Size, Speed, and Security: An Ed25519 Case Study

* Cesar Pereida García[1], Sampo Sovio[2]

[1] Tampere University, Tampere, Finland
cesar.pereidagarcia@tuni.fi
[2] Huawei Technologies Oy, Helsinki, Finland
sampo.sovio@huawei.com

Abstract. Ed25519 has significant performance benefits compared to ECDSA using Weierstrass curves such as NIST P-256, therefore it is considered a good digital signature algorithm, specially for low performance IoT devices. However, such devices often have very limited resources and thus, implementations for these devices need to be as small and as performant as possible while being secure. In this paper we describe a scenario in which an obvious strategy to aggressively optimize an Ed25519 implementation for code size leads to a small memory footprint that is functionally correct but vulnerable to side-channel attacks. This strategy serves as an example of aggressive optimizations that might be considered by cryptography engineers, developers, and practitioners unfamiliar with the power of Side-Channel Analysis (SCA). As a solution to the flawed implementation example, we use a computer-aided cryptography tool generating formally verified finite field arithmetic to generate two secure Ed25519 implementations fulfilling different size requirements. After benchmarking and comparing these implementations to other widely used implementations our results show that computer-aided cryptography is capable of generating competitive code in terms of security, speed, and size.

Keywords: applied cryptography; public key cryptography; EdDSA; Ed25519; side-channel analysis; computer-aided cryptography

1 Introduction

The growing number of IoT devices around us is ever increasing, and thus the need to secure these devices and their communication is of utmost importance. Moreover, due to their nature, the attack surface of IoT devices is higher compared to commodity PCs and servers, as attackers are able to get physical access to them, thus exposing them to both physical and remote attacks, resulting in new threat scenarios. Cryptography engineers face multiple challenges when securing these devices as cryptography implementations must be not only secure

* This research was done while the author was an intern at Huawei Technologies Oy.

but also competitive in terms of speed and size since these are constrained devices with limited power, memory, and processing resources.

EdDSA, and more specifically Ed25519 [7], is a popular algorithm choice for digital signatures in the IoT world as it is small, fast, and it does not require fresh randomness per signature thus reducing the risk of using a faulty random number generator (RNG). EdDSA instead computes a deterministic nonce as a function of the hashed message and the private key, and in general, it provides a more robust security against several attacks when compared to ECDSA. During the development of Ed25519, choices were made to decrease the chances of implementation flaws and unintentional information leakage. However, these secure choices need to be clearly understood by cryptography engineers, as a small deviation from the original specification can lead back to insecure implementations.

Cryptography engineers must follow general recommendations and coding best practices when implementing algorithms that receive confidential information as input values. These recommendations are mostly to protect against Side-Channel Analysis (SCA), and failing to follow any of the best practices can have devastating effects on the practical security of any implementation. Some of these best practices include: (i) using algorithms that execute in constant-time, i.e., the runtime of the algorithm is independent from the input secret value; (ii) avoiding branching based on secret values; (iii) avoiding table-lookups indexed by secret values; (iv) avoiding looping through a piece of code with a bound dictated by a secret value. Generally speaking, cryptography engineers must be aware that any line of code they write dealing with secret values, must not leak any information through either execution time, EM emanations, power consumption, microarchitecture components, temperature, or any other so-called side-channel, thus all of them must be considered for IoT security.

In this work we present a case study on how aggressive optimizations aiming for a small memory footprint can lead to SCA vulnerabilities on an otherwise secure Ed25519 implementation. We describe the rationale for the aggressive optimizations from the point of view of a cryptography engineer without SCA expertise, trying to meet the requirements, and then we briefly analyse why the approach is insecure against an adversary with SCA expertise. As a countermeasure, we replace the flawed implementation with a secure one generated with the help of `ECCKiila` [5], a computer-aided cryptography tool. We compare the performance of the computer-aided Ed25519 against other well established implementations on Intel and ARM architectures.

In summary, Section 2 gives an overview of background information and related work. Section 3 describes the example of a flawed implementation due to aggressive optimizations and its implications, we give a brief side-channel analysis. Section 4 describes two implementations generated with the help of `ECCKiila` and provides a performance comparison against other well established Ed25519 implementations. We conclude in Section 5.

2 Background

2.1 EdDSA

The Edwards-curve Digital Signature Algorithm (EdDSA) is an elliptic curve variant of the Schnorr signature system [27], thus it is a deterministic digital signature scheme constructed over twisted Edwards curves. Despite being a relatively new cryptographic primitive, EdDSA has gained traction over the last five years on both, the research community, and industry, due to being fast, secure, and hard to implement wrong—at least compared to ECDSA. Notably, EdDSA does not require fresh randomness for each signature generated—therefore it is more resilient against side-channel analysis (SCA)—and no special cases for the point at infinity need to be handled due to exception free formulas for point addition. EdDSA is generally defined by eleven parameters. An odd prime p defining the Galois field $GF(p)$, two elements $a, d \in GF(p)$ defining the twisted Edwards curve.

$$E : ax^2 + y^2 = 1 + dx^2y^2 \tag{1}$$

An element $B \in E$ different from the neutral element. An integer c and an odd prime ℓ such that $\#E = 2^c\ell$. An integer b defining the size of the EdDSA public keys and EdDSA signatures in bits, an integer n defining the scalar size, an encoding of the elements in $GF(p)$, a hash function H and an optional "prehash" function PH. Choosing parameter is outside of the scope of our work, but we refer the reader to Bernstein et al. [7], RFC 8032 [19] and FIPS 186-5 [1].

Generally speaking, EdDSA is composed by three algorithms, namely, key generation, signature generation, and signature verification. Each of these algorithms is composed by several specific algorithms, which when converted to code, dictate the security properties of the whole EdDSA scheme.

2.2 Ed25519

Originally described by Bernstein et al. [7], Ed25519 is EdDSA instantiated with a twisted Edwards curve that is birationally equivalent to Curve25519 [6]. Ed25519 is allegedly the most widely used instance of EdDSA, and it is instantiated with the parameters present in Table 1.

Key Generation. Given a random and uniformly chosen private key k, the user hashes it using the chosen hash function H such that $H(k) = (h_0, h_1, ..., h_{2s-1}) = (a, b)$ where a is a private scalar value, b is an auxiliary key, and then computes the public key $A = [a]B$.

Signature Generation. Given the private scalar a, the auxiliary key b, and a hash function H, the signature (R, S) on the message M is created by

$$\begin{aligned} r &= H(b, M) & R &= [r]B \\ h &= H(R, A, M) & S &= (r + ha) \bmod \ell \end{aligned} \tag{2}$$

Description	Symbol	Value
Power for $GF(p)$	p	$2^{255} - 19$
Element in $GF(p)$	a	-1
Non-square element in $GF(p)$	d	$-121665/121666$
Base point	B	(x, y) from [7]
Order of base point	ℓ	$2^{252} + 27742317777372353535851937790883648493$
Key length	s	256
$log_2(cofactor)$	c	3
Scalar size	n	254
Hash function	H	SHA-512
Prehash function	PH	None

Table 1. Ed25519 domain parameters.

Signature Verification. Given the base point B, the public key A, and the signature tuple (R, S), on the message M, the EdDSA signature is valid if $h = H(R, A, M)$ satisfies the equation

$$8SB = 8R + 8hA \tag{3}$$

Security. The mathematical security of EdDSA (and Ed25519) is similar to that of other ECC primitives, namely, it relies on the hardness of the *Elliptic Curve Discrete Logarithm Problem (ECDLP)*, i.e., given a known base B and an elliptic curve element $[r]B$, it is infeasible to compute the integer r. Additionally, during EdDSA construction several choices were made to avoid common flaws and vulnerabilities affecting the well established ECDSA. Some of these choices include: (i) the usage of a deterministic ephemeral nonce instead of a random nonce per signature, avoiding potential issues with a faulty RNG; (ii) the usage of twisted Edwards curves providing complete addition law and thus avoiding special cases that can be exploited; (iii) provide an open and freely available reference implementation that is secure in the mathematical model and also against SCA.

2.3 Ed25519 Implementations

In 2011, together with the original mathematical and technical description of the new Ed25519, Bernstein et al. [7] released multiple implementations of their new digital signature algorithm to the public domain through the eBACS project [8]. The objective was to promote the widespread adoption of the new primitive by providing implementations suitable for different architectures and systems with different requirements. The original release included a portable, slow but secure implementation written in *C* language named `ref`, a portable and faster implementation with competitive performance also written in *C* language named `ref10`, and two additional `x86_64`-specific, fast and highly optimized implementations written in *assembly* language named `amd64-64-24k` and `amd64-51-30k` using radix 2^{64} and 2^{51} for field element representation, respectively. Shortly

after, the so-called `donna`[3] implementation was released which included high performance, portable 32-bit and 64-bit implementations for Ed25519.

Built on top of the reference implementation, Bernstein et al. [9] released NaCl as an easy-to-use high-speed software library offering several state-of-the-art implementations for several types of cryptographic primitives such as encryption, decryption, and signing. Similarly, libsodium[4] was born as a fork of NaCl to expand on the original API while supporting a variety of compilers and operating systems, becoming the de-facto library for Ed25519 at that time.

In 2015, BoringSSL added support[5] for Ed25519 to its codebase. Different to previous implementations, instead of using custom finite field arithmetic code, BoringSSL adopted formally verified finite field arithmetic generated with `fiat-crypto`[6] [15].

In 2017 `monocypher`[7] was released, including its own implementation of Ed25519 influenced by the `ref10` implementation. This specific implementation targets devices with limited resources, and it offers a compact and portable implementation compatible with `libsodium`.

In 2018 Tuveri and Brumley [28] added unofficial support for Ed25519 in OpenSSL through their `libsuola` ENGINE, by leveraging Ed25519 computations to the libsodium library. Official support was later added to the code base during the same year with the release of the newer version OpenSSL 1.1.1[8].

At the time of writing, most of the widely used general-purpose TLS and cryptography libraries support Ed25519. One notable exception is the `mbedTLS` library which is currently under development with an unspecified release date[9]. It is worth noting that despite the variety of implementations, most of them use code from the original reference implementations. This ultimately confirms that the original goal of widespread adoption of Ed25519 was achieved by providing robust reference implementations for other projects[10].

2.4 Related Work

Although EdDSA is a secure signature algorithm, it is susceptible to attacks derived from implementation flaws and SCA. Despite making secure design choices to minimize the probability of bad implementations, EdDSA still requires specialized knowledge and attention to detail during implementation, to avoid leaking confidential information that renders the primitive insecure.

Samwel et al. [26] demonstrate that failing to protect the auxiliary key b for any given signature can potentially lead to full key recovery, allowing an attacker to forge signatures. More specifically, the authors apply Differential

[3] https://github.com/floodyberry/ed25519-donna
[4] https://github.com/jedisct1/libsodium
[5] https://boringssl.googlesource.com/boringssl/+/4fb0dc4b031df7c9ac9d91fc34536e4e08b35d6a
[6] https://github.com/mit-plv/fiat-crypto
[7] https://monocypher.org/
[8] https://www.openssl.org/blog/blog/2018/09/11/release111/
[9] https://github.com/ARMmbed/mbedtls/pull/3245
[10] https://ianix.com/pub/ed25519-deployment.html

Power Analysis (DPA) on the underline SHA-512 function of the WolfSSL library to recover the auxiliary key b during the computation of the ephemeral nonce r, which allows them to ultimately forge signatures for any message of their choosing.

Romailler and Pelissier [24] propose the first differential fault attack (DFA) on Ed25519 against an 8-bit Arduino nano device. The authors introduce a fault to the output of the hash function, however this value is not public, thus they need to bruteforce the value in order to exploit it to forge signatures.

Following the same attack principle, Samwel and Batina [25] introduce a fault during the computation of R, resulting in R' and therefore in a faulty hash computation h'. Using a single pair of correct and faulty signatures, the authors are able to recover the private scalar a solving a simple system of equations and consequentially, forge signatures for any given message.

Similarly, Ambrose et al. [3] study the effects of DFA on deterministic digital signature schemes, including EdDSA. In their work, the authors propose several attacks against EdDSA using DFA and describe the place and the type of fault that is needed allowing them to recover enough confidential information to forge signatures. Moreover, the authors discuss practical countermeasures against DFA, and possible changes to EdDSA to protect against this type of attacks.

On the software side, Poddebniak et al. [23] demonstrate a practical cross-VM fault attack against EdDSA by using the Rowhammer technique from a malicious VM to introduce faults to the target VM running `Minisign`. The authors successfully recover the private scalar a that allows them to forge signatures.

Exploiting the hardware translation lookaside buffers (TLBs), Gras et al. [17] recover the full keys for an insecure Ed25519 implementation on `libgcrypt v1.6.3`. Finally, Gras et al. [16] show a system that synthesizes new (port) contention-based side-channels. The authors demonstrate the working system on both, secure and insecure implementations of Ed25519 on `libgcrypt`.

3 When Optimization Goes Wrong

In this section we focus on how a relatively small change to the reference implementation trying to reduce the memory footprint, leads to a functionally correct but insecure implementation of Ed25519. The implementation that we describe in here is a custom implementation, thus this is not a real implementation affecting any system nor an open source cryptography library. However, we believe that this flawed implementation is a representative of aggressive optimizations that might be considered by cryptography engineers and practitioners in order to achieve specific memory requirements.

Implementation description. In the original work, Bernstein et al. [7] describe two different algorithms for scalar multiplication to be used during Ed25519: a fixed-point scalar multiplication for key and signature generation, and a double-scalar multiplication for signature verification. Additionally, each scalar multiplication algorithm requires a recoded scalar value in a suitable form for the chosen

scalar multiplication algorithm, thus this involves additional recoding algorithms which also affect the overall implementation size.

For fixed-point scalar multiplication, the original implementation follows a standard technique first discussed by Pippenger [22]. The technique consists of computing the scalar multiplication as a sum of precomputed values with the addition of supporting negative coefficients. This algorithm by itself does not prevent nor protect against SCA, but instead allows to load all the precomputed values into memory and then compute the correct value by using arithmetic operations that do not branch or otherwise reveal the secret value through the index accessed. After an analysis, the authors decide that a balance on performance versus memory size is reached by storing 256 curve points consuming a total of 30 kilobytes of RAM. In fact, the authors mention that is possible to reduce the table size by half at the expense of 8 additional elliptic curve doubles. While this change already potentially reduces the size of the table by half, it might not be enough for a constrained device, and more aggressive optimizations for code size might be considered.

For the double-scalar algorithm the original implementation uses standard techniques similar to the windowed Non-Adjacent Form (wNAF) scalar multiplication [21] which allows them to compute the result for both scalar values in a single call, instead of performing a more costly fixed-point and variable-point scalar multiplications. This algorithm achieves a fast result at a low memory cost, as it does not require a precomputed table. However the algorithm execution is highly dependent on its inputs, thus it is specially suitable during signature verification where all the input values are public.

Considering these two algorithms to achieve the same result, namely a scalar multiplication, an appealing approach to reduce code size is not only to use the algorithm with the smallest memory footprint but also the most flexible algorithm that can be adapted for multiple use cases. Therefore, the double-scalar multiplication algorithm is a good candidate that can be adapted for usage in key generation, signature generation, and signature verification.

Our custom implementation continues with this idea, by simply adding conditional branches at the top of the double-scalar multiplication algorithm, we are able to cover all use cases for Ed25519: if the input value to the function containing the variable-point is empty then the algorithm is equivalent to a fixed-point scalar multiplication; if the input value to the function containing the fixed-point is empty then the algorithm is equivalent to a variable-point scalar multiplication; otherwise it is the standard double-scalar multiplication. By following this approach the implementation saves more than 30 KB of code since it does not require a 30 KB precomputed table, and it only uses a single algorithm for fixed-point, variable-point, and double-point scalar multiplication—and consequently only one algorithm for scalar recoding.

SCA Analysis. We now give a brief analysis from a SCA perspective to demonstrate the vulnerabilities enabled by the previous modifications to the original implementation.

Recall that prior to the scalar multiplication computation, the integer representing the scalar value must be recoded into a different form. The algorithm used for recoding, as any other algorithm dealing with secret information, must behave in a constant-time fashion, that is, no correlation must be observable between the input value and the execution of the algorithm in order to prevent SCA leakage. The reference implementation achieves this by cleverly using arithmetic and bitwise operations to recode the scalar as digits in the range $[-8, 7]$. These arithmetic and bitwise operations do not branch nor loop based on the scalar value, thus they are secure against SCA. However, one problem arises when using the double-scalar multiplication as a fixed-point scalar multiplication, as the recoding algorithm used for the latter is different than in the former. The recoding algorithm for double-scalar multiplication is based on the work by Avanzi [4], a left-to-right recoding variant commonly used during wNAF scalar multiplication. This variant in particular branches out according to individual bits of the scalar value, therefore its usage is only suitable when the scalar is a public value and does not need SCA protection, however this is not the case when using it for fixed-point scalar multiplications as in our vulnerable implementation. Despite being known to leak information, recoding algorithms were mostly ignored on SCA research as attacking them requires techniques with fine granularity allowing to capture leakage at a single-branch level. It was until recently, when ul Hassan et al. [18] demonstrated that it is possible to recover the scalar value by performing a microarchitecture attack on the wNAF recoding algorithm used in Mozilla's NSS during `secp384r1` ECDSA computation.

A second, and more well known, SCA vulnerability in our example implementation is the double-scalar multiplication algorithm itself. Even if a SCA secure recoding algorithm is in use, the scalar multiplication algorithm itself is vulnerable against a SCA attacker, since its execution is highly dependent on the wNAF representation of the scalar value. While the double-scalar multiplication algorithm has not been exploited in the past, it follows the same execution flow of the wNAF scalar multiplication algorithm, which has been repeatedly shown to be vulnerable [12, 2, 29]. On a high level, this scalar multiplication algorithm performs an elliptic curve point double for each recoded scalar digit, and an elliptic curve point addition only when the recoded scalar digit is non-zero, thus the general idea is that a SCA attacker is able to recognize the zero and non-zero digits of the recoded scalar value, as well as being able to identify which was the value of the digit since it is the index of the multiplier accessed from the precomputed table during the elliptic curve point addition, giving enough information to ultimately recover the private key.

Scalar multiplications are a basic operation for digital signature algorithms, thus an attacker with SCA capabilities would have opportunity to recover a secret key not only during key generation but also during signature generation. We reckon that specifically for Ed25519, an attacker would require (near) perfect traces as no practical lattice attacks have been demonstrated against it but we speculate is only a matter of time before it is possible.

4 Computer-Aided Ed25519

It is easy to see from the analysis presented in Section 3 that an easy fix to the SCA flaws presented is to either use a well established cryptography library providing an Ed25519 implementation, revert back to the reference implementation best suited to our needs, or implement constant-time versions of those algorithms leaking sensitive information. However, we decided to explore a different approach. We decided to make use of a cryptography tool to generate "new" Ed25519 implementations, and then we compared them to other available implementations. This approach serves two purposes, it allows us: (i) to analyze the easiness of producing and implementing different SCA-secure Ed25519 implementations with the added benefit of (partial) formal verification; (ii) and to compare the performance among computer-aided and widely used implementations.

For the computer-aided Ed25519 implementations, we used the `ECCKiila`[11] cryptography tool created by Belyavsky et al. [5]. The tool uses the `fiat-crypto` project to generate formally verified Galois Field (GF) arithmetic [15] for many ECC curves including Ed25519, and on top of this layer it generates complete EC arithmetic. Everything generated as portable code for 32-bit and 64-bit architectures, therefore useful for several use cases.

New implementations. Harnessing the power of `ECCKiila`, we created two portable and SCA-secure Ed25519 implementations with different memory size requirements targeting different architectures: (i) a full-fledge portable implementation with a 30 KB precomputed table filling up an average L1 memory cache which we call `ecckiila-precomp`; and a lighter 32-bit implementation with a small 2.5 KB precomputed table suitable for smaller devices which we call `ecckiila-no-precomp`. The `ecckiila-precomp` implementation uses a constant-time fixed-point scalar multiplication based on the *comb* method [14, 9.3.3] and regular-NAF scalar recoding [20], while `ecckiila-no-precomp` uses a constant-time variable-point scalar multiplication and regular-NAF scalar recoding. Both implementations use the variable-time double-point scalar multiplication based on textbook wNAF [4] and Shamir's trick [14, 9.1.5].

Once we generated all the EC arithmetic using the tool, we were left with the task of adding EdDSA specific algorithms and creating the upper API layer. For the missing Ed25519 specific algorithms—i.e., point decompression, multiply and add, and modular reduction by the order of the base point—we ported them from the `ref10` implementation and adapted them accordingly. Then we implemented the public API layer on top them, resulting in a working implementation.

4.1 Benchmarking

After generating two computer-aided Ed25519 implementations, we decided to benchmark their performance and compare them against our aggressively optimized implementation from Section 3 (called `overoptimized`) and against

[11] https://gitlab.com/nisec/ecckiila/

other widely used implementations. For benchmarking we used the SUPERCOP[12] framework developed as part of the EBACS [8] project. SUPERCOP is a well established and well known cryptography benchmarking framework containing several different implementations for all types of cryptographic primitives, including hash functions, stream ciphers, block ciphers, key exchange, digital signatures, etc. Moreover, SUPERCOP runs on several architectures, allowing us to expand our comparison of implementations to include Intel and ARM architectures for 32 and 64 bits.

SUPERCOP already ships with the original reference implementations in its code, and in addition we included and adapted `donna`, `monocypher`, `ecckiila--no-precomp` and `ecckiila-precomp` to its required API in order to benchmark their performance. It is worth mentioning that adapting these implementations to SUPERCOP's required API does not affect their performance, but the reported values in this work might differ from each project self-reported values. This is due to each implementation using different RNG and hash function implementations—i.e., for our benchmarks all of the Ed25519 implementations use SUPERCOP's own RNG and hash functions.

Intel Setup. For both 64-bit and 32-bit benchmarks our setup consists of an Intel Xeon E5-1650 v2 Ivy Bridge EP at 3.50GHz running Ubuntu 18.04 LTS "Bionic Beaver". We disabled TurboBoost and set the frequency scaling governor to performance.

ARM Setup. For both 64-bit and 32-bit benchmarks our setup consists of a Raspberry Pi 3B equipped with a quad-core 1.2GHz Broadcom BCM2837 64-bit CPU and 1GB RAM, running Ubuntu 18.04 LTS "Bionic Beaver". The 64-bit `aarch64` has Linux kernel version `5.4.0-1026-raspi`, and the 32-bit `armv7l` has Linux kernel version `5.4.0-1015-raspi`. We disabled frequency scaling via software.

SUPERCOP Setup. SUPERCOP and all the implementations were compiled with stock `gcc` version 7.5.0, and using the `-O3` optimization level. The reported values are in thousands of clock cycles and they correspond to the median value of many measurements (as defined by SUPERCOP) for an operation on a 59-byte message.

Results. Table 2 and Table 3 show the results of our benchmarks for Intel and ARM architectures, respectively. Without surprise, `donna` is the most performant among all the implementations on both architectures, and it specially excels on the Intel architecture, where it is twice as fast as `ref10`. Another observation is that `monocypher` shows good results for being an implementation with a smaller memory footprint targeting IoT devices.

Our results confirm that optimizing for memory size not only has detrimental results for security, but also for speed, as observed in the `overoptimized` results where we observe a decreased performance by 2.5x at the cost of saving

[12] https://bench.cr.yp.to/supercop.html

Architecture	Implementation	Sign	Verify	KeyGen
x86_64	ref10	140 (□ base)	455 (□ base)	135 (□ base)
	ref	1560 (▽11.1x)	5218 (▽11.4x)	1531 (▽11.3x)
	amd64-64-24k	64 (▲2.18x)	225 (▲2.02x)	60 (▲2.25x)
	amd64-51-30k	66 (▲2.12x)	210 (▲2.16x)	62 (▲2.17x)
	donna	64 (▲2.18x)	217 (▲2.09x)	59 (▲2.28x)
	monocypher	230 (▽1.64x)	525 (▽1.15x)	210 (▽1.55x)
	overoptimized	264 (▽1.88x)	455 (▽1.00x)	227 (▽1.68x)
	ecckiila-precomp	101 (▲1.38x)	280 (▲1.62x)	96 (▲1.4x)
x86	ref10	399 (□ base)	1155 (□ base)	374 (□ base)
	ref	4137 (▽10.3x)	14105 (▽12.2x)	4086 (▽10.9x)
	amd64-64-24k	–	–	–
	amd64-51-30k	–	–	–
	donna	310 (▲1.28x)	962 (▲1.20x)	291 (▲1.28x)
	monocypher	533 (▽1.33x)	1347 (▽1.16x)	471 (▽1.25x)
	overoptimized	958 (▽2.40x)	1155 (▽1.00x)	914 (▽2.44x)
	ecckiila-no-precomp	1133 (▽2.83x)	1231 (▽1.06x)	1075 (▽2.87x)
	ecckiila-precomp	427 (▽1.07x)	1228 (▽1.06x)	368 (▲1.01x)

Table 2. Comparison of timings on Intel architecture. □ is the baseline. ▲ means a speedup (better) w.r.t. baseline. ▽ means a slowdown (worst) w.r.t. baseline. Timings are given in clock cycles (thousands).

Architecture	Implementation	Sign	Verify	KeyGen
aarch64	ref10	245 (□ base)	688 (□ base)	238 (□ base)
	ref	2924 (▽11.9x)	9579 (▽13.9x)	2425 (▽10.1x)
	amd64-64-24k	–	–	–
	amd64-51-30k	–	–	–
	donna	196 (▲1.25x)	638 (▲1.07x)	162 (▲1.46x)
	monocypher	422 (▽1.72x)	812 (▽1.18x)	366 (▽1.53x)
	overoptimized	726 (▽2.96x)	688 (▽1.00x)	635 (▽2.66x)
	ecckiila-precomp	270 (▽1.10x)	808 (▽1.17x)	261 (▽1.09x)
armv7l	ref10	597 (□ base)	1755 (□ base)	582 (□ base)
	ref	9933 (▽16.6x)	28642 (▽16.3x)	8442 (▽14.5x)
	amd64-64-24k	–	–	–
	amd64-51-30k	–	–	–
	donna	508 (▲1.17x)	1508 (▲1.16x)	495 (▲1.17x)
	monocypher	983 (▽1.64x)	2505 (▽1.42x)	987 (▽1.69x)
	overoptimized	1622 (▽2.71x)	1800 (▽1.02x)	1534 (▽2.63x)
	ecckiila-no-precomp	2134 (▽3.57x)	2237 (▽1.27x)	2050 (▽3.52x)
	ecckiila-precomp	815 (▽1.36x)	2213 (▽1.26x)	732 (▽1.25x)

Table 3. Comparison of timings on ARM architecture. □ is the baseline. ▲ means a speedup (better) w.r.t. baseline. ▽ means a slowdown (worst) w.r.t. baseline. Timings are given in clock cycles (thousands).

slightly more than 30 KB of memory used for precomputed tables during scalar multiplication.

For our two computer-aided implementations the results show, on the one hand, that `ecckiila-no-precomp` achieves similar size and performance results as `overoptimized` on the Intel 32-bit architecture, with the added benefit of being secure against SCA. On the other hand, we were positively surprised to observe that `ecckiila-precomp` outperforms `ref10` on the Intel 64-bit architecture and has very similar results on the Intel 32-bit architecture.

We note that on ARM architecture both `ecckiila-no-precomp` and `ecckiila-precomp` clearly lag behind when compared to their Intel counterpart. We speculate that these underperforming results on ARM are due to internal parameters in the `ECCKiila` tool used during code generation. These parameters try to calculate the correct size of the precomputed tables, however these parameters were not fine-tuned for our ARM benchmark devices. Our devices were incapable of internally generating the precomputed tables due to intensive computation by `fiat-crypto`, thus we generated them externally. We believe the ARM results could improve by correctly tweaking these parameters. In light of our results, it is interesting to observe that `ECCKiila` generates competitive portable ECC code that can potentially outperform handwritten, highly optimized code.

Finally, one more thing to consider for Ed25519 is that widely used implementations such as `ref10` and `donna` were originally published almost a decade ago, so these implementations do not consider new research results [13, 11, 10, 30] that further improve the security and performance of Ed25519.

5 Conclusion

Our toy example demonstrates, yet again, that implementing one's own cryptography is a complex task with a small margin for error, specially when strict requirements must be met. Aggressive optimizations can easily lead to a situation where both security and speed are greatly reduced at the cost of size as observed from our experiments, so we hope this serves as a lesson of a strategy to avoid. If implementation size is the main concern, a possible strategy to adopt is to use a SCA-secure variable-point scalar multiplication algorithm for key generation, signature generation, and signature verification. This reduces substantially the speed of all the operations but is secure and saves memory by avoiding precomputation tables, and additional recoding and scalar multiplication algorithms.

More generally, we recommend cryptography engineers, developers, and practitioners to avoid the usage of variable time algorithms on confidential inputs; the mix usage of SCA-secure and SCA-insecure algorithms, and we recommend to consider SCA good practices and recommendations if implementing cryptography is a must.

Additionally, our results show that computer-aided cryptographic tools have reached a maturity level where they can compete against code written by cryptography researchers with advanced skills on software and hardware engineering—

as reflected on their adoption on BoringSSL and NSS[18]—and thus we highly recommend adopting them as part of the development process.

Acknowledgments. We would like to thank Philip Ginzboorg for the comments during the development of this research.

The first author thanks the Nokia Foundation for the generous support through a Nokia Scholarship.

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. Digital signature standard (DSS). FIPS-PUB 186-5, National Institute of Standards and Technology (October 2019), https://doi.org/10.6028/NIST.FIPS.186-5-draft
2. Allan, T., Brumley, B.B., Falkner, K.E., van de Pol, J., Yarom, Y.: Amplifying side channels through performance degradation. In: Schwab, S., Robertson, W.K., Balzarotti, D. (eds.) Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016. pp. 422–435. ACM (2016), http://dl.acm.org/citation.cfm?id=2991084
3. Ambrose, C., Bos, J.W., Fay, B., Joye, M., Lochter, M., Murray, B.: Differential attacks on deterministic signatures. In: Smart, N.P. (ed.) Topics in Cryptology - CT-RSA 2018 - The Cryptographers' Track at the RSA Conference 2018, San Francisco, CA, USA, April 16-20, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10808, pp. 339–353. Springer (2018), https://doi.org/10.1007/978-3-319-76953-0_18
4. Avanzi, R.M.: A note on the signed sliding window integer recoding and a left-to-right analogue. In: Handschuh, H., Hasan, M.A. (eds.) Selected Areas in Cryptography, 11th International Workshop, SAC 2004, Waterloo, Canada, August 9-10, 2004, Revised Selected Papers. Lecture Notes in Computer Science, vol. 3357, pp. 130–143. Springer (2004), https://doi.org/10.1007/978-3-540-30564-4_9
5. Belyavsky, D., Brumley, B.B., Chi-Domínguez, J., Rivera-Zamarripa, L., Ustinov, I.: Set it and forget it! turnkey ECC for instant integration. In: ACSAC '20: Annual Computer Security Applications Conference, Virtual Event / Au stin, TX, USA, 7-11 December, 2020. pp. 760–771. ACM (2020), https://doi.org/10.1145/3427228.3427291
6. Bernstein, D.J.: Curve25519: New Diffie-Hellman speed records. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings. Lecture Notes in Computer Science, vol. 3958, pp. 207–228. Springer (2006), https://doi.org/10.1007/11745853_14
7. Bernstein, D.J., Duif, N., Lange, T., Schwabe, P., Yang, B.: High-speed high-security signatures. J. Cryptogr. Eng. 2(2), 77–89 (2012), https://doi.org/10.1007/s13389-012-0027-1
8. Bernstein, D.J., Lange, T.: eBACS: ECRYPT Benchmarking of Cryptographic Systems (September 2020), https://bench.cr.yp.to

9. Bernstein, D.J., Lange, T., Schwabe, P.: The security impact of a new cryptographic library. In: Hevia, A., Neven, G. (eds.) Progress in Cryptology - LATINCRYPT 2012 - 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7-10, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7533, pp. 159–176. Springer (2012), https://doi.org/10.1007/978-3-642-33481-8_9
10. Bernstein, D.J., Yang, B.: Fast constant-time gcd computation and modular inversion. IACR Trans. Cryptogr. Hardw. Embed. Syst. 2019(3), 340–398 (2019), https://doi.org/10.13154/tches.v2019.i3.340-398
11. Brendel, J., Cremers, C., Jackson, D., Zhao, M.: The provable security of ed25519: Theory and practice. IACR Cryptol. ePrint Arch. 2020, 823 (2020), https://eprint.iacr.org/2020/823
12. Brumley, B.B., Hakala, R.M.: Cache-timing template attacks. In: Matsui, M. (ed.) Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5912, pp. 667–684. Springer (2009), https://doi.org/10.1007/978-3-642-10366-7_39
13. Chalkias, K., Garillot, F., Nikolaenko, V.: Taming the many eddsas. In: van der Merwe, T., Mitchell, C.J., Mehrnezhad, M. (eds.) Security Standardisation Research - 6th International Conference, SSR 2020, London, UK, November 30 - December 1, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12529, pp. 67–90. Springer (2020), https://doi.org/10.1007/978-3-030-64357-7_4
14. Cohen, H., Frey, G., Avanzi, R., Doche, C., Lange, T., Nguyen, K., Vercauteren, F. (eds.): Handbook of Elliptic and Hyperelliptic Curve Cryptography. Chapman and Hall/CRC (2005), https://doi.org/10.1201/9781420034981
15. Erbsen, A., Philipoom, J., Gross, J., Sloan, R., Chlipala, A.: Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In: 2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019. pp. 1202–1219. IEEE (2019), https://doi.org/10.1109/SP.2019.00005
16. Gras, B., Giuffrida, C., Kurth, M., Bos, H., Razavi, K.: Absynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures. In: 27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020. The Internet Society (2020), https://www.ndss-symposium.org/ndss-paper/absynthe-automatic-blackbox-side-channel-synthesis-on-commodity-microarchitectures/
17. Gras, B., Razavi, K., Bos, H., Giuffrida, C.: Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In: Enck, W., Felt, A.P. (eds.) 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018. pp. 955–972. USENIX Association (2018), https://www.usenix.org/conference/usenixsecurity18/presentation/gras
18. ul Hassan, S., Gridin, I., Delgado-Lozano, I.M., García, C.P., Chi-Domínguez, J., Aldaya, A.C., Brumley, B.B.: Déjà vu: Side-channel analysis of mozilla's NSS. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020. pp. 1887–1902. ACM (2020), https://doi.org/10.1145/3372297.3421761
19. Josefsson, S., Liusvaara, I.: Edwards-curve digital signature algorithm (EdDSA). RFC 8032, 1–60 (2017), https://doi.org/10.17487/RFC8032
20. Joye, M., Tunstall, M.: Exponent recoding and regular exponentiation algorithms. In: Preneel, B. (ed.) Progress in Cryptology - AFRICACRYPT 2009, Second International Conference on Cryptology in Africa, Gammarth, Tunisia, June 21-25,

2009. Proceedings. Lecture Notes in Computer Science, vol. 5580, pp. 334–349. Springer (2009), https://doi.org/10.1007/978-3-642-02384-2_21
21. Möller, B.: Algorithms for multi-exponentiation. In: Vaudenay, S., Youssef, A.M. (eds.) Selected Areas in Cryptography, 8th Annual International Workshop, SAC 2001 Toronto, Ontario, Canada, August 16-17, 2001, Revised Papers. Lecture Notes in Computer Science, vol. 2259, pp. 165–180. Springer (2001), https://doi.org/10.1007/3-540-45537-X_13
22. Pippenger, N.: On the evaluation of powers and related problems (preliminary version). In: 17th Annual Symposium on Foundations of Computer Science, Houston, Texas, USA, 25-27 October 1976. pp. 258–263. IEEE Computer Society (1976), https://doi.org/10.1109/SFCS.1976.21
23. Poddebniak, D., Somorovsky, J., Schinzel, S., Lochter, M., Rösler, P.: Attacking deterministic signature schemes using fault attacks. In: 2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018. pp. 338–352. IEEE (2018), https://doi.org/10.1109/EuroSP.2018.00031
24. Romailler, Y., Pelissier, S.: Practical fault attack against the ed25519 and eddsa signature schemes. In: 2017 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2017, Taipei, Taiwan, September 25, 2017. pp. 17–24. IEEE Computer Society (2017), https://doi.org/10.1109/FDTC.2017.12
25. Samwel, N., Batina, L.: Practical fault injection on deterministic signatures: The case of eddsa. In: Joux, A., Nitaj, A., Rachidi, T. (eds.) Progress in Cryptology - AFRICACRYPT 2018 - 10th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 7-9, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10831, pp. 306–321. Springer (2018), https://doi.org/10.1007/978-3-319-89339-6_17
26. Samwel, N., Batina, L., Bertoni, G., Daemen, J., Susella, R.: Breaking ed25519 in wolfssl. In: Smart, N.P. (ed.) Topics in Cryptology - CT-RSA 2018 - The Cryptographers' Track at the RSA Conference 2018, San Francisco, CA, USA, April 16-20, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10808, pp. 1–20. Springer (2018), https://doi.org/10.1007/978-3-319-76953-0_1
27. Schnorr, C.: Efficient identification and signatures for smart cards. In: Brassard, G. (ed.) Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings. Lecture Notes in Computer Science, vol. 435, pp. 239–252. Springer (1989), https://doi.org/10.1007/0-387-34805-0_22
28. Tuveri, N., Brumley, B.B.: Start your ENGINEs: Dynamically loadable contemporary crypto. In: 2019 IEEE Cybersecurity Development, SecDev 2019, Tysons Corner, VA, USA, September 23-25, 2019. pp. 4–19. IEEE (2019), https://doi.org/10.1109/SecDev.2019.00014
29. Tuveri, N., ul Hassan, S., García, C.P., Brumley, B.B.: Side-channel analysis of SM2: A late-stage featurization case study. In: Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018. pp. 147–160. ACM (2018), https://doi.org/10.1145/3274694.3274725
30. de Valence, H., Grigg, J., Tankersley, G., Valsorda, F., Lovecruft, I.: The ristretto255 group. Tech. rep., IETF CFRG Internet Draft (2019)