Note #1: Update to the Sca25519 Library

Mitigating Tearing-based Side-channel Attacks

Łukasz Chmielewski and Lubomír Hrbáček

Masaryk University, Brno, Czech Republic
chmiel@fi.muni.cz,493077@mail.muni.cz

Abstract.
This short note describes an update to the sca25519 library, an ECC implementation computing the X25519 key-exchange protocol on the Arm Cortex-M4 microcontroller. The sca25519 software came with extensive mitigations against various side-channel and fault attacks and was, to our best knowledge, the first to claim affordable protection against multiple classes of attacks that are motivated by distinct real-world application scenarios.
This library is protected against various passive and active side-channel threats. However, both classes of attacks were considered separately, i.e., combining the attacks is considered out-of-scope because to successfully execute such a combined attack, the adversary would need to be very powerful (e.g., a very well-equipped security laboratory). Protection against such powerful adversaries is considered infeasible without using dedicated protected hardware with which Arm Cortex-M4 is not equipped.
However, there exists a particular class of easy and cheap active attacks: they are called tearing, and they are well known in the smartcard context. In this paper, we extend the scope of the library to also consider a combination of tearing and side-channel attacks. In this note, we show how we can mitigate such a combination by performing a small code update. The update does not affect the efficiency of the library.

Keywords: X25519 library · Tearing Attack · Side-Channel Analysis · Fault Injection

1 Introduction

Originally, the library was presented in a Systematization-of-Knowledge paper in IACR Transactions on Cryptographic Hardware and Embedded Systems 2023 [BCH+22a]; the corresponding eprint with some updates was also published [BCH+21]. This software came with extensive protections against both side-channel and fault attacks while being at least as efficient as widely-deployed ECC libraries. The authors were optimistic that the implementation is secure within the assumed attacker model [1], given the experimental evidence from, but left more extensive investigation for future work. In particular, they were mostly concerned about the threats posed by attackers equipped with EM microbes and applications of complex single-trace profiled attacks.

The sca25519 [BCH+22b] repository contains three implementations of X25519 in C and assembly for the Cortex-M4 with countermeasures against side-channel and fault injection attacks. The first implementation is unprotected, the second implementation contains countermeasures required for the case of ephemeral scalar multiplication, and

[1] To be precise, the authors considered an attacker according to level SL3 as defined in IEC-62443 [IEC13], which covers capabilities of academic attackers with well-equipped lab, but no powerful state agencies.

the third implementation contains the most countermeasures for the case of static scalar multiplication. The three implementations are stored under the following names:

STM32F407-unprotected contains the implementation for the unprotected X25519; this implementation contains no side-channel protections besides being constant-time;

STM32F407-ephemeral contains our implementation for the ephemeral X25519; this implementation contains some side-channel protections;

STM32F407-static contains our implementation for the static X25519; this implementation contains multiple side-channel protections.

This update concerns only the STM32F407-static implementation.

Software availability.

The update described in this note is in the public domain, uploaded to the repository [BCH+22b], release 1.1. The original code corresponding to the paper [BCH+22a] is marked as release 1.0.

Organization of the note.

First, we describe the tearing attacks in general, as well as the attack that this update protects against, in Section 2. Second, we describe the proposed code update in Section 3. Finally, we conclude the note in Section 4.

2 Tearing Attacks

One of the most straightforward and easy fault injection methods involves cutting power to the processor during critical computations. The aim is either to prevent the system from responding to a detected attack or to put the system in a vulnerable state when power is restored. We call such attacks tearing attacks. This kind of attack is the most common in the smartcard context [HP04, HMP06]. For example, an attacker can attempt to bypass the transaction mechanism by powering down (i.e., tearing) a card before a PIN try counter is increased in the case of an incorrectly provided PIN.

Such attacks on their own are considered in [BCH+22a]. However, the library is not susceptible to them.

2.1 A Combination Attack

Now, we can describe a combination of a tearing-based attack and a side-channel one for the STM32F407-static library. Let us recall how protected scalar multiplication in STM32F407-static works on a high level. First, the private key, which consists of a 64-bit blinding, blinded scalar, and two blinding points, is loaded. Second, the blinding and the scalar are additionally randomized and used in the scalar multiplication to compute the output. Finally, the key is updated and saved [2].

The whole idea of the attack is to collect side-channel traces but power down the target (running the library) before the final update of the key. This way, the scalar, blinding, and blinded points are not updated, while multiple side-channel traces can be collected. As a consequence of this tearing, the output of the operation is not obtained, but that is not required for many side-channel attacks.

In Figure 1, we point to the moment at which the card needs to be powered down. The arrow points at the beginning of the update procedure, and before this moment, the attacker needs to power down the device. This update does not always happen exactly

[2] Note that the sca25519 does not actually save the key to the flash memory but to the RAM. However, if the library is used in the practical application, then the key should be saved to the flash in a protected way (e.g., encrypted or saved to a restricted flash partition).

at the same moment in time due to the randomizations used during inversions. Still, the jitter caused by that is tolerable since it is only necessary to power down the target before the beginning of the procedure.

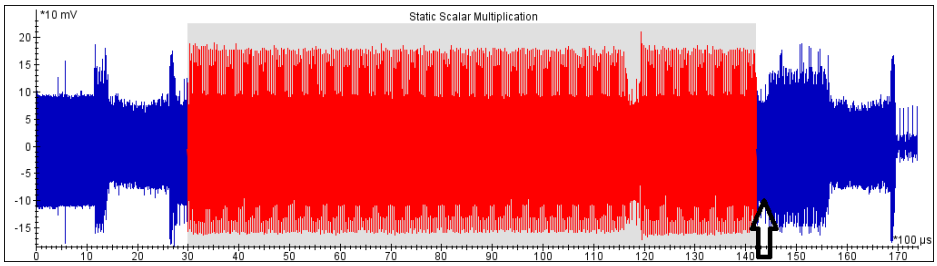

Figure 1: The power profile of the static (STM32F407-static) implementation marked the beginning of the update procedure.

The consequence of this attack is turning off point blinding but not scalar blinding (because of the additional randomization). Consequently, turning off countermeasures might enable the execution of side-channel attacks. Furthermore, it is possible to attack the parts of the private key not only during the initial transfer but also when it is being re-randomized. In particular, these parts will be loaded to register multiple times.

3 Code Update

The main idea is to move the update procedure from after the scalar multiplication to before. The resulting new algorithm is presented in Algorithm 1. The reshuffled steps are marked with blue color. With orange, we marked the code that we believe could be removed without increasing the leakage. However, this modification might modify a number of points of interest for single-trace attacks and would require side-channel evaluation. Therefore, we decided not to include it in release 1.1; we leave analyzing this extra optimization as a future work.

We performed a brief efficiency analysis, and the results are the same as for the original algorithm[3]. The results are in Table 1. We measured the clock cycles of the static implementation on an STM32F407 Discovery development board in the same way as in [BCH+22a]. In particular, for all our measurements, we use the `gcc` compiler, `arm-none-eabi-gcc version 9.2.1 20191025`, with the `-O2` optimization flag.

Table 1: Short performance evaluation of the updated static implementation (release 1.1).

Implementation/Component	**Clock Cycles:**
Complete static scalar multiplication:	2 338 126
Cost per cswaprr iteration:	1047
Updating static key:	362 958

Since there is no effective modification to the core of the algorithm, we did not perform a side-channel evaluation on this update.

Comment on removing blinding (i.e., the orange code in Algorithm 1).

We have also tested the code to remove the extra masking/blinding of the scalar after it is updated — see Table 2. As expected, the code is slightly more efficient[4]. However,

[3]There are some minor differences in the total length of the algorithm since it is randomized. The difference is, however, negligible.

[4]Updating static key is unchanged; therefore, the costs are approximately the same. Minor differences are due to randomizations used in this procedure.

Algorithm 1 Pseudocode of the updated side-channel and fault-attack protected static X25519

Input: the x-coordinate x_P of point P. **Output:** $x_{[k]P}$.
Secure Input: a 64-bit blinding f, blinded scalar $k_{f-1} = k \cdot f^{-1} \mod l$, and blinding points $R, S = [-k]R$.

$ctr \leftarrow 0$; $x_P \xleftarrow{\$} \{0, \ldots, 2^{256} - 1\}$ ▷ Initialize iteration counter to 0 and output buffer to random bytes
`Update`(R, S) ▷ 2 double-and-add scalar multiplications with the same 8-bit random scalar for R and S
`Randomize`(k_{f-1}, f) ▷ Generate new 64-bit random value f, securely compute f^{-1} and update k_{f-1}
`Save`(R, S, k_{f-1}, f)
Copy k_f to internal state while increasing ctr. ▷ Updating ctr in a loop protects copying against FI
$y_P \leftarrow$ ycompute(x_P)
`Increase`(ctr)
$(X_P, Y_P, Z_P) \leftarrow$ ecadd$((x_P, y_P), R)$ ▷ Point blinding, output of addition of R is projective
$(X_P, Y_P, Z_P) \leftarrow$ ecdouble(ecdouble(ecdouble$((X_P, Y_P, Z_P))))$ ▷ 3 doublings to multiply by co-factor 8
$r \xleftarrow{\$} \{1, \ldots, 2^{64} - 1\}$ ▷ Sample 64-bit non-zero random value for scalar blinding
$b \xleftarrow{\$} \{0, \ldots, \ell\}$ ▷ Sample blinding factor of non-constant-time inversion
$t \leftarrow r \cdot b \mod \ell$ ▷ Invert using extended binary gcd
$s \leftarrow t^{-1} \cdot b \mod \ell$ ▷ Unblind result of inversion
$k'_{f-1} \leftarrow k_{f-1} \cdot s \mod l$ ▷ Multiplicatively blind scalar k_{f-1}
$k' \leftarrow k'_{f-1} \cdot f \mod l$ ▷ Multiplicatively unblind scalar k'_{f-1} with f
`Increase`(ctr)
$x_P \leftarrow X_P \cdot Z_P^{-1}$; $y_P \leftarrow Y_P \cdot Z_P^{-1}$ ▷ Return to affine x and y coordinates
$X_1 \leftarrow 1, Z_1 \leftarrow 0$
$Z_2 \xleftarrow{\$} \{0, \ldots, 2^{255} - 20\}$; $X_2 \leftarrow x_P \cdot Z_2$ ▷ Initial randomization of projective representation
$k' \leftarrow k' \oplus 2k'$ ▷ Precompute condition bits for cswap
$a \xleftarrow{\$} \{0, \ldots, 2^{253} - 1\}$ ▷ Sample mask for address-randomization
$k' \leftarrow k' \oplus a$ ▷ Mask the scalar
`Increase`(ctr)
$(X_1, Z_1, X_2, Z_2) \leftarrow$ cswaprr $(X_1, Z_1, X_2, Z_2, a[252])$ ▷ Projective re-rand.+cswap based on masking a
for i from 253 downto 0 **do** ▷ scalar multiplication by $k' = k \cdot r^{-1}$
    $(X_1, Z_1, X_2, Z_2) \leftarrow$ cswaprr $(X_1, Z_1, X_2, Z_2, k'[i])$ ▷ Projective re-rand.+cswap based on masked k
    if $i \geq 1$ **then**
        $(X_1, Z_1, X_2, Z_2) \leftarrow$ ladderstep $(x_P, X_1, Z_1, X_2, Z_2)$
        $(X_1, Z_1, X_2, Z_2) \leftarrow$ cswaprr $(X_1, Z_1, X_2, Z_2, a[i-1])$ ▷ Projective re-rand.+cswap based on a
        `Increase`(ctr)
$y_P \leftarrow$ yrecover$(X_1, Z_1, X_2, Z_2, x_P, y_P)$
$x_P \leftarrow X_2 \cdot Z_2^{-1}$
$X_1 \leftarrow 1, Z_1 \leftarrow 0$
$Z_2 \xleftarrow{\$} \{0, \ldots, 2^{255} - 20\}$; $X_2 \leftarrow x_P \cdot Z_2$ ▷ Again randomize projective representation
$a' \xleftarrow{\$} \{0, \ldots, 2^{65} - 1\}$ ▷ Sample additional mask for address-randomization
$r \leftarrow r \oplus 2r$ ▷ Precompute condition bits for cswap
$r \leftarrow r \oplus a'$ ▷ Mask the random scalar r
`Increase`(ctr)
$(X_1, Z_1, X_2, Z_2) \leftarrow$ cswaprr $(X_1, Z_1, X_2, Z_2, a'[64])$ ▷ Projective re-rand.+cswap based on masking a'
for i from 64 downto 0 **do** ▷ Scalar multiplication by r
    $(X_1, Z_1, X_2, Z_2) \leftarrow$ cswaprr $(X_1, Z_1, X_2, Z_2, r[i])$ ▷ Projective re-rand.+cswap based on masked r
    if $i \geq 1$ **then**
        $(X_1, Z_1, X_2, Z_2) \leftarrow$ ladderstep $(x_P, X_1, Z_1, X_2, Z_2)$
        $(X_1, Z_1, X_2, Z_2) \leftarrow$ cswaprr $(X_1, Z_1, X_2, Z_2, a'[i-1])$ ▷ Projective re-rand.+cswap based on a'
        `Increase`(ctr)
$Y_2 \leftarrow$ yrecover$(X_1, Z_1, X_2, Z_2, x_P, y_P)$
$(X_2, Y_2, Z_2) \leftarrow$ ecadd$((X_2, Y_2, Z_2), S)$ ▷ Remove point blinding, add in $S = [-k]R$
$x_P \leftarrow X_2 \cdot Z_2^{-1}$
`Increase`(ctr)
if ! `Verify`(ctr) **then** ▷ Detected wrong flow, including iteration count
    $x_P \xleftarrow{\$} \{0, \ldots, 2^{256} - 1\}$ ▷ Set output buffer to random bytes
return x_P

we did not add this update to the release since we believe it requires an extra side-channel evaluation, and we leave it for future work.

Table 2: Short performance evaluation of the static implementation experimental branch (`removing_unnecessary_blinding`).

Implementation/Component	Clock Cycles:
Complete static scalar multiplication:	2 128 112
Cost per cswaprr iteration:	1047
Updating static key:	363 257

4 Conclusions

In this short note, we described an update to the sca25519 library, particularly to its static X25519 key-exchange component, that mitigates a combination of tearing-based attacks and side-channel attacks with an impact on efficiency.

References

[BCH+21] Lejla Batina, Łukasz Chmielewski, Björn Haase, Niels Samwel, and Peter Schwabe. SCA-secure ECC in software — mission impossible? Cryptology ePrint Archive, Paper 2021/1003, 2021. https://eprint.iacr.org/2021/1003.

[BCH+22a] Lejla Batina, Łukasz Chmielewski, Björn Haase, Niels Samwel, and Peter Schwabe. SCA-secure ECC in software — mission impossible? *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(1):557–589, Nov. 2022.

[BCH+22b] Lejla Batina, Łukasz Chmielewski, Björn Haase, Niels Samwel, and Peter Schwabe. The sca25519 library ("SCA-secure ECC in software — mission impossible?"), 2022.

[HMP06] Engelbert Hubbers, Wojciech Mostowski, and Erik Poll. Tearing Java Cards. In *Proceedings, e-Smart 2006, Sophia-Antipolis, France, September 20–22*, 2006. Available on-line.

[HP04] Engelbert Hubbers and Erik Poll. Reasoning about card tears and transactions in java card. In Michel Wermelinger and Tiziana Margaria, editors, *Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004 Barcelona, Spain, March 29 - april 2, 2004, Proceedings*, volume 2984 of *Lecture Notes in Computer Science*, pages 114–128. Springer, 2004.

[IEC13] Industrial communication networks – network and system security – part 3-3: System security requirements and security levels. Standard, International Electrotechnical Commission, 2013. https://webstore.iec.ch/preview/info_iec62443-3-3%7Bed1.0%7Db.pdf.