Enabling Efficient Threshold Signature Computation via Java Card API

Antonín Dufka
dufkan@mail.muni.cz
Masaryk University
Brno, Czech Republic

Petr Švenda
svenda@fi.muni.cz
Masaryk University
Brno, Czech Republic

ABSTRACT

Threshold signatures are becoming an increasingly popular method of signing key protection, primarily due to their ability to produce signatures that require the cooperation of multiple parties yet appear indistinguishable from a regular signature. This unique feature allows for their easy integration with existing systems, making them highly desirable in applications like national identity systems and transaction authorization, where they are being gradually deployed; their growing importance is further attested by NIST's recently initiated efforts to standardize threshold schemes [19].

An issue often encountered in the deployment of threshold schemes is that their execution is not supported by current secure hardware, which is necessary for the secure handling of secrets, as storing the shares in regular memory puts them at an increased risk of compromise. This raises the question of whether it is possible to run state-of-the-art threshold protocols with current secure hardware that we attempt to answer for cryptographic smartcards.

We analyzed algorithms available on smartcards with the Java Card platform and repurposed them to construct operations needed in threshold protocols. We use these derived operations to implement FROST, a state-of-the-art threshold signature scheme currently in a standardization process, making it the first open smartcard implementation of a threshold protocol supporting an arbitrary threshold. We demonstrate the practicality of this approach on the latest smartcards with no requirement for proprietary libraries.

CCS CONCEPTS

• **Security and privacy** → *Hardware security implementation*; *Digital signatures*; *Multi-factor authentication*; *Key management*.

KEYWORDS

threshold cryptography, Schnorr signatures, elliptic curves, smartcards, Java Card

ACM Reference Format:
Antonín Dufka and Petr Švenda. 2023. Enabling Efficient Threshold Signature Computation via Java Card API. In *The 18th International Conference on Availability, Reliability and Security (ARES 2023), August 29-September 1, 2023, Benevento, Italy.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3600160.3600180

1 INTRODUCTION

Threshold signature protocols are being increasingly deployed in practical applications to better secure private keys by decentralizing their storage and eliminating a single point of failure. With threshold protocols, a private key is split using secure secret sharing into multiple parts stored on different devices so that as long as less than a certain threshold of the devices is compromised, the private key remains secure. A signature is then created through an interactive protocol conducted among devices controlling the shares, during which they do not need to reveal their shares.

Since these protocols can produce signatures indistinguishable from single-party ones, they are easily integrable with existing systems without introducing any incompatible changes. Thanks to this property, threshold signatures are already deployed in applications including national identity scenarios [25] or transaction authorizations both in traditional finance and cryptocurrencies [16, 36]. Furthermore, the recently published call for threshold schemes by NIST [19] testifies to their increasing importance.

A commonly faced problem in the practical deployment of threshold schemes is that interfaces for their execution or their implementations are not available on secure hardware devices that are otherwise used to handle secrets securely. Because of this, computation with the secret shares needs to be performed by a general-purpose processing unit, increasing the risk of their compromise. This situation raises the question of whether it is possible to execute threshold protocols with the current secure hardware devices and interfaces they provide or if there is some limitation in their capabilities or interface programmability.

We focus on answering this question in the area of cryptographic smartcards, particularly those based on the Java Card platform [22], as they have the largest market share. Even the most recent revision (3.2) of the Java Card platform [21] has no explicit support for threshold protocols. While there exist some implementations of simpler n-party protocols [14, 17] for Java Card-based smartcards, no open implementation of protocols with a threshold lower than a total number of parties n is known. Therefore, whether the currently available Java Card smartcards are capable of computing state-of-the-art threshold protocols is an open question.

Java Card smartcards provide a generic programming interface capable of arbitrary computations, yet quite slow to achieve cryptographic computation at current security levels, especially for asymmetric cryptography. For that reason, the devices are typically equipped with cryptographic coprocessors, accelerating particular operations needed by algorithms like RSA decryption or ECDH key agreement. While the high-level algorithms can be accessed via Java Card API, the underlying operations are not directly accessible, severely limiting smartcard's programmability. To mitigate this issue, we analyzed the high-level algorithms available on the latest smartcards and repurposed them to compute lower-level operations.

To identify which operations are commonly needed in threshold protocols, we analyzed FROST [15], a state-of-the-art protocol computing threshold Schnorr signatures (or EdDSA signatures) currently in a standardization process [10]. FROST is an efficient protocol consisting of two rounds, the first of which can be securely precomputed, which is convenient for low-performance devices like smartcards or when communicating over slow networks. The second round, however, relies heavily on elliptic curve operations that need to be computed in dependence on the message content and are computationally demanding. Still, as we show later in this work, these operations can be computed efficiently on the latest smartcards.

Next, we implemented the FROST protocol, relying solely on the public Java Card API while fully complying with the current version of the standardization draft [10]. Additionally, we proposed and incorporated smartcard-specific optimizations in the implementation, significantly improving its performance without introducing any incompatibility. To the best of our knowledge, this is the first public open-source implementation of a threshold protocol on cryptographic smartcards supporting an arbitrary threshold.

We evaluated the efficiency of our implementation on three different smartcard models and dissected the computational cost of its components. Our findings reveal that FROST can be computed efficiently on the latest smartcards that provide elliptic curve algorithms outputting plain points. Yet, we demonstrate that even older smartcard models are capable of computing the protocol in full compliance with its current specification, albeit substantially slower.

1.1 Our contribution

This paper makes the following contributions:

(1) Identification of operations needed in FROST, a state-of-the-art threshold Schnorr signature protocol, and their mapping to operations computable via public Java Card API.
(2) Extension of capabilities of the JCMathLib library [18] to support operations needed for FROST implementation.
(3) The first public threshold protocol implementation capable of execution on Java Card-based smartcards with an arbitrary threshold, compliant with the FROST specification [10].
(4) Performance evaluation on three smartcard models with various threshold settings, demonstrating the practicality of the implementation.
(5) Discussion of application scenarios for threshold protocols computed on smartcards.

1.2 Related work

To the best of our knowledge, we are presenting the first public threshold protocol implementation allowing an arbitrary threshold and using only public Java Card API. However, there have been previous works implementing multi-party protocols on Java Card-based smartcards like n-party signatures and decryption [14, 17], which are notably simpler and more efficient.

Apart from multi-party protocols, there have been various implementations utilizing Java Card API to perform more complex protocols focusing primarily on identity management and privacy like anonymous credential scheme implementation by Bichsel et al. [3], utilizing RSA algorithm accessible by public Java Card API to achieve faster modular multiplication. A similar technique and its variation has been used by Sterckx et al. [26] in the implementation of direct anonymous attestation scheme on smartcards. Vullers implemented self-blindable credentials while exploiting Java Card API to perform scalar multiplication on an elliptic curve [30]. Balli et al. [1] implemented a suite of protocols allowing a smartcard to be used as a biometric identity document, building on elliptic curve scalar multiplication and point addition implementation provided by JCMathLib [18]. The most recent works in this area presents an efficient implementation of revocable keyed-verification anonymous credentials [7] and privacy-preserving digital identity token suitable for use in humanitarian aid distribution [31].

In parallel to these applications, there have been attempts to create reusable libraries for the Java Card platform providing low-level operations needed for cryptographic implementations relying only on public Java Card API. One of the first such projects is the implementation of modular arithmetic by Tews and Jacobs [29], who first published the method for accelerating modular multiplication by reusing RSA algorithm outputs. This library was used as a foundation for JCMathLib [18], which extended it by including support for elliptic curve operations, tools for improving memory management and performance profiling. We further extend this library as a part of this work by new operations that we use for the implementation of the threshold signature scheme.

The rest of this paper is organized as follows. After the introductory section, Section 2 presents the relevant background, covering the Java Card platform, secure secret sharing, and threshold signatures. In Section 3, we inspect the FROST signature scheme specification [10], identify required operations to comply with the specification, and describe how the computations can be realized via public Java Card API. Sections 4 and 5 present the implementation and its experimental evaluation while discussing various optimization options. In Section 6, we discuss possible applications utilizing the unique combination of threshold schemes executed on smartcards; and we conclude the paper with Section 7.

2 BACKGROUND

In this section, we briefly describe the basic characteristics of the Java Card platform, introduce the necessary background and notation for secure secret sharing, and describe how it is involved in a threshold signature computation.

2.1 Java Card platform

The Java Card platform allows for implementing Java-like applications and their execution on devices capable of running a Java Card virtual machine, typically smartcards. The implemented applications (called applets) are portable to other devices that run compatible virtual machines. This simplifies the development and deployment of smartcard applications [8].

The Java Card specification [22] includes standard packages that provide developers with a unified interface to various cryptographic functionalities, including hash algorithms, symmetric and asymmetric ciphers, signatures, key agreement protocols, and random number generation. Via these interfaces, Java Card virtual machine can provide algorithm implementations leveraging underlying hardware accelerators, enabling significant performance improvement of costly operations without having to introduce platform-dependent interfaces. However, implementations of algorithms not included in the standard packages cannot access these accelerators, at least not directly.

2.2 Secret sharing and threshold protocols

Secure secret sharing allows for splitting a secret into n shares in a way that at least a certain number t (called threshold) of the shares is needed to reconstruct it, while any set of less than t shares provides no information about the secret. Secure secret sharing of a secret s from the finite field of q elements $\mathbb{F}_q$ is typically realized using Shamir's secret sharing [24]. In Shamir's secret sharing, a random polynomial $f(x)$ of degree $t-1$ over $\mathbb{F}_q$ is sampled so that $f(0) = s$, and the shares are computed as evaluations of the polynomial in points $1, \ldots, n$.

Given a set $S = \{(x_1, f(x_1)), \ldots, (x_t, f(x_t))\}$ of t evaluations of the polynomial in different points, it can be uniquely reconstructed using Lagrange interpolation as:

$$f(x) = \sum_{(x_i, f(x_i)) \in S} f(x_i)\lambda_i(x)$$

where

$$\lambda_i(x) = \frac{(x-x_1)}{(x_i-x_1)} \cdots \frac{(x-x_{i-1})}{(x_i-x_{i-1})} \cdot \frac{(x-x_{i+1})}{(x_i-x_{i+1})} \cdots \frac{(x-x_t)}{(x_i-x_t)}.$$

Evaluation of this reconstructed polynomial at 0 yields the secret. We use λ_i to denote $\lambda_i(0)$ in the rest of this work.

Threshold signature schemes allow a group of at least t signers holding secret shares of a private key to compute a signature verifiable with the public key corresponding to the securely shared private key without having to reconstruct it, while any set of less than t parties is not able to do so. Signatures produced this way may be indistinguishable from standard single-party signatures and thus compatible with all preexisting applications that rely on signatures of a particular form.

There have been proposed threshold protocols for all practically used signature schemes; in particular, there have been recently several works on threshold ECDSA [6, 11, 12] and threshold Schnorr signatures [15]. In this work, we focus on threshold schemes based on Schnorr signatures, which are becoming popular also due to their properties suitable for threshold signature construction.

3 OPERATIONS REQUIRED BY FROST

In this section, we identify what operations are required in FROST computation and discuss how they can be efficiently computed via the public interface provided by the Java Card platform.

The current version of the specification [10] defines the scheme FROST(E, H) parameterized by two values – an elliptic curve E and a hash function H, and describes five ciphersuites:

- FROST(Ed25519, SHA-512),
- FROST(Ed448, SHAKE256),
- FROST(ristretto255, SHA-512),
- FROST(P-256, SHA-256), and
- FROST(secp256k1, SHA-256).

We will discuss the required operations in general for FROST(E, H) but highlight properties specific to particular instantiations when necessary.

3.1 Modular arithmetic

Independently on a given parametrization, all instantiations of FROST need to perform modular arithmetic with large numbers. The modular arithmetic is used to construct the signature share of party i, defined as:

$$s_i = r_{0,i} + b_i r_{1,i} + \lambda_i e x_i \pmod q$$

where $r_{0,i}$ and $r_{1,i}$ represent the hiding and binding nonces generated by signer i; the value b_i is its binding coefficient; the Lagrange coefficient λ_i corresponds to the signer's secret share x_i; the signature challenge is denoted by e; and q is the order of the elliptic curve group. From the expression, it is apparent that FROST implementations need to perform modular additions and multiplications. Furthermore, to compute λ_i the computation of modular subtraction and inversion is needed. Lastly, the computation of b_i and e involves hashing a bit string to a field element, requiring modular reductions in compliance with the specification.

The Java Card platform does not provide any direct support for modular arithmetic with large numbers. It natively supports only word-sized arithmetic (typically 2- or 4-byte wide). Some implementations may optionally provide class `BigNumber` that allows performing basic operations with larger integers, but it is typically constrained to eight bytes and cannot perform modular arithmetic required by cryptographic implementations.

Therefore, another approach has to be used to support modular arithmetic without relying on any proprietary interface. The basic word-sized arithmetic may suffice for the computation of simpler operations, but more complex operations require the aid of cryptographic accelerators to achieve reasonable performance. The accelerators are accessible only via high-level algorithms provided by the Java Card API. Fortunately, outputs of these algorithms can often be repurposed for constructing more generic operations, though with an unavoidable overhead.

Addition and subtraction. No algorithms in the Java Card API allow accelerating modular addition and subtraction. However, as long as the inputs to the computation are smaller than the modulus, these operations can be computed efficiently with just the basic word-sized arithmetic.

Multiplication. Modular multiplication can also be implemented using word-sized arithmetic; however, its quadratic complexity substantially affects computation time. Java Card API provides two algorithms that allow accelerating modular multiplication: RSA decryption (`ALG_RSA_NOPAD`) and plain Diffie-Hellman key agreement (`ALG_DH_PLAIN`). Both these algorithms compute modular exponentiation, which is by an order of magnitude more complex than multiplication. But since a dedicated coprocessor performs this computation, it is able to outperform multiplication implemented with the basic arithmetic in a Java Card virtual machine.

The ability to perform modular exponentiation does not immediately yield multiplication, but it can be transformed to it using the well-known binomial formula:

$$2ab \equiv (a+b)^2 - a^2 - b^2 \pmod{q}$$

using which the modular multiplication can be constructed at the cost of three squarings, one addition, two subtractions, and a division by two, that can be efficiently realized by a bit shift. Counterintuitively, this computation is still faster than software implementation of modular multiplication, and has been previously utilized in a number of works [3, 18, 26, 29]. Furthermore, Sterckx et al. [26] have proposed an alternative formula for the multiplication:

$$4ab \equiv (a+b)^2 - (a-b)^2 \pmod{q}$$

but by their experiments, the division by four has turned out to be slower than the additional squaring on some cards.

Inversion. Computation of modular inversion is equivalent to the ability to perform modular exponentiation as by Fermat's little theorem $a \cdot a^{q-2} \equiv 1 \pmod{q}$ for prime q. The value $a^{q-2} \pmod{q}$ can be computed directly with the aforementioned algorithms for RSA decryption or plain Diffie-Hellman key agreement.

Modular reduction. The Java Card API does not provide any algorithm that would allow speeding up the computation of modular reduction, as all the algorithms require the base to be smaller than the modulus to run the algorithm. Therefore, this operation has to be implemented fully within the Java Card virtual machine, requiring remainder division performed only with word-sized arithmetic. The complexity of this computation is quadratic in the difference between the bit lengths of the reduced number and the modulus, rendering it very costly for large inputs.

3.2 Elliptic curves

Depending on the given parametrization, FROST(E, H) needs to perform computations with points in the group of E. The most significant number of elliptic curve point operations is performed in the aggregation of a group commitment:

$$R = \sum_{i=1}^{t} R_{0,i} + \sum_{i=1}^{t} b_i R_{1,i}$$

where b_i is the same binding coefficient as in the signature share computation; $R_{0,i}$ and $R_{1,i}$ are hiding and binding commitments, corresponding to nonces generated by party i for its signature share. Apart from the group commitment aggregation, scalar multiplication is also needed for the computation of hiding and binding commitments by a signing party. Additionally, the specification [10] states that points should be encoded using SEC1 compressed encoding [4] or a similar compressed encoding in case of curves `Ed25519`, `Ed448`, and `ristretto255`, in which one of the point coordinates has to be reconstructed.

Java Card platform supports specifying parameters of curves in the short Weierstrass form, allowing for the direct support of curves like `secp256k1` and `P-256`. These curves can then be used with algorithms for plain elliptic curve Diffie-Hellman key agreement and the generic mapping computation according to specification TR-03110 [5][1] used in electronic passports, both of which output points without additional processing and thus can be used in further computations.

As any elliptic curve whose field characteristic is different from 2 and 3 can be mapped to a curve in the short Weierstrass form [32], it is possible to convert twisted Edwards curves like `Ed25519` to their Weierstrass equivalents (like `Wei25519` [27]) and perform operations on them. This will incur additional overhead with point transformations, but in some applications, the transformations can be trustlessly offloaded to a different device. Such an approach has already been demonstrated with Java Card smartcards [23].

Starting with Java Card API version 3.1 [20], a new API introducing a set of named elliptic curves became available. This API allows using curves like `Ed25519` and `Ed448` directly, but their usage is limited only to specific operations, e.g., EdDSA signing [2], which do not allow reusing them to derive more generic operations.

Point addition. Algorithm `ALG_EC_PACE_GM` outputs the point $sP + Q$, where s is a scalar from $\mathbb{F}_q$ and P, Q are points of E. Since the scalar can be set to 1, this algorithm can be used to perform point addition on smartcards that do support it. Furthermore, in case scalar multiplication and point addition operations need to be performed in a sequence in some application, with this algorithm, it can be achieved in a single API call without increasing the cost.

In case this algorithm is not supported, it is still possible to implement point addition using modular arithmetic over the elliptic curve's field, requiring several modular subtractions and multiplications, and one modular inversion. However, this approach is significantly more costly as the modular arithmetic has to be constructed as described in the previous section.

Scalar multiplication. The generic mapping from the previous paragraph can be repurposed for just scalar multiplication, though another slightly more efficient algorithm can be used just for scalar multiplication: `ALG_EC_SVDP_PLAIN_XY`. The algorithm is intended for computing the elliptic curve Diffe-Hellman key agreement, but as it outputs the resulting point without any further processing, it can be directly used for scalar multiplication.

If the previous algorithms are not supported, it may still be possible to compute the result using `ALG_EC_SVDP_DH_PLAIN` or `ALG_EC_SVDP_DHC_PLAIN`[2] algorithms, which, unlike the previous algorithms, output just the X coordinate of a point. In this case, it is necessary to compute the corresponding Y coordinate, but since there are two such values, additional checks to determine the correct sign are needed. This computation is quite costly, as it involves the computation of modular square root requiring modular multiplications and exponentiations. Still, this approach has already been successfully used [18, 30].

Point encoding and decoding. Although Java Card API supports SEC1 encoding used with curves `P-256` and `secp256k1`, the support of compressed points is only optional and rarely available. Encoding of `Ed25519` and `Ed448` points is supported only for the named curves API, that does not seem usable for generic computations. In any case, both encoding and decoding can be implemented using modular arithmetic; however, decoding can be quite costly.

[1]Introduced in Java Card API version 3.0.5.
[2]This algorithm additionally performs cofactor clearing.

3.3 Hash functions

FROST(E, H) uses the cryptographic hash function H in many contexts, separating each of them by a different domain separation tag. In total, the hash function is used in five parts of the implementation: three times to hash to a scalar in $\mathbb{F}_q$ and two times as a hash to a byte string. The defined ciphersuites include three different hash functions: SHA-256, SHA-512, and SHAKE-256.

The Java Card API contains a wide range of standard cryptographic hash functions, including SHA-256 and SHA-512; however, SHAKE-256 is not among them, while similar hash functions from the SHA-3 family are. Furthermore, even though SHA-512 is included in the Java Card API specification, it is not available as commonly as SHA-256 [28].

If the hash function of a ciphersuite is not supported by the Java Card API on a particular card, it would require implementing the hash function inside the application code executed by a Java Card virtual machine. Such implementation would probably turn out to be too slow in practice, especially since hashing is used heavily in the FROST specification.

4 IMPLEMENTATION

Based on the requirements discussed in the previous section, we decided to implement one of the ciphersuites that uses a curve in the short Weierstrass form without the need for transformations, to get performance closer to a native implementation, and SHA-256, as its support among available smartcards is more common and it is faster than SHA-512. Out of the two possible options FROST(P-256, SHA-256) and FROST(secp256k1, SHA-256), we decided to implement the latter, as the curve is used in Bitcoin, and may be used for signing Taproot transactions (see Section 6). The switch to the other curve is only a matter of changing a few constants in the code. The resulting implementation[3] is compliant with the FROST specification [10] and was verified by the supplemented test vectors.

4.1 JCMathLib extension and improvements

Our implementation is based on the JCMathLib library [18], an open-source library for the Java Card platform providing modular arithmetic with large numbers and operations with points on elliptic curves. The library depends only on public Java Card API, so it resorts to some of the approaches described in the previous section.

However, as the library was originally developed and tested on smartcards supporting only Java Card API 3.0.1 and it was not expanded since, it lacked the support of new algorithms introduced in later versions, especially in version 3.0.5, allowing much more efficient point addition and scalar multiplication. Furthermore, constraints on inputs to the algorithms utilizing modular exponentiation accelerators have changed in time, and input preprocessing had to be reinvented for the latest smartcards. In the rest of this subsection, we give an overview of changes made to the library to enable the implementation of FROST.

Backward-compatible extensions. The main building block of most of the functionality of JCMathLib is the modular exponentiation utilizing the ALG_RSA_NOPAD algorithm. However, since then, the constraints on inputs passed to the API have evolved, and new cards could not run the library. There are minor differences in what various Java Card implementations are able to accept, and it needs to be addressed on a per-card basis. We identified which input transformations are needed for the API call to succeed for each of our tested cards and changed the implementation to be able to accommodate all permissible execution paths, depending on which card it is being executed on. This way, we maintain backward compatibility with older smartcards and are able to support various constraints of the new ones.

To make the code compile even with older versions of Java Card SDK that do not support constants defined in later versions, we use the corresponding numbers directly. If an algorithm is not supported and thus cannot be instantiated, the code either selects an alternative implementation of the operation or fails. Furthermore, smartcards sometimes support algorithms introduced in later versions of Java Card API even though they do not fully support this API version, and they can be used this way. For example, this approach allows us to use algorithms for efficient operations on elliptic curves on an NXP J3H145 card with API version 3.0.4.

Compressed point encoding and decoding. As none of our tested cards supported compressed points, and we wanted to fully comply with the FROST specification [27], we had to implement SEC1 compressed point encoding and decoding [4]. Encoding is straightforward, as it only needs to output the X coordinate prefixed with a value depending on parity of Y. Decoding requires recomputing the Y coordinate, which involves costly modular square root computation and several modular multiplications and additions.

Hardware-accelerated point addition. When JCMathLib was originally implemented, Java Card API did not include the algorithm for computing the elliptic curve generic mapping (ALG_EC_PACE_GM), and the point addition had to be computed using modular arithmetic. We integrated the new algorithm into the library to enable more efficient point addition on smartcards that support it, resulting in more than 7x speedup on such cards. We also extended JCMathLib API to expose multiplication and addition in a single call so that its users can take advantage of a single Java Card API invocation in their applications.

Faster scalar multiplication. Similarly to the previous paragraph, the algorithm for elliptic curve Diffie-Hellman computation that outputs full point (ALG_EC_SVDP_DH_PLAIN_XY) was not available when the library was implemented; instead, the authors had to resort to recomputing Y coordinate with significant impact on the performance. We integrated this new API call in the library and set it as the default option for scalar multiplication for smartcards that do support it, resulting in more than 4x speedup on tested cards.

Faster modular multiplication. The original modular multiplication implementation in JCMathLib was based on the $2ab$ formula and performed the computations with a modulus of bit length greater than $(a + b)^2$, requiring an extra remainder division of the result. This was done for two reasons. The first reason is that constraints on inputs to the RSA algorithm on older smartcards caused modular squaring with modulus of a lower bit length than

[3]The source code is available at https://github.com/crocs-muni/JCFROST.

the minimal allowed by the RSA algorithm[4] to be inefficient, as its each output would need to get reduced using remainder division. The second reason is that this way, the intermediary values would not get reduced in any of the squarings, obtaining even $2ab$ which could be bit-shifted to get ab.

For older smartcards, we changed this functionality to use the $4ab$ formula, as the unreduced variant does not suffer from the issue with slow division by four, reported by Sterckx et al. [26], as the bit shift can be performed directly by 2 bits. This change resulted in only 8% speed improvement as the cost is still dominated by the remainder division.

For newer smartcards, we reimplemented the functionality so that the computations are performed directly mod q, lowering the cost of additions and subtractions and avoiding remainder division entirely. Furthermore, we based the computation on the $4ab$ formula and managed to avoid dividing by four using the following trick:

$$c \equiv (a+b)/2 \pmod q$$
$$ab \equiv c^2 - (c-b)^2 \pmod q$$

With this approach, we were able to achieve more than 4.5x speedup over the previous implementation of which around 0.7x can be attributed to using the $4ab$ formula.

4.2 Implementation optimizations

We implemented the protocol in full compliance with the specification [10], but in practical usage with smartcards, it is reasonable to make minor compromises to achieve substantially better performance. In particular, we propose the following two optimizations.

Decompressed points. Since smartcards are always used in combination with a typically much more computationally powerful host device, the device may perform additional precomputations to decrease the computational load on the smartcard. This can be used in the case of point decompression. As discussed in the previous sections, point decompression requires a sequence of (for a smartcard) costly operations. We can rely on the host device to decompress points as they are being sent to the smartcard. This computation can be done without any impact on security, as smartcard implementations should check the validity of points upon their loading into its `ECPublicKey` object.

Additionally, if the implementation were to use a ciphersuite that does not use an elliptic curve in the short Weierstrass form, and thus needed to convert it (as discussed in Section 3.2), the host could perform the conversion along with the decompression, saving even more computation of the smartcard.

Efficient lambda computation. Lagrange coefficient computation requires a number of modular multiplications linear in t, which is a significant factor for large signing groups. However, if the maximal number of parties is reasonably small, most of the intermediate multiplications can be performed without modular reductions (up to $n \leq 57$). Furthermore, with $n \leq 12$, the intermediate computations can fit within a 4-byte wide type, and with $n \leq 7$, within a 2-byte wide type. One additional subtraction needs to be performed if the result is negative, but it is a minimal price for the performance gain.

[4]The `ALG_RSA_NOPAD` algorithm typically requires at least 512- or 1024-bit modulus.

Table 1: Time (ms) required to compute signature share on the analyzed smartcards in dependence on threshold t.

t	J3R200	J3H145	J2E145
2	1290	2481	25724
3	1649	3219	38435
4	1940	4137	51180
5	2297	4933	63934
6	2597	5682	76597
7	2956	6481	89523
8	3257	7276	—
9	3544	8023	—
10	3902	8798	—
11	4196	9610	—
12	4556	10352	—

5 EXPERIMENTAL EVALUATION

We measured the performance of the signing round of the implementation on three smartcards by NXP: J3R200, J3H145, and J2E145. The measurement was done using JCProfilerNext [35], with each measurement repeated 100 times. All inputs to the protocol[5] were freshly sampled for each measurement and then fixed for the rest of the profiling to obtain precise results. In this section, we present only the results of the implementation with optimizations proposed in the previous section; for measurement results of unoptimized implementation, see the Appendix.

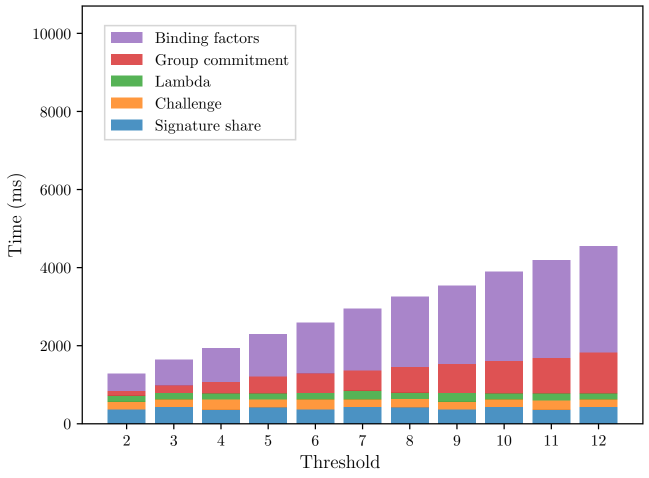
Table 1 shows the mean values of the measurements in dependence only on threshold t, as the computation is independent of the group size n. From the results, we can see that smartcards that do support efficient computation with elliptic curves are able to produce a signature share within a few seconds for all measured values of t. In contrast, older smartcards with lower processing speeds that do not support these algorithms require more than 25 seconds to complete the computation even in the simplest case.

To identify how different parts of the computation contribute to the overall cost, we present Figure 1a (NXP J3R200) and Figure 1b (NXP J3H145). These figures display the computation time divided into its five components: binding factors computation, group commitment computation, lambda computation, challenge computation, and signature share computation. Figures for the remaining tested cards are provided in the Appendix.

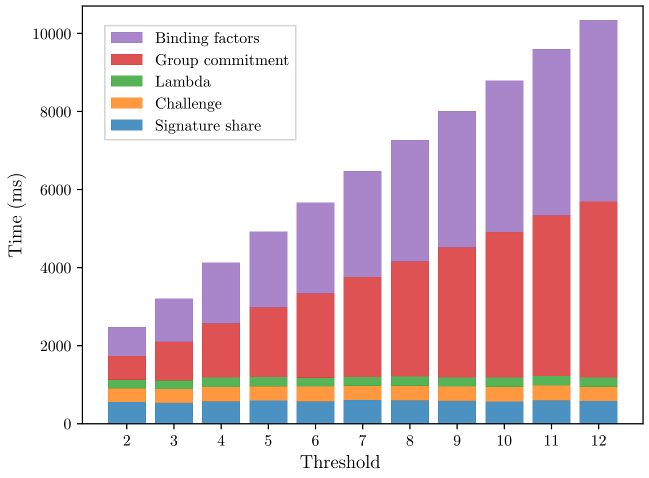
From the figures, it is apparent that computations of the resulting signature share and challenge are not correlated with t, thus setting a lower bound on the computation time. It may appear that the computation of lambda (the Lagrange interpolation coefficient λ_i) is also uncorrelated with t, but that is only due to the proposed optimization that allows most of the computation to be performed only with the platform's native type. Otherwise, this computation would cause a significant slowdown with increasing t.

Group commitment computation is performing on the native performance level on both J3R200 and J3H145 cards, as most of the computational load of point addition and multiplication is realized by the underlying hardware-accelerated algorithms. However,

[5]Except for the message, which was fixed to the string `"frost"`.



(a) NXP J3R200 (API version 3.0.5)



(b) NXP J3H145 (API version 3.0.4)

Figure 1: Time measurement (average of 100 runs, in milliseconds) of signing round components with different threshold values on smartcards that support efficient elliptic curve operations.

this computation is a major bottleneck for older cards that do not support efficient elliptic curve operations.

Apart from the operations with elliptic curve points, binding factor computation is the most demanding component. The computation requires deriving a binding coefficient for each of the t parties by processing inputs to the protocol using H and then mapping the result to a scalar in $\mathbb{F}_q$, requiring costly modular reduction. These operations cannot be precomputed as they rely on fresh inputs, and the modular reduction cannot cannot be avoided without diverging from the specification. The performance of hash computations could be improved by storing the internal state of the hashing object and reusing it, since many of the computations share the same prefix. However, this cannot be done with `MessageDigest` class provided by the Java Card API.

5.1 Further possible optimizations

Beyond the optimizations incorporated in the presented results, we considered other optimizations, but decided to not include them in the implementation, as they may deviate from the FROST specification in the compatibility of the produced signature shares or even in the protocol's security model. Still, we present those optimizations in this section in case they are suitable for some use cases.

Precomputed lambda. As long as the signing subgroup remains fixed, the Lagrange interpolation coefficient λ_i remains the same. This can be taken advantage of to speed up repeated signing by the same subgroup by storing the intermediate result of the multiplication of λ_i and the secret key share x_i. Furthermore, if the number of eligible signing groups $\binom{n}{t}$ is small enough, the intermediate results can be precomputed for all of them, decreasing the signature share computation cost by the lambda computation and one modular multiplication. This optimization would result in saving of more than 250ms on the measured smartcards.

Binding factors without modular reduction. The most limiting component of the compliant implementation of FROST is modular reduction, which needs to be performed for each binding coefficient. Binding coefficient computation involves hashing a sequence of inputs of the protocol and then mapping the result to a field element. This mapping requires, by the specification [10], to produce 16 more bytes from the hash function than is the size of the field and reduce it modulo the field's order, to minimize the introduced bias. However, as is also discussed in the specification [10], if the order of the field to which the hash result should be mapped is close to a power of two, it may be good enough to simply get the corresponding number of bits from the hash function and interpret it as a scalar. This change would allow for avoiding the modular reduction in binding factor computation, significantly improving the performance of our implementation. Nonetheless, since binding factors are derived differently with this optimization, it would no longer be compatible with the specified ciphersuites.

Offloaded group commitment aggregation. The group commitment aggregation is a major bottleneck for smartcards that do not support algorithms for efficient operations with points on elliptic curves. This computation can be avoided at the cost of changing the security model and enforcing sequential execution. Drijvers et al. [13] have shown that threshold Schnorr signature schemes accepting a group commitment computed externally can be secure under stronger assumptions in a setting with limited concurrency. Utilizing this result, the group commitment could be computed by the host device and sent together with the signing request.

Furthermore, with this change, binding factors would not provide any utility to the smartcard, so their computation could be avoided and, consequently, even the generation of the binding nonce. Not even the commitments of other parties would need to be received. The computation would be reduced to the bare minimum required to construct a threshold Schnorr signature share, i.e., the computation $r_{0,i} + \lambda_i e x_i \pmod q$. Interestingly, as discussed in a related work [14], such a signature share can be made compatible with signature shares of other parties running compliant FROST implementations.

6 APPLICATION SCENARIOS

The implementation presented in this work is primarily intended to demonstrate that it is possible to efficiently compute a state-of-the-art threshold signature protocol with current Java Card-based smartcards, relying only on public Java Card API. As the implementation has not been properly analyzed and is most likely susceptible to side-channel attacks due to implementation of modular operations aided by computations executed within a Java Card virtual machine, it should not be deployed in practical applications where side-channel attacks are part of the attacker model. With this in mind, in the rest of this section, we discuss application scenarios that would benefit from having threshold protocols run on smartcards or other secure hardware devices.

6.1 Cryptocurrency hardware wallets

Cryptocurrency hardware wallet designs have been utilizing multiple secure elements to prevent a signing key breach in case one of the elements turns out vulnerable [9]. However, the elements are used only for storage in currently deployed devices. They do not perform actual computations with the signing key, which is thus at increased risk once loaded into the device's memory. By utilizing more programmable secure elements like Java Card-based smartcards, threshold signature protocol could be computed directly with the elements without loading the key to an unprotected memory.

As a variant of Schnorr signatures [33] has been deployed in Bitcoin since the activation of Taproot [34], our implementation can be applied in this area with minor modifications. The signatures, specified in BIP 340 [33], are defined over the `secp256k1` curve, which is already used in our implementation, and the main difference is in point encoding. By the specification, all points are encoded only as their X coordinate, leaving two options for Y coordinate when decoding a point and implicitly choosing one of them. This may require protocol participants to negate their nonces if a group commitment cannot be decoded properly. Still, the introduced overhead should be minor.

6.2 Security of keys on smartphones

Secure storage of keys on modern smartphones relies on specialized interfaces[6], often backed by secure hardware. While these interfaces support certain most common cryptographic algorithms like ECDSA or RSA signatures, they may not support more recent or application-specific schemes like EdDSA or BIP 340 signatures. In case application developers need to use one of the unsupported schemes, they cannot benefit from these secured interfaces and have to resort to less secure key storage methods.

The security of those approaches can be enhanced using smartcards and threshold signatures by splitting the key into two shares, one held by a smartphone and the other by a smartcard. Whenever the key would need to be used, a user would tap the smartphone with the smartcard, which would execute the signing protocol and output a signature. In order to obtain the signing key, an attacker would need to compromise both devices, thereby the overall security would be increased. Even if the implementation on the smartcard could be compromised, it would not weaken the security beyond the original setup, and vice versa.

[6]Keystore on Android and Secure Enclave on iOS.

7 CONCLUSION

In this work, we have shown that it is possible to efficiently execute a state-of-the-art threshold signature protocol FROST on currently available smartcards, even using only the restrictive interface that Java Card API provides. We achieved this by repurposing algorithms exposed by the API that are accelerated on the hardware level to compute operations required by the protocol. Our implementation is the first open-source threshold protocol implementation for Java Card smartcards supporting threshold values lower than the number of eligible parties, and it is compliant with the current version of the protocol standardization draft [10].

While our performance results are practical for many applications, an implementation utilizing a low-level interface to the underlying hardware could enable running the protocol significantly faster. A rough estimate could be made as the time required to create two regular signatures plus the time required to compute the group commitment (which is computed close to the native performance on J3R200 and J3H145 cards).

Furthermore, this work can attest to the practicality of FROST to the upcoming threshold scheme standardization process showing that even with current secure hardware devices, which were not designed to support threshold protocols, it is possible to execute FROST protocol on them. On the other side, this implementation's bottlenecks should not influence FROST design decisions because the cost of certain operations is magnified by the costly emulation via public API.

During the work on this paper, we made several improvements and extensions to the JCMathLib library [18] that are now merged in the upstream repository[7]. The changes focused on making the library more efficient and usable, supporting more recent smartcards, and allowing easy integration within other applets. We hope these efforts enhance research and development of proof-of-concept applications utilizing smartcards, and help the ecosystem to be more open.

ACKNOWLEDGMENTS

This research was supported by the Ministry of the Interior of the Czech Republic under grant VJ01010084 in program IMPAKT I. We also thank Jakub Janků for fruitful discussions on JCMathLib improvements.

[7]Available at https://github.com/OpenCryptoProject/JCMathLib.

REFERENCES

[1] Fatih Balli, F. Betül Durak, and Serge Vaudenay. 2019. BioID: A Privacy-Friendly Identity Document. In *Security and Trust Management: 15th International Workshop, STM 2019, Luxembourg City, Luxembourg, September 26–27, 2019, Proceedings* (Luxembourg, Luxembourg). Springer-Verlag, Berlin, Heidelberg, 53–70. https://doi.org/10.1007/978-3-030-31511-5_4

[2] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. 2012. High-speed high-security signatures. *Journal of cryptographic engineering* 2, 2 (2012), 77–89.

[3] Patrik Bichsel, Jan Camenisch, Thomas Groß, and Victor Shoup. 2009. Anonymous Credentials on a Standard Java Card. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '09)*. ACM, New York, NY, USA, 600–610. https://doi.org/10.1145/1653662.1653734

[4] Daniel R. L. Brown. 2009. Elliptic Curve Cryptography, Standards for Efficient Cryptography Group, ver. 2. https://www.secg.org/sec1-v2.pdf. Accessed: 2023-06-27.

[5] BSI. 2016. TR-03110 Technical Guideline Advanced Security Mechanisms for Machine Readable Travel Documents and eIDAS Token. https://www.bsi.bund.de/dok/TR-03110-en. Accessed: 2023-06-27.

[6] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. 2020. UC Non-Interactive, Proactive, Threshold ECDSA with Identifiable Aborts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. ACM, New York, NY, USA, 1769–1787. https://doi.org/10.1145/3372297.3423367

[7] Raúl Casanova-Marqués, Petr Dzurenda, and Jan Hajny. 2022. Implementation of Revocable Keyed-Verification Anonymous Credentials on Java Card. In *Proceedings of the 17th International Conference on Availability, Reliability and Security* (Vienna, Austria) *(ARES '22)*. ACM, New York, NY, USA, Article 140, 8 pages. https://doi.org/10.1145/3538969.3543798

[8] Zhiqun Chen. 2000. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide.* Addison-Wesley Longman Publishing Co., Inc., USA.

[9] Coinkite Inc. 2022. Dual Secure Elements. https://github.com/Coldcard/firmware/blob/master/docs/mk4-secure-elements.md. Accessed: 2023-06-27.

[10] Deirdre Connolly, Chelsea Komlo, Ian Goldberg, and Christopher A. Wood. 2023. *Two-Round Threshold Schnorr Signatures with FROST.* Internet-Draft draft-irtf-cfrg-frost-13. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-irtf-cfrg-frost/13/ (WIP).

[11] Ivan Damgård, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Jakob Illeborg Pagter, and Michael Bæksvang Østergaard. 2020. Fast Threshold ECDSA with Honest Majority. In *Security and Cryptography for Networks*, Clemente Galdi and Vladimir Kolesnikov (Eds.). Springer International Publishing, Cham, 382–400.

[12] Jack Doerner, Yashvanth Kondi, Eysa Lee, and Abhi Shelat. 2019. Threshold ECDSA from ECDSA Assumptions: The Multiparty Case. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 1051–1066. https://doi.org/10.1109/SP.2019.00024

[13] Manu Drijvers, Kasra Edalatnejad, Bryan Ford, Eike Kiltz, Julian Loss, Gregory Neven, and Igors Stepanovs. 2019. On the Security of Two-Round Multi-Signatures. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 1084–1101. https://doi.org/10.1109/SP.2019.00050

[14] Antonin Dufka, Vladimir Sedlacek, and Petr Svenda. 2022. SHINE: Resilience via Practical Interoperability of Multi-party Schnorr Signature Schemes. In *Proceedings of the 19th International Conference on Security and Cryptography*, Sabrina De Capitani di Vimercati (Ed.). SCITEPRESS, Portugal, 305–316.

[15] Chelsea Komlo and Ian Goldberg. 2021. FROST: Flexible Round-Optimized Schnorr Threshold Signatures. In *Selected Areas in Cryptography*, Orr Dunkelman, Michael J. Jacobson, Jr., and Colin O'Flynn (Eds.). Springer International Publishing, Cham, 34–65.

[16] Yehuda Lindell. 2022. How Smart Cryptography Makes Coinbase More Secure. https://www.coinbase.com/blog/how-smart-cryptography-makes-coinbase-more-secure. Accessed: 2023-03-30.

[17] Vasilios Mavroudis, Andrea Cerulli, Petr Svenda, Dan Cvrcek, Dusan Klinec, and George Danezis. 2017. A Touch of Evil: High-Assurance Cryptographic Hardware from Untrusted Components. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. ACM, New York, NY, USA, 1583–1600. https://doi.org/10.1145/3133956.3133961

[18] Vasilios Mavroudis and Petr Svenda. 2020. JCMathLib: Wrapper Cryptographic Library for Transparent and Certifiable JavaCard Applets. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, Genoa, Italy, 89–96. https://doi.org/10.1109/EuroSPW51379.2020.00022

[19] NIST. 2023. NIST First Call for Multi-Party Threshold Schemes. https://csrc.nist.gov/publications/detail/nistir/8214c/draft. Accessed: 2023-06-27.

[20] Oracle. 2018. Java Card Platform 3.1 Specification Release Notes. https://docs.oracle.com/en/java/javacard/3.1/specnotes/index.html. Accessed: 2023-06-27.

[21] Oracle. 2023. Java Card Platform 3.2 Specification Release Notes. https://docs.oracle.com/en/java/javacard/3.2/specnotes/index.html. Accessed: 2023-06-27.

[22] Oracle. 2023. Oracle Java Card technology. https://www.oracle.com/java/java-card/. Accessed: 2023-06-27.

[23] David Oswald. 2015. Repository jc_curve25519. https://github.com/david-oswald/jc_curve25519. Commit: fa65318c. Accessed: 2023-06-27.

[24] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (1979), 612–613.

[25] Smart ID. 2017. The smart way to identify yourself. https://www.smart-id.com/about-smart-id/. Accessed: 2023-06-27.

[26] Michaël Sterckx, Benedikt Gierlichs, Bart Preneel, and Ingrid Verbauwhede. 2009. Efficient implementation of anonymous credentials on Java Card smart cards. In *2009 First IEEE International Workshop on Information Forensics and Security (WIFS)*. IEEE, London, UK, 106–110. https://doi.org/10.1109/WIFS.2009.5386474

[27] Rene Struik. 2022. *Alternative Elliptic Curve Representations.* Internet-Draft draft-ietf-lwig-curve-representations-23. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-lwig-curve-representations/23/ (WIP).

[28] Petr Svenda, Rudolf Kvasnovsky, Imrich Nagy, and Antonin Dufka. 2022. JCAlgTest: Robust identification metadata for certified smartcards. In *Proceedings of the 19th International Conference on Security and Cryptography*, Sabrina De Capitani di Vimercati (Ed.). SCITEPRESS, Portugal, 597–604.

[29] Hendrik Tews and Bart Jacobs. 2009. Performance Issues of Selective Disclosure and Blinded Issuing Protocols on Java Card. In *Information Security Theory and Practice. Smart Devices, Pervasive Systems, and Ubiquitous Networks*, Olivier Markowitch, Angelos Bilas, Jaap-Henk Hoepman, Chris J. Mitchell, and Jean-Jacques Quisquater (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 95–111.

[30] Pim Vullers. 2014. *Efficient Implementations of Attribute-based Credentials on Smart Cards.* Ph. D. Dissertation. Radboud University Nijmegen.

[31] Boya Wang, Wouter Lueks, Justinas Sukaitis, Vincent G. Narbel, and Carmela Troncoso. 2023. Not Yet Another Digital ID: Privacy-preserving Humanitarian Aid Distribution. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 645–663. https://doi.org/10.1109/SP46215.2023.00174

[32] Lawrence C. Washington. 2008. *Elliptic Curves: Number Theory and Cryptography, Second Edition* (2 ed.). Chapman & Hall/CRC, USA.

[33] Pieter Wuille, Jonas Nick, and Anthony Towns. 2020. Schnorr signatures for secp256k1. https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki. Accessed: 2023-06-27.

[34] Pieter Wuille, Jonas Nick, and Anthony Towns. 2020. Taproot: SegWit Version 1 Spending Rules. https://github.com/bitcoin/bips/blob/master/bip-0341.mediawiki. Accessed: 2023-06-27.

[35] Lukáš Zaoral. 2023. *Automatic Performance Profiler for Security Analysis of Cryptographic Smart Cards.* Master's thesis. Masaryk University, Faculty of Informatics. https://is.muni.cz/th/v7l30/

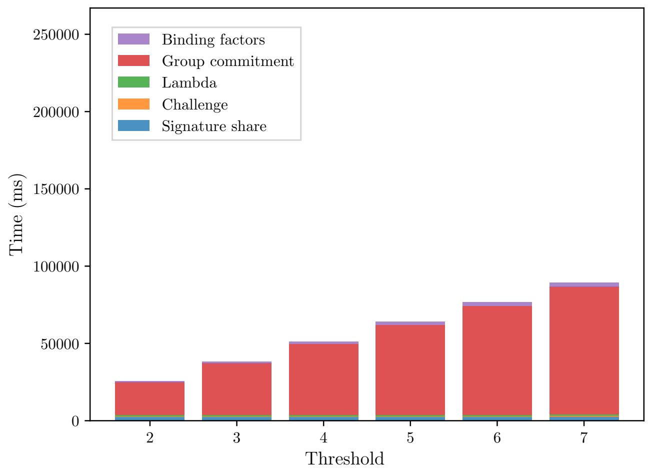
[36] ZenGo Ltd. 2022. The smart way to identify yourself. https://zengo.com/mpc-wallet/. Accessed: 2023-06-27.

A FURTHER MEASUREMENTS

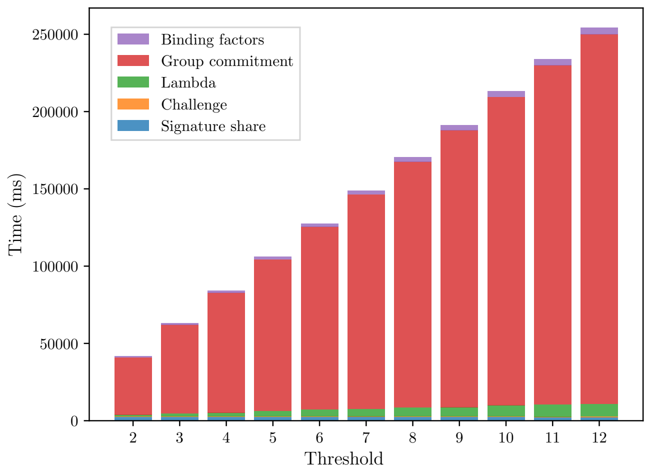
In this section, we present the decomposition of computation measurement for the remaining card with the optimized implementation (Figure 2a) and also for all the cards with the unoptimized implementation. Measurements of the unoptimized implementation were repeated only 10 times, as they take significantly longer to measure with the used approach.

Table 2: Time (ms) required to compute signature share with the unoptimized implementation depending on threshold t.

t	J3R200	J3H145	J2E145
2	3174	5041	41624
3	4676	7383	63184
4	6193	9739	84323
5	7662	12115	106090
6	9141	14450	127663
7	10652	16760	148783
8	12206	19072	170448
9	13620	21474	191292
10	15099	23816	213168
11	16711	26152	233887
12	18122	28467	254293

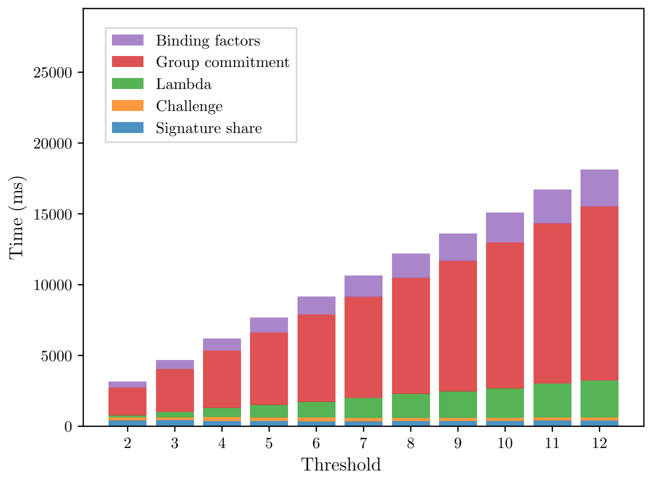


(a) NXP J2E145 (with optimizations)

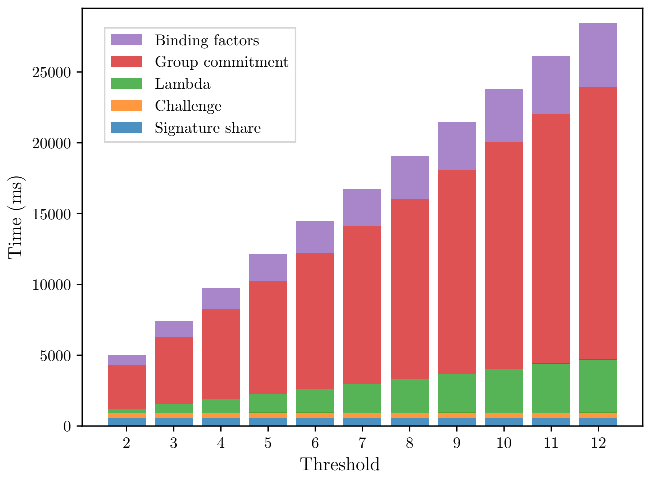


(b) NXP J2E145 (without optimizations)

Figure 2: Time measurement (in milliseconds) of the optimized (average of 100 runs) and the unoptimized (average of 10 runs) implementations of signing round components with different threshold values on NXP J2E145 smartcard (API version 3.0.1). Note that this card does not support efficient elliptic curve operations.



(a) NXP J3R200 (without optimizations)



(b) NXP J3H145 (without optimizations)

Figure 3: Time measurement (average of 10 runs, in milliseconds) of the unoptimized implementation of signing round components with different threshold values on smartcards that support efficient elliptic curve operations.

Table 2 shows the mean values of measurements of the unoptimized implementation depending on threshold values. By inspecting the performance composition closer, we can see in Figures 2b, 3a, and 3b that the point decoding has a tremendous impact on the computation time of the group commitment. Similarly, performing modular multiplications in $\mathbb{F}_q$ during the lambda computation significantly impacts the performance.