Surnaming Protocols, Fast Verification, and Applications to SGX Technology Preprint

Preprint of
Dan Boneh and Shay Gueron. Surnaming schemes, fast verification, and applications to SGX technology, in Topics in Cryptology - CT-RSA 2017 - The Cryptographers' Track at the RSA Conference 2017, San Francisco, CA, USA, February 14-17, 2017, Proceedings, 2017, pp. 149:164
https://link.springer.com/chapter/10.1007/978-3-319-52153-4_9

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

ABSTRACT

We introduce a new cryptographic primitive that we call *Surnaming*, which is closely related to digital signatures, but has a different syntax and security requirements. We study the properties of the Surnaming primitive and show that, while it can be constructed from a digital signature, a direct construction can be somewhat simpler.

Surnaming plays a central role in Intel's new Software Guard Extensions (SGX) technology, and we present the specific SGX Surnaming implementation as a special case. This explains why SGX does not require a PKI or pinned keys for its particular usage.

SGX motivates an interesting question in digital signature design: for reasons explained in the paper, SGX requires a digital signature scheme where the verification time must be minimized, the public key must be short, but signature size is unimportant. We review the RSA-based method that is currently used by SGX, and discuss its security and efficiency to show how this design meets the requirements.

Finally, we propose a new signature scheme where verification time is about 40% faster than the current SGX scheme. Our new scheme, which employs a hash-based signature, can be scaled to also provide post-quantum security. It offers a viable alternative to the current Surnaming scheme used by SGX, if post-quantum security becomes a desired goal.

1. INTRODUCTION

Intel has recently introduced a powerful security architecture called Software Guard Extensions (SGX for short) that enables one to run a "secure enclave" (application) on the processor, so that nothing else running on the processor can access the enclave's memory. The root of trust is the processor itself – even memory is considered untrusted and all reads and writes to memory are encrypted with integrity and replay protection. This technology enables applications to operate on secret data without fear of compromise. SGX is available on the 6^{th} Generation Intel® Core™ processor (microarchitecture codename Skylake).

An enclave is initialized by loading executable code into a segment of memory using special SGX instructions and then calling the `EINIT` instruction to initialize it. The `EINIT` instruction verifies the enclave author's digital signature on the enclave. The enclave code can subsequently invoke the `EGETKEY` instruction to derive a secret key that is unique to all enclaves written by this author and running on this platform. We explain this mechanism, which we call *Surnaming*, in Section 2.

Surprisingly, SGX does not use a PKI to verify signatures,

for this usage. This seems impossible: without a PKI or pinned keys, signature verification is meaningless, and yet SGX does neither. The reason this usage works is that the process of verifying an enclave is not using digital signatures as digital signatures are normally used.

In this paper we formalize the mechanism used to verify enclaves. We call this cryptographic primitive a *Surnaming* mechanism. We define it precisely in Section 3 and study its properties. We show that Surnaming can be constructed from digital signatures and vice versa, but surprisingly, a Surnaming mechanism can be implemented with no conditional statements. Digital signature verification, in contrast, always require conditional statements to output true or false depending on the signature's validity. Moreover, Surnaming requires no PKI or pinned keys. We explain how these properties are used by the SGX architecture.

Verification performance. The `EINIT` instruction verifies a digital signature atomically. Because the processor has a hard upper limit on the number of cycles that a single instruction can take, SGX must use a digital signature with the fastest possible verification. Signature size is unimportant in these settings. Thus, SGX gives rise to an unusual design requirement: construct a secure signature scheme with the fastest possible verification time, irrespective of signature size. For a technical reason explained in the next section, the signature scheme must also generate short public-keys.

Towards this goal we review the choice of signature primitives currently used by SGX. We show how the current scheme, which is based on RSA3072 for 128 bit security with special optimizations, that is discussed in Section 4), can execute signature verification in only 12,000 cycles – much faster than other known alternatives.

Taking on the performance challenge posed by the SGX usage model, we design and implement an alternative method. It is based on hash-based signatures, and designed to minimize verification time. After much optimization work we obtain a scheme where signature verification takes only about 7,000 cycles, roughly 40% faster than what is currently used in SGX. We explain our design and the optimizations that went into achieving this speed in Section 5.

Moreover, by scaling the parameters of our hash-based scheme appropriately, the system can be made post-quantum secure. While this slows down verification time to about 20,000 cycles, it provides a viable post-quantum alternative for (the relevant part of) SGX. In the same section we also discuss other alternatives, such as lattice-based signatures.

Our contributions.

- We formalize the concept of Surnaming and show secure methods to construct a Surnaming scheme from a signature scheme and vice versa. We also show that we can obtain a Surnaming scheme that is somewhat simpler than the signature scheme from which it is derived.
- We analyze the security of the Surnaming mechanism used in SGX, review the root cause of the performance challenge, and explain how the current SGX scheme addresses it.
- We show that using a specific variant of hash-based signatures leads to a signature with even faster verification time. Moreover, this variant can be extended to give a post-quantum secure scheme.
- We experiment with all our proposals and report on running times and parameter settings that provide optimal performance on current processor architectures.

2. SGX AND ITS SURNAMING MECHANISM

SGX is a security technology, designed to allow a general purpose computer platform to run application software in a trustworthy manner, and to handle secrets that are inaccessible to anyone outside the defined trust boundaries. These trust boundaries encompass only the CPU internals, implying, in particular, that the system memory is untrusted. SGX is a complex technology that involves many details (see [2], [3], [20], [17], [18]). We provide here *only* a simplified outlined description of the essential elements that are necessary for the paper's focus on the use of Surnaming in SGX.

Enclaves. The basic primitive in SGX is the "enclave". An enclave consists of code, data, and metadata (`CDM` hereafter) that realize some application that the enclave's author (`A`) prepares. The enclave is organized as a collection of 4KB "pages". The identity of an enclave consists of the following information on its construction (see Remark 1): a) the `CDM` inside the enclave (before it is initialized); b) the order of loading the pages into memory (and the linear addresses of the pages); and c) the security attributes (Read/Write/eXecute) of each page.

The SHA-256 digest of this information is called `MRENCLAVE`, and represents the cryptographic identity of the enclave. Two enclaves with the same cryptographic identity are considered equivalent.

Remark 1. Not all the data in `EADD`-ed pages must be measured. For example, non-initialized data, or SSA pages, do not need to be measured. An author (`A`) may decide which parts of the `CDM` should be actually included as the enclave's identity, by specifying which pages are to be measured and taken into account in `MRENCLAVE`.

To prepare an enclave after the author `A` has written the actual application code, `A` is expected to do the following:

a) compute `MRENCLAVE` by hashing the appropriate data;
b) generate a private-public key pair (`pk`, $\mathtt{sk}_{\mathrm{sign}}$);
c) sign `MRENCLAVE` using $\mathtt{sk}_{\mathrm{sign}}$, to obtain signature `s`;
d) ship `s`, `pk`, and the expected `MRENCLAVE`, together with the enclave.

In the context of SGX, the SHA-256 digest of `pk` is called `MRSIGNER`.

Remark 2. SGX ships with a Software Development Kit (SDK) that automates the execution of some of the above steps. `A` is responsible for generating the private-public key pair securely, and to sign `MRENCLAVE` (which is computed by the SDK). The SDK processes the information, produces the required outputs, and wraps them in the required format.

Remark 3. `MRENCLAVE` and `MRSIGNER` are different identifiers. `MRENCLAVE` identifies the enclave's contents (i.e., its `CDM`), thus reflects its intended functionality, while `MRSIGNER` identifies `A`.

The enclave code and data are available in the clear before instantiation. That is, the CDM is visible, and – more importantly – auditable (some pieces of the CDM can be encrypted, but the decryption key should not be pre-installed). Consequently, the entity (Service-Provider) that hands secrets to the enclave can work with the author to pre-approve the enclave (i.e., its intended functionality). Secrets such as keys, passwords, and other sensitive data, need to be handed to the enclave from a 3rd party (either from another enclave or from outside the platform), *after* it is loaded and instantiated on some platform. To this end, the enclave must convince a remote secret service provider (Service-Provider) who owns a secret, that it is trustworthy, and can be provisioned with secrets. Furthermore, after an enclave is provisioned with secrets, it should be able to securely store them outside of the enclave for subsequent use.

Instantiating an enclave [1]**.** SGX includes special CPU instructions that are used to "build" an enclave: ECREATE (sets up and records the configuration information), EADD (records the offset of a page inside the enclave, and its security attributes, and copies the CDM page from non trusted memory to trusted (protected) memory (see Remark 1)), EEXTEND (records the pointer and the data stored in a 128 byte chunk of the enclave page), and EINIT (described below).

The enclave is built by invoking ECREATE, and then, for each page of its CDM, invoking EADD, followed by 32 invocations of EEXTEND. This flow copies the CDM, incrementally, from general purpose (unprotected) memory, and locks it in a protected memory region, while (incrementally) measuring MRENCLAVE and logging the size of its CDM. The build process ends by invoking the EINIT instruction. EINIT has several roles, and we describe only those that are relevant to our discussion: a) "finalize" the SHA-256 computation of MRENCLAVE (i.e., add in the padding block, using the recorded enclave size); b) "verify" (see Remark 6 below), using the input pk, that the input s is the signature on the (measured) MRENCLAVE; and c) compute MRSIGNER and store it in the protected memory region (only after completing a successful verification).

After the build process terminates successfully, the initialized enclave is considered "instantiated", and ready to run.

Isolation during run time. An instantiated enclave runs in a special "secure enclave" mode where a hardware based access control mechanism isolates it from all other processes (at all privilege levels) that run on the platform, and from external hardware devices that are attached to the system. Furthermore, the enclave operates from a memory region that is cryptographically protected by a dedicated hardware unit (the Memory Encryption Engine [14]), that protects privacy, integrity and freshness (anti-replay). In other words, the enclave can protect its secrets during run-time.

Acquiring secrets. Secrets need to be delivered to the enclave after its origin, identity, and execution environment are verified. Service-Provider is expected to vet the enclave's Trusted Computing Base (TCB) before it trusts it and provisions it with secrets (in particular, to guarantee that the enclave would perform its pre-approved intended functionality). To this end, SGX offers the means for an enclave to prove to an off-platform party its MRENCLAVE value, its MRSIGNER value, and its execution environment (namely, enclave mode, the CPU security level, and the Security Version Number (SVN)). The details of the tools, the protocols, and the Provisioning and Attestation services are outside the scope of this paper (details appear in [3] and [18]).

Remark 4. In this paper, the term "enclave" refers to enclaves that are written by developers/users of the SGX technology. We mention that SGX has another type of enclaves, called *Architectural Enclaves* (SGX defines a few). The Architectural Enclaves are written and signed by Intel, and the processor has an internal copy of the (hash of the) matching public key. They are designed for a specific functionality, in particular, they are used for the process of Provisioning and Attestation. A useful way to consider the Architectural Enclaves is to view them as an extension of the processor's hardware, an separate their behavior/functionality from general purpose users' enclaves.

Handling secrets The enclave needs the ability to store its secrets to non-volatile memory, in order to use them in subsequent runs. For this purpose, SGX includes the EGETKEY instruction that the (instantiated) enclave software can invoke. EGETKEY instruction can be used to obtain a *Sealing key* which is unique to the specific platform, to the enclave (its identity of its author), and to the SGX security version. The enclave software can use the Sealing key to encrypt its secret information (*Seal*) before it stores it on untrusted media, and decrypt it (*UnSeal*) in subsequent runs. EGETKEY computes the Sealing key by applying a PRF (Pseudo Random Function) where the PRF key is derived from a secret key (*PlatformKey*) that is unique to the platform (among other things. See Remark **??**). The PRF runs over several non-secret fields, including either MRENCLAVE or MRSIGNER. The selection between the two is determined by the software selected EGETKEY parameters (determined, originally, by A). We discuss here only the Sealing keys that are produced by using MRSIGNER (and ignore Sealing keys that are produced by using MRENCLAVE). Theses keys have the following desirable property: enclaves running on the same platform with the same SVN, and written by the same author A, will obtain the same Sealing key when calling EGETKEY. This lets two enclaves written by the same developer, running on the same platform, share secret state. As a result, the process of software update is simplified — all versions of an enclave share the same secret "Sealing key". This is considered to be a significant feature.

Remark 5. The value of MRSIGNER represents A, who is the enclave's software developer. Service-Provider, who owns the secrets that need to be delivered to the instantiated enclave, is not necessarily A. However, Service-Provider can communicate, offline, with A and establish trust in A's MRSIGNER identity. This implies that Service-Provider would: trust all enclave software that A produces (e.g., advanced versions of the same application[2]) to allow these applications to share secrets (on a given platform) through the common Sealing key. Under this viewpoint, A is also referred to as the *Sealing Authority*.

[1]This is a conceptual flow, but actual software might implement a different one.

[2]not that ISV SVN is a component in the key derivation. The enclave developer can choose to anchor it to zero.

REMARK 6. While executing EINIT, the processor *does not* (and cannot) check the origin and validity the input pk. It follows that the processor *does not* perform a signature verification during EINIT. As we show here, the primitive needed for enclave verification is quite different from a digital signature. We call the required primitive a *Surnaming* scheme.

The strict performance constraints. The SGX usage model raises a special set of constraints. Since EINIT is a processor instruction, its allowed latency is very limited. Consequently, it is essential for SGX to specify a Surnaming scheme that has a very efficient verification. By contrast, the performance of signing is rather irrelevant, because it occurs offline (by A). Similarly, the signature size is also not a significant concern: the argument to EINIT is a pointer to a memory location where the inputs are stored (this structure is called SIGSTRUCT). This structure can have an arbitrary size. Note that EINIT also computes MRSIGNER by hashing the (input) public key. This hashing step adds to the overall latency of EINIT. The specific scheme that SGX employs is described in Section 4.

3. SURNAMING SCHEMES

The discussion in the previous section explains that when initializing an enclave, the processor cannot check the authenticity of the signature on the enclave content, because it cannot validate the relevant public-key pk. Instead, the processor uses pk and the signature, along with other data, to derive a secret key (the *Surnaming key*) that is only known to enclaves written by the same author. In this section we define the syntax and security properties required of a Surnaming scheme that model the processor's operation.

3.1 Signatures and Surnaming

Recall that a digital signature is made up of three algorithms (G, S, V). Algorithm G generates a private-public key pair, $(\mathtt{pk}, \mathtt{sk}_{\text{sign}})$. Algorithm $S(\mathtt{sk}_{\text{sign}}, m)$ signs the message m and outputs a signature s. Algorithm $V(\mathtt{pk}, m, \mathtt{s})$ verifies the signature and outputs true or false. An example is a software/firmware update procedure, where the software vendor signs the update using its public-key pk and every device verifies the signature prior to installing the update. Clearly, establishing trust in pk is essential.

A Surnaming mechanism has a different syntax and security requirements. The purpose of Surnaming is to allow an author (A) to use its public and private key pair $(\mathtt{pk}, \mathtt{sk}_{\text{sign}})$ to sign multiple messages $(\mathtt{m}_1, \ldots, \mathtt{m}_k)$ and distribute triples $(\mathtt{pk}, \mathtt{m}_1, \mathtt{s}_1)$, ..., $(\mathtt{pk}, \mathtt{m}_k, \mathtt{s}_k)$. The author is assured of the following properties:

- if a verifier (V) is presented with a triplet $(\mathtt{pk}, \mathtt{m}_j, \mathtt{s}_j)$, it can apply some pre-agreed algorithm Surname to the given values to generate a constant c (that depends only on pk but not on m or s).
- if V is presented with a triplet $(\mathtt{pk}', \mathtt{m}', \mathtt{s}')$ such that $\mathtt{m}' \notin \{\mathtt{m}_1, \ldots, \mathtt{m}_k\}$, and $\mathtt{s}'$ is any signature and $\mathtt{pk}'$ is any public key, then Surname outputs a constant $\mathtt{c}' \neq \mathtt{c}$.

The resulting constant c is subsequently used by V as (part of some) input to a PRF with a secret key owned by V, in order to generate a new secret key (called Sealing Key) that is shared across, and only across, $\mathtt{m}_1, \ldots, \mathtt{m}_k$. The time it takes V to produce c is hereafter referred to as the *verification time*. Note the significant difference from signatures: in the Surnaming protocol, V does not need to trust neither $\mathtt{m}_j$ nor pk.

3.2 Surnaming scheme: definition

Context. a Surnaming scheme operates over a message space $\mathcal{M}$ and is defined as follows.

DEFINITION 1. *A Surnaming scheme is a triple of algorithms* (*Setup*, *Authorize*, *Surname*) *where*

- *Setup: outputs* sk.
- *Authorize*$(\mathtt{sk}, m)$ $(m \in \mathcal{M})$ *outputs* σ.
- *Surname*(m, σ) $(m \in \mathcal{M})$*: outputs* id *or* $\perp$.

For correctness, we require that for all sk *which is output by Setup, and all* $m, m' \in \mathcal{M}$*:*

if $\sigma \leftarrow$ *Authorize*$(\mathtt{sk}, m)$ *and* $\sigma' \leftarrow$ *Authorize*$(\mathtt{sk}, m')$ *then* *Surname*$(m, \sigma) =$ *Surname*(m', σ').

In other words, if both m and m' are authorized, then calling Surname on either one produces the same constant id.

Mapping to SGX. To understand the relation to SGX, it is helpful to think of the following mapping. The Surnaming scheme is used by two parties: A (enclave author) and V (verifier, namely the processor during execution of EINIT). The authorization key sk is A's private key and m is the enclave's CDM. The author runs Authorize(sk, CDM) to obtain the enclave authorization token σ which is packed into the enclave metadata. On the SGX machine, the EINIT instruction runs $\mathtt{Surname}(m, \sigma)$ and the output is some constant id. The "consumer" of the output id is the EGETKEY instruction: it uses id to generate a local secret by computing Sealing Key = PRF(*PlatformKey*, id). The correctness property of the Surnaming scheme ensures that all enclaves that are authorized by a single developer will lead to the generation of the same identifier id, and therefore obtain the same Sealing Key when running on the same platform.

Security definition. define the following game between challenger and adversary:

1. Challenger generates random $\mathtt{sk} \xleftarrow{R} \mathtt{Setup}$.
2. adversary adaptively submits messages $m_1, m_2, \ldots,$ and gets back $\sigma_i := \mathtt{Authorize}(\mathtt{sk}, m_i)$ for $i = 1, 2, \ldots$. Adversary must submit at least one query
3. eventually, adversary outputs (m, σ) where m is not in $\{m_1, m_2, \ldots\}$.

Adversary wins if $\mathtt{Surname}(m, \sigma) = \mathtt{Surname}(m_1, \sigma_1)$.

DEFINITION 2. *A Surnaming scheme* (*Setup*, *Authorize*, *Surname*) *is secure if no efficient adversary can win the game with non-negligible probability.*

The security definition captures the intuition that an adversary who obtains authorization tokens for arbitrary enclaves of its choice, cannot construct a useful authorization token σ for some other enclave m. That is, $\mathtt{Surname}(m, \sigma)$

will be different from the output of $\texttt{Surname}$ for the valid enclaves $m_1, m_2, \ldots$.

Secure Surnaming from a secure signature scheme. It is not difficult to see that a secure Surnaming scheme can be constructed from a secure digital signature. Let (G, S, V) be a signature scheme, where V outputs 0 or 1. The derived Surnaming scheme is defined as follows:

- $\texttt{Setup}$: run G to get $\texttt{sk}_{\text{sign}}$ and $\texttt{vk}$. Set the secret Surnaming key to be $\texttt{sk} := (\texttt{sk}_{\text{sign}}, \texttt{pk})$.
- $\texttt{Authorize}(\texttt{sk}, m)$: run $\texttt{sig} \leftarrow S(\texttt{sk}_{\text{sign}}, m)$ and output $\sigma \leftarrow (\texttt{sig}, \texttt{pk})$.
- $\texttt{Surname}(m, \sigma)$: output $\big(\texttt{pk},\ V(\texttt{pk}, m, \texttt{sig})\big)$.

The scheme is correct: $\texttt{Surname}(m, \sigma)$ outputs $(\texttt{pk}, 1)$ whenever σ is a valid authorization for m.

Security follows from the following simple theorem whose proof is immediate.

THEOREM 1. *The derived Surnaming scheme is secure assuming (G, S, V) is a signature scheme that is existentially unforgeable under a chosen message attack.*

PROOF SKETCH. An adversary that defeats the derived Surnaming scheme queries the challenger on a sequence of messages $m_1, m_2, \ldots$ and finally produces a pair $\big(m,\ \sigma = (\texttt{sig}, \texttt{pk})\big)$ such that $\texttt{Surname}(m, \sigma)$ outputs $(\texttt{pk}, 1)$ and m is new. But then $V(\texttt{pk}, m, \texttt{sig}) = 1$, which is an existential forgery for the underlying signature scheme. □

Surnaming implies signatures. Next we show that every secure Surnaming scheme implies a secure signature scheme. Let $(\texttt{Setup}, \texttt{Authorize}, \texttt{Surname})$ be a Surnaming scheme. Define the following signature scheme (G, S, V):

- Algorithm G works as follows:
  - run $\texttt{Setup}$ to get $\texttt{sk}$,
  - run $\texttt{Authorize}(\texttt{sk}, 0)$ to get σ,
  - run $\texttt{Surname}(m, \sigma)$ to get $\texttt{id}$.

  Set $\texttt{pk} := \texttt{id}$ and outputs $(\texttt{pk}, \texttt{sk})$.
- Algorithm $S(\texttt{sk}, m)$: output $\sigma \leftarrow \texttt{Authorize}(\texttt{sk}, m)$.
- Algorithm $V(\texttt{pk}, m, \sigma)$: accept if $\texttt{Surname}(m, \sigma) = \texttt{pk}$.

The following simple theorem shows that the constructed signature scheme is secure.

THEOREM 2. *If $(\texttt{Setup}, \texttt{Authorize}, \texttt{Surname})$ is a secure Surnaming scheme then (G, S, V) is a signature scheme secure against existential forgery under a chosen message attack.*

PROOF SKETCH. Suppose there is an attacker $\mathcal{A}$ on the signature scheme. We use it to build an attacker $\mathcal{B}$ on the underlying Surnaming scheme. $\mathcal{B}$ begins by choosing a random message m' and asking its challenger to authorize m', thereby receiving $\sigma' \xleftarrow{R} \texttt{Authorize}(\texttt{sk}, m')$. Then

$$\texttt{Surname}(m', \sigma') = \texttt{pk}.$$

Next, $\mathcal{B}$ runs the signature attacker $\mathcal{A}$. It responds to $\mathcal{A}$'s signature queries by asking $\mathcal{B}$'s challenger to authorize the messages output by $\mathcal{A}$. Eventually $\mathcal{A}$ outputs an existential forgery (m, σ). Since (m, σ) is a valid signature, we know that $\texttt{Surname}(m, \sigma) = \texttt{pk}$, even though m was never authorized. This breaks the underlying Surnaming scheme because $\texttt{Surname}(m, \sigma) = \texttt{Surname}(m', \sigma')$. Note that we are assuming the message space is sufficiently large so that $m \neq m'$ with high probability. □

3.3 Surnaming with conditional-free verification

Signature verification algorithms necessarily require conditional statements to decide if a given signature is valid. Remarkably, the $\texttt{Surname}$ algorithm in a Surnaming scheme can be implemented with no conditional statements. Nothing needs to be checked. This is a significant advantage of Surnaming schemes over traditional signatures, primarily because signature verification checks have often been implemented incorrectly in practice. Bleichenbacher's attack on low-exponent RSA signatures [1] is a famous example of faulty signature verification, where the error was a result of an incorrect PKCS1 padding check. Another example is the large subgroup attack on some discrete-log based signature schemes where the verifier forgets to check if the given signature components are in the prescribed subgroup [21]. Even the original DSA specification from NIST contained a security error in signature verification where the verifier did not properly verify that the size of the two signature elements are in the required range [25].

Let us see how to implement a Surnaming scheme where $\texttt{Surname}$ requires no conditional statements. We demonstrate this by example using RSA signatures with PKCS1 padding.

RSA Surnaming scheme:

- $\texttt{Setup}$: Run the RSA key generation algorithm to obtain $\texttt{pk} = (N, e)$ and $\texttt{sk}_{\text{sign}} = (N, d)$. Here N is the RSA modulus and e is the RSA public exponent, and d is the RSA private exponent. Output $\texttt{sk} = (\texttt{pk}, \texttt{sk}_{\text{sign}})$.
- $\texttt{Authorize}(\texttt{sk}, m)$: RSA sign m, that is set

  $$m' := \text{PKCS1PAD} \parallel \text{SHA256}(m)$$

  as the PKCS1 padded message and treat m' as an integer. Then use $\texttt{sk}$ to compute $s := (m')^d \pmod N$ and output $\sigma := (\texttt{pk},\ s)$.
- $\texttt{Surname}(m, \sigma)$:
  - compute $s' := s^e \pmod N$,
  - remove the message hash, namely set

    $$s'' := s' - \text{SHA256}(m) \pmod N,$$

    when σ is valid, this zeroes out the 256 least-significant bits of s'',
  - output $\texttt{id} := (\texttt{pk},\ s'')$

Note that no conditional statements are used in $\texttt{Surname}$. The point is that instead of checking the pad, as required during RSA signature verification, we simply output the pad as part of the $\texttt{id}$. This eliminates the consequences of an incorrect pad checking implementation.

In the next section we show that the SGX $\texttt{EINIT}$ instruction essentially uses this RSA Surnaming mechanism to derive the constant $\texttt{id}$, and $\texttt{EGETKEY}$ uses the result in order to provide a $\texttt{Sealing Key}$ to an enclave that invokes it.

The RSA Surnaming scheme is a correct Surnaming scheme: when σ is a valid authorization for m then $\texttt{Surname}(m, \sigma)$ produces an `id` containing the public key and the PKCS1 pad. The same `id` is obtained for every properly authorized message.

The following theorem captures the security property of the RSA Surnaming scheme.

THEOREM 3. *The RSA Surnaming scheme is a secure Surnaming scheme assuming RSA-PKCS1 is existentially unforgeable under a chosen message attack.*

PROOF. Suppose there is an attacker $\mathcal{A}$ on the RSA Surnaming scheme. We use it to build an attacker $\mathcal{B}$ on the RSA signature scheme. $\mathcal{B}$ runs the Surnaming attacker $\mathcal{A}$. It responds to $\mathcal{A}$'s authorization queries by asking $\mathcal{B}$'s challenger to sign the messages output by $\mathcal{A}$ using RSA-PKCS1. Eventually $\mathcal{A}$ outputs a valid forgery (m, σ). We know that $\texttt{Surname}(m, \sigma)$ outputs $(\texttt{pk}, \text{PKCS1}_{\text{PAD}} \| 0^{256})$. But this means that σ is a valid RSA-PKCS1 signature for m and therefore (m, σ) is a valid existential forgery for RSA-PKCS1. □

More generally, any signature scheme with message recovery can be converted into a Surnaming scheme with conditional-free `Surname`. Examples include RSA, Rabin [4], and Nyberg-Reupell [23] signatures.

Even pairing-based BLS [8] signatures give a Surnaming scheme with a conditional-free `Surname`. Recall that in BLS signatures are verified by testing that

$$e(g, \texttt{pk}) = e(H(m), \sigma)$$

where e is a paring function, H is a hash function and g is a fixed group generator. When used in $\texttt{Surname}(m, \sigma)$ one can instead output

$$e(g, \texttt{pk})/e(H(m), \sigma)$$

so that `Surname` contains no conditional statements. If the signature is valid the ratio will be 1. Otherwise, it will be some other value. Since the output of a pairing function is never zero, we need not worry about division by zero.

4. THE SGX SURNAMING SCHEME

As explained above, the performance of the *processor instruction* `EINIT` limits the possible choice of signature primitives that SGX can use. To understand the available options, consider Table 1 which shows the verification performance of some standard 128-bit security signature schemes. The measurements were done on the latest Intel processor generation (Architecture Codename Skylake), which is the first processor that supports SGX.

The performance of ECDSA and RSA3072 with public exponent $e = 2^{16} + 1$ is prohibitive. RSA3072 with a short public exponent $e = 3$ is $\approx 2.76x$ faster than with $e = 2^{16} + 1$. Finally, note that RSA3072 with $e = 3$ using a QVRSA verification method [11] is the fastest option, by a wide margin. Indeed, SGX uses a Surnaming scheme that it based on these primitives. We analyze it here.

The RSA3072 primitives used in SGX. SGX constructs a Surnaming scheme from RSA signatures as explained in the previous section. The version of RSA signatures used is PKCS1-v1.5 format and DER encoding of the

Scheme	Cycles per verification	Comments
ECDSA (P256)	264,609	OpenSSL 1.0.2
ECDSA (P256) optimized	226,986	OpenSSL patched [15]
RSA3072 $e = 2^{16} + 1$	122,928	OpenSSL
RSA3072 $e = 3$	44,500	50,400 with padding check
RSA3072 $e = 3$ $e = 3$ with QVRSA	12,000	Optimized implementation

Table 1: Signature verification performance for several signature schemes.

"DigestInfo" specified in PKCS#1 v2.1 [19]. The hash function `SHA-256`, and the key length is 3072 bits (384 bytes). We denote the PKCS1-v1.5 padding patten (352 bytes) by `PADDING`[3], the private key by d, and the public modulus by N. The public exponent is set to $e = 3$. The signature (S) of a message `m` is computed by:

a) Computing $\texttt{H} = \texttt{SHA-256}(\texttt{m})$ (32 bytes);
b) Constructing the 384 bytes Encoded Message $\texttt{EM} = \texttt{PADDING} \| \texttt{H}$;
c) Output $S = (\texttt{EM})^d \pmod N$.
Here `EM`, d, and N are parsed as integers.

For verification, the verifier receives $\texttt{m}'$, S', N', computes $H' = \texttt{SHA-256}(\texttt{m}')$, $T = (S')^e \pmod{N'}$, and then checks that $T = \texttt{EM}$ as two 384-byte strings. In particular, checks that $T[31\texttt{B} : 0\texttt{B}] = H'$ and $T[383\texttt{B} : 32\texttt{B}] = \texttt{PADDING}$ (signature verification requires the verifier to trust N').

The QVRSA optimization. Because SGX needs a signature with fast verification, it uses a variant of RSA verification called *Quick Verification RSA*, or QVRSA, as proposed by Gueron [11] (see last row of Table 1). This optimization is a way to speed up RSA verification for any public exponent, in particular, of the form $2^k + 1$. QVRSA is based on handing the verifier some pre-computed constants with which the verifier can compute $T = (S)^e \pmod N$ using only integer arithmetic instead of modular arithmetic. Computing these (public) constants does not require knowledge of d, and can even be done by post processing a signature that a secure platform (e.g., an HSM) generates. QVRSA is especially effective with $e = 3$, which is our case. Here, only two constants $q1$, $q2$ are needed:

$$q1 = \left\lfloor S^2/N \right\rfloor, \quad q2 = \left\lfloor (S^3 - q1 \cdot S \cdot N)/N \right\rfloor \tag{1}$$

The verifier is given `m`, S, N, $q1$, $q2$, and applies the following algorithm to computed $S^3 \pmod N$.

Algorithm 1. QVRSA computations
Input: $\texttt{m}, S, N, q1, q2$ (s.t., $0 < S, q1, q2 < N < 2^{3072}$)
(1) if $\neg(S < N)$ then verification = FAILURE
(2) $T1 = S^2 - q1 \cdot N$
(3) if $\neg(0 < T1 < N)$ then verification = FAILURE
(4) $T2 = S \cdot T1 - q2 \cdot N$

[3]By [19] (with `SHA-256` hash) , `PADDING` = $\text{PKCS1}_{\text{PAD}}$ $00 \| 01 \| ff[330B] \| 00 \|$ $3031300d060960864801650304020105000420$

(5) if $\neg(0 < T2 < N)$ then verification = FAILURE

Output: if (verification = FAILURE) output FAILURE else output $T2$.

THEOREM 4. *Algorithm 1 returns $T2$ if and only if $q1$ and $q2$ satisfy Condition (1), and in that case, $T2 = S^3 \pmod N$.*

PROOF. Assume that Algorithm 1 does not return FAILIURE. From Step 2, we have $T1 \pmod N = S^2 \pmod N$. Together with the condition in Step 3, this implies that $T1 = S^2 \pmod N$. Then, from Step 4, we have $T2 \pmod N = S \cdot T1 \pmod N = S \cdot S^2 \pmod N = S^3 \pmod N$. Together with condition in Step 5, we conclude $T2 = S^3 \pmod N$. On the other direction, note that for $0 < S < N$, we have $S^2 \bmod N = S^2 - \alpha \cdot N$ for some integer $\alpha \geq 0$. Also, $0 < S^2 \bmod N < N$. Therefore, $A = \lfloor S^2/N \rfloor$, and this is the value of $q1$. Finally, $S^3 \pmod N = S \cdot (S^2 - \alpha \cdot N) \pmod N = S^3 - \beta \cdot N$ for some $\beta \geq 0$. Since, in addition, $0 < S^3 \bmod N < N$, we get the value of $\beta = \lfloor (S^3 - q1 \cdot S \cdot N)/N \rfloor$. This is the value of $q2$. □

REMARK 7. *Algorithm 1 computes $S^3 \pmod N$ by means of four integer multiplications and two subtractions (some of the multiplications can be parallelized). This does not obviate the need to assure that $T =$ `EM` for a signature verification, or the incorporate the conditional-free technique for Surnaming.*

REMARK 8. *Step 1 of Algorithm 1 is not mandatory, if the verification flow can handle larger inputs. A reasonable implementation could easily enforce the condition $S, q1, q2, < 2^{3072}$ (instead of $< N$), use 3072-bit arithmetic, and enforce Steps 3 and 5.*

The SGX conditional-free verification. We can now explain the rationale behind how RSA is used in `EINIT` and `EGETKEY` for Surnaming. Section 3.3 discussed Surnaming with a conditional-free verification, and RSA Surnaming in particular. This technique is employed in SGX and makes it possible to generate a Surnaming key without relying on a correct check of the PKCS1 padding (although `EINIT` executes the correct padding check anyway). To understand the details, we refer the readers to the usage of the 352 bytes strings "`PKCS Padding Buffer`" and the "`HARDCODED_PKCS1_5_PADDING`" in the `EINIT` and `EGETKEY` instructions (see [2]).

Comment on the Bleichenbacher attack. It is interesting to recall the famous Bleichenbacher attack [6] on a *flawed* verification of an RSA signature with a short public exponent. This is not an attack on the RSA signature, but a demonstration that a faulty padding check can facilitate forgery (especially if $e = 3$). Surprisingly, several real implementations turned out to have a mistaken padding check. This led to the statement in [1], and to removing $e = 3$ from the list of "allowed" RSA public exponents (i.e., requiring $\geq 2^{16} + 1$). The irony is that excluding the shortest public exponent ($e = 3$) slows down the fastest possible RSA signature verification (see Table 1). At the same time, we point out that fast verification is probably the only remaining advantage of RSA over ECDSA. Fortunately, a Surnaming protocol can be made provably agnostic to a (hypothetical) padding check mistake, and therefore enjoy the fast verification. SGX leverages this property to its advantage.

5. NEW HASH-BASED SIGNATURES WITH FAST VERIFICATION

Our discussion of Surnaming, and its usage in SGX, motivates the need for signatures with the fastest possible verification time, while ignoring the signature size. This suggests the following two questions:

1. Are there signature schemes with fast verification time, that would outperform RSA3072 with $e = 3$ and QVRSA that is used in SGX (irrespective of the signature size).

2. In a post-quantum scenario, what signature has the fastest verification time on a modern processor?

We answer these questions in this section.

In the context of SGX, the enclave author uses a Surnaming scheme to authorize the set of enclaves that it writes, including the different versions of each enclave. As such, a single public `pk` is used to authorize (sign) only a relatively small number of enclaves, say a maximum of 1000 enclaves. Therefore, a scheme that is secure for only a limited number of signatures is sufficient in this context.

Another SGX constraint on the Surnaming mechanism is that the output of $\texttt{Surname}(m, \sigma)$, called `MRSIGNER`, needs to be short, say at most 32 bytes. When constructing a Surnaming from a digital signature, this constraint translates to requiring a short public-key on the signature scheme. The reason we need a short `MRSIGNER` is that this information is stored in a protected memory region (called Enclave Page Cache), which is limited in size. In addition, recall that the `EGETKEY` instruction feeds this information into a PRF to obtain the `Sealing Key`, thus its performance depends on the length of `MRSIGNER`.

To summarize, our goal is to design a signature scheme with the fastest possible signature verification time, a small public key, while the signature size can be ignored. We observe that any signature scheme (G, S, V) can be easily transformed into one where the public key is short:

- At key generation time, set the public key $\texttt{pk}'$ to be the `SHA-256` hash of the actual public key `pk` that is output by algorithm G, that is, $\texttt{pk}' := \texttt{SHA-256}(\texttt{pk})$[4],
- when signing a message m, prepend the full public key `pk` to the signature, and
- when verifying a signature, first check that the full public key `pk` supplied with the signature is consistent with the given hashed public key $\texttt{pk}'$, then check the given signature relative to the full public-key `pk`.

In light of this observation we see that the main design challenge is constructing a signature scheme with a fast verification time.

5.1 Starting point: one-time signatures

Our goal is to build a signature scheme with super-fast verification, as needed for SGX. Our plan is to build these signatures from a hash-based scheme such as Merkle signatures [22]. By tuning the parameters appropriately, we could obtain a post-quantum secure signature.

The need for fast-verification, irrespective of signature size, is an unusual point in the design space. Existing schemes, such as Sphincs [5] and others [7, 9], primarily try to minimize signature size so as to reduce network transmission

[4]or a hash function with a longer digest, for Post Quantum scenarios.

overhead. Here we design a hash-based signature optimized for fast verification.

Construction. Hash-based signatures are built from two underlying primitives:

- A **one-way function** $f : X \to X$, and
- A **collision resistant hash** $h : \{0,1\}^* \to Y$.

Typically $X := \{0,1\}^{128}$ and $Y := \{0,1\}^{256}$. In a post-quantum settings one needs to increase the parameters to $X := \{0,1\}^{256}$ and $Y := \{0,1\}^{384}$, as discussed in the next section.

The design of the fastest verification-time scheme depends on the relative speeds of running f vs. h. We start with a general design and then optimize for the different options for f and h. The results are reported in the next sections.

We first construct a one-time signature with the fastest verification possible. Roughly speaking, one-time signatures belong to one of two families: Winternitz [22, 9] and HORS [24] (both are extensions of Lamport's one-time signature). HORS is designed to produce shorter signatures, but verification time for HORS is slower than for Winternitz because HORS verification generally requires more hashes than Winternitz. We therefore opt to optimize a Winternitz-like construction for our purposes.

We first describe a variant of the general Winternitz one-time signature scheme. The scheme is parametrized by a small constant d, called the chain depth, where typically $d = 2, 4$ or 8. This d refers to the length of a hash chain based on the function $f$used in the signature scheme. Setting $d = 2$ means that message bits are processed one at a time, resulting in many shallow chains. Setting $d = 4$ means that message bits are processed in pairs, which halves the number of chains, but doubles the chain length. When $d = 8$ message bits are processed in groups of 3, further reducing the number of chains and increasing the chain length to 8. To obtain fast verification there is never a need to go beyond $d = 8$ (on current processors).

We describe the signature scheme below. Recall that Y is the range of our collision resistant hash function h (e.g., $Y = \{0,1\}^{256}$). Let

$$\begin{aligned} n :=& \lceil \log(|Y|)/\log(d) \rceil \quad \text{and} \\ \ell :=& \lceil \log(n(d-1)+1)/\log(d) \rceil. \end{aligned} \tag{2}$$

For example, when $Y = \{0,1\}^{256}$ and $d = 2$, we have that $n = 256$ and $\ell = 9$.

The signature scheme works as follows. For our purposes, the verification algorithm is the central issue.

- Algorithm G:
  (1) choose random $x_0, \ldots, x_{n+\ell-1}$ in X,
  (2) for $i = 0, \ldots, n+\ell-1$ compute $y_i := f^{(d-1)}(x_i)$,[5] that is, we construct $n + \ell$ hash chains,
  (3) output

$$\begin{aligned} \mathtt{sk} :=& (x_0, \ldots, x_{n+\ell-1}) \in X^{n+\ell} \\ \mathtt{pk} :=& h(y_0, \ldots, y_{n+\ell-1}) \in Y \end{aligned}$$

  Note that $\mathtt{pk}$ is short. When $Y = \{0,1\}^{256}$ and $d = 2$ we constructed $256 + 9 = 265$ chains. With the same size Y, increasing d to 4 reduces the number of chains to $128 + 5 = 133$.

- Algorithm $S(\mathtt{sk}, m)$:
  (1) compute $h(m) \in Y$ and treat the result as a positive integer written in base d with digits $0 \le m_0, \ldots, m_{n-1} < d$, so that

$$h(m) = m_0 + m_1 d + \ldots + m_{n-1} d^{n-1},$$

  (2) let $w := n(d-1) - (m_0 + \ldots + m_{n-1})$ and write w in base d with digits $0 \le m_n, \ldots, m_{n+\ell-1} < d$,
  (3) for $i = 0, \ldots, n+\ell-1$ set $s_i := h^{(m_i)}(x_i)$,
  (4) output the signature $\sigma := (s_0, \ldots, s_{n+\ell-1})$.

- Algorithm $V(\mathtt{pk}, m, \sigma)$:
  (1) compute $0 \le m_0, \ldots, m_{n+\ell-1} < d$ as in the signing algorithm (this step takes negligible time),
  (2) for $i = 0, \ldots, n+\ell-1$ let $y_i := f^{(d-1-m_i)}(s_i)$,
  (3) accept the signature if $\mathtt{pk} = h(y_0, \ldots, y_{n+\ell-1})$ and reject otherwise.

Security. Security of the scheme follows from the collision resistance of h and the one-wayness of f on $(d-1)$-iterates (i.e., that $f^{(d-1)}$ is a one-way function). See [9, 7] for a security analysis of the scheme. Here we briefly outline the intuition for why this is a secure one-time signature. The adversary is given the public-key and the signature σ on a message m of his choice. His goal is to then forge the signature on some other message m'. Suppose the attacker could find an m' whose signature can be derived from σ by applying f to the components of σ. Then we say that m dominates this m'. The pair of messages (m, m') lets the attacker win the existential forgery game: the attacker can request the signature on m and from it derive the signature on m'. Step (2) in the signing algorithm S ensures that no message m dominates another m' which prevents this potential forgery attack. This can be expanded to a full proof of one-time signature security from the assumption that $f^{(d-1)}$ is one-way function and h is collision resistant.

Performance. The run time of signature verification is dominated by steps (2) and (3): evaluating the one-way function f at $(d-1)(n+\ell)$ points in the worst case (and half that on average), and computing the collision resistant hash h given $n+\ell$ quantities in X as input. Step (2) can be done in parallel: each of the $(n+\ell)$ chains can be computed in parallel. This gives a further speed-up on modern architectures where the AES-NI instructions, used for evaluating AES, are fully pipelined. Step (3) can be similarly parallelized by using a Merkle tree (instead of a linear Merkle-Damgard chain) to evaluate the function h on the $(n+\ell)$ blocks. We discuss the significant impact of parallelism in the next section.

The only free parameter we can play with is the chain depth d. All other parameters are determined by d and $|Y|$ in Eq. (2). Note that different values of d offer different balance between the number of evaluations of f and the length of the string that needs to be hashed (using h). Thus, the question is how to choose the optimal d for different choices of h and f. We do so in the next section.

5.2 From one-time to t-times signatures.

As mentioned earlier, in the context of SGX it suffices to allow only a small number of signatures per public key, say

[5] Here, the notation $f^{(v)}(x)$ means composing the function f with itself v times. For example, $f^{(2)}(x) = f(f(x))$.

a thousand signatures per public key (i.e., $t = 1000$). A one-time signature can be extended to a thousand-time (stateful) signature by simply generating a thousand one-time public-keys and publishing a Merkle tree root of all these public-keys as the global public-key. Each leaf of the tree can sign one enclave. A signature will include a Merkle proof of inclusion of the relevant public-key in the Merkle tree. During verification, the verification algorithm will check the proof, which adds another $\lceil \log_2(1000) \rceil = 10$ sequential hash computations. This is small compared to the amount of hashing needed to verify the one-time signature.

5.3 Concrete constructions

We use the general construction presented in the previous section to derive several concrete signature schemes with fast verification. We implement and experiment with each to compare their verification times. We study two classes of signatures:

1. signatures that target classical 128-bit security, where we set $X = \{0,1\}^{128}$, $Y = \{0,1\}^{256}$;
2. signatures that target post quantum 128-bit security where we set $X = \{0,1\}^{256}$, $Y = \{0,1\}^{384}$. These parameters are needed for generic 128-bit post-quantum collision resistance for h and one-wayness for f.

Our goal is to explore f and h candidates that lead to high performance signature verification. To this end, we leverage the availability of AES instructions (known as "AES-NI" [12], [10]) on modern processors. They offer high performance AES computations.

In some of our experiments we use the Simpira permutations [16], a recently proposed family of cryptographic permutations that support inputs of $128 \times b$ bits for any $b \geq 1$. *Simpira* is designed to provide high throughput on modern processors that have native AES instructions, and therefore, *Simpira* constructions use only the AES round as the basic primitive. *Simpira* is assumed to provide 128-bit security and can be modeled as a random permutation. We use $Simpira_b$ to denote the value of b in a specific parameters choice (e.g., $Simpira_4$ is a permutation of $4 \times 128 = 512$ bits). These cryptographic permutations can be used to define one way functions of the form $Simpira_b(x) \oplus x$, and hash functions $SimpiraHash$ of the from

$$SimpiraHash(x) = [Simpira_b(x) \oplus x]_{t-truncated} \qquad (3)$$

that consumes inputs of $128b$ bits and produces (by truncation) a t-bit digest. We use $SimpiraHash_{(a,b)}$ to denote a specific choice of parameters (e.g., $SimpiraHash_{(4,256)}$ hashes a 512-bit input to a 256-bit digest).

Experiments. Table 2 lists six candidates for the function f, denoted $f1, \ldots, f6$ and four candidate for the function h, denoted $h1, \ldots, h4$. The functions $f1, f2, h1, h2$ target classical 128-bit security, where as the other functions target 128-bit post quantum security.

The functions $f1, f2$ use two standard ways to build a one-way function out of AES. The one-wayness of $f2$ follows from the standard assumption that AES is a secure pseudorandom permutation (PRP), while the one-wayness of $f1$ follows from modelling AES as an ideal cipher. In this sense, $f2$ is preferable, however, $f1$ is considerably faster than $f2$ because $f1$ uses a fixed key. The $f2$ function requires computing a new AES key schedule for every input which has a significant impact on performance. For both functions it is possible to achieve good performance by parallelizing the function evaluation. Notice that when using $f1$ and $f2$ in our signature scheme, we need these functions to be one-way on iterates (that is, $f^{(d)}$ is one-way for $d \leq 8$). This is still quite a natural assumption on these functions.

For the post-quantum security, where $X = \{0,1\}^{256}$, we need a new one-way function. We experiment with the four functions $f3$, $f4$, $f5$, $f6$, listed in Table 2.

- $f3$ mirrors the function $f1$, but using the Simpira function mentioned earlier.
- $f4$ is a simple construction based on AES256. Its one-wayness follows directly from the one-wayness of $f(x) = \text{AES256}_x(0)$, which is a standard assumption about AES256. It is therefore natural to assume that this $f4$ is also one-way on iterates, that is $f4^{(d)}$ is one-way for $d \leq 8$.
- $f5$ and $f6$ mirror $f1$ and $f2$, but for the 256-bit block Rijndael cipher. As before, $f5$ has better performance, but one-wayness of $f6$ is based on a standard assumption about Rijndael.

For the collision resistant hash functions h, we explore the candidates $h1, h2, h3, h4$ that are shown in Table 2 . We use them in a tree construction to exploit the parallelism provided by the hardware.

bits	X	$f : X \to X$
256	$\{0,1\}^{128}$	$f1(x) = AES128_{K0}(x) \oplus x$
256	$\{0,1\}^{128}$	$f2(x) = AES128_x(0)$
384	$\{0,1\}^{256}$	$f3(x) = Simpira_2(x) \oplus x$
384	$\{0,1\}^{256}$	$f4(x) = a \Vert b$ s.t., $a = AES256_x(0)$ $b = AES256_a(0)$
384	$\{0,1\}^{256}$	$f5(x) = Rijndael256_{K0}(x) \oplus x$
384	$\{0,1\}^{256}$	$f6(x) = Rijndael256_x(0)$

	Y	$h : \{0,1\}^* \to Y$
256	$\{0,1\}^{256}$	$h1(x) = \texttt{SHA-256}(x)$
256	$\{0,1\}^{256}$	$h2(x) = SimpiraHash(x)$
384	$\{0,1\}^{384}$	$h3(x) = \texttt{SHA-512}(x)$
384	$\{0,1\}^{384}$	$h4(x) = SimpiraHash(x)$

Table 2: Different options for one way functions ($f : X \to X$) and collision resistant functions ($h : \{0,1\}^* \to Y$) for different choices of X and Y

Implementation. To measure the performance for signature verification, we wrote an optimized implementation for each of the variants. The measurements were done on the latest Intel processor generation (Architecture Codename Skylake), where AES instructions (e.g., `AESENC`) have latency of 4 cycles and are fully pipelined so that throughput is 1 cycle. When implementing hashing for large inputs, we implemented a hash-tree (rather than the sequential Merkle-Damgard) so as to take advantage of the parallelism provided by the hardware. The functions $h2$ and $h4$ use *Simpira*, which is designed to leverage parallelism.

d		f1	f2
2	h2	9,984	19,764
2	h1	18,444	28,485
4	h2	4,849	10,667
4	h1	10,763	16,581
8	h2	5,200	14,415
8	h1	9,461	18,677

Table 3: Signature verification performance for 128-bit non-quantum secure parameters, for $d = 2, 4, 8$ and $h \in \{h1, h2\}$ and $f \in \{f1, f2\}$. The reported numbers measure processor cycle counts for signature verification (lower is better).

On a processor where the `AESENC` has 4 cycles latency, interleaving the computations of 4 permutations reaches the theoretical performance limit (see details in [16]). For $h1$ and $h3$, we used the techniques shown in [13] for both `SHA-256` and `SHA-512`, to leverage 4-way and 8-way parallelism.

We discuss the results in the next two subsections.

5.3.1 Performance for non-quantum 128-bit security

Table 3 gives the cycle count for signature verification using different chain depth values (d) and the non-quantum one-way functions f and hash-functions h. It is useful to compare these number to the running times in Table 1, where verification time is reported for RSA and ECDSA. Clearly hash-based signatures are far faster than all these methods, except for RSA3072 ($e = 3$) with QVRSA.

The best hash-based signature performance ($d = 4$ using $f1$ and $h2$) is faster than RSA3072 with QVRSA (4,849 vs. 12,000 cycles). However, these are one-time signatures and one would need to include the time to compute additional hashes to support a 1000-time signature. As explained in Section 5.2, supporting a 1000-time signature requires an additional 10 sequential hashes during verification, where each hash is on a 64 byte payload. These hashes add ~ 2400 cycles to verification time (using *Simpira*), showing that hash-based signatures are possibly still faster than RSA3072 with QVRSA. One can argue that RSA signatures are stateless and therefore easier to use, which is true. However, if fast verification is the top requirement, as is the case in SGX, then our results show that hash-based signatures are the way to go.

The combination $(f2, h1)$ is the most conservative in terms of the assumptions needed for security. Again $d = 4$ gives the best performance (16,581 cycles). While this is faster than a *basic* implementation of RSA3072 verification, it is not competitive with RSA3072 with QVRSA.

5.3.2 Performance for 128-bit quantum security

Table 4 provides the cycle counts for signature verification for 128-bit quantum security parameters. We use chain depth $d = 2, 4, 8$, the functions $f3, f4, f5, f6$ and the hash functions $h3, h4$.

Clearly running times are slower than in the non-quantum settings. Here, RSA and ECDSA are not competitors because of their insecurity against quantum attacks. Lattice based signatures may provide a viable alternative, but they require large public keys and hashing those keys during verification may dominate verification time. Moreover, lattice based signatures are based on specific algebraic assumptions which may or may not hold in a post-quantum world. In contrast, hash-based signatures are unlikely to be affected by quantum machines.

d		f3	f4	f5	f6
2	h4	40,964	95,281	51,781	203,477
2	h3	59,473	113,789	70,290	222,496
4	h4	18,341	45,919	23,843	99,568
4	h3	30,889	58,468	36,392	112,116
8	h4	20,410	62,993	28,899	147,937
8	h3	28,937	71,520	37,426	156,465

Table 4: Signature verification performance for 128-bit quantum secure parameters, for $d = 2, 4, 8$ and $h \in \{h3, h4\}$ and $f \in \{f3, f4, f5, f6\}$. The reported numbers measure processor cycle counts for signature verification (lower is better).

In Table 4 we again see that the optimal chain depth is $d = 4$. The functions $(f3, h4)$ give the best performance, but also require the strongest security assumptions. The functions $(f4, h3)$ are the most conservative, but are slower than than fastest Rijndael-based construction. These results may renew the interest in the Rijndael cipher with a 256-bit block.

6. CONCLUSIONS

We formalized the concept of a Surnaming mechanism, as needed for in Intel's SGX technology. Although the concept is closely related to a digital signature, there are important subtle differences. For example, a Surnaming mechanism can be implemented with no conditional statements and this can greatly reduce the potential for implementation errors.

We explained that SGX gives rise to an unusual design constraint: we need a signature scheme with the fastest possible verification time and a short public-key. Signature size is immaterial. We explained the mechanism currently used by SGX, namely RSA3072 with $e = 3$ and QVRSA, compared its performance to other signature schemes, to demonstrate its advantage. We later explored an possible alternative design using some newly proposed hash-based signatures. After much work in optimizing the hash-based signature, we showed that there are (only) a few specific combinations can be faster than the current SGX mechanism.

To make SGX post-quantum secure, SGX will need to move away from RSA3072 (in addition to other necessary changes in the underlying cryptographic protocols). Our experiments with quantum-secure hash-based signatures show that they are a viable replacement option.

Acknowledgments

Suppressed for anonymity.

7. REFERENCES

[1] An attack on RSA digital signature. A NIST document, 2006. http://csrc.nist.gov/groups/ST/toolkit/documents/dss/RSAstatement_10-12-06.pdf.

[2] Intel ® Software Guard Extensions Programming Reference, 2014. https://software.intel.com/en-us/isa-extensions/intel-sgx.

[3] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, volume 13, 2013.

[4] M. Bellare and P. Rogaway. The exact security of digital signatures-how to sign with RSA and Rabin. In *Proceedings of the 15th Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'96, 1996.

[5] D. J. Bernstein, D. Hopwood, A. Hülsing, T. Lange, R. Niederhagen, L. Papachristodoulou, M. Schneider, P. Schwabe, and Z. Wilcox-O'Hearn. SPHINCS: practical stateless hash-based signatures. In *Proceedings of EUROCRYPT 2015*, pages 368–397, 2015.

[6] D. Bleichenbacher. Forging some RSA signatures with pencil and paper. Crypto 2006, Rump Session, 2006. https://www.iacr.org/conferences/crypto2006/rumpsched.html.

[7] D. Bleichenbacher and U. M. Maurer. On the efficiency of one-time digital signatures. In *Advances in Cryptology - ASIACRYPT '96, International Conference on the Theory and Applications of Cryptology and Information Security, Kyongju, Korea, November 3-7, 1996, Proceedings*, pages 145–158, 1996.

[8] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. *J. Cryptology*, 17(4):297–319, 2004.

[9] J. Buchmann, E. Dahmen, S. Ereth, A. Hülsing, and M. Rückert. On the security of the Winternitz one-time signature scheme. Cryptology ePrint Archive, Report 2011/191, 2011. http://eprint.iacr.org/.

[10] S. Gueron. Intel's New AES Instructions for Enhanced Performance and Security. In *Fast Software Encryption, 16th International Workshop, FSE 2009, Leuven, Belgium, February 22-25, 2009, Revised Selected Papers*, pages 51–66, 2009.

[11] S. Gueron. Quick verification of RSA signatures. In *Information Technology: New Generations (ITNG), 2011 Eighth International Conference on*, pages 382–386, April 2011.

[12] S. Gueron. Intel® Advanced Encryption Standard (AES) New Instructions Set. Available at: https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set, September 2012. Revision 3.01.

[13] S. Gueron. A j-Lanes Tree Hashing Mode and j-Lanes SHA-256. *J. Information Security*, 4(1):7–11, 2014.

[14] S. Gueron. A memory encryption engine suitable for general purpose processors. Cryptology ePrint Archive, Report 2016/204, 2016. http://eprint.iacr.org/.

[15] S. Gueron and V. Krasnov. Improved P256 ECC performance by means of a dedicated function for modular inversion modulo the P-256 group order. OpenSSL patch, 2015. .

[16] S. Gueron and N. Mouha. Simpira v2: A family of efficient permutations using the AES round function. Cryptology ePrint Archive, Report 2016/122, 2016. http://eprint.iacr.org/.

[17] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 11:1–11:1, New York, NY, USA, 2013. ACM.

[18] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen. Intel ®Software Guard Extensions: EPID provisioning and attestation services. *White Paper*, April 2016.

[19] B. S. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447, Oct. 2015.

[20] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 10:1–10:1, New York, NY, USA, 2013. ACM.

[21] A. Menezes. Another look at HMQV. Cryptology ePrint Archive, Report 2005/205, 2005. http://eprint.iacr.org/.

[22] R. C. Merkle. A certified digital signature. In *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, pages 218–238, 1989.

[23] K. Nyberg and A. Rueppel. A new signature scheme based on the DSA giving message recovery. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, CCS '93, 1993.

[24] L. Reyzin and N. Reyzin. Better than biba: Short one-time signatures with fast signing and verifying. In *Information Security and Privacy, 7th Australian Conference, ACISP 2002, Melbourne, Australia, July 3-5, 2002, Proceedings*, pages 144–153, 2002.

[25] R. L. Rivest, M. E. Hellman, J. C. Anderson, and J. W. Lyons. Responses to NIST's proposal. *Communications of the ACM*, 35(7):41–54, July 1992.