Is Java Card ready for hash-based signatures?

Ebo van der Laan[1], Erik Poll[2], Joost Rijneveld[2], Joeri de Ruiter[2], Peter Schwabe[2], and Jan Verschuren[1] ⋆

[1] Netherlands National Communication Security Agency (NLNCSA)
`ebo.laan@nlncsa.nl`, `jan.verschuren@nlncsa.nl`
[2] Digital Security Group, Radboud University, The Netherlands
`erikpoll@cs.ru.nl`, `joost@joostrijneveld.nl`, `joeri@cs.ru.nl`, `peter@cryptojedi.org`

Abstract. The current Java Card platform does not seem to allow for fast implementations of hash-based signature schemes. While the underlying implementation of the cryptographic primitives provided by the API can be fast, thanks to implementations in native code or in hardware, the cumulative overhead of the many separate API calls results in prohibitive performance for many common applications. In this work, we present an implementation of XMSS^{MT} on the current Java Card platform, and make suggestions how to improve this platform in future versions.

Keywords. Post-quantum cryptography, hash-based signatures, Java Card, XMSS^{MT}

1 Introduction

Over the past years, cryptographic schemes that promise resilience against quantum cryptanalysis have been getting more and more attention. With the start of NIST's Post-Quantum Standardization project [17] in 2017, the focus is starting to shift from an academic niche towards real-world use.

The most immediate concern with respect to quantum attacks is long-term confidentiality of data: an adversary that records ciphertext today may come back to decrypt it later, when sufficiently large quantum computers are available. Attacks against authentication, on the other hand, would require access to a quantum computer today. Nevertheless, one should be careful not to dismiss work towards practical applications of post-quantum authentication as premature. In this work, we show that even though standardization of post-quantum signature schemes is underway, widespread application still requires bridging serious gaps.

Perhaps the first images that come to mind when considering cryptographic software in the real world are large data centers full of servers that terminate TLS, full disk encryption on laptops, or intricate PKI systems. It is easy to forget that most people carry several cryptographic devices in their pockets: smart cards.

⋆ This work has been supported by the European Commission through the ICT program under contract ICT-645622 (PQCRYPTO), and by the Netherlands Organisation for Scientific Research (NWO) through Veni 2013 project 13114. Date: June 14, 2018

With estimates of over 10 billion[3] “secure elements” sold globally in 2018 [7], this is undeniably an important market.

Smart cards are often used as authentication token – in an asymmetric-key setting, the card then stores the private key and uses it to generate signatures. We examine a practical use case: setting up VPN connections using the popular OpenVPN application, for which we implemented a state-of-the-art post-quantum signature scheme. Specifically, we implement the hash-based signature scheme XMSS^{MT} [11,10] (described in Section 2) on the Java Card platform. XMSS^{MT} is a stateful signature scheme; a smart card implementation can conveniently record this state alongside the key material, hiding the complexity of the statefulness from the applications that use the functionality offered by the card. Section 3 describes the Java Card platform and OpenVPN use case in more detail.

We are not the first to implement hash-based signatures on a smart card. In 2013, Hülsing, Busold and Buchmann implemented a variant of XMSS on an Infineon-produced smart card [6]; their work makes even on-card key generation practical – something that cannot possibly be said of our implementation. This is done by building upon earlier work [18] where so-called ‘BDS traversal’ [5] was used on an 8-bit AVR, and expanding it to the multi-tree scheme that would later evolve into XMSS^{MT}. Crucially, their work uses low-level access to the underlying hardware, which is not publicly available or portable across manufacturers.

As alluded to earlier, and as is reflected by the question in the title of this paper, the results of this work are somewhat demoralizing. With signatures taking just shy of a minute (and a subsequent preparation step well over a minute and a half), for many use cases this is impractical; see Section 4.2 for a more detailed analysis. The main contribution of this work is clearly not to present speed records, but instead to provide a proof-of-concept and directions on how to improve the situation. Section 4 discusses our implementation of XMSS^{MT}; the issues we identify carry over into Section 5, where we provide suggestions for future improvements that could help make hash-based signature schemes more practical on the Java Card platform. The Java Card API has been extended in the past to support new protocols (notably the SAC/PACE protocol used in passports [14,3]), so we can expect future extensions when applications begin to require support for post-quantum cryptography.

Availability of software. We place all software presented in this paper into the public domain to maximize reusability of our results. It is available for download at `https://joostrijneveld.nl/papers/javacard-xmss`.

2 XMSS^{MT}

In [11], Hülsing, Rausch and Buchmann propose XMSS^{MT}, the current state of the art in (stateful) hash-based signatures. This scheme does not stand on its own, though, as hash-based signatures go back all the way to the 1970s when

[3] Half of these are SIM cards; financial and governmental applications make up most of the remainder.

they were first described by Merkle [16]. These schemes are characterized by their very conservative security assumptions, relying only on the existence of a secure one-way function; the provably minimal assumption required for any signature scheme to exist [19].

As research is progressing and the need for post-quantum cryptography is becoming more urgent, standardization efforts are starting to take shape. This is not only limited to the aforementioned project by NIST, but also ISO and the IETF have shown interest. The latter is relevant in particular, since at the time of writing a ‘Request For Comments’ describing XMSS and XMSS^{MT} [10] has just been published. Our implementation is compatible with XMSS^{MT} as specified in RFC 8391, and we refer to this document for a detailed technical specification. We limit the description in this section to that which is required as a preliminary for the discussions in the remainder of this paper.

2.1 WOTS$^+$

Before describing XMSS^{MT}, it is useful to separately define WOTS$^+$ [9], a variant of the Winternitz One-Time Signature (WOTS) scheme.

Parameters. WOTS$^+$ is a one-time signature scheme: a private key must not be used to sign more than one n-byte message, where n is a parameter defined at the time of key generation. Additionally, a parameter w signifies a trade-off between signature size and computation time.

To describe the resulting scheme, we use derived values ℓ_1, ℓ_2 and ℓ, defined as $\ell_1 := \lceil \frac{8n}{\log_2(w)} \rceil$, $\ell_2 := \lfloor \frac{\log_2(\ell_1 \cdot (w-1))}{\log_2(w)} \rfloor + 1$, and $\ell := \ell_1 + \ell_2$. For the remainder of this paper, we fix the parameters[4] $n = 32$ and $w = 16$. This leads to $\ell_1 = 64$ and $\ell_2 = 3$, and thus $\ell = 67$.

Keys. A WOTS$^+$ private key consists of ℓ random values of n bytes. In practice, these are derived from an n-byte seed using a pseudo-random generator. The corresponding public key is derived by applying a so-called chaining function F to the values in private key $w - 1$ times. The result consists of the ℓ chain heads (i.e. the last computed nodes) of n bytes each.

This public key can be compressed to an n-byte value by interpreting the ℓ heads as leaf nodes of a hash tree. Note that this tree is almost a binary tree; this so-called ℓ-tree is constructed by hashing two neighboring nodes to construct a parent node on a higher layer, and simply raising the last node to the next layer if the number of nodes on that layer is odd. The root of this tree is the de facto WOTS$^+$ public key, and we will refer to it as such.

Signatures. Assuming an n-byte message m, this is split into ℓ_1 chunks of $\log_2(w)$ bits, which are interpreted as integers m_1 to m_{ℓ_1}. The chaining function F is then applied m_i times to the i-th value of the private key, and the output is included

[4] One could consider $w = 4$, to speed up the computation at the cost of additional signature size. While the RFC [10] does not specify a specific parameter set, it does explicitly mention $w = 4$ as an option for this purpose.

as part of the signature. Given a message m and such outputs, verification means completing the chains by applying the chaining function an additional $w-1-m_i$ times and checking that these values combine to form the public key.

The careful reader will have noticed two issues: there were ℓ (and not ℓ_1) chain heads that make up the public key, and given a signature on a message m, it is easy to forge a signature for some messages m' by simply applying the chaining function $m'_i - m_i$ times[5]. To remedy this, WOTS$^+$ signatures include a checksum, computed by signing the base-w representation of $C = \Sigma_{i=1}^{\ell_1}(w-1-m_i)$, i.e. $(C_1, \ldots, C_{\ell_2})$. This prevents forgery, since an increase in any m_i results in a decrease in C and thus at least one C_j.

Functions. In the above description, we have left the chaining function F unspecified, and have not defined how the hash tree is constructed. These functions are instantiated using a tweaked variant of SHA-256 in the parameter sets we consider in this work[6]. To ensure collision resilience and to mitigate multi-function and multi-target attacks [13], each application of this function not only hashes the above-described input, but additionally includes a domain separator, a unique 'address' and a key, as well as applying a mask.

For ease of exposition, we omit the specifics of these constructions here, and only touch upon the relevant aspects in Section 5.

2.2 Hash trees

Having established WOTS$^+$ as a one-time signature scheme, we now expand this into the many-time signature scheme XMSS [4]. In essence, XMSS consists of many instances of WOTS$^+$ and a hash tree to authenticate them.

Keys. Consider a binary hash tree of height h, i.e., a tree with 2^h leaf nodes. We associate a WOTS$^+$ key pair with each leaf, allowing for 2^h signatures. The XMSS private key simply provides a seed from which to generate the WOTS$^+$ private keys; the WOTS$^+$ public keys are then derived by applying the chaining function, as described above. Then, by computing a binary hash tree on top of the WOTS$^+$ public keys, one derives the XMSS public key: the root node of the tree.

Signatures. By the above construction, it is straight-forward to see that an XMSS signature mostly consists of a WOTS$^+$ signature, complemented by an index to indicate which WOTS$^+$ private key was used. However, the verifier does not hold the corresponding WOTS$^+$ public key required to verify the signature. Instead they compute what the WOTS$^+$ public key should be, based on the presented signature – note that this process is exactly the same as verifying a WOTS$^+$ signature, omitting actual comparison to the public key. This effectively gives the verifier one leaf node in the hash tree. To compare to the root node (i.e. the XMSS public key), the verifier requires nodes along the path to the root of the tree. This path is referred to as the 'authentication path', and it can be seen that the signer must include h additional nodes. See Figure 1.

[5] This requires that $m'_i \geq m_i$ for all i, but this is sufficiently likely even for random m.
[6] A common and often more natural instantiation relies on the Keccak-based SHAKE [2]

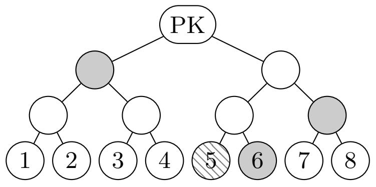

Fig. 1. The authentication path to authenticate the fifth leaf is shown in grey. [12]

The state. When introducing the notion of many WOTS$^+$ key pairs linked to one XMSS public key, it is crucial to prevent re-using these one-time key pairs. Conceptually, this is trivially accomplished by iterating through the leaf nodes sequentially. It is important to remark, however, that this implies maintaining a persistent *and changing* state across different signing operations. Depending on the specific usage scenario, this may or may not be a problem – it adds complexity in settings where a key is used by multiple processes or needs to be kept in sync across servers, but is not a concern when a key is embedded in a single system. The latter scenario describes the use case discussed in this paper. In [6], the authors demonstrate that there is a strong synergy between this property and achieving forward security (but this is not part of the 'standard' XMSS scheme).

2.3 Chaining trees

In order to be able to perform many signatures using the same public key, one could instantiate XMSS with a large tree. This comes at a considerable cost, as the signer needs to compute all leaf nodes when generating a signature. Specialized tree traversal algorithms [5] move a large part of this cost to the key generation, but it remains a limiting factor. This is mitigated in XMSS^{MT}, the multi-tree variant of XMSS. As the name suggests, this scheme makes use of a structure of trees.

On the bottom layer, a WOTS$^+$ key pair is used to sign the message. Along with the WOTS$^+$ signature, the signer supplies an authentication path to the root of that subtree. Rather than interpreting this root as the public key, it is signed using a WOTS$^+$ leaf of a new tree, one layer 'above' the current layer. This signature is authenticated by a path leading to the next root node, et cetera.

Considering d layers of trees of height h/d, this allows for 2^h signatures while only requiring h/d leafs on each layer to be computed to construct the authentication path (as well as opening up a whole new range of time-memory trade-offs with tree traversal [5]).

Note that this trade-off leads to increased signature sizes. While an XMSS signature consists roughly of a WOTS$^+$ signature (i.e. $67 \cdot 32$ bytes) and a number of intermediate nodes (say, $20 \cdot 32$ bytes), an XMSS^{MT} signature consists of multiple WOTS$^+$ signatures. For the sake of simplicity, we now consider XMSS^{MT} to be a direct generalization of XMSS, i.e. XMSS is the specific class of instances where $d = 1$.

3 Java Card platform and limitations

Java Card defines a standardized, vendor-independent programming platform for multi-application smart cards produced by different manufacturers. While the specification is controlled by Oracle, many of the large smart-card manufacturers[7] collaborate in the 'Java Card Forum' [8] in defining the platform. The platform has proven popular, with over 20 billion cards sold at the time of its twentieth anniversary in 2016 [20]. Java Card is often found in SIM cards and passports.

As the name suggests, Java Card is based on Java, but with many language features restricted due to the limited resources. This shows prominently in the limited availability of types – a Java Card platform is only required to support 8-bit `bytes` and 16-bit `shorts`. Similarly, Java Card inherits the class-based object-oriented style of Java, but using objects is discouraged because of size constraints; moreover, garbage collection is optional for Java Card.

The APIs for Java and Java Card differ vastly. The Java Card API is extremely limited, but does provide a range of high-level methods for standard cryptographic use cases (e.g. signature generation, key storage, block encryption). This enables developers to quickly construct applets to perform basic cryptographic operations. The implementation of the API is left to the smart-card manufacturer, allowing implementations in native code or directly in hardware. This is crucial for performance: the Java Card VM introduces considerable overhead, so implementing cryptographic primitives in Java Card bytecode would be unacceptably slow. Still, considerable overhead remains when calling these API functions, and this turns out to be a recurring theme in the rest of this paper.

An important consideration is the limited amount of memory. Typical Java Cards have in the order of tens of KiB persistent memory (EEPROM or Flash), but the transient (RAM) memory is typically only a few KiB, which is a serious bottleneck. Memory sizes can vary significantly between cards, so memory requirements should be carefully taken into account when developing applications.

Java Card is compatible with the ISO 7816 standard. This means that communication is done using APDUs (Application Protocol Data Units). These traditionally support a payload of up to 256 bytes, although recent cards support extended-length APDUs allowing longer payloads.

In this work, we focus on compatibility with Java Card version 2.2.2 to 3.0.4.

3.1 Considerations for the OpenVPN use case

This work was done as part of a project involving a Java Card applet to provide authentication when establishing a VPN connection, tightly integrated into OpenVPN. The projected benefit of this was twofold: increased security and increased usability. Smart cards typically provide much more secure storage of the key material. By selecting the Java Card platform, the cross-platform applet can be easily combined with existing deployed systems. The tight integration with

[7] At the time of writing, the Java Card Forum consists of Gemalto, Giesecke & Devrient, IDEMIA, Infineon, jNet ThingX, NXP Semiconductors and STMicroelectronics [8]

OpenVPN aims to improve the user experience: we avoid third-party middleware (which would be required for the use of more generic solutions, such as hardware tokens relying on standards like PKCS#11) and store the configuration files for OpenVPN on the card to simplify the setup process for the user.

This use case implies a set of assumptions and limitations. There is some margin in terms of signing time, as signing operations are fairly infrequent and users would expect some latency when establishing a connection. More importantly, the required throughout is low: after signing once, typical usage scenarios suggest a period of time during which the card is connected and powered, but not used to produce a new signature. Furthermore, we note that key generation can be done during issuance, and even outside of the card (assuming a secure issuance environment – this is a reasonable assumption given that initialization also involves, e.g., PIN codes). In principle, there is a nice match between these properties and the XMSS^{MT} signature scheme. There are many time-memory trade-offs that can be flexibly tweaked, and there is ample opportunity for precomputation either during key generation or idle time. However, it is important to reiterate that memory (in particular the fast RAM) is a scarce resource on the card. The next section details these trade-offs.

4 Implementation

When designing a smart card application, it is important to consider natural 'commands' that divide up and structure the computation. For a traditional RSA-2048 or ECC signature, signing a message could be a single command with a single APDU as response. For XMSS^{MT}, signatures are several kilobytes in size and must be spread out over multiple 256-byte response APDUs. This behavior is typical for hash-based signatures on small devices [12]; they are too large to comfortably fit in RAM but are very sequential in their construction, strongly suggesting an interface where the signature is streamed out incrementally.

There is much repetition of small subroutines to be found in the scheme. After initializing the signing routine by computing a message digest, a signature consists of a sequence of WOTS$^+$ signatures and authentication paths. Internally, the WOTS$^+$ signatures can be decomposed further into their separate chains. The $\ell = 67$ chains split naturally into 8 sets of 8 chains for the $\ell_1 = 64$ message digest chains, and $\ell_2 = 3$ chains for the checksum. For hashes of 32 bytes and $h/d \leq 8$, authentication paths within a subtree fit into one response APDU, and choosing $h/d > 8$ is not realistic on this platform because of resource constraints[8]. In order to reduce the latency of signature generation, we ensure that all relevant leaf nodes for the authentication path in each subtree on each layer are available in memory. We address this later in this section, and for now only note that maintaining this invariant introduces a preparation step after a signature is produced (and thus: a leaf node is consumed).

[8] This would imply either computing or storing hundreds of WOTS$^+$ leaf nodes per tree layer.

Figure 2 represents these states visually. Note that each state is triggered by a command APDU, of which only the initial command contains auxiliary data (i.e. the message). These have been omitted for simplicity.

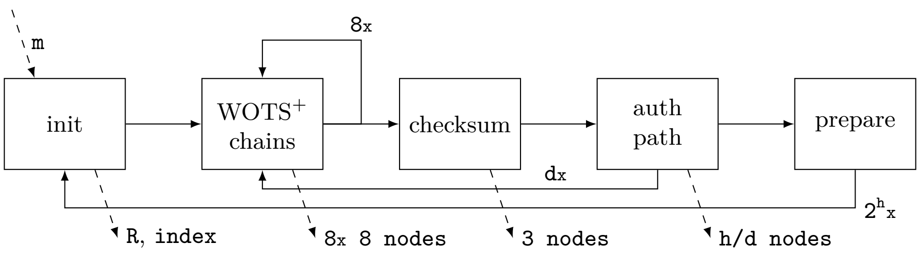

Fig. 2. State diagram of the signing routine.

Indices. As it is crucial that the smart card cannot be coerced into re-using a leaf, the first operation should be incrementing the state index. Because Java Card does not guarantee a native 32-bit integer type, all indices are stored as tuples of two `shorts`, interpreted as 15-bit unsigned values (effectively ignoring the high bit). As a consequence, atomic increments are not possible without use of expensive transactions, and special care has to be taken in case of overflows – the conservative approach skips 2^{15} leafs in case of card tear[9], rather than rolling back. Similar considerations apply when deriving indices of 'next' and 'previous' nodes during state generation. As we have limited h/d previously, internal tree indices can be represented with a single `short`.

WOTS$^+$ leaf generation. To generate a WOTS$^+$ leaf, an ℓ-tree must be computed over the heads of all chains. As memory is limited, the natural choice here is to use the treehash algorithm [16, Section 7]. Since $\ell = 67$ is not a power of 2, bringing the tree out of balance, there are some special cases to consider. As the value of ℓ is constant for all parameters we account for, this can be simplified by manually handling these special cases after performing treehash. Altogether, this ensures that we require only 416 bytes of RAM for intermediate results when deriving a WOTS$^+$ public key.

State (re)generation. At least one WOTS$^+$ computation needs to be performed whenever a message is signed: exactly when signing the message digest. Without proper state management, however, one would be required to compute $d \cdot 2^{h/d}$ WOTS$^+$ leafs to derive the authentication paths. Instead, we keep a persistent array of the leaf nodes of the current tree on each of the d layers. If the secret key is generated off-card, the leaf nodes of the first trees can be preloaded; alternatively, they can be computed during issuance. Similarly, the $d-1$ WOTS$^+$ signatures that join the subtrees together can also be precomputed and cached.

[9] The physical attack of interrupting the power supply to the card, e.g., by removing it from the reader

By keeping an additional array of such nodes and signatures for the ‘next’ tree on every layer[10] and computing one new node whenever one is consumed, it can be easily seen that we are guaranteed to always have all leafs available before they are consumed. Note that this introduces an imbalance in signing time cost (as consuming indices that introduce new nodes on multiple layers adds linear leaf generation cost), but that this computation be performed *after* outputting a signature. Careful administration is required to guarantee that this is not neglected. Intuitively, one might consider decoupling the signing and preparation step, and allow the signature routine to effectively consume the nodes up to the point at which they were prepared. While this is certainly possible, the involved bookkeeping is more complicated than it may seem at first: memory requirements imply re-using arrays, the leafs currently in use cannot be overwritten, and the next layer of leafs needs to be completed precisely when switching to the next subtree. Verifying these conditions combines poorly with the convoluted arithmetic on tuples of `short`s that represent indices.

4.1 Hash functions

Performance is dominated by the cost of a call to the chaining function in WOTS$^+$ and the hash function in the binary trees. In essence, these functions consist of many applications of SHA-256 to small arrays of data (i.e. 32 to 128 bytes) and some `xor` operations. This is not a particularly common pattern of operations in traditional cryptography – a signature operation typically requires just one hash function call to digest the message, often negligible in the overall performance of the signing operation. Note also that there is significant cost associated with a single call to a hash function that is constant in the length of the input, likely representing the overhead of the function call, as shown in Table 1.

AES-based hashing. Instead of using a cryptographic hash function as a building block for the described functions, a block cipher can be used to construct a similar primitive using common constructions such as Davies-Meyer and Matyas-Meyer-Oseas (the latter being used in [6]). Some care would need to be taken to transform these to a security level equivalent to the second pre-image resistance derived from SHA-256 in the context of XMSS^{MT}. This would break compatibility with the RFC [10], but in principle this is not unsurmountable.

Some Java Cards appear to be equipped with an AES implementation in hardware, speeding up its performance significantly. This is evidenced by an even larger unbalance between constant and variable costs: encrypting large blocks of data is only marginally more costly than smaller blocks, as shown in Table 2. The base cost of a single call to AES is still significant, however, putting the performance in the same ballpark as SHA-256 on short inputs. Note that these numbers cannot be directly compared to the cost of SHA-256 as listed in Table 1, as multiple iterations of AES would be required for one compression block.

[10] This is only required on layers where there is still a ‘next’ tree, which is trivially false for the top-most tree.

Table 1. 1000 iterations of SHA-256

data (bytes)	32	64	128	256
runtime (seconds)	3.94	5.83	8.02	12.40

Table 2. 1000 iterations of AES-128 in ECB mode

data (bytes)	16	32	64	128	256	1024
runtime (seconds)	2.97	3.30	3.96	5.30	7.97	23.87

There is another avenue to explore when relying on AES as a primitive, as the Java Card API supports a range of modes of operation for AES. Combining this with the fact that we process a large amount of data at once suggests opportunities for parallel data streams; encrypting a large data stream using AES in ECB mode is functionally equivalent to performing independent AES encryption in parallel – under the same key. This last restriction is crucial, as a message-dependent block is used as key, ruling out precisely the constructions available to turn AES into a compression function. Other modes of operation suffer a similar faith. As a result, there is no clear way to exploit the available AES implementation for parallel data streams.

4.2 Memory usage and benchmarks

This section outlines the performance when running the applet on a Java Card. For this, we performed measurements and ran tests on NXP-produced JCOP cards, as well as a card of unclear origin (`ICFabricator=0005`). While this is somewhat indicative of relative performance, we note that measurements may vary wildly when comparing different cards by different manufacturers. Tables 1 and 2 give the individual benchmarks for the primitives on the cards we used.

For a WOTS$^+$ signature operation with the parameters described in Section 2.1, we measure an average time of approximately 33 seconds. In the best case, the preparation step requires one WOTS$^+$ key generation, which requires approximately a minute.

When we consider a realistic parameter set, where $h = 20$ and $d = 4$, i.e. subtrees with 32 leaf nodes, we notice that the cost of authentication path generation starts to come into play. In particular, the access to nodes stored in persistent memory makes this more costly than a back-of-the-envelope computation would predict[11]. For these parameters, a signature takes roughly 54 seconds in the best case: every 32nd signature adds an additional WOTS$^+$ signature generation, every 256th signature adds two WOTS$^+$ signatures, et cetera. Similarly, preparation takes 85 seconds in the best case. Varying to $d = 5$ results in a slightly shorter

[11] A WOTS$^+$ signature costs 536 applications of the chaining function on average, versus 63 hash function calls in the tree.

signing time, coming in at 50 seconds in the best case (but more frequently requires new WOTS$^+$ signatures).

Besides a small number of bytes to store the keys and index, the requirements on persistent memory follow from the storage of WOTS$^+$ signatures and leaf nodes: $32 \cdot \ell \cdot (d-1)$ bytes for the WOTS$^+$ signatures, and $32 \cdot (2 \cdot d - 1) \cdot 2^{\frac{h}{d}}$ bytes for the leaf nodes. For $d = 4$, this comes down to $6432 + 7168 = 13600 = 13.28\,\text{KiB}$. Similarly, for $d = 5$, this adds up to $8576 + 4608 = 13.18\,\text{KiB}$. Note that increasing d also increases signature size by additional WOTS$^+$ signatures, but decreasing d while maintaining $h = 20$ sharply increases the memory requirements for node storage, as well as the cost of (off-card) key generation.

Considering the signing states described in Section 4, in particular in Figure 2, it can be easily seen that the signature is output in stages as computation progresses. With the WOTS$^+$ chain computation taking up most of the computation, splitting this over eight APDUs levels out communication costs.

5 Java Card API recommendations and considerations

In the previous section, we touched upon several issues with implementing XMSS^{MT} (and hash-based signatures in general) using the current Java Card API (i.e. version 3.0.5 or below). This section discusses potential extensions to improve support for hash-based signatures. In the past the Java Card API has been extended to support new cryptographic algorithms[12]. If and when hash-based signatures become widely used in the future, one would expect extensions of the API for this, either as proprietary extensions of manufactures or ultimately as extensions of the standard.

An important design choice in such an API is the level of abstraction. One can opt for low-level methods providing more fine-grained (i.e. more primitive) operations, or for higher levels of abstractions, where the API methods provide bigger building blocks, or possibly even a complete signatures scheme[13]. Below we present four alternatives with an increasing level of abstraction.

Generally speaking, a more fine-grained API is likely to be easier to implement for manufacturers and offers more flexibility to applet developers. On the other hand, higher level, more monolithic API methods make it easier for developers that are less versed in the relevant cryptography to make the correct choices, allow for faster implementation in hardware, and enable manufacturers to provide more comprehensive side-channel countermeasures. Also, an API implementation may need memory to record state between API calls and scratchpad memory to record temporary results. Given that transient memory is extremely scarce, it is not acceptable that API methods need large amounts of RAM.

[12] For example, version 3.0.5 introduces support for SAC/PACE [14,3], a protocol used in electronic passports.

[13] For example, in the case of the PACE protocol, the choice has been made not to provide a generic API method for elliptic curve point addition, which would enable applet developers to implement PACE, but rather to provide more higher-level operations to directly provide PACE as primitive.

Another factor to take into consideration when making an abstraction level trade-off is the fact that standardization efforts are still ongoing, and a high-level API leads to less agility to account for future scheme changes.

5.1 Parallel hashing

Performance of hash-based signatures is completely dependent on the ability to efficiently compute many hash digests over small amounts of data. While this can be sped up by implementing the hash function in hardware, Section 4.1 illustrates that this is only part of the solution. More critically, the execution time depends on being able to exploit the extreme levels of parallelism that are inherent to hash-based signatures. Here parallelism does not necessarily imply parallel execution, but rather independent parallel data streams.

The current interface to hash functions is provided in the form of the `MessageDigest` class. After instantiating an object for a specific digest function, say SHA-256, a user can add additional data by calling the `update(byte[] inBuff, short inOffset, short inLength)` method, and obtain the final digest by calling `doFinal(byte[] inBuff, short inOffset, short inLength, byte[] outBuff, short outOffset)`.

We propose duals of these methods, following an almost identical API: `updateParallel(byte[] inBuff, short inOffset, short inBlockLength, short numberOfBlocks)`, and `doFinalParallel(byte[] inBuff, short inOffset, short inBlockLength, short numberOfBlocks, byte[] outBuff, short outOffset)`. Here, `inBuff` provides `numberOfBlocks` sequential inputs of `inBlockLength` bytes, and output is written to `outBuff` analogously.

Providing an inconsistent number of inputs (i.e. different `numberOfBlocks`) for `update` and `doFinal` calls could be treated as an error but it may be beneficial to instead fix the `numberOfBlocks` at the time of construction of the `MessageDigest` object. For hash-based signatures this decision is equivalent, as the relevant hash function calls all require arguments of the same form. Both options have serious effects on the underlying implementations, as these modifications suggest maintaining a (runtime-determined) number of intermediate hash function states. If this proves to be infeasible, a natural restriction would be to drop the parallel `updateParallel` method[14]. While this reduces flexibility for the applet developer, in particular when memory is constrained and rearranging input is costly, this allows underlying hardware to sequentially process each instance of the hash function without maintaining a variable-length intermediate state in addition to the caller-provided input and output buffers. This does not contradict the goal of achieving a speedup through internal parallelism, as the majority of the cost can be attributed to the Java stack on top of the underlying implementation (see Section 4.1 and 4.2). As a result, implementations that would support a parallel `update` method would still likely opt for a sequential underlying hashing primitive to reduce area cost.

[14] It is also possible to reach a similar invariance by fixing `numberOfBlocks`, but this still requires multiple hash-function intermediate states in transient memory.

5.2 Complete WOTS$^+$ chains

Rather than providing a narrow API that allows a developer to efficiently make use of the underlying hash function primitive, the parallelism can be made transparent to the implementer through a more abstract API: computing WOTS$^+$ chains and authentication paths.

In the WOTS$^+$ chains, there is a lot of opportunity for shared execution. Besides the natural 'horizontal' parallelism across many chains (which would be the primary candidate for optimizations discussed in the previous subsection, 5.1), there is potential gain in coupling the 'vertical' computations that take place during WOTS$^+$ public key and signature generation. On top of the benefits achieved from only passing through the Java stack once, rather than repeatedly for every application of the chaining function, the input to many of the underlying SHA-256 compression function calls overlaps significantly. In particular, for a single WOTS$^+$ key pair, the input to the first compression call is completely identical across all $16 \cdot 67 = 1072$ calls of the chaining function. Note that this is a consequence of the specific instantiation of the compression function in XMSS^{MT} as defined in [10], however, and does not immediately carry over to other function designs (in particular, the SPHINCS+ proposal [1] to the NIST standardization project [17] does not benefit from this).

Such an API goes beyond a straight-forward parameter for the hash function specifying the number of iterations, as the iterated function is not simply SHA-256, but rather the address- and key-aware chaining function. Furthermore, to make it effective for WOTS$^+$ signature generation, it would require specifying the length of each individual chain, as well as a variable number of chains (as an entire WOTS$^+$ signature will likely not fit in RAM on most Java Cards).

This middle ground between abstracting away the parallelism of the hash functions but still requiring (or, indeed, allowing) the developer to puzzle together the pieces has its upsides, but is clearly not without added complexities. We stress that the API of such a hybrid solution needs to be carefully thought through to be sufficiently fine-grained to provide a benefit over an all-in-one API (as described later, in Section 5.4), yet convenient to use so that it actually reduces boilerplate code and development overhead when compared to a more straight-forward parallel hashing API.

5.3 WOTS$^+$ nodes and hash trees

Another unit of abstraction is a hash tree. In XMSS^{MT}, there are two specific instances of hash trees: the tree in XMSS, and the ℓ-tree in the WOTS$^+$ nodes.

The computation of WOTS$^+$ nodes can be hidden behind an API with relative ease. Given its position in the hypertree and the secret seed, the only relevant output is the root node of the ℓ-tree, easily fitting a single APDU (and thus appropriate as the result of a single function call). We note that in the SPHINCS+ proposal, ℓ-trees have been eliminated altogether. It is not inconceivable that future updates to XMSS will include the same change.

Abstracting the hash trees in the hypertree behind a single function is somewhat more complicated. The reason for this is twofold. First and foremost, preventing recomputation of such trees is crucial to make XMSS^{MT} practical, which implies carefully maintaining a state (either by storing leaf nodes, as is done in the current implementation, or through more involved tree traversal techniques [5]). This introduces a time/memory trade-off that strongly depends on the parameter choice – allowing more flexibility in terms of tree height and multi-tree depth significantly increases complexity of the underlying implementation. Secondly, as the relevant output comes in the form of an authentication path of multiple nodes, APDU size (and thus state machine management) becomes relevant as soon as $h > 8$.

Conversely, there is much to gain in terms of simplicity for the user if this is abstracted, as this prevents the users from having to re-implement the treehash algorithm and make complex state management decisions. We argue that this is a crucial requirement for non-expert usage.

5.4 Complete XMSS^{MT} signatures

At the end of the spectrum, we consider an API that abstracts away as much of the scheme's internals as possible. Intuitively this matches the current approach of the Java Card API for public-key primitives; given a parameterized and keyed object and a message, there is a single API call that produces a signature. To allow for longer messages, an `update` mechanism is available similar to how message digests work (see Section 5.1). Crucially, this is made possible by the small size of signatures; for typical parameters, the resulting signature easily fits in RAM and even in a single output APDU.

When considering the multiple kilobytes of a typical XMSS^{MT} signature, such an API suggests writing the signature to persistent memory. This requires additional EEPROM/Flash and adds the extra cost of slow memory access. However, this is likely to compare favorably when considering the potential for performance improvement by implementing the entire scheme natively.

Alternatively, the API could be split up in a similar way as is done in this implementation; we refer to the states described in Figure 2 – each state could represent an API call. This would still require the applet developer to implement the state machine, but makes conversion to output APDUs more natural.

Perhaps the most compelling argument for this high-level API is usability for applet developers. XMSS^{MT}, and tree traversal in general, is administratively notoriously tedious, and wrongly managing indices can easily lead to degraded security. In particular, a high-level API is required to properly abstract the state preparation step, as this would otherwise heavily depend on implementation choices (i.e. what part of the state is cached, and how it is iterated). Ease of use should not be underestimated as a critical factor towards adoption in real-world applications.

5.5 Side-channel countermeasures

Smart cards are a common target for physical attacks. To remedy this, manufacturers commonly implement a wide variety of platform-specific countermeasures. An API that abstracts away the usage of secret data is paramount for this to be effective. This requirement aligns well with the considerations of the rest of this section when considering the simplicity of the API exposed to the applet developer: a fine-grained API that requires the developer to implement the overarching scheme creates many potential pitfalls. To illustrate, the current lack of API required us to abuse the `AESKey` object to store sensitive key material in EEPROM, extracting it into RAM before use (although more recent versions of Java Card provide the `SensitiveArray` class for this purpose). Similarly, without API support, the expanded WOTS$^+$ seeds live plainly in transient memory. While in general hash-based signatures have a history of robustness against side-channel attacks, it is precisely this usage of the PRF that has recently been under scrutiny [15].

References

1. Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, and Peter Schwabe. SPHINCS+. Submission to NIST's post-quantum crypto standardization project, 2017. https://sphincs.org. 13
2. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The Keccak reference, January 2011. http://keccak.noekeon.org/. 4
3. Advanced Security Mechanisms for Machine Readable Travel Documents and eIDAS Token. Technical Report TR-03110, German Federal Office for Information Security (BSI), 2015. Version 2.20. 2, 11
4. Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS - a practical forward secure signature scheme based on minimal security assumptions. In Bo-Yin Yang, editor, *Post-Quantum Cryptography*, volume 7071 of *LNCS*, pages 117–129. Springer, 2011. https://eprint.iacr.org/2011/484. 4
5. Johannes Buchmann, Erik Dahmen, and Michael Schneider. Merkle tree traversal revisited. In Johannes Buchmann and Jintai Ding, editors, *Post-Quantum Cryptography*, volume 5299 of *LNCS*, pages 63–78. Springer, 2018. https://www.cdc.informatik.tu-darmstadt.de/reports/reports/AuthPath.pdf. 2, 5, 14
6. Christoph Busold, Johannes Buchmann, and Andreas Hülsing. Forward secure signatures on smart cards. In Lars R. Knudsen and Huapeng Wu, editors, *Selected Areas in Cryptography – SAC 2012*, volume 7707 of *LNCS*, pages 66–80. Springer, 2013. https://huelsing.files.wordpress.com/2013/05/xmss-smart.pdf. 2, 5, 9
7. Eurosmart. Digital security industry to pass the 10 billion mark in 2018 for worldwide shipments of secure elements. Press Release, 2017. http://www.eurosmart.com/news-publications/press-release/296. 2
8. Java Card Forum. About the JCF, 2018. https://javacardforum.com, accessed 2018-03-12. 6

9. Andreas Hülsing. W-OTS+ – shorter signatures for hash-based signature schemes. In Amr Youssef, Abderrahmane Nitaj, and Aboul-Ella Hassanien, editors, *Progress in Cryptology – AFRICACRYPT 2013*, volume 7918 of *LNCS*, pages 173–188. Springer, 2013. https://eprint.iacr.org/2017/965. 3
10. Andreas Hülsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. XMSS: eXtended Merkle Signature Scheme. Request for Comments 8391, 2018. https://tools.ietf.org/html/rfc8391. 2, 3, 9, 13
11. Andreas Hülsing, Lea Rausch, and Johannes Buchmann. Optimal parameters for XMSS^{MT}. In Alfredo Cuzzocrea, Christian Kittl, Dimitris E. Simos, Edgar Weippl, and Lida Xu, editors, *Security Engineering and Intelligence Informatics*, volume 8128 of *LNCS*, pages 194–208. Springer, 2013. https://eprint.iacr.org/2017/966. 2
12. Andreas Hülsing, Joost Rijneveld, and Peter Schwabe. ARMed SPHINCS – computing a 41KB signature in 16KB of RAM. In Giuseppe Persiano and Bo-Yin Yang, editors, *Public Key Cryptography – PKC 2016*, volume 9614 of *LNCS*, pages 446–470. Springer, 2016. https://eprint.iacr.org/2015/1042. 5, 7
13. Andreas Hülsing, Joost Rijneveld, and Fang Song. Mitigating multi-target attacks in hash-based signatures. In Giuseppe Persiano and Bo-Yin Yang, editors, *Public Key Cryptography – PKC 2016*, volume 9614 of *LNCS*, pages 387–416. Springer, 2016. https://eprint.iacr.org/2015/1256. 4
14. Supplemental Access Control for Machine Readable Travel Documents. Technical report, International Civil Aviation Organization (ICAO), 2014. Version 1.1. 2, 11
15. Matthias J Kannwischer, Aymeric Genêt, Denis Butin, Juliane Krämer, and Johannes Buchmann. Differential power analysis of XMSS and SPHINCS. In *Constructive Side-Channel Analysis and Secure Design – COSADE 2018*, LNCS. Springer, 2018. https://kannwischer.eu/papers/2018_hbs_sca.pdf, to appear. 15
16. Ralph Merkle. A certified digital signature. In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO '89*, volume 435 of *LNCS*, pages 218–238. Springer, 1990. www.merkle.com/papers/Certified1979.pdf. 3, 8
17. NIST. Post-quantum cryptography: NIST's plan for the future, 2016. http://csrc.nist.gov/groups/ST/post-quantum-crypto/documents/pqcrypto-2016-presentation.pdf. 1, 13
18. Sebastian Rohde, Thomas Eisenbarth, Erik Dahmen, Johannes Buchmann, and Christof Paar. Fast hash-based signatures on constrained devices. In Gilles Grimaud and François-Xavier Standaert, editors, *Smart Card Research and Advanced Applications (CARDIS'08)*, volume 5189 of *LNCS*, pages 104–117. Springer, 2008. https://www-old.cdc.informatik.tu-darmstadt.de/reports/reports/REDBP08.pdf. 2
19. John Rompel. One-way functions are necessary and sufficient for secure signatures. In *Proceedings of the twenty-second annual ACM symposium on theory of computing*, pages 387–394. ACM, 1990. https://www.cs.princeton.edu/courses/archive/spr08/cos598D/Rompel.pdf. 3
20. Safran Identity & Security. The impact of Java Card technology yesterday and tomorrow: Safran Identity & Security celebrates 20 years with the Java Card Forum. Press Release. https://www.morpho.com/en/media/impact-java-card-technology-yesterday-and-tomorrow-safran-identity-security-celebrates-20-years-java-card-forum-20170302, accessed 2018-03-12. 6