Certified Side Channels

Cesar Pereida García, Sohaib ul Hassan, Nicola Tuveri, and Iaroslav Gridin, *Tampere University;* Alejandro Cabrera Aldaya, *Tampere University and Universidad Tecnológica de la Habana;* Billy Bob Brumley, *Tampere University*

https://www.usenix.org/conference/usenixsecurity20/presentation/garcia

This paper is included in the Proceedings of the 29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

Open access to the Proceedings of the 29th USENIX Security Symposium is sponsored by USENIX.

Certified Side Channels

Cesar Pereida García[1], Sohaib ul Hassan[1], Nicola Tuveri[1],
Iaroslav Gridin[1], Alejandro Cabrera Aldaya[1,2], and Billy Bob Brumley[1]
[1]*Tampere University, Tampere, Finland*
{cesar.pereidagarcia,n.sohaibulhassan,nicola.tuveri,iaroslav.gridin,billy.brumley}@tuni.fi
[2]*Universidad Tecnológica de la Habana (CUJAE), Habana, Cuba*
aldaya@gmail.com

Abstract

We demonstrate that the format in which private keys are persisted impacts Side Channel Analysis (SCA) security. Surveying several widely deployed software libraries, we investigate the formats they support, how they parse these keys, and what runtime decisions they make. We uncover a combination of weaknesses and vulnerabilities, in extreme cases inducing completely disjoint multi-precision arithmetic stacks deep within the cryptosystem level for keys that otherwise seem logically equivalent. Exploiting these vulnerabilities, we design and implement key recovery attacks utilizing signals ranging from electromagnetic (EM) emanations, to granular microarchitecture cache timings, to coarse traditional wall clock timings.

1 Introduction

Academic SCA tends to focus on implementations of cryptographic primitives in isolation. With this view, the assumption is that any higher level protocol or system built upon implementations of these primitives will naturally benefit from SCA mitigations in place at lower levels.

Our work questions this assumption, and invalidates it with several concrete vulnerabilities and attacks against modern software libraries: we dub these *Certified Side Channels*, since the novel attack vector is deeply rooted in cryptography standards. For this vector, "certified" is in the certificate sense (e.g. X.509), not in the Common Criteria sense. Counter-intuitively, we demonstrate that the format in which keys are stored plays a significant role in real world SCA security. Detailed security recommendations for key persistence are scarce; e.g. FIPS 140-2 vaguely states "*Cryptographic keys stored within a cryptographic module shall be stored either in plaintext form or encrypted form [..] Documentation shall specify the key storage methods employed by a cryptographic module*" [1, 4.7.5].

There are (at least) two high level dimensions at play regarding key formats as an SCA attack vector: (i) Among the multitude of standardized cryptographic key formats to choose from when persisting keys: *which one to choose, and does the choice matter?* Surprisingly, it does—we demonstrate different key formats trigger different behavior within software libraries, permeating all the way down to the low level arithmetic for the corresponding cryptographic primitive. (ii) At the specification level, alongside required parameters, standardized key formats often contain optional parameters: *does including or excluding optional parameters impact security?* Surprisingly, it does. We demonstrate that omitting optional parameters can cause extremely different execution flows deep within a software library, and also that two keys seemingly mathematically identical at the specification level can be treated by a software library as inequivalent, again reaching very different arithmetic code deep within the library.

Furthermore, we demonstrate that key parsing in general is a lucrative SCA attack vector. This is due mostly to software engineering constraints. Complex libraries inevitably stray to convoluted data structures containing generous nesting levels to meet the demands of broad standardized cryptography. This is exacerbated by the natural urge to handle keys generically when faced with extremely diverse cryptographic standards spanning RSA, DSA, ECDSA, Ed25519, Ed448, GOST, SM2, etc. primitives. The motivation behind this generalization is to abstract away underlying cryptographic details from application developers linking against a library—more often than not, these developers are not cryptography experts. Nevertheless, we observe that when loading keys modern security libraries make varying design choices that ultimately impact SCA security. From the functionality perspective, these design choices are sensible; from the security perspective, we demonstrate they are often questionable.

Outline. Section 2 gives an overview of the related background and previous work. Section 3 discusses the vulnerabilities discovered as a result of our analysis, with microarchitecture SCA evaluations on OpenSSL RSA, DSA, and mbedTLS RSA. We also demonstrate end-to-end attacks on OpenSSL ECDSA using timing and EM side channels in Section 4. We

conclude in Section 5.

2 Background

2.1 Public Key Cryptography

ECDSA. Denote an order-n generator $G \in E$ of an elliptic curve group E with cardinality fn and n a large prime and f the small cofactor. The user's private key α is an integer uniformly chosen from $\{1..n-1\}$ and the corresponding public key is $D = [\alpha]G$. With approved hash function Hash(), the ECDSA digital signature (r,s) on message m (denoting with $h < n$ the representation of Hash(m) as an integer) is

$$r = ([k]G)_x \bmod n, \quad s = k^{-1}(h + \alpha r) \bmod n \qquad (1)$$

where k is a nonce chosen uniformly from $\{1..n-1\}$.

RSA. According to the PKCS #1 v2.2 standard (RFC 8017 [55]), an RSA private key consists of the eight parameters $\{N, e, p, q, d, d_p, d_q, i_q\}$ where all but the first two are secret, and $N = pq$ for primes p, q. Public exponent e is usually small and the following holds:

$$d = e^{-1} \bmod \operatorname{lcm}(p-1, q-1) \qquad (2)$$

In addition, Chinese Remainder Theorem (CRT) parameters are stored for speeding up RSA computations:

$$d_p = d \bmod p, \quad d_q = d \bmod q, \quad i_q = q^{-1} \bmod p \qquad (3)$$

2.2 Key Formats

Interoperability among different software and hardware platforms in handling keys and other cryptographic objects requires common standards to serialize and deserialize such objects. `ASN.1` or *Abstract Syntax Notation One* is an interface description language to define data structures and their (de/)serialization, standardized [69] jointly by ITU-T and ISO/IEC since 1984 and widely adopted. It supports several encoding rules, among which the *Distinguished Encoding Rules* (DER), a binary format ensuring uniqueness and concision, has been preferred for the representation of cryptographic objects. `PEM` (RFC 7468 [45]) is a textual file format to store and trasmit cryptographic objects, widespread despite being originally developed as part of the now obsoleted IETF standards for *Privacy-Enhanced Mail* after which it is named. `PEM` uses `base64` to encode the binary DER serialization of an object, providing some degree of human readability and support for text-based protocols like e-mail and HTTP(S).

Object Identifiers. The `ASN.1` syntax also defines an `OBJECT IDENTIFIER` primitive type which represents a globally unique identifier for an object. ITU-T and ISO jointly manage a decentralized hierarchical registry of object identifiers or `OID` s. The registry is organized as a tree structure, where every node is authoritative for its descendants, and decentralization is obtained delegating the authority on subtrees to entities such as countries and organizations. This mechanism solves the problem of assigning globally unique identifiers to entities to facilitate global communication.

RSA private keys. PKCS #1 (RFC 8017 [55]) also defines the `ASN.1` DER encoding for an RSA private key, defining an item for each of its eight parameters. As further discussed in Section 3.4, the standard does not strictly require implementations to include all the eight parameters during serialization, nor to invalidate the object during deserialization if one of the parameters is not included.

EC private keys. The ANSI X9.62 standard [51] is the normative reference for the definition of the ECDSA cryptosystem and the encoding of ECDSA public keys, but omits a serialization for private keys. The SEC1 standard [2] follows ANSI X9.62 for the public key `ASN.1` and provides a DER encoding also for EC private keys, but allows generous variation as it seems to assume different encapsulating options depending on different protocols in which the EC private key can be used. Flexibility in the format brings complexity in the deserializer implementation, that needs to be stateful w.r.t. parsing of the container of the private key encoding and flexible enough to interoperate with other implementations and interpretations of the standards: this already suggests that the parsing stage shows potential as a lucrative SCA attack vector. The SEC1 `ASN.1` notation for `ECPrivateKey` contains the private scalar as an octet string, an optional (depending on the container) `ECDomainParameters` field, and an optional bit string field to include the public part of the key pair. The `ECDomainParameters` can be null, if the curve parameters are specified in the container encapsulating the `ECPrivateKey`, or contain either an `OID` for a "named" curve, or a `SpecifiedECDomain` structure. The latter, simplifying, contains a description of the field over which the EC group is defined, the definition of the curve equation in terms of the coefficients of its Short Weierstrass form, an encoding of the EC base point, and its order n. Finally it can *optionally* contain a component to represent a small cofactor f as defined at the beginning of this section. In Section 3.1 we will further discuss about the security consequences caused in actual implementations by the logic required to support the cofactor as an optional field.

MSBLOB key format. MSBLOB is the OpenSSL implementation of Microsoft's private key BLOB format[1] supporting different cryptosystems, using custom defined structures and fields. DSS key BLOB uses an arbitrary structure, while RSA key BLOBs follows PKCS #1 with minor differences. To identify each cryptosystem, a "magic member" is used in the key BLOB structure—the member is the hexadecimal representation of the ASCII encoding of the cryptosystem name, e.g. "RSA1", "RSA2", "DSS1", "DSS2", etc., where the integer

[1] https://docs.microsoft.com/en-us/windows/win32/seccrypto/base-provider-key-blobs

dictates if it is a public or a private key. Public and private key BLOBs are stored as binary files in little-endian order and by default the private key BLOBs are not encrypted—it is up to the developers to choose whether to encrypt the key. Microsoft created the public and private key BLOBs in order to support cryptographic service providers (CSP), i.e. third party cryptographic software modules. It is worth noting that both private and public BLOBs are independent from each other, thus allowing a CSP to only support and implement the desired format according to the cryptosystem in use, meaning that public keys can be computed on-demand using the private key BLOB information.

PVK key format. The PriVate Key (PVK) format is a Microsoft proprietary key format used in Windows supporting signature generation using both DSA and RSA private keys. Little information is available about this format but a key is typically composed of a header containing metadata, and a body containing a private key BLOB structure as per the previous description. Following the same idea as in the private key BLOB, the PVK header metadata contains the "magic" value `0xb0b5f11e`[2] to uniquely identify this key format. Additionally, PVK's header contains metadata information for key password protection, preventing the storage of private key information in plain text. Unfortunately, PVK is an outdated format and it only supports RC4 encryption, moreover, in some cases PVK keys use a weakened encryption key to comply with the US export restrictions imposed during the 90's [3].

2.3 Side-Channel Analysis

SCA is a cryptanalysis technique used to target software and hardware implementations of cryptographic primitives. The main goal of SCA is to expose hidden algorithm state by measuring variations in time, power consumption, electromagnetic radiation, temperature, and sound. These variations might leak data or metadata that allows the retrieval of confidential information such as private keys and passwords. The history of SCA is long and rich—from the military program called TEMPEST [31] to current commodity PCs, SCA has deeply impacted security-critical systems and it has reached the most popular and widely used cryptosystems over the years such as AES, DSA, RSA, and ECC, implemented in the most widely used cryptographic libraries including OpenSSL, BoringSSL, LibreSSL, and mbedTLS.

SCA can be broadly categorized (w.r.t. signal procurement techniques) in two specific research fields: hardware and software. Both fields have evolved and developed their own techniques, and the line separating them has blurred as research improves, and attacks become more complex. Nevertheless, the ultimate goal is still the same: extract confidential information from a device executing vulnerable cryptographic code. A brief overview follows.

Hardware. Ever since their inception, System-on-Chip (SoC) embedded devices have become passively ubiquitous in the form of mobile devices and IoT, performing security critical tasks over the Internet. Their basic building blocks—in terms of performing computations—are the CMOS transistors, drawing current during the switching activity to depict the behavior of logic gates. Power analysis attacks introduced by Kocher et al. [49] rely on the fact that accumulated switching activity of these transistors influence the overall power fluctuations while secret data dependent computations take place on the processor and memory subsystems.

While power analysis is one way to perform SCA, devices may also leak sensitive information through other means such as EM [5], acoustic [34], and electric potential [36]. In contrast to the power side channels which require physically tapping onto the power lines, EM and acoustic based SCA add a spatial dimension. There may be slight differences when it comes to acquiring and processing these signals, but in essence the concept is similar to traditional power analysis, hence the hardware based SCA techniques generally apply to all.

Over the years more powerful SCA techniques have emerged such as differential power analysis [49], correlation power analysis [19], template attacks [25], and horizontal attacks [12]. Most of these techniques rely on statistical methods to find small secret data dependent leakages.

Traditionally, hardware SCA research mainly focuses on architecturally simpler devices such as smart cards and microcontrollers [52, 65, 66]. Being simple here does not imply that developing and deploying such cryptosystems is simpler, rather in terms of their functionality and hardware architecture. Modern consumer electronics (e.g. smart phones) are more feature rich, containing SoC components, memory subsystems and multi-core processors with clock speeds in gigahertz. These devices are often running a full operating system (several in fact) making it possible to deploy software libraries such as OpenSSL. More recently, a new class of hardware side channel attacks on embedded, mobile devices and even PCs has emerged, targeting crypto software libraries such as OpenSSL [38, 50], GnuPG [34, 35, 36, 37], PolarSSL [29], Android's Bouncy Castle [13], and WolfSSL [68]. They employ various signal processing tools to counter the noise induced by complex systems and microarchitectures. For further details, Tunstall [71] present an elaborate discussion on hardware based SCA techniques, while Danger et al. [27] and Abarzúa et al. [3] sum up various SCA attacks and their countermeasures.

Software. The widespread use of e-commerce and the need for security on the Internet sparked the development of cryptographic libraries such as OpenSSL. Researchers quickly began analyzing these libraries and it took a short time to find security flaws in these libraries. Impulsed by Kocher's

[2] Leetspeak for "bobsfile"!

[3] http://justsolve.archiveteam.org/wiki/PVK

work [48], SCA timing attacks quickly gained traction. By measuring the amount of time required to perform private key operations, the author demonstrated that it was feasible to find Diffie-Hellman exponents, factor RSA keys, and recover DSA keys. Later Brumley and Boneh [23] demonstrated that it was possible to do the same but remotely, by measuring the response time from an OpenSSL-powered web server. Other TLS-level timing attacks include [47] with a software target and [53] with a hardware target.

As software SCA became more complex and sophisticated, a new subclass of attacks denominated "microarchitecture attacks" emerged. Typically, a modern CPU executes multiple programs either concurrently or via time-sharing, increasing the need to optimize resource utilization to obtain high performance. To achieve this goal, microarchitecture components try to predict future behavior and future resource usage based on past program states. Based on these observations, researchers [15, 60] discovered that some microarchitecture components—such as the memory subsystem—work wonderfully as communication channels. Due to their shared nature between programs, some of the microarchitecture components can be used to violate access control and achieve inter-process communication. Among these components, researchers noticed that the memory subsystem is arguably the easiest to exploit: by observing the memory footprint an attacker can leak algorithm state from an executing cryptographic library in order to obtain secret keys. Since the initial discovery, several SCA techniques have been developed to extract confidential data from different memory levels and under different threat models. Some of these techniques include FLUSH+RELOAD [78], PRIME+PROBE [59], EVICT+TIME [59], and FLUSH+FLUSH [42]. Moreover, recent research [9, 24, 74] shows that most (if not all) microarchitecture components shared among programs are a security hazard since they can potentially be used as side-channels. Ge et al. [33] provide a great overview on software SCA, including the types of channels, microarchitecture components, side-channel attacks, and mitigations.

2.4 Lattice Attacks

In Section 4 we present two attacks against ECDSA signing that differ in SCA technique, but share a common pattern: (i) gathering several (r,s,m) tuples in a collection phase, using SCA to infer partial knowledge about the nonce used during signature generation; (ii) a recovery phase combines the collected tuples and the associated partial knowledge to retrieve the long-term secret key.

To achieve the latter, we recur to the common strategy of constructing *hidden number problem* (HNP) [18] instances from the collected information, and then use lattice techniques to find the secret key. In this section we discuss the lattice technique used to recover the private keys.

We follow the formalization used in [61], which itself builds on the work by Nguyen and Shparlinski [57, 58], that assumed a fixed amount of known bits (denoted ℓ) for each nonce used in the lattice, but also includes the improvements by Benger et al. [14], using ℓ_i and a_i to represent, respectively, the amount of known bits and their value on a per-equation basis.

The collection phase of [61] as well as our Section 4.2 attack recovers information regarding the LSBs of each nonce, hence it annotates the nonce associated with i-th equation as $k_i = W_i b_i + a_i$, with $W_i = 2^{\ell_i}$, where ℓ_i and a_i are known, and since $0 < k_i < n$ it follows that $0 \leq b_i \leq n/W_i$. Denote $\lfloor x \rfloor_n$ modular reduction of x to the interval $\{0..n-1\}$ and $|x|_n$ to the interval $\{-(n-1)/2..(n-1)/2\}$. Combining (1), define (attacker-known) values $t_i = \lfloor r_i/(W_i s_i) \rfloor_n$ and $\hat{u}_i = \lfloor (a_i - h_i/s_i)/W_i \rfloor_n$, then $0 \leq \lfloor \alpha t_i - \hat{u}_i \rfloor_n < n/W_i$ holds. Setting $u_i = \hat{u}_i + n/2W_i$ we obtain $v_i = |\alpha t_i - u_i|_n \leq n/2W_i$, i.e. integers λ_i exist such that $\mathrm{abs}(\alpha t_i - u_i - \lambda_i n) \leq n/2W_i$ holds. Thus u_i approximate αt_i since they are closer than a uniformly random value from $\{1..n-1\}$, leading to an instance of the HNP [18]: recover α given many (t_i, u_i) pairs.

Consider the rational $d+1$-dimension lattice generated by the rows of the following matrix.

$$B = \begin{bmatrix} 2W_1 n & 0 & \dots & \dots & 0 \\ 0 & 2W_2 n & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \dots & 0 & 2W_d n & 0 \\ 2W_1 t_1 & \dots & \dots & 2W_d t_d & 1 \end{bmatrix}$$

Denoting $\vec{x} = (\lambda_1, \dots, \lambda_d, \alpha)$, $\vec{y} = (2W_1 v_1, \dots, 2W_d v_d, \alpha)$, and $\vec{u} = (2W_1 u_1, \dots, 2W_d u_d, 0)$, then $\vec{x}B - \vec{u} = \vec{y}$ holds. Solving the Closest Vector Problem (CVP) with inputs B and $\vec{u}$ yields $\vec{x}$, and hence the private key α. Finally, as in [61], we embed the CVP into a Shortest Vector Problem (SVP) using the classical strategy [39, Sec. 3.4], and employ an extended search space heuristic [32, Sec. 5].

The presence of outliers among the results of the collection phase usually has a detrimental effect on the chances of success of the lattice attack. The traditional solution is to oversample, filtering $t > d$ traces from the collection phase if d traces are required to embed enough leaked information in the lattice instance to solve the HNP. Indicating with e the amount of traces with errors in the filtered set of size t, picking a subset of size d uniformly at random, the probability for any such subset to be error-free is $\hat{p} = \binom{t-e}{d} / \binom{t}{d}$. For typical values of $\{t, e, d\}$, $\hat{p}$ will be small. Viewing the process of randomly picking a subset and attempting to solve the resulting lattice instance as a Bernoulli trial, the number of expected trials before first success is $1/\hat{p}$. So an attacker can compensate for small $\hat{p}$ by running $j = 1/\hat{p}$ jobs in parallel.

2.5 Triggerflow

Triggerflow [40] is a tool for tracking execution paths, previously used to facilitate SCA of OpenSSL. After users mark up source code with annotations of Points Of Interest (POI) and filtering rules for false positive considerations, Triggerflow runs the binary executable under a debugger and records the execution paths that led up to POIs. The user supplies binary invocation lines called "triggers". These techniques are useful in SCA of software, where areas that do not execute in constant time are known and the user needs to find code that leads up to them. The authors designed Triggerflow with continuous integration (CI) in mind, and maintain an automatic testing setup which continuously monitors all non-EOL branches of OpenSSL for new vulnerabilities by watching execution flows that enter known problematic areas.

Triggerflow is intended for automated regression testing and has no support for automatic POI detection. Thus offensive leakage detection methodologies including (but certainly not limited to) CacheAudit [28], templating [21, 41], CacheD [75], and DATA [77] complement Triggerflow to establish POIs. One approach is to apply these leakage detection methodologies, filter out false positives, limit to functions deemed security-critical and worth tracking, then use the result to add Triggerflow source code annotations for CI. See [40, Sec. 7] for a more extensive discussion.

3 Vulnerabilities

We used Triggerflow to analyze several code paths on multiple cryptographic libraries, discovering SCA vulnerabilities across OpenSSL and mbedTLS. In this section, we discuss these vulnerabilities, including the unit tests we developed for Triggerflow that detected each of them, then identify the root cause in each case. Following Figure 1, Triggerflow executes each line of the unit tests given in a text file. Triggerflow will trace the execution of lines beginning with `debug` to detect break points getting hit at SCA-critical points in the code. Each such line is security critical—in these examples, generating a key pair or using the private key to e.g. digitally sign a message. Hence if Triggerflow encounters said break points during execution, it represents a potential SCA vulnerability. We compiled the target executables (and shared libraries) with debug symbols, and source code annotated using Triggerflow's syntax to mark previously known SCA-vulnerable functions. Lines that do not begin with `debug` are not traced by Triggerflow, merely executed as preparation steps for subsequent triggers (e.g. setting up public fixed parameters).

Vulnerability-wise, the main results of this section are as follows: (i) bypassing SCA countermeasures using ECC explicit parameters (Section 3.1, OpenSSL); (ii) bypassing SCA countermeasures for DSA using PVK and MSBLOB key formats (Section 3.2, OpenSSL); (iii) bypassing SCA countermeasures for RSA by invoking key validation (Section 3.3, OpenSSL); (iv) bypassing SCA countermeasures for RSA through key loading (Section 3.4, mbedTLS).

```
# ECDSA with explicit curve parameters, zero cofactor
debug openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:P-256
  ↪ -pkeyopt ec_param_enc:explicit -outform DER -out p256.der
sed -i 's/\x25\x51\x02\x01\x01/\x25\x51\x02\x01\x00/' p256.der
debug openssl dgst -sha256 -sign p256.der -keyform DER -out /dev/null
  ↪ /etc/lsb-release
```

```
# DSA with PVK key format
openssl genpkey -genparam -algorithm DSA -out dsa.params -pkeyopt
  ↪ dsa_paramgen_bits:1024 -pkeyopt dsa_paramgen_q_bits:160
debug openssl genpkey -paramfile dsa.params -out dsa.pkey
debug openssl dsa -in dsa.pkey -outform PVK -pvk-none -out dsa.pvk
debug openssl dgst -sha1 -sign dsa.pvk -keyform PVK -out /dev/null
  ↪ /etc/lsb-release
```

```
# DSA with MSBLOB key format
openssl genpkey -genparam -algorithm DSA -out dsa.params -pkeyopt
  ↪ dsa_paramgen_bits:1024 -pkeyopt dsa_paramgen_q_bits:160
debug openssl genpkey -paramfile dsa.params -out dsa.pkey
debug openssl dsa -in dsa.pkey -outform MS\ PRIVATEKEYBLOB -out dsa.blob
debug openssl dgst -sha1 -sign dsa.blob -keyform MS\ PRIVATEKEYBLOB
  ↪ -out /dev/null /etc/lsb-release
```

```
# RSA key validation in OpenSSL
openssl genrsa -out rsa.pem 2048
debug openssl rsa -in rsa.pem -check
debug openssl pkey -in rsa.pem -check
```

```
# RSA key loading in mbedTLS
create_rsa_pem.sh without_d > custom.pem
debug mbedtls_pk_sign custom.pem
```

Figure 1: New Triggerflow unit tests.

3.1 ECC: Bypass via Explicit Parameters

From a standardization perspective, curve data for ECC key material gets persisted in one of two ways: either including the specific OID that points to a named curve with fixed parameters, or explicitly specifying the curve with `ASN.1` syntax. Mathematically, they seem equivalent. To explore the potential difference in security implications between these options, we constructed three keys: (i) a NIST P-256 private key as a named curve, using the `ec_param_enc:named_curve` argument to the OpenSSL `genpkey` utility; (ii) a NIST P-256 private key with explicit curve parameters, using the `ec_param_enc:explicit` argument; (iii) a copy of the previous key, but post-modified with the OpenSSL `asn1parse` utility to remove the optional cofactor. The first two keys additionally used the `ec_paramgen_curve:P-256` argument to specify the target curve. We highlight that, from a standards perspective, all three of these keys are valid. We then integrated the commands to produce these keys into the Triggerflow framework as unit tests. Finally, we added an OpenSSL `dgst` utility unit test for each of these keys in Triggerflow, to induce ECDSA signing. What follows is a discussion on the three distinct control flow cases for each key, regarding the security-critical scalar multiplication operation.

Named curve. Triggerflow indicated `ecp_nistz256_points_mul` handled the operation. The reason for this is OpenSSL uses an `EC_METHOD` structure for legacy ECC; the assignment of structure instances to specific curves happens at library compile time, allowing different curves to have different (optimized) implementations depending on archi-

tecture and compiler features. This particular function is part of the `EC_GFp_nistz256_method`, an `EC_METHOD` optimized for AVX2 architectures [43]. The implementation is constant time, hence this is the best case scenario.

Explicit parameters. Triggerflow indicated `ec_scalar_mul_ladder` handled the operation, through the default `EC_GFp_simple_method`, the generic implementation for curves over prime fields. In fact this is the oldest `EC_METHOD` in the codebase, present since ECC support appeared in 2001. The implementation of this particular function was mainlined in 2018 [72] as a result of CVE-2018-5407 [9], SCA-hardening generic curves with the standard Montgomery ladder. Interpreting this Triggerflow result, we conclude OpenSSL has no runtime mechanism to match explicit parameters to named curves present in the library. Ideally, it would match the explicit parameters to `EC_GFp_nistz256_method` for improved performance and SCA resistance. Failure to do so bypasses one layer of SCA mitigations, but in this particular case the default method still features sufficient SCA hardening.

Explicit parameters, no cofactor. Triggerflow indicated `ec_wNAF_mul` handled the operation through the same `EC_METHOD` as the previous case. This is a known SCA-vulnerable function since 2009 [21], and is a POI maintained in the Triggerflow patchset to annotate OpenSSL for automated CI. Root causing the failed Triggerflow unit test, the function only early exits to the SCA-hardened ladder if both the curve generator order and the curve cardinality cofactor are non-zero. Since the optional cofactor is not present in the key, the library assigns zero as the default, indicating either the provided cofactor was zero or not provided at all. The OpenSSL ladder implementation utilizes the cofactor as part of SCA hardening, hence the code unfortunately falls through to the SCA-insecure version in this case, bypassing the last layer of SCA defenses for scalar multiplication. This is the path we will exploit in Section 4.

Keys in the wild. While we reached a vulnerable code path through a standards-compliant, valid, non-malicious key, the fact is the OpenSSL CLI will not organically emit a key in this form. One can argue that OpenSSL is far from the only security tool that produces keys conforming to the specification, that it must subsequently parse since they are valid. Nevertheless, this leaves us with the question: *do keys like this exist—does this vulnerability matter?* Investigating, we at least found two deployment classes this vulnerability affects: (i) The GOST engine[4] for OpenSSL, dynamically adding support for Russian cryptographic primitives in RFC 4357 [64]. Since the curves from the standard are not built-in to OpenSSL, the engine programatically constructs the curve based on fixed parameters inside the engine. However, since the cofactor parameter to the OpenSSL `EC_GROUP_set_generator` API is optional, the engine developers omit it in earlier versions, passing `NULL`. When GOST keys are persisted, they have their own OID distinct from legacy ECC standards and only support named curves; however, the usage of these curves within the engine hits the same exact code path. (ii) GOSTCoin[5] is the official software stack for a cryptocurrency. It links against OpenSSL for cryptographic functionality, but does not support the GOST engine. Examining the digital wallet, we manually extracted several DER-encoded legacy (OID-wise) ECC private keys from the binary. Parsing these keys revealed they are private keys with explicit parameters from RFC 4357 [64], "Parameter Set A". Upon closer inspection, the cofactor is present in the `ASN.1` encoding, yet explicitly set to zero. Similar to the previous case, this is due to failure to supply the correct cofactor to the OpenSSL EC API when constructing the curve.

From this brief study, we can conclude that failure to provide the valid cofactor to the OpenSSL EC API when constructing curves programmatically (the only choice for curves not built-in to the library), or importing a (persisted) ECC private key with explicit parameters containing a zero or omitted (spec-optional) cofactor are characteristics of applications affected by this vulnerability.

Related work. Concurrent to our work, Takahashi and Tibouchi [70] utilize explicit parameters in OpenSSL to mount a fault injection attack. They invasively induce a fault during key parsing to change OpenSSL's representation of a curve coefficient. This causes decompression of the explicit generator point to emit a point on a weaker curve, subsequently mounting a degenerate curve attack [56]. At a high level, the biggest differences from our work are the invasive attack model and limited set of applicable curves.

Subsequent to our work, CVE-2020-0601 tracks the "Curveball" vulnerability. It affects the Windows CryptoAPI and uses ECC explicit parameters to match a named curve in all but the custom generator point, allowing to spoof code-signing certificates.

3.2 DSA: Bypass via Key Formatting

As the Swiss knife of cryptography, OpenSSL provides support for PVK and MSBLOB key formats to perform digital signatures using DSA. In fact, OpenSSL has supported these formats since version 1.0.0, hence the library has a dedicated file in `crypto/pem/pvkfmt.c` for parsing these keys. The file contains all the logic to parse Microsoft's DSA and RSA private key BLOBs, common to both PVK and MSBLOB key formats. Unfortunately, the bulk of code for parsing the keys has seen few changes throughout the years, and more importantly it has missed important SCA countermeasures that other parts of the code base have received [62], allowing this vulnerability to go unnoticed in all OpenSSL branches until now.

[4] https://github.com/gost-engine/engine

[5] https://github.com/GOSTSec/gostcoin

As mentioned previously, PVK and MSBLOB key files contain only private key material but OpenSSL expects the public key to be readily available. Thus every time it loads any of these key formats, the library computes the corresponding public key. More specifically, the upper level function `b2i_dss` reads the private key material and subsequently calls the `BN_mod_exp` function to compute the public key using the default modular exponentiation function, without first setting the constant-time flag `BN_FLG_CONSTTIME`. Note that this vulnerability does not depend on whether the PVK key is encrypted or not, because when the code reaches the `b2i_dss` function, the key has been already decrypted, and the modular exponentiation function is already leaking private key material. This default SCA-vulnerable modular exponentiation algorithm follows a square-and-multiply approach—first precomputes a table of multipliers, and then accesses the table during the square-and-multiply step. Already in 2005 Percival [60] demonstrated an L1 data cache-timing attack against this function during RSA decryption. We found that the original flaw is still present, but this time in the context of DSA.

Figure 2 demonstrates the side-channel leakage obtained by our L1 data-cache malicious spy process running in parallel with OpenSSL during a modular exponentiation operation while computing the DSA public key using PVK and MSBLOB key formats. Using the PRIME+PROBE technique, our spy process is able to measure the latency of accessing a specific cache set (y-axis) over time (x-axis) to obtain a sequence of pre-computed multipliers accessed during computation. In OpenSSL a multiplier is represented as a BIGNUM structure spanning approximately across three different cache sets. Reading from top-to-bottom and left-to-right, and after a brief period of noise, the figure shows that every block of approximately three continuous high latency cache sets corresponds to a multiplier access. An attacker can not only trace the multipliers accessed, but also the order in which they were accessed during the exponentiation, leaking more than half of the exponent bits. This information greatly reduces the effort to perform full key recovery. Moreover, the public key is computed every time the private key is loaded, thus an attacker has several attempts at tracing the sequence of operations performed during the exponentiation. Our experiments reveal that cache sets stay constant across multiple invocations of modular exponentiation, reducing the attacker's effort and permitting the use of statistical techniques to improve the leakage quality.

Keys in the wild. PVK and MSBLOB are based on MS proprietary private key formats—nevertheless they are widely found in use in open source software. MSBLOB keys are supported by MS Smart Card CSP and OpenSC[6], an open source software library for smart cards linking to OpenSSL. In fact, OpenSC has a function[7] that creates a key container—by calling the OpenSSL vulnerable function—whenever "the card either does not support internal key generation or the caller requests that the key be archived in the card", facilitating the attack in a smart card setting. On the other hand, MS Visual Studio 2019 provides tools[8] to generate, convert, and sign Windows drivers, libraries, and catalog files using the PVK format. In a typical workflow, `MakeCert` generates certificates and the corresponding private key, then `Pvk2Pfx` encapsulates private keys and certificates in a PKCS #12 container, and finally `SignTool` signs the driver. Interestingly, `MakeCert` and `SignTool` successfully generate keys and signatures using RSA and DSA, but `Pvk2Pfx` fails to accept any key that is not RSA—a gap filled by the vulnerable OpenSSL, creating compliant PKCS #12 keys. Other libraries such as `jsign`[9], `osslsigncode`[10], and the Mono Project[11] exist to provide signing capabilities using MS proprietary private key formats outside of Windows. We can expect this vulnerability to be exploitable by an attacker targeting Windows developers.

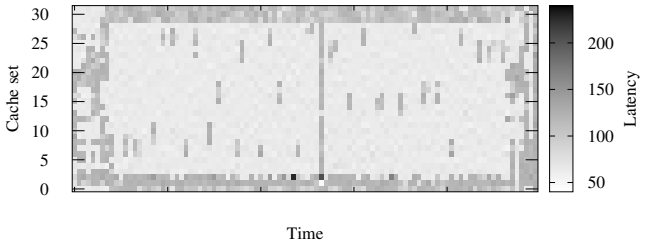

Figure 2: L1 dcache trace showing distinctive access patterns to pre-computed multipliers in cache sets 6-8, 9-11, 13-16, 15-17, 22-24, 25-27, 28-30 during DSA public key computation.

3.3 RSA: Bypass via Key Validation

RSA key validation is a common operation required in a cryptography library supporting RSA to verify that an input key is indeed a valid RSA key. We found that OpenSSL function `RSA_check_key_ex` located at `crypto/rsa/rsa_chk.c` contains several SCA vulnerabilities. In fact, we found that the affected function `RSA_check_key_ex` can be accessed by two public entry points: a direct call to `RSA_check_key`, and through the public EVP interface calling `EVP_PKEY_check` on an RSA key. Figure 1 shows the commands in OpenSSL leading to the affected code path through the two different public functions. Note that any external, OpenSSL-linking application calling any of these two public functions is also affected.

The check function takes as input an RSA key, parses the

[6] https://github.com/OpenSC/OpenSC

[7] https://github.com/OpenSC/OpenSC/blob/master/src/minidriver/minidriver.c#L3308

[8] https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/tools-for-signing-drivers

[9] https://ebourg.github.io/jsign/

[10] https://sourceforge.net/projects/osslsigncode/files/osslsigncode/

[11] https://www.mono-project.com/

key, and reads all of the private and public components, checking the correctness of all the components. In general, the function validates the primality of p and q, then it recomputes the rest of the values $\{N, d, d_p, d_q, i_q\}$ to compare against the parsed values and check their validity. Unfortunately, we found that in several cases OpenSSL uses by default SCA-vulnerable functions to recompute these secret values.

Primality testing vulnerabilities. The prime values p and q are the first components verified during the process. The verification is done using the Miller-Rabin primality test [67] as implemented in the function `BN_is_prime_fasttest_ex`. This function calls a lower level *witness* function named `bn_miller_rabin_is_prime`[12] where a b base value is chosen randomly to compute $b^m \bmod p$, in which p is the candidate prime and the relation $2^a m = p-1$ holds. The *witness* exponentiation is performed using the `BN_mod_exp_mont` function, where unfortunately the `BN_FLG_CONSTTIME` is not set beforehand. Thus a variable-time sliding window exponentiation is used, allowing a malicious process to potentially perform a data cache-timing attack to recover half of the bits from the exponent [60]. This is enough information to recover both prime values p and q. Moreover, the exponentiation function gets called several time by the *witness* function with different b values in order to obtain confidence about the prime values, providing multiple attempts for an attacker to capture the leakage and perform error correction during its key recovery attack.

In addition to the previous vulnerability, as part of the *witness* function, a Montgomery setup phase occurs in `BN_MONT_CTX_set`, where the inverse of $2^w \bmod p$ for w-bit architectures is computed. The modular inverse function `BN_mod_inverse` is called without setting the constant-time flag. The inverse operation uses a variation of the greatest common divisor (GCD) algorithm, which is dependent on its inputs $\{2^w, p \bmod 2^w\}$, thus leaking algorithm state equivalent to the least significant word of both p and q.

Secret value vulnerabilities. Once the prime values p and q are deemed correct, the key validation continues by computing the rest of the secret components where more vulnerabilities are found. To compute the private exponent d during the verification code path, OpenSSL uses the least common multiple (LCM) of $p-1$ and $q-1$. Nevertheless, this operation is computed as

$$\mathrm{lcm}(p-1, q-1) = \frac{(p-1)\cdot(q-1)}{\gcd(p-1, q-1)} \tag{4}$$

performing the GCD computation using the `BN_gcd` function. This function does not have an early exit to a constant-time function, instead it completely ignores the flag existence, so even if it was set it would not have any effect on the code path taken. Finally, the last vulnerability is observed during CRT i_q computation. OpenSSL computes this parameter using the `BN_mod_inverse` function, which yet again fails to properly set the constant-time flag, leaving the computation $q^{-1} \bmod p$ unprotected.

It is worth noting that variable-time GCD functions, and variants, potentially leak all the algorithm state. Depending on the attacker capabilities [6], an attacker is fully capable of recovering the input values, i.e. p and q.

As can be observed, all of the vulnerabilities leak on p and q at different degrees, but by combining all the leaks, an attacker can use the redundancy and number-theoretic constraints to correct errors and obtain certainty on the bits leaked.

Keys in the wild. Surprisingly, the vulnerabilities presented in this section do not depend on a special key format. In fact, the vulnerabilities are triggered whenever an RSA key is checked for validity using the OpenSSL library, thus a potential attacker could simply wait for the right moment to exploit these vulnerabilities. The potential impact of these vulnerabilities is large, but it is minimized by two important factors: the user must trigger an RSA key validation; and the attacker must be collocated in the same CPU as the user. Nevertheless, this is not a rare scenario, and thus exploitation is very much possible.

3.4 RSA: Bypass via Missing Parameters

Recalling Section 2, an RSA private key is composed by some redundant parameters while at the same time not all of them are mandatory per RFC 8017 [55]: *"An RSA private key should be represented"*. This implies that cryptography implementations must deal with RSA private keys that do not contain all parameters, requiring potentially computing them on demand. Natural questions arise: (i) *How do software libraries handle this uncertainty?* (ii) *Does this uncertainty mask SCA threats?* Shifting focus from OpenSSL, the remainder of this section analyzes the open source mbedTLS library in this regard.

Fuzzing RSA private key loading. Following the Triggerflow methodology, we developed unit tests for the mbedTLS library, specifically for targeting RSA key loading code paths. To this end, we analyzed the mbedTLS v2.18.1 `bignum` implementation and set three POIs for Triggerflow: (i) GCD computation, `mbedtls_mpi_gcd`; (ii) Modular multiplicative inverse, `mbedtls_mpi_inv_mod`; (iii) Modular exponentiation, `mbedtls_mpi_exp_mod`. We arrived at these POIs from state-of-the-art SCA applied to cryptography libraries where these operations are commonly exploited. The first two functions are based on the binary GCD algorithm, previously shown weak to SCA [4, 7, 10, 61, 76], while exponentiation is a classical SCA target [17, 26, 35, 49, 62].

With these POIs, we fuzz the RSA mbedTLS private key loading code path to identify possible vulnerabilities. The fuzzing consists of testing the loading of an RSA private key when some parameters are equal to zero (i.e. empty PKCS #1 parameter).

[12] In OpenSSL 1.0.2 the function is called `witness`.

After configuring the potential leaking functions as Triggerflow POIs, we created an RSA private key fuzzing utility that generates all possible combinations of PKCS #1-compliant private keys. This ranges from a private key that includes all PKCS #1 parameters to none. While the latter is clearly invalid as it carries no information, other missing combinations could be interesting regarding SCA. As PKCS #1 defines eight parameters, the number of private key combinations compliant with this standard is 256.

Triggerflow provides a powerful framework for testing all these combinations smoothly. Using Triggerflow for each of these private keys, we tested the generic function of mbedTLS for loading public keys: `mbedtls_pk_parse_keyfile`. The advantage of using Triggerflow for this task is that we can automate the whole process of testing each code corner of this execution path, searching for SCA threats. Figure 1 (bottom) shows a Triggerflow unit test of one of these parameter combinations, with a private key missing d. Unit tests for the other combinations are similar.

Results. For each combination, we obtained a report that indicates if and where POIs were hit or not, also recording the program return code. A quick analysis of the generated reports indicates the 256 combinations group in four classes (i.e. only four unique reports were generated for all 256 private key parameter combinations). Table 1 shows the number of keys for each group. The majority of private key combinations yield an "Invalid" return code without hitting a POI before returning.

The group "Public" contains those remaining *valid* private keys for which $\{d, p, q\}$ is not a subset of included parameters. In this case, mbedTLS recognized the key as a public key even if the CRT secret parameters are present. Nevertheless, identified as "Public" by mbedTLS, we ignore them, since no secret data processing takes place.

Table 1: Report groups for the 256 private keys.

Group	Number of keys
Invalid	216
Public	8
POI-hit (CRT)	16
POI-hit (CRT & d)	16

The last two groups in Table 1 contain those private keys (32 in total) that indeed hit at least one POI. Analyzing both reports on these groups, we identified two potential leakage points. One is related to processing of the CRT parameters, and the other to computation of the private exponent d. We now investigate if these hits represent an SCA threat. Appendix A details the complete list of parameter combinations that hit a POI.

Leakage analysis: CRT. The last two report groups have at least one hit at a Triggerflow POI in a CRT related computation. In both groups, the report regarding this code path is identical, hence the following analysis applies to both.

The Triggerflow report reveals hitting the modular inverse POI; the parent function is `mbedtls_rsa_deduce_crt`, computing the CRT parameters in (3) as $i_q = q^{-1} \bmod p$ using `mbedtls_mpi_inv_mod`. It is a variant of the binary extended Euclidean algorithm (BEEA) with an execution flow highly dependent on its inputs, therefore an SCA vulnerability. This is similar to OpenSSL's Section 3.3 vulnerability. Yet in contrast to OpenSSL, this code path in mbedTLS executes every time this library loads a private key: the vulnerability exists regardless of missing parameters in the private key.

Leakage analysis: private exponent. The last group in Table 1 contains the CRT leakage previously described in addition to one related to private exponent d processing. The targeted POIs hit by all private key parameter combinations in this group are `mbedtls_mpi_gcd` and `mbedtls_mpi_inv_mod`. Both are called by the parent function `mbedtls_rsa_deduce_private_exponent`, that aims at computing the private exponent if it is missing in the private key using (2), involving a modular inversion. However, for computing $\mathrm{lcm}(p-1, q-1)$ using (4), the value $\gcd(p-1, q-1)$ needs to be computed first. Therefore, the report indicates a call first to `mbedtls_mpi_gcd` with inputs $p-1$ and $q-1$. This call represents an SCA vulnerability as the binary GCD algorithm is vulnerable in these instances [4, 8, 10]. Note, this leakage is also present in OpenSSL (Section 3.3), however the contexts differ. We observed OpenSSL leakage when verifying d correctness, whereas mbedTLS computes d because it is missing. This difference is crucial regarding SCA, because OpenSSL verifies by checking if $de = 1 \bmod \mathrm{lcm}(p-1, q-1)$ holds; yet mbedTLS indeed computes d, executing a modular inversion (2). Therefore this vulnerability is present in mbedTLS, and absent in OpenSSL.

After obtaining $\mathrm{lcm}(p-1, q-1)$, it computes d using (2) through a call to `mbedtls_mpi_inv_mod`. [61, 76] exploit OpenSSL's BEEA using microarchitecture attacks, so at a high level it represents a serious security threat. A deeper analysis follows for this mbedTLS case.

Summarizing, the private exponent computation in mbedTLS contains two vulnerable code paths: (i) GCD computation of $p-1$ and $q-1$; and (ii) modular inverse computation of e modulo $\mathrm{lcm}(p-1, q-1)$. Next, we investigate which of these represents the most critical threat.

The inputs of the first code path (GCD computation) are roughly the same size. This characteristic implies that, for some SCA signals, the number of bits that can be recovered is small and not sufficient to break RSA. [7, 61] practically demonstrated this limitation using different SCA techniques: the former power consumption, the latter microarchitecture timings.

However, note the inputs of the second code path (modular inversion) differ considerably in size. The public exponent e is typically small, e.g. 65537. Following (4), $\mathrm{lcm}(p-1, q-1)$ has roughly the same number of bits as $(p-1)(q-1)$;

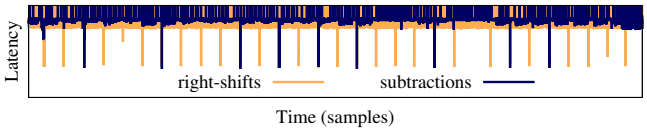

Figure 3: Sequence of right-shifts and subtractions from a FLUSH+RELOAD attack targeting mbedTLS modular inversion.

more than 1024 because $\gcd(p-1, q-1)$ is small with high probability [44]. This significant bit length difference between `mbedtls_mpi_inv_mod` inputs makes this algorithm extremely vulnerable to SCA [8]. This difference implies the attacker knows part of the algorithm execution flow beforehand, and it is exactly this part that is usually difficult to obtain and considerably limits the number of bits that can be recovered employing some SCA techniques as demonstrated in [7, 61]. This characteristic means the attacker only needs to distinguish the main two arithmetic operations present in this algorithm (i.e. right-shift and subtraction) to fully recover the input $\mathrm{lcm}(p-1, q-1)$ that yields d.

Regarding microarchitecture attacks, this distinction lends itself to a FLUSH+RELOAD attack. As part of our validation, we attacked this implementation using a FLUSH+RELOAD attack paired with a performance degradation technique [11]. We probed two cache lines: one detecting right-shift executions, the other subtractions. Figure 3 shows the start of a trace, demonstrating the sequence extraction of right-shifts and subtraction is straightforward.

In addition, the key loading application threat model allows capturing several traces corresponding to the processing of the same secret data. Therefore, the attacker can correct errors that may appear in captured traces (e.g. fix errors produced by preemptions) by combining the information as they are redundant.

Recap. After the analysis of both leaking code paths we detected, we conclude the private exponent leakage is easier to exploit than that of CRT due to the large bitlength difference between the modular inversion algorithm inputs in the former [8, 10, 76]. On the other hand, the private exponent leakage is only present when the private key does not include d; whereas the CRT-related leakage always represents a threat regardless of missing parameters [6]. The number of bits that can be recovered exploiting these leaking code paths depends on the side-channel signal employed. However, these code paths potentially leak all the bits of the processed secrets, as demonstrated in [6, 8, 10, 76].

Keys in the wild. As such, in the context of mbedTLS the simplest example of a vulnerable RSA key is the default key typically generated by libraries, including *all* parameters. We verified this default behavior on e.g. mbedTLS, OpenSSL, and BoringSSL. Hence such keys are ubiquitous in nature. For example, Let's Encrypt's `certbot` tool for automated certificate renewal only supports RSA keys. We conclude that any application linking to mbedTLS for RSA functionality including key parsing is potentially vulnerable, including (but certainly not limited to) ACME-backed web servers relying on mbedTLS for TLS functionality.

4 Two End-to-End Attacks

As highlighted in Section 3, the format used to encode a private key can lead to the bypass of side-channel countermeasures in cryptographic libraries: these are *Certified Side Channels*. In this section we concretely instantiate the threat in Section 3.1 with two SCA attacks against ECDSA signature generation over the popular NIST P-256 curve against OpenSSL 1.1.1a: a remote timing attack and an EM attack.

Target application. For computing the ECDSA signatures from the protocol stack application layer we chose RFC 3161 [79] Time Stamp Protocol. The protocol ensures the means of establishing a time stamping service: a time stamp request message from a client and the corresponding time stamp response from the Trusted Timestamp Authority (TSA). In short, the TSA acts as a trusted third party that binds the Time Stamp Token (TST) to a valid client request message—one way hash of some information—and digitally signs it with the private key. Anyone with a valid TSA certificate can thus verify the existence of the information with the particular time stamp, ensuring timeliness and non-repudiation.

In principle, the client generates a time stamp request message containing the version information, OID of the one way hash algorithm, and a valid hash of the data. Optionally, the client may also send TSA policy OID to be used for creating the time stamp instead of TSA default policy, a random nonce for verifying the response time of the server, and additionally request the signing public key certificate in the TSA response message. The server timestamp response contains a status value and a TST with the OID for the content type and the content itself composed of DER-encoded TST information (TSTinfo). The TSTinfo field incorporates the version number info, the TSA policy used to generate the time stamp response, the message imprint (same as the hashed data in the client request), a unique serial number for the TST, and the UTC based TST generation time along with the accuracy in terms of the time granularity. Depending on the client request, the server response may additionally contain the signing certificate and the client provided nonce value. For further details on TSP, the reader may refer to RFC 3161 [79].

Our attack exploits point multiplication in the ECDSA signature generation during the TSA response phase to recover the long term private key of the server. As a protocol-level target, we compiled and deployed unmodified uts-server[13] v0.2.0 without debug symbols, an open source TSA server linking against an unmodified debug build of OpenSSL 1.1.1a.

[13] https://github.com/kakwa/uts-server

We configured the server with a NIST P-256 X.509 digital certificate, using the private key containing explicit parameters with a zero cofactor, i.e. the preconditions for our Section 3.1 vulnerability. We used the OpenSSL time stamp utility `ts` to create time stamp requests with SHA256 as the hash function, along with a request for the server's public key certificate for verification. We used the provided HTTP configuration for uts-server, hence the TSP messages between the (victim) server and our (attacker) client were transported via standard HTTP.

Target device. We selected a Linux-based PINE A64-LTS board with an Allwinner A64 Quad Core SoC based on Cortex-A53 which supports a 64-bit instruction set with a maximum clock frequency of 1.15 GHz. The board runs Ubuntu 16.04.1 LTS without any modifications to the stock image. We set the board's frequency governor to "performance".

Threat model. As discussed (Section 3.1), when handling such a key in OpenSSL 1.1.1a, the underlying implementation for the EC scalar multiplication is based on a wNAF algorithm, which has been repeatedly targeted in SCA works over the last decade, usually focusing on the recovery of the LSBs of the secret scalar. Contributions from Google [46] partially mitigated the attack vector for select named curves with new `EC_METHOD` implementations, then fully even for generic curves due to the results and contributions from [72]. With the attack vector now open again, this section presents two end-to-end attacks with different signal procurement methods: (i) a novel remote timing attack (Section 4.1), where it is assumed the attacker can measure the overall wall clock time it takes for the TSA server to respond to a request—note this attacker is indistinguishable from a legitimate user of the service; (ii) an EM attack (Section 4.2), similar in spirit to [38, 72], which has the same aforementioned threat model but additionally assumes physical proximity to non-invasively measure EM emanations. The motivation for the two different threat models is due to both practicality and the number of required samples, which will become evident by the end of this section.

4.1 ECDSA: Remote Timing Attack

In contrast to previous work on this code path and to widen potential real-world application, we performed a remote timing attack on the TSA server application via TCP. Instead of taking measurements on this code path server side like e.g. PRIME+PROBE [20, 21] and FLUSH+RELOAD [11, 14, 30, 73], we (as a non-priviliged, normal user of the service), make network requests and measure the wall clock response time.

Experiment setup. We connected the PINE A64-LTS board directly by Ethernet cable to a workstation equipped with an Intel i5-4570 CPU and an onboard I217-LM (rev 04) Ethernet controller. To measure the remote wall clock latency and reduce noise, we created a custom HTTP client for time stamp requests. Its algorithm is as follows: (i) establish a TCP connection to the server; (ii) write the HTTP request and the body, sans a single byte; (iii) start the timer; (iv) write the last body byte—now the server can begin computing the digital signature; (v) read the HTTP response headers—the server might write at least part of them before computing the digital signature; (vi) read one byte of HTTP response body—the digital signature is received by the server directly from linked OpenSSL in an octet string, so reading one byte guarantees it has been generated; (vii) stop the timer; (viii) finish reading the HTTP response; (ix) record the timing information and digital signature in a database; (x) close the TCP connection; (xi) repeat until the requested number of samples has been gathered. We implemented the measurement software in C to achieve maximum performance and control over operations. For the client timer, we used the x86 `rtdtsc` instruction that is freely accessible from user space. In recent Intel processors the `constant_tsc` feature is available—a frequency-independent and easily accessible precision timer.

Performing a traditional timing analysis under the above assumptions, we discovered a direct correlation between the wall clock execution time of ECDSA signature generation and the bitlength of the nonce used to compute the signature, as shown in Figure 4. This happens because given a scalar k and its recoded NAF representation $\hat{k}$, the algorithm execution time is a function of both the NAF length of $\hat{k}$ and its Hamming weight. While the NAF length is a good approximation for the bitlength of k (in fact at most one digit longer), its Hamming weight masks the NAF length linearly so it is not obvious how to correlate these two factors with the precise bitlength of k. Nevertheless, the empirical results (by sampling) shown in Figure 4 clearly demonstrate the latter is directly proportional to the overall algorithm execution time.

This result shares similarity to the one exploited in CVE-2011-1945 [22] (that built the foundation for the recent Minerva[14] and TPM-FAIL [54] attacks), and in fact suggests that CVE applied to not only binary curves using the Montgomery ladder, but prime curves as well. Following their attack methodology, we devise an attack in two phases: (i) The collection phase exploits the timing dependency between the execution time and the bitlength of the nonce used to generate a signature, thus selecting (r,s,m) tuples associated with shorter-than-average nonces; (ii) The recovery phase then combines the partial knowledge inferred from the collection phase to instantiate an HNP instance and solve it through a lattice technique (Section 2.4).

Collection phase. Using our custom TCP time stamp client, across Ethernet we collect 500K traces for a single attack, sorting by the measured latency, and filter the first $t = 128$ items: empirically this is closely related to the selection by a fixed threshold suggested by Figure 4. We prefer the for-

[14] https://minerva.crocs.fi.muni.cz/

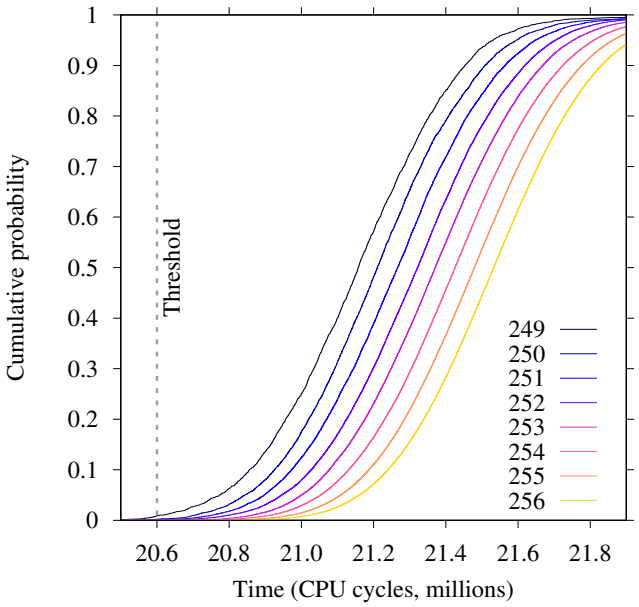

Figure 4: Direct correlation between wall-clock execution time of ECDSA signature generation and the bitlength of the nonce. Plots from left to right correspond to legend keys from top to bottom. Measured on NIST P-256 in OpenSSL on a Pine64-LTS, bypassing all SCA hardening countermeasures via a private key parsing trigger.

mulation where we set the dimension t of the filtered set and the total number of collected signatures, as these numbers are more significant for comparison with other works or directly used in the formalization of the subsequent lattice phase.

Lattice phase. As noted above, the collection phase in this attack selects shorter-than-average nonces, i.e. looking at the nonce k_i as a string of bits with the same bitlength of the generator order n,

$$0 < k_i < 2^{(\lg(n)-\ell_i)} < 2^{(\lg(n)-\ell)} < n/2^{\ell} \equiv n/W < n$$

for some $W = 2^{\ell}$ bound, representing that at least ℓ consecutive MSBs are equal to 0. This is in contrast with the Section 2.4 formalization, which instead implies knowledge of nonce LSBs, so we need to slightly revise some definitions to frame the lattice problem using the same notation. Therefore, we can define $W_i = W = 2^{\ell}$ and, similarly to the formalization in Section 2.4, rearrange (1) as $k_i = \alpha(r_i/s_i) - (-h_i/s_i) \bmod n$ and then redefine $t_i = \lfloor r_i/s_i \rfloor_n$, $\hat{u}_i = \lfloor -h_i/s_i \rfloor_n$ which leads once again to $0 \leq \lfloor \alpha t_i - \hat{u}_i \rfloor_n < n/W_i$, from which the rest of the previous formalization follows unchanged.

Although it used a different lattice description, [22] also dealt with a leak based on nonce MSBs, which led to an interesting property that is valid also for the formalization used in this particular lattice attack. Comparing the definitions of t_i, $\hat{u}_i$, and u_i above with the ones from Section 2.4, we note that in this particular attack no analogue of the a_i term features in the equations composing the lattice problem, from which follows that even if some k_i does not strictly satisfy the bound $k_i < n/W$ there is still a chance that the attack will succeed, leading to a better resilience to *errors* (i.e., entries in the lattice that do not strictly satisfy the bound above) in this lattice formulation. From the attacker perspective, higher W is desirable but requires more leakage from the victim.

Since in this formulation the attacker does not use a per-equation W_i as the distributions are partially overlapping, the question remains how to set W. Underestimating W is technically accurate for approximating zero-MSBs for most of the filtered traces, but forces higher lattice dimensions and slower computation for each job. Using a larger set of training samples, analyzing the ground truth w.r.t. the actual nonce of each sample, we empirically determined that the distribution of nonce bitlengths on the average set filtered by our collection phase is a Gaussian distribution with mean $\overline{\lg(k_i)} = 247.80$ and s.d. 3.81, which suggests $W = 2^8$ ($8 = 256 - 248$) is a better approximation of the bound on most nonces. Given the s.d. magnitude, by trial-and-error we set $W = 2^7$ as a good trade-off for lattice attack execution time vs. success rate.

Combining the better resilience to errors of this particular lattice formulation and the higher amount of information carried by each trace included in the lattice instance by pushing W, we fixed the lattice attack parameters to $d = 60$ and $j = 55\text{K}$ and limit the maximum number of attempted lattice reductions per job to 100 (in practice on our cluster, less than a single minute), as we observed the overwhelming majority of instances returned success within this time frame or not at all.

Attack results. With these parameters, and repeating the attack 100 times, we observed a 91% success rate in our remote timing attack over Ethernet. The median number of jobs needed over all attack instances was 1377 (i.e. $j = 1377$ was sufficient for key recovery in the majority of cases). Those reductions that led to successful key recovery (i.e. 91 in number) had $\overline{\lg(k_i)} = 246.85$ and s.d. 3.13, while the $j = 55\text{K}$ reductions per *each* of the 9 failed overall attack instances had $\overline{\lg(k_i)} = 247.96$ and s.d. 3.87. This difference suggests: (i) the better resilience to errors in this lattice formulation is empirically valid, as given the stated s.d. not all k_i satisfied the bound W; (ii) in our environment, even the failed instances would likely succeed by tweaking lattice parameters (i.e. decreasing W and increasing d) and providing more parallel computation power (i.e. increasing j).

In case of success, the attacker obtains the long-term secret key. On failure she can repeat the collection phase (accumulating more traces and improving the filtering output and the probability of success of another lattice phase) or iteratively tune the lattice parameters (decreasing W and increasing d) to adapt to the features of the specific output of the collection phase, thus improving the lattice attack's success probability.

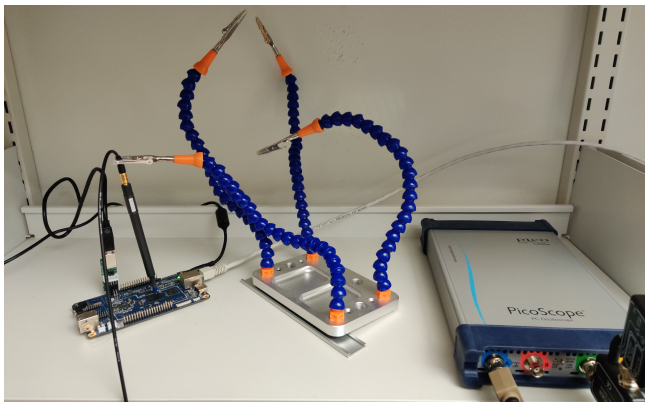

Figure 5: Experiment setup capturing EM traces using Picoscope USB oscilloscope with the Langer EM probe positioned on the Pine64-LTS SoC: a TSP server connected via Ethernet serving requests over HTTP.

4.2 ECDSA: EM Attack

In a much stronger (yet still SCA-classical) attack model assuming physical proximity, we now perform an EM attack on OpenSSL ECDSA. As far as we are aware, we are the first to exploit this code path in the context of OpenSSL and NIST P-256: [38] target the 256-bit Bitcoin curve, and [72] the 256-bit SM2 curve. The reason for this is our Section 3.1 vulnerability allows us to bypass the dedicated `EC_METHOD` instance on this architecture, `EC_GFp_nistz256_method` which is constant time and optimized for AVX and ARMv8 architectures. The wNAF Double and Add operations have a different set of underlying finite field operations—square, multiply, add, sub, inversion—resulting in distinguishable EM signatures.

Experiment setup. To capture the EM traces, we positioned the Langer LF-U 2.5 near field probe head on the SoC where it resulted in the highest signal quality. For digitizing the EM emanations, we used Picoscope 6404*C* USB digital oscilloscope with a bandwidth of 500 MHz and maximum sampling rate of 5 GSps. However, we used a lower sampling rate of 125 MSps as the best compromise between the trace quality and processing overhead. To acquire the traces while ensuring that the entire ECDSA trace was captured, we synchronized the oscilloscope capture with the time stamp request message: initiate the oscilloscope to start acquiring traces, query a time stamp request over HTTP to the server and wait for the server response, and finally stop the trace acquisition. We stored the EM traces along with the DER-encoded server response messages. We parsed the messages to retrieve the hash from the client request and the DER-encoded ECDSA signatures, used to generate metadata for the key recovery phase. Figure 5 shows the setup we used for our attack.

Signal analysis. After capturing the traces, we moved to offline post processing of the EM traces for recovering the partial nonce information. This essentially means identifying the position of the last Add operation. The problem is twofold: finding the end of the point multiplication (end trigger), then identifying the last Add operation therein. We divided the complete signal processing phase mainly into four steps: (i) Remove traces with errors due to acquisition process; (ii) Find the end of the ECDSA point multiplication; (iii) Remove traces encountering interrupts; (iv) Identifying the position of the last Add operation. We started by selecting only those traces which had peak magnitude to the root mean square ratio within an emphatically selected confidence interval, evidently removing traces where the point multiplication operation was not captured or trace was too noisy to start with.

In the next step, we used a specific pattern at the end of ECDSA point multiplication as our soft end trigger. To isolate this trigger pattern from the rest of the signal, we first applied a low pass FIR filter followed by a phase demodulation using the digital Hilbert transform. We further enhanced this pattern while suppressing the rest of the operations by applying root mean square envelope with a window size roughly half its sample size. We created a template by extracting this pattern from 20 random traces and taking their average. We used the Euclidean distance between the trace and template to find the end of point multiplication. We dropped all traces where the Euclidean distance was above an experimental threshold value, i.e. no soft trigger found. The traces also encountered random interrupts due to OS scheduling clearly identifiable as high amplitude peaks. Any traces with an interrupt at the end of point multiplication were also discarded to avoid corrupting the detection of the Add operation.

To recover the position of the last Add operation, we applied a different set of filters on the raw trace, keeping the end of point multiplication as our starting reference. Since the frequency analysis revealed most of the Add operations energy is between 40 MHz and 50 MHz, we applied a band pass FIR filter around this band. Performing a digital Hilbert transform, additional signal smoothing and peak envelope detection, the Add operations were clearly identifiable (Figure 6).

To automatically extract the Add operation, we first used peak extraction. However it was not as reliable since the signals occasionally encountered noisy peaks or in some instances the Add peaks were distorted. We again resorted to the template matching method used in the previous step, i.e. create an Add template and use Euclidean distance for pattern matching. For each peak identified, we also applied the template matching and measured the resulting Euclidean distance against a threshold value. Anything greater than the threshold was considered a false positive peak.

These steps ensured that the error rate stays low, consequently increasing the success rate of the key recovery lattice attack. We estimated the number of Double operations using the total sample length from the middle of the last Add operation to the end of trace as illustrated in Figure 6. To effectively reduce the overlap between the sample length metric of different Double and Add sequences, we applied K-means

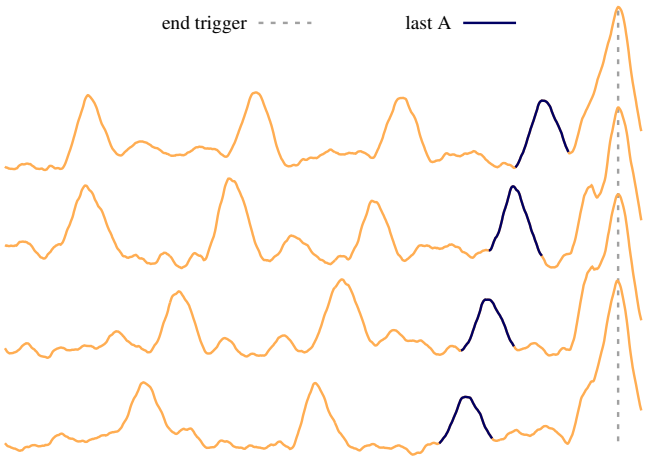

Figure 6: Four different EM traces showing the last Add (A) operations relative to the soft end trigger. The distance in terms of samples between the last Add and trigger gives the number of Double (D) operations. Top to Bottom: Trace ends with an A, AD, ADD, ADDD.

clustering to keep sequences which were close to the cluster mean.

Attack results. We acquired a total of 500 signatures, and after performing the signal processing steps we were left with 422 traces. Additionally, after filtering out signatures categorized as "A" and hence not useful lattice-wise, we were left with $t = 172$ signatures suitable for building lattice problem instances. We chose $d = 120$ as the number of signatures to populate the lattice basis. We then constructed $j = 48$ instances of the lattice attack, randomly selecting d-size subsets from the t signatures for each instance. We then ran these instances in parallel on a 2.10 GHz dual CPU Intel Xeon Silver 4116 (24 cores, 48 threads across 2 CPUs). The first instance to succeed in recovering the private key did so in just over three minutes. Checking the ground truth afterwards, $e = 4$ out of the t signatures were categorized incorrectly, for a suitably small error rate of about 2.3%.

5 Conclusion

In this work, we evaluated how different choices of private key formats and various optional parameters supported by them can influence SCA security. We employed the automated tool Triggerflow to analyze vulnerable code paths in well known cryptographic libraries for various combinations of key formats and optional parameters. The results uncovered several *Certified Side Channels*, circumventing SCA hardened code paths in OpenSSL (ECC with explicit parameters, DSA with MSBLOB and PVK formats, RSA during key validation) and mbedTLS (RSA with missing parameters). To demonstrate the severity of these vulnerabilities, we performed microarchitecture leakage analysis on RSA and DSA and also presented end-to-end key recovery attacks on OpenSSL ECDSA using traditional timing and EM side channels. We publish our data set for the remote timing attack to support Open Science [63].

In the OpenSSL case, Pereida García et al. [62] conclude the fundamental design issue around `BN_FLG_CONSTTIME` is due to its insecure default nature, hypothesizing inverted logic with secure-by-default behavior provides superior assurances. While that would indeed have prevented CVE-2016-2178, our work shows that runtime secure-by-default is still not enough: simply the presence of known SCA-vulnerable code alongside SCA-hardened code poses a security threat. For example, in this light, in our Section 3.1 vulnerability the zero cofactor masquerades as a virtual `BN_FLG_CONSTTIME`, since the exploited code path is oblivious to the flag's value by design.

Mitigations. As part of the responsible disclosure process, we notified OpenSSL and mbedTLS of our findings. At the same time, we made several FOSS contributions to help mitigate these issues in OpenSSL, who assigned CVE-2019-1547 based on our work. For the Section 3.1 vulnerability, we implemented a fix that manually computes the cofactor from the field cardinality and generator order using the Hasse Bound. This works for all standards-compliant curves—named or with explicit parameters. To mitigate the vulnerabilities in Section 3.2 and Section 3.3, we submitted simple patches that set the `BN_FLG_CONSTTIME` correctly, steering the computations to existing SCA-hardened code. Moreover, we replaced the variable-time GCD function in OpenSSL by a constant-time implementation[15] based on the work by Bernstein and Yang [16]. To further reduce the SCA attack surface, we implemented changes[16] in the way OpenSSL creates EC key abstractions when the associated curve is defined by explicit parameters. The explicit parameters are matched against the internal table of known curves, switching to an internal named curve representation for matches, ultimately enabling the use of specialized implementations where available. Finally, we integrated the new Triggerflow unit tests (Figure 1). Applying all these fixes across non-EOL OpenSSL branches as well as the development branch, no Triggerflow POIs are subsequently triggered, indicated the patches are effective.

Astute readers may notice the above fixes do not remove the vulnerable functions in question. In general, indeed this is one option, but such a strategy requires analysis on a case-by-case basis. These are real-world products that come with real-world performance constraints. For example, an SCA-secure modular exponentiation function that protects both the length and values of the exponent can likely meet the performance requirements for e.g. DSA signing, but not RSA verification with a short, low-weight, and public exponent. This is the main reason why libraries often feature multiple versions of the same functionality with different security characteristics.

[15] https://github.com/openssl/openssl/pull/10122
[16] https://github.com/openssl/openssl/pull/9808

Future work. In Section 4.1, discussing the lattice formulation for our attack, we highlight an increased resilience to lattic errors compared to similar previous works. We note here that an analysis of error resilience of different lattice constructions and of strategies to increase the overall success rate of lattice attacks in the presence of errors in collected traces would constitute a valuable future contribution to this area of research.

Our vulnerabilities in Section 3 cover only a very small subset of possible inputs, combinations, architectures, and settings. Another interesting research question is how to extend test coverage in a reasonable way, even considering other libraries.

Acknowledgments. We thank Tampere Center for Scientific Computing (TCSC) for generously granting us access to computing cluster resources. The second author was supported in part by the Tuula and Yrjö Neuvo Fund through the Industrial Research Fund at Tampere University of Technology. The third author was supported in part by a Nokia Scholarship from the Nokia Foundation. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 804476).

References

[1] Security requirements for cryptographic modules. FIPS PUB 140-2, National Institute of Standards and Technology, May 2001. URL https://doi.org/10.6028/NIST.FIPS.140-2.

[2] SEC 1: Elliptic Curve Cryptography. SEC 1, Standards for Efficient Cryptography Group, May 2009. URL http://www.secg.org/sec1-v2.pdf.

[3] Rodrigo Abarzúa, Claudio Valencia Cordero, and Julio López Hernandez. Survey for performance & security problems of passive side-channel attacks countermeasures in ECC. *IACR Cryptology ePrint Archive*, 2019(10), 2019. URL https://eprint.iacr.org/2019/010.

[4] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. In *IMACC*, volume 4887 of *LNCS*, pages 185–203. Springer, 2007. URL https://doi.org/10.1007/978-3-540-77272-9_12.

[5] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The EM side-channel(s). In *CHES*, volume 2523 of *LNCS*, pages 29–45. Springer, 2002. URL https://doi.org/10.1007/3-540-36400-5_4.

[6] Alejandro Cabrera Aldaya and Billy Bob Brumley. When one vulnerable primitive turns viral: Novel single-trace attacks on ECDSA and RSA. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(2):196–221, 2020. URL https://doi.org/10.13154/tches.v2020.i2.196-221.

[7] Alejandro Cabrera Aldaya, Alejandro J. Cabrera Sarmiento, and Santiago Sánchez-Solano. SPA vulnerabilities of the binary extended Euclidean algorithm. *J. Cryptographic Engineering*, 7(4):273–285, 2017. URL https://doi.org/10.1007/s13389-016-0135-4.

[8] Alejandro Cabrera Aldaya, Raudel Cuiman Márquez, Alejandro J. Cabrera Sarmiento, and Santiago Sánchez-Solano. Side-channel analysis of the modular inversion step in the RSA key generation algorithm. *I. J. Circuit Theory and Applications*, 45 (2):199–213, 2017. URL https://doi.org/10.1002/cta.2283.

[9] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *IEEE S&P*, pages 870–887. IEEE, 2019. URL https://doi.org/10.1109/SP.2019.00066.

[10] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. Cache-timing attacks on RSA key generation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(4):213–242, 2019. URL https://doi.org/10.13154/tches.v2019.i4.213-242.

[11] Thomas Allan, Billy Bob Brumley, Katrina E. Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *ACSAC*, pages 422–435. ACM, 2016. URL http://doi.acm.org/10.1145/2991079.2991084.

[12] Aurélie Bauer, Éliane Jaulmes, Emmanuel Prouff, Jean-René Reinhard, and Justine Wild. Horizontal collision correlation attack on elliptic curves – extended version. *Cryptography and Communications*, 7(1):91–119, 2015. URL https://doi.org/10.1007/s12095-014-0111-8.

[13] Pierre Belgarric, Pierre-Alain Fouque, Gilles Macario-Rat, and Mehdi Tibouchi. Side-channel analysis of Weierstrass and Koblitz curve ECDSA on Android smartphones. In *CT-RSA*, volume 9610 of *LNCS*, pages 236–252. Springer, 2016. URL https://doi.org/10.1007/978-3-319-29485-8_14.

[14] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "Ooh Aah... Just a Little Bit": A small amount of side channel can go a long way. In *CHES*, volume 8731 of *LNCS*, pages 75–92. Springer, 2014. URL https://doi.org/10.1007/978-3-662-44709-3_5.

[15] Daniel J. Bernstein. Cache-timing attacks on AES, 2005. URL http://cr.yp.to/papers.html#cachetiming.

[16] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(3):340–398, 2019. URL https://doi.org/10.13154/tches.v2019.i3.340-398.

[17] Daniel J. Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding right into disaster: Left-to-right sliding windows leak. In *CHES*, volume 10529 of *LNCS*, pages 555–576. Springer, 2017. URL https://doi.org/10.1007/978-3-319-66787-4_27.

[18] Dan Boneh and Ramarathnam Venkatesan. Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes. In *CRYPTO*, volume 1109 of *LNCS*, pages

129–142. Springer, 1996. URL https://doi.org/10.1007/3-540-68697-5_11.

[19] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *CHES*, volume 3156 of *LNCS*, pages 16–29. Springer, 2004. URL https://doi.org/10.1007/978-3-540-28632-5_2.

[20] Billy Bob Brumley. Faster software for fast endomorphisms. In *COSADE*, volume 9064 of *LNCS*, pages 127–140. Springer, 2015. URL https://doi.org/10.1007/978-3-319-21476-4_9.

[21] Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. In *ASIACRYPT*, volume 5912 of *LNCS*, pages 667–684. Springer, 2009. URL https://doi.org/10.1007/978-3-642-10366-7_39.

[22] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *ESORICS*, volume 6879 of *LNCS*, pages 355–371. Springer, 2011. URL https://doi.org/10.1007/978-3-642-23822-2_20.

[23] David Brumley and Dan Boneh. Remote timing attacks are practical. In *USENIX Sec.* USENIX Association, 2003. URL https://www.usenix.org/conference/12th-usenix-security-symposium/remote-timing-attacks-are-practical.

[24] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant CPUs. In *ACM CCS*, pages 769–784. ACM, 2019. URL https://doi.org/10.1145/3319535.3363219.

[25] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In *CHES*, volume 2523 of *LNCS*, pages 13–28. Springer, 2002. URL https://doi.org/10.1007/3-540-36400-5_3.

[26] Christophe Clavier, Benoit Feix, Georges Gagnerot, Mylène Roussellet, and Vincent Verneuil. Horizontal correlation analysis on exponentiation. In *ICICS*, volume 6476 of *LNCS*, pages 46–61. Springer, 2010. URL https://doi.org/10.1007/978-3-642-17650-0_5.

[27] Jean-Luc Danger, Sylvain Guilley, Philippe Hoogvorst, Cédric Murdica, and David Naccache. A synthesis of side-channel attacks on elliptic curve cryptography in smart-cards. *J. Cryptographic Engineering*, 3(4):241–265, 2013. URL https://doi.org/10.1007/s13389-013-0062-6.

[28] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A tool for the static analysis of cache side channels. *ACM Trans. Inf. Syst. Secur.*, 18(1):4:1–4:32, 2015. URL https://doi.org/10.1145/2756550.

[29] Margaux Dugardin, Louiza Papachristodoulou, Zakaria Najm, Lejla Batina, Jean-Luc Danger, and Sylvain Guilley. Dismantling real-world ECC with horizontal and vertical template attacks. In *COSADE*, volume 9689 of *LNCS*, pages 88–108. Springer, 2016. URL https://doi.org/10.1007/978-3-319-43283-0_6.

[30] Shuqin Fan, Wenbo Wang, and Qingfeng Cheng. Attacking OpenSSL implementation of ECDSA with a few signatures. In *ACM CCS*, pages 1505–1515. ACM, 2016. URL https://doi.org/10.1145/2976749.2978400.

[31] Jeffrey Friedman. Tempest: A signal problem. *NSA Cryptologic Spectrum*, 2(3):26–30, 1972. URL https://www.nsa.gov/Portals/70/documents/news-features/declassified-documents/cryptologic-spectrum/tempest.pdf.

[32] Nicolas Gama, Phong Q. Nguyen, and Oded Regev. Lattice enumeration using extreme pruning. In *EUROCRYPT*, volume 6110 of *LNCS*, pages 257–278. Springer, 2010. URL https://doi.org/10.1007/978-3-642-13190-5_13.

[33] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptographic Engineering*, 8(1):1–27, 2018. URL https://doi.org/10.1007/s13389-016-0141-6.

[34] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *CRYPTO*, volume 8616 of *LNCS*, pages 444–461. Springer, 2014. URL https://doi.org/10.1007/978-3-662-44371-2_25.

[35] Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. Stealing keys from PCs using a radio: Cheap electromagnetic attacks on windowed exponentiation. In *CHES*, volume 9293 of *LNCS*, pages 207–228. Springer, 2015. URL https://doi.org/10.1007/978-3-662-48324-4_11.

[36] Daniel Genkin, Itamar Pipman, and Eran Tromer. Get your hands off my laptop: physical side-channel key-extraction attacks on PCs - extended version. *J. Cryptographic Engineering*, 5(2):95–112, 2015. URL https://doi.org/10.1007/s13389-015-0100-7.

[37] Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. ECDH key-extraction via low-bandwidth electromagnetic attacks on PCs. In *CT-RSA*, volume 9610 of *LNCS*, pages 219–235. Springer, 2016. URL https://doi.org/10.1007/978-3-319-29485-8_13.

[38] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. ECDSA key extraction from mobile devices via nonintrusive physical side channels. In *ACM CCS*, pages 1626–1638. ACM, 2016. URL http://doi.acm.org/10.1145/2976749.2978353.

[39] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. Public-key cryptosystems from lattice reduction problems. In *CRYPTO*, volume 1294 of *LNCS*, pages 112–131. Springer, 1997. URL https://doi.org/10.1007/BFb0052231.

[40] Iaroslav Gridin, Cesar Pereida García, Nicola Tuveri, and Billy Bob Brumley. Triggerflow: Regression testing by advanced execution path inspection. In *DIMVA*, volume 11543 of *LNCS*, pages 330–350. Springer, 2019. URL https://doi.org/10.1007/978-3-030-22038-9_16.

[41] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Sec.*, pages

897–912. USENIX Association, 2015. URL https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss.

[42] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In *DIMVA*, volume 9721 of *LNCS*, pages 279–299. Springer, 2016. URL https://doi.org/10.1007/978-3-319-40667-1_14.

[43] Shay Gueron and Vlad Krasnov. Fast prime field elliptic-curve cryptography with 256-bit primes. *J. Cryptographic Engineering*, 5(2):141–151, 2015. URL https://doi.org/10.1007/s13389-014-0090-x.

[44] M. Jason Hinek. *Cryptanalysis of RSA and its variants*. Chapman & Hall/CRC Cryptography and Network Security. CRC Press, 2010. ISBN 978-1-4200-7518-2. URL https://doi.org/10.1201/9781420075199.

[45] Simon Josefsson and Sean Leonard. Textual encodings of PKIX, PKCS, and CMS structures. RFC 7468, RFC Editor, April 2015. URL https://datatracker.ietf.org/doc/rfc7468/.

[46] Emilia Käsper. Fast elliptic curve cryptography in OpenSSL. In *Financial Cryptography Workshops*, volume 7126 of *LNCS*, pages 27–39. Springer, 2011. URL https://doi.org/10.1007/978-3-642-29889-9_4.

[47] Vlastimil Klíma, Ondrej Pokorný, and Tomás Rosa. Attacking RSA-based sessions in SSL/TLS. In *CHES*, volume 2779 of *LNCS*, pages 426–440. Springer, 2003. URL https://doi.org/10.1007/978-3-540-45238-6_33.

[48] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996. URL https://doi.org/10.1007/3-540-68697-5_9.

[49] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999. URL https://doi.org/10.1007/3-540-48405-1_25.

[50] Jake Longo, Elke De Mulder, Dan Page, and Michael Tunstall. SoC it to EM: ElectroMagnetic side-channel attacks on a complex System-on-Chip. In *CHES*, volume 9293 of *LNCS*, pages 620–640. Springer, 2015. URL https://doi.org/10.1007/978-3-662-48324-4_31.

[51] Accredited Standards Committee X9, editor. *ANSI X9.62-2005: Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*. ANSI American National Standards Institute, 2005.

[52] Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. Examining smart-card security under the threat of power analysis attacks. *IEEE Trans. Computers*, 51(5):541–552, 2002. URL https://doi.org/10.1109/TC.2002.1004593.

[53] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel, and Erik Tews. Revisiting SSL/TLS implementations: New Bleichenbacher side channels and attacks. In *USENIX Sec.*, pages 733–748. USENIX Association, 2014. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/meyer.

[54] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL: TPM meets timing and lattice attacks. In *USENIX Sec.* USENIX Association, 2020. URL https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi.

[55] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. PKCS #1: RSA cryptography specifications version 2.2. RFC 8017, RFC Editor, November 2016. URL https://datatracker.ietf.org/doc/rfc8017/.

[56] Samuel Neves and Mehdi Tibouchi. Degenerate curve attacks: extending invalid curve attacks to Edwards curves and other models. *IET Information Security*, 12(3):217–225, 2018. URL https://doi.org/10.1049/iet-ifs.2017.0075.

[57] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the Digital Signature Algorithm with partially known nonces. *J. Cryptology*, 15(3):151–176, 2002. URL https://doi.org/10.1007/s00145-002-0021-3.

[58] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the Elliptic Curve Digital Signature Algorithm with partially known nonces. *Des. Codes Cryptogr.*, 30(2):201–217, 2003. URL https://doi.org/10.1023/A:1025436905711.

[59] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006. URL https://doi.org/10.1007/11605805_1.

[60] Colin Percival. Cache missing for fun and profit. In *BSDCan*, 2005. URL http://www.daemonology.net/papers/cachemissing.pdf.

[61] Cesar Pereida García and Billy Bob Brumley. Constant-time callees with variable-time callers. In *USENIX Sec.*, pages 83–98. USENIX Association, 2017. ISBN 978-1-931971-40-9. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/garcia.

[62] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. "Make sure DSA signing exponentiations really are constant-time". In *ACM CCS*, pages 1639–1650. ACM, 2016. URL http://doi.acm.org/10.1145/2976749.2978420.

[63] Cesar Pereida García, Sohaib ul Hassan, Nicola Tuveri, Iaroslav Gridin, Alejandro Cabrera Aldaya, and Billy Bob Brumley. *CVE-2019-1547: research data and tooling*. Zenodo, April 2020. URL https://doi.org/10.5281/zenodo.3736311.

[64] Vladimir Popov, Serguei Leontiev, and Igor Kurepkin. Additional cryptographic algorithms for use with GOST 28147-89, GOST R 34.10-94, GOST R 34.10-2001, and GOST R 34.11-94 algorithms. RFC 4357, RFC Editor, January 2006. URL https://datatracker.ietf.org/doc/rfc4357/.

[65] Thomas Popp, Stefan Mangard, and Elisabeth Oswald. Power analysis attacks and countermeasures. *IEEE Design & Test of Computers*, 24(6):535–543, 2007. URL https://doi.org/10.1109/MDT.2007.200.

[66] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): measures and counter-measures for smart cards. In *E-smart*, volume 2140 of *LNCS*, pages 200–210. Springer, 2001. URL https://doi.org/10.1007/3-540-45418-7_17.

[67] Michael O. Rabin. Probabilistic algorithm for testing primality. *J. Number Theory*, 12(1):128–138, 1980. ISSN 0022-314X. URL https://doi.org/10.1016/0022-314x(80)90084-0.

[68] Niels Samwel, Lejla Batina, Guido Bertoni, Joan Daemen, and Ruggero Susella. Breaking Ed25519 in WolfSSL. In *CT-RSA*, volume 10808 of *LNCS*, pages 1–20. Springer, 2018. URL https://doi.org/10.1007/978-3-319-76953-0_1.

[69] ITU-T Telecommunication standardization sector of ITU, editor. *ITU-T X.690 Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*. ITU International Telecommunication Union, August 2015. URL http://handle.itu.int/11.1002/1000/12483.

[70] Akira Takahashi and Mehdi Tibouchi. Degenerate fault attacks on elliptic curve parameters in OpenSSL. In *EuroS&P*, pages 371–386. IEEE, 2019. URL https://doi.org/10.1109/EuroSP.2019.00035.

[71] Michael Tunstall. Smart card security. In *Smart Cards, Tokens, Security and Applications*, pages 217–251. Springer, second edition, 2017. URL https://doi.org/10.1007/978-3-319-50500-8_9.

[72] Nicola Tuveri, Sohaib ul Hassan, Cesar Pereida García, and Billy Bob Brumley. Side-channel analysis of SM2: A late-stage featurization case study. In *ACSAC*, pages 147–160. ACM, 2018. URL https://doi.org/10.1145/3274694.3274725.

[73] Joop van de Pol, Nigel P. Smart, and Yuval Yarom. Just a little bit more. In *CT-RSA*, volume 9048 of *LNCS*, pages 3–21. Springer, 2015. URL https://doi.org/10.1007/978-3-319-16715-2_1.

[74] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: rogue in-flight data load. In *IEEE S&P*, pages 88–105. IEEE, 2019. URL https://doi.org/10.1109/SP.2019.00087.

[75] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. CacheD: Identifying cache-based timing channels in production software. In *USENIX Sec.*, pages 235–252. USENIX Association, 2017. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-shuai.

[76] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. Single trace attack against RSA key generation in Intel SGX SSL. In *AsiaCCS*, pages 575–586. ACM, 2018. URL http://doi.acm.org/10.1145/3196494.3196524.

[77] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. DATA - differential address trace analysis: Finding address-based side-channels in binaries. In *USENIX Sec.*, pages 603–620. USENIX Association, 2018. URL https://www.usenix.org/conference/usenixsecurity18/presentation/weiser.

[78] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Sec.*, pages 719–732. USENIX Association, 2014. ISBN 978-1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[79] Robert Zuccherato, Patrick Cain, Carlisle Adams, and Denis Pinkas. Internet X.509 public key infrastructure time-stamp protocol (TSP). RFC 3161, RFC Editor, August 2001. URL https://datatracker.ietf.org/doc/rfc3161/.

A mbedTLS vulnerable RSA keys

Table 2: RSA keys that follow leaking mbedTLS code paths. Missing parameters are marked with •. Note, the first row belongs to a key with all included parameters indicating that it leaks through CRT computation.

Group	N	e	p	q	d	d_p	d_q	i_q
CRT								
						•	•	•
						•	•	
						•		•
						•		
							•	•
							•	
								•
	•							
	•					•	•	•
	•					•	•	
	•					•		•
	•					•		
	•						•	•
	•						•	
	•							•
CRT & d					•			
					•	•	•	•
					•	•	•	
					•	•		•
					•	•		
					•		•	•
					•		•	
					•			•
	•				•			
	•				•	•	•	•
	•				•	•	•	
	•				•	•		•
	•				•	•		
	•				•		•	•
	•				•		•	
	•				•			•