no. 2, pp. 79–82, 2009.

[16] B. B. Brumley and N. Tuveri, "Remote timing attacks are still practical," in *ESORICS*, 2011, pp. 355–371.

[17] D. Brumley and D. Boneh, "Remote timing attacks are practical," in *USENIX Security*, 2003.

[18] R. Callan, A. G. Zajic, and M. Prvulovic, "A practical methodology for measuring the side-channel signal available to the attacker for instruction-level events," in *MICRO*, 2014, pp. 242–254.

[19] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking data on Meltdown-resistant CPUs," in *CCS*, 2019, pp. 769–784.

[20] J. Coron, "Resistance against differential power analysis for elliptic curve cryptosystems," in *CHES*, 1999, pp. 292–302.

[21] M. Dugardin, L. Papachristodoulou, Z. Najm, L. Batina, J. Danger, and S. Guilley, "Dismantling real-world ECC with horizontal and vertical template attacks," in *COSADE*, F. Standaert and E. Oswald, Eds., 2016, pp. 88–108.

[22] J. Fan and I. Verbauwhede, "An updated survey on secure ECC implementations: Attacks, countermeasures and cost," in *Cryptography and Security: From Theory to Applications - Essays Dedicated to Jean-Jacques Quisquater on the Occasion of His 65th Birthday*, 2012, pp. 265–282.

[23] J. Fan, X. Guo, E. D. Mulder, P. Schaumont, B. Preneel, and I. Verbauwhede, "State-of-the-art of secure ECC implementations: A survey on known side-channel attacks and countermeasures," in *HOST*, 2010, pp. 76–87.

[24] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: Concrete results," in *CHES*, 2001, pp. 251–261.

[25] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *J. Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, 2018.

[26] D. Genkin, I. Pipman, and E. Tromer, "Get your hands off my laptop: Physical side-channel key-extraction attacks on PCs," in *CHES*, 2014, pp. 242–260.

[27] D. Genkin, A. Shamir, and E. Tromer, "RSA key extraction via low-bandwidth acoustic cryptanalysis," in *CRYPTO'14*, 2014, pp. 444–461.

[28] D. Genkin, L. Pachmanov, I. Pipman, and E. Tromer, "Stealing keys from PCs using a radio: Cheap electromagnetic attacks on windowed exponentiation," in *CHES*, 2015, pp. 207–228.

[29] D. Genkin, L. Pachmanov, I. Pipman, and E. Tromer, "ECDH key-extraction via low-bandwidth electromagnetic attacks on PCs," in *CT-RSA*, 2016, pp. 219–235.

[30] D. Genkin, L. Pachmanov, I. Pipman, E. Tromer, and Y. Yarom, "ECDSA key extraction from mobile devices via nonintrusive physical side channels," in *CCS*, 2016, pp. 1626–1638.

[31] D. Genkin, L. Pachmanov, E. Tromer, and Y. Yarom, "Drive-by key-extraction cache attacks from portable code," in *ACNS*, 2018, pp. 83–102.

[32] D. Goruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR," in *CCS*, 2016, pp. 368–379.

[33] L. Goubin, "A refined power-analysis attack on elliptic curve cryptosystems," in *PKC*, 2003, pp. 199–

210.
[34] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the line: Practical cache attacks on the MMU," in *NDSS*, 2017.
[35] N. Homma, A. Miyamoto, T. Aoki, A. Satoh, and A. Shamir, "Collision-based power analysis of modular exponentiation using chosen-message pairs," in *CHES*, 2008, pp. 15–29.
[36] K. Itoh, T. Izu, and M. Takenaka, "A practical countermeasure against address-bit differential power analysis," in *CHES*, 2003, pp. 382–396.
[37] M. Joye and M. Tunstall, Eds., *Fault Analysis in Cryptography*, ser. Information Security and Cryptography. Springer, 2012.
[38] H. A. Khan, M. Alam, A. G. Zajic, and M. Prvulovic, "Detailed tracking of program control flow using analog side-channel signals: a promise for IoT malware detection and a threat for many cryptographic implementations," in *SPIE Defense + Security, 2018*, 2018.
[39] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *IEEE SP*, 2019, pp. 1–19.
[40] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *CRYPTO'96*, 1996, pp. 104–113.
[41] P. C. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, "Introduction to differential power analysis," *J. Cryptographic Engineering*, vol. 1, no. 1, pp. 5–27, 2011.
[42] A. Langley, M. Hamburg, and S. Turner, "Elliptic curves for security," RFC 7748, Internet Engineering Task Force, 2016. [Online]. Available: http://www.ietf.org/rfc/rfc7748.txt
[43] D. Lee, D. Jung, I. T. Fang, C.-C. Tsai, and R. A. Popa, "An off-chip attack on hardware enclaves via the memory bus," in *USENIX Security*, 2020.
[44] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache attacks on mobile devices," in *USENIX Security*, 2016, pp. 549–564.
[45] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *USENIX Security*, 2018, pp. 973–990.
[46] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE SP*, 2015, pp. 605–622.
[47] J. Longo, E. De Mulder, D. Page, and M. Tunstall, "SoC it to EM: electromagnetic side-channel attacks on a complex system-on-chip," in *CHES*, 2015, pp. 620–640.
[48] S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks — Revealing the Secrets of Smart Cards*. Springer, 2007.
[49] T. S. Messerges, E. A. Dabbish, and R. H. Sloan, "Power analysis attacks of modular exponentiation in smartcards," in *CHES*, 1999, pp. 144–157.
[50] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.
[51] P. L. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization," in *Mathematics of Computation*, vol. 13, no. 3, 1987, pp. 243–264.
[52] E. D. Mulder, S. B. Örs, B. Preneel, and I. Verbauwhede, "Differential power and electromagnetic attacks on a FPGA implementation of elliptic curve cryptosystems," *Computers & Electrical Engineering*, vol. 33, no. 5–6, pp. 367–382, 2007.
[53] E. Nascimento and L. Chmielewski, "Applying horizontal clustering side-channel attacks on embedded ECC implementations," in *CARDIS*, T. Eisenbarth and Y. Teglia, Eds., 2017, pp. 213–231.
[54] E. Nascimento, L. Chmielewski, D. Oswald, and P. Schwabe, "Attacking embedded ECC implementations through cmov side channels," in *SAC*, 2016, pp. 99–119.
[55] A. Nazari, N. Sehatbakhsh, M. Alam, A. G. Zajic, and M. Prvulovic, "EDDIE: EM-based detection of deviations in program execution," in *ISCA*, 2017, pp. 333–346.
[56] P. Q. Nguyen and I. Shparlinski, "The insecurity of the elliptic curve digital signature algorithm with partially known nonces," *Des. Codes Cryptography*, vol. 30, no. 2, pp. 201–217, 2003.
[57] Olimex, "A13-OLinuXino-MICRO user manual," https://www.olimex.com/Products/OLinuXino/A13/A13-OLinuXino-MICRO/open-source-hardware, accessed April 3, 2016.
[58] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: Practical cache attacks in JavaScript and their implications," in *CCS*, 2015, pp. 1406–1418.
[59] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *CT-RSA*, 2006, pp. 1–20.
[60] C. Percival, "Cache missing for fun and profit," in *BSDCan (2005)*., 2005.
[61] C. Pereida García and B. B. Brumley, "Constant-time callees with variable-time callers," in *USENIX Security*, 2017, pp. 83–98.
[62] J. van de Pol, N. P. Smart, and Y. Yarom, "Just a little bit more," in *CT-RSA*, 2015, pp. 3–21.
[63] J.-J. Quisquater and D. Samyde, "Electromagnetic analysis (EMA): measures and counter-measures for smart cards," in *International Conference on Research in Smart Cards: Smart Card Programming and Security*, 2001, pp. 200–210.
[64] N. Roelofs, N. Samwel, L. Batina, and J. Daemen, "Online template attack on ECDSA: Extracting keys via the other side," in *AFRICACRYPT*, 2020, pp. 323–336.
[65] S. Saab, P. Rohatgi, and C. Hampel, "Side-channel protections for cryptographic instruction set extensions," IACR ePrint archive 2016/700, 2016.
[66] N. Sehatbakhsh, A. Nazari, A. G. Zajic, and M. Prvulovic, "Spectral profiling: Observer-effect-free profiling by monitoring EM emanations," in *MICRO*, 2016, pp. 59:1–59:11.
[67] N. Sehatbakhsh, M. Alam, A. Nazari, A. G. Zajic, and M. Prvulovic, "Syndrome: Spectral analysis for anomaly detection on medical IoT and embedded devices," in *HOST*, 2018, pp. 1–8.
[68] N. Sehatbakhsh, A. Nazari, M. Alam, F. Werner, Y. Zhu, A. Zajic, and M. Prvulovic, "Remote: Ro-

bust external malware detection framework by using electromagnetic signals," *IEEE Transactions on Computers*, vol. 69, no. 3, pp. 312–326, 2019.
[69] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Robust website fingerprinting through the cache occupancy channel," in *USENIX Security*, 2019, pp. 639–656.
[70] F.-X. Standaert, L. Oldeneel tot Oldenzeel, D. Samyde, and J.-J. Quisquater, "Power analysis of FPGAs: How practical is the attack?" in *FPL*, 2003, pp. 701–711.
[71] J. Stecklina and T. Prescher, "LazyFP: Leaking FPU register state using microarchitectural side-channels," *CoRR*, vol. abs/1806.07480, 2018.
[72] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *USENIX Security*, 2018, pp. 991–1008.
[73] W. Van Eck and N. Laborato, "Electromagnetic radiation from video display units: An eavesdropping risk?" *Computers & Security*, vol. 4, pp. 269–286, 1985.
[74] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *IEEE SP*, 2019, pp. 88–105.
[75] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "CacheOut: Leaking data on Intel CPUs via cache evictions," in *IEEE SP*, 2021.
[76] L. Weissbart, S. Picek, and L. Batina, "One trace is all it takes: Machine learning-based side-channel attack on EdDSA," in *SPACE*, 2019, pp. 86–105.
[77] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, "Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution," https://foreshadowattack.eu/foreshadow-NG.pdf, 2018.
[78] Y. Yarom and N. Benger, "Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack," IACR Cryptology ePrint Archive 2014/140, 2014.
[79] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security*, 2014, pp. 719–732.
[80] S.-M. Yen, W.-C. Lien, S.-J. Moon, and J. Ha, "Power analysis by exploiting chosen message and internal collisions — vulnerability of checking mechanism for RSA-decryption," in *Mycrypt 2005*, 2005, pp. 183–195.
[81] B. B. Yilmaz, R. L. Callan, M. Prvulovic, and A. G. Zajic, "Quantifying information leakage in a processor caused by the execution of instructions," in *2017 IEEE Military Communications Conference, MILCOM 2017*, 2017, pp. 255–260.
[82] B. B. Yilmaz, R. L. Callan, M. Prvulovic, and A. Zajić, "Capacity of the EM covert/side-channel created by the execution of instructions in a processor," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 3, pp. 605–620, 2018.
[83] A. Zajic and M. Prvulovic, "Experimental demonstration of electromagnetic information leakage from modern processor-memory systems," *IEEE Transactions on Electromagnetic Compatibility*, vol. 56, no. 4, pp. 885–893, 2014.
[84] A. G. Zajic, "Impact of moving scatterers on vehicle-to-vehicle narrow-band channel characteristics," *IEEE Trans. Vehicular Technology*, vol. 63, no. 7, pp. 3094–3106, 2014.
[85] ZTE, "ZTE ZFIVE 2 LTE," https://www.zteusa.com/zfive2, Jan 17, 2019.