AUTONOMOUS SECURE REMOTE ATTESTATION EVEN WHEN ALL USED AND TO BE USED DIGITAL KEYS LEAK

Marten van Dijk
CWI Amsterdam & University of Connecticut
marten.van.dijk@cwi.nl

Deniz Gurevin
University of Connecticut
deniz.gurevin@uconn.edu

Chenglu Jin
CWI Amsterdam
chenglu.jin@cwi.nl

Omer Khan
University of Connecticut
omer.khan@uconn.edu

Phuong Ha Nguyen
eBay
phuongha.ntu@gmail.com

May 8, 2021

ABSTRACT

We provide a new remote attestation scheme for secure processor technology, which is secure in the presence of an All Digital State Observing (ADSO) adversary. To accomplish this, we obfuscate session signing keys using a silicon Physical Unclonable Function (PUF) with an extended interface that combines the LPN-PUF concept with a repetition code for small failure probabilities, and we introduce a new signature scheme that only needs a message dependent subset of a session signing key for computing a signature and whose signatures cannot be successfully forged even if one subset per session signing key leaks. Our solution for remote attestation shows that results computed by enclaves can be properly verified even when an ADSO-adversary is present. For $N = 2^l$ sessions, implementation results show that signing takes $934.9 + 0.6 \cdot l$ ms and produces a signature of $8.2 + 0.03 \cdot l$ KB, and verification by a remote user takes $118.2 + 0.4 \cdot l$ ms. During initialization, generation of all session keys takes $819.3 \cdot N$ ms and corresponding storage is $3 \cdot 10^{-5} + 0.12 \cdot N$ MB.

Keywords Remote Attestation · One Time Signatures · Secure Processor Architecture · Physical Unclonable Function

1 Introduction

Secure processor architecture design [1–9] is based on two core principles: hardware isolation and remote attestation. Hardware isolation allows one to run a code snippet in a so-called enclave that is isolated from the OS and other enclaves with the goal to keep its internal computations private. Hardware isolation implements access control which, for example, disallows the OS to access reserved DRAM for enclaves. Besides being able to execute code in a trusted execution environment that guarantees privacy, a remote user also needs to be able to verify whether a computed result originated from the executed code. Remote attestation is a protocol that signs and binds a computed result to the enclave code that produced it and the processor identity.

Hardware isolation has shown to be elusive. With the developing capabilities of adversaries and side channel attacks, vulnerabilities of secure processors continuously keep getting exploited leaking private digital state of the processor. A recent survey [10] has shown that Intel SGX has been susceptible to a wide range of attacks in the recent years [11–39]. We may conclude that hardware isolation as is implemented today for executing enclave code cannot guarantee privacy. In fact, any internally computed enclave value may potentially leak; we cannot make any solid privacy guarantee.

Nevertheless hardware isolation does offer the guarantee that unmodified enclave code executes. That is, the enclave code itself may have vulnerabilities and can possibly be exploited through its own I/O interactions, but the secure processor architecture does not add additional attack points that can be used (by, for example, the OS) to modify enclave code. So, while privacy cannot be guaranteed, we can still guarantee untampered enclave execution.

A remote user needs remote attestation to verify the results produced by such an enclave. However, since all digital states of the secure processor can potentially leak, how can we design a Remote Attestation (RA) scheme which remains secure under a strong All Digital State Observing (ADSO) adversary? From the perspective of the ADSO adversary, the processor is a white box with no hidden functionalities, which reveals all its internal digital state all the time, and which allows it to be used through its interface specifications. This seems to suggest that we cannot rely on any secret digital key, hence, we cannot use our crypto tool kit. Current remote attestation protocols [40–46] commonly rely on some level of digital privacy and therefore seem not applicable.

If a secret digital key leaks in a forward secure scheme, then all next keys are potentially exposed implying that the corresponding public key needs to be revoked. Additional mechanisms to minimize the damage of key exposure, i.e., intrusion detection and prompt key revocation, are required [47]. To overcome this shortcoming, key-insulated schemes (KIS) [48–52] and intrusion-resilient schemes (IRS) [53, 54] are proposed in order to protect *future* as well as past secret keys even if the current key is leaked. However, KIS and IRS assume for signing a 'user' and a 'base' that are not compromised together at the same time. We want to resist the much stronger ADSO adversary who can observe all digital state (of both the 'user' and 'base' at the same time). We do not want to depend on distributed trust or a trusted third party in our RA protocol.

We propose a RA protocol between an RA enclave at the secure processor and a remote user. Even though the enclave cannot base security on secret digital state, we allow it to have access to a Physical Unclonable Function (PUF) which implements secret 'analogue' state. However, as soon as any analogue state is interpreted in digital state for computation, the digital state leaks to the ADSO adversary. Nevertheless, by introducing a new signature scheme, we show that we can use such digital state to build a secure RA protocol.

The ADSO adversary

- can compromise and alter the OS, run own enclave code, and can execute or interact with instantiations of the RA enclave,
- can observe all digital state, which includes all intermediate digital values computed by the RA enclave as well as all digital storage together with register values, permanent storage, and fused (endorsement) keys,
- cannot circumvent hardware isolation in that it must adhere to the access control implemented for the PUF, for persistent on-chip digital storage, and for the RA enclave. In particular, the ADSO adversary cannot circumvent the usual access to digital state and tamper with values computed inside the RA enclave or stored in the on-chip digital storage.
- We assume that the secure processor keeps on functioning according to its specification. This implies no embedded hardware Trojan or other physical attack that changes processor functionality.

We stress that our RA solution does not involve a Trusted Third Party (TTP) who, for example, may provides new keys and is in charge of key management. Relying on a TTP requires trust in some company out of users' control. When we buy secure processors, we do not want continued dependence on a third party that can make unilateral decisions, instead we want *digital autonomy*. Our proposed RA enclave can be bootstrapped by any user without need for permission from another third-party. In particular, the owner of the processor does not need to ask for help/permission from another entity and can remain digitally autonomous.

1.1 Approach

Figure 1 illustrates our RA solution that resists the powerful ADSO adversary. At the core of our solution we use (1) an obfuscation mechanism based on a PUF with tight access control, that is, each enclave has access to its own subset of the challenge response space of the PUF, and we use (2) a new signing primitive, called a One Time Signature with Secret Key Exposure (OTS-SKE) scheme consisting of KEYGEN, SIGN, and VERIFY. The OTS-SKE primitive has the property that signing a message during a session uses a subset of the "session signing key", where the subset is message specific. Its security guarantee states that if each session signs at most one message, then the collection of all the sub keys over all sessions (from past, current, and future) cannot help an adversary forge a signature for a new message. Therefore, the ADSO adversary cannot learn useful information from digital state computed during the RA enclave for spoofing remote attestation. Also, the untrusted memory only stores obfuscated session keys and is of no help to the ADSO adversary as it cannot have access to the corresponding challenge response space (of the PUF).

We do not assume any adversary during initialization mode where KEYGEN generates a public key and a sequence of session keys, and stores all the session keys in an obfuscated form. An ADSO adversary during initialization would learn much more than only a subset of each session key per session, and would be able to break the remote attestation solution. Initialization mode is a one-time event during which RA is bootstrapped. It happens before the continuously active operation mode where signatures are being generated and the ADSO adversary is present.

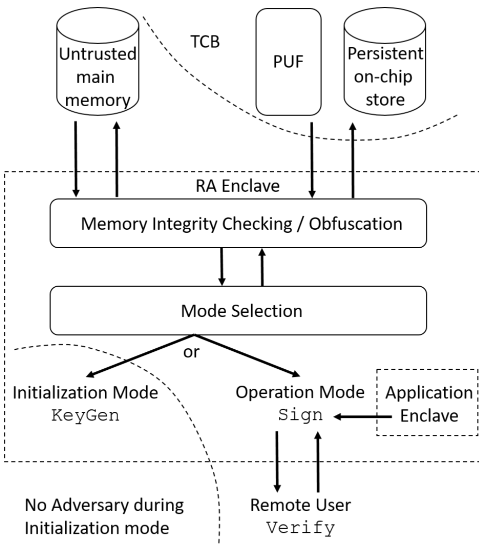

Figure 1: Remote Attestation Enclave.

The remote user uses the public key (which is universal for all sessions) to verify the received signature. During the remote attestation protocol, the remote user uses a fresh nonce and this convinces the remote user that the signature is fresh and must come from the remote attestation enclave. Since the message is a combination of the result of a computation by an application enclave together with the application enclave's hash/measurement, the remote user is able to identify both the application enclave and the processor (which corresponds to a unique PUF) on which it is executed.

1.2 Contributions and Outline

Our contributions and outline are as follows:

- Section 2 carefully describes the Trusted Computing Base (TCB) which consists of a PUF, a persistent tamper-resistant on-chip store, hardware isolation for access control, and a True Random Number Generator (TRNG). Our solution uses a strong silicon PUF and we explain an extended PUF interface (for any enclave) which combines in a new way the LPN-PUF concept [55, 56] together with a repetition code in order to reach a small failure probability.
- Section 3 introduces the new OTS-SKE signature primitive. We show a post-quantum secure OTS-SKE primitive by adopting a second pre-image resistance (SPR) Merkle tree construction [57].
- In Section 4 we provide a new remote attestation protocol that can resist the above described ADSO adversary who can observe all digital states. The protocol does not rely on a trusted third party and offers digital autonomy instead.
- Simulation results are in Section 5: For $N = 2^l$ sessions, signing takes $934.9 + 0.6 \cdot l$ ms and produces a signature of $8.2 + 0.03 \cdot l$ KB, and verification by a remote user takes $118.2 + 0.4 \cdot l$ ms. During initialization, generation of all session keys takes $819.3 \cdot N$ ms and corresponding storage is $3 \cdot 10^{-5} + 0.12 \cdot N$ MB.

Our RA solution offers secure verification of computed results. In a way it offers an audible heartbeat of the processor showing that it is alive and healthy.

2 Trusted Computing Base (TCB)

The TCB consists of a PUF, a persistent tamper-resistant on-chip store, hardware isolation for access control, and a TRNG:

Physical Unclonable Function (PUF). Since we cannot rely on digital secret state alone, we use a PUF as a sort of *analogue master key* which can be used to produce a one time padding mechanism for obfuscating keys (in our case obfuscating session signing keys for remote attestation). In order to avoid an adversary from “reading out” all the PUF’s ‘key material’ or challenge-response pairs, we require a so-called strong PUF in the sense that it has an ‘exponentially’ large challenge-response space (that cannot be read out in ‘feasible time’) and the behavior of the strong PUF cannot be cloned (based on assuming the hardness of machine-learning the PUF’s challenge-response behavior and based on assuming that manufacturing variations that produce the PUF’s behavior cannot be duplicated in hardware). By using the PUF we get rid off persistent secret digital state (like a master key).

Persistent Tamper-Resistant On-Chip Store. Instead of a persistent secret digital state, we may use a small persistent tamper-resistant on-chip store for maintaining *persistent public digital state*. This can be used to store for example a root-hash of an authentication tree for memory integrity checking, which in turn can be used to store, as we will propose for remote attestation, an irreversible counter and a corresponding universal public key. The on-chip store cannot be modified by the powerful ADSO adversary (as this would require certain physical access to the processor which we assume is not one of the attacker capabilities). The on-chip store is non-volatile memory which stores persistent state across context switches or enclave removal after which an enclave’s execution resumes or restarts. Such persistent state is necessary for implementing access control of managed keys.

Hardware Isolation. Hardware isolation provides two security properties. First, under a much less capable ‘default’ adversarial model hardware isolation provides secure enclave execution such that the OS or other enclaves cannot access intermediate computed values or reserved DRAM. Under the ADSO adversarial model, all digital state can be leaked and we assume that in this sense the hardware isolation barrier is crossed and does not offer privacy.

Second, hardware isolation is at its core implemented as an access control system. E.g., the virtual to physical (DRAM) address translation is mixed with an efficient access control mechanism that disallows the OS to access reserved enclave DRAM. This type of access control is not based on any secrets and is hard coded in hardware. Under the ADSO adversarial model this type of hardware isolation is not compromised. In particular, access to the PUF and the persistent tamper-resistant on-chip store can be made enclave dependent – and the strong adversary cannot circumvent such access control (since we assume that it cannot modify the processor’s hardware).

We will propose to automatically embed the enclave’s measurement in any challenge that is input to the PUF – this guarantees that different enclaves get access to different ‘partitions’ of the challenge-response space. We will also propose to allow an enclave to allocate and reserve memory in the on-chip store such that no other enclave can modify (overwrite) this memory (notice that the ADSO adversary can read this memory since all digital state becomes visible to him/her).

True Random Number Generator (TRNG). In order to generate new keys, we may use some pseudo random number generator based on some secret seed. Since the ADSO adversary can read all digital state, possibly including such a seed, one will need to discard the seed immediately after new keys have been generated. However, if the seed was extracted out of a mix of previous digital state of the system as a whole, then the ADSO adversary may now be able to successfully predict (the whole or a large fraction of) the seed. For this reason, it is important to use a TRNG such that the ADSO-adversary is guaranteed not to have any information about such a seed.

Trusted Computing Base (TCB). The TCB consists of (1) assuming secrecy of the internal analogue processing of the PUF (i.e., the PUF’s analogue internal working in the form of a predictive model that has learned about the manufacturing variations that cause each fabricated PUF to illicit unique behavior remains hidden and ‘unclonable’), (2) assumes tamper resistance of a small persistent on-chip store, and (3) assumes tamper resistance of the hardware isolation functionality of the secure processor that regulates access control to modules (such as the PUF and on-chip store above) and to enclave code instructions (making sure its flow is uninterrupted and only enclave code is executed), and (4) assumes a TRNG. Only when considering a less capable adversary, the TCB may be extended with hardware isolation that allows executing secure enclaves without other enclaves or the OS being able to observe its intermediate computations. Such privacy property is not assumed for the ADSO adversary.

Below we detail our proposed PUF interface, on-chip store interface, and explain shortly that our schemes are compatible with any existing secure processor architecture technology.

2.1 PUF Interface

Intuitively, a strong silicon PUF is a fingerprint of a chip, that leverages process manufacturing variation to generate a unique function taking “challenges” as input and generating “responses” as output, which cannot be cloned in hardware (the PUF’s internal behavior, e.g. its unique physical characteristics or behavior of its wires, cannot be read out accurately enough; also it is not feasible to manufacture two PUFs with the same responses to a significant subset of challenges) and cannot be efficiently learned given a “polynomial number” of challenge response pairs (making it impossible to impersonate/clone the function’s behavior to a new random challenge in software). This informal definition implies that a ‘strong’ PUF must have a large number of challenge response pairs so that only an insignificant fraction can be read out within a practical time window. One possibility is to create an ‘exponentially’ large challenge response space and assume that machine learning is computational hard for the PUF design. Another option is to throttle access to the PUF so that it must take a too long time window for reading out a sufficient number of challenge response pairs. For a more detailed discussion see [58]. Our PUF interface does not throttle access because we want to achieve as high as a throughput as possible for key generation and signing such that our solution has wider applicability. For completeness, we notice that responses are measured and are subject to measurement noise due to voltage and temperature variations and aging. Since responses are subject to measurement noise, one can observe response reliability rather than just the plain response and this allows more accurate machine learning [59, 60].

We introduce a secure processor hardware extension, called EPUF, which can only be called from inside an enclave. Given a challenge c, EPUF takes the unique measurement MRENCLAVE of the enclave (computed as a hash of the enclave code) and concatenates it with the challenge. Then, it takes the hash of this value and feeds this new input to the PUF, and finally returns the PUF response. This partitions access to the PUF among enclaves: Two different enclaves can only access the PUF for the same input if their challenges c and c' together with their measurements M and M' form a hash collision $\text{Hash}(M\|c) = \text{Hash}(M'\|c')$. Assuming a collision resistant hash function, even an intelligently designed enclave will only find collisions with negligible probability. As a result, it is as if the PUF’s actual challenge response space is partitioned among different enclaves. In this way we can make the EPUF functionality available to all enclaves, and not have to restrict it to just certain specific ‘legitimate’ enclaves (so that adversarially designed enclaves are prevented from obtaining challenge response pairs that are used by legitimate enclaves). The EPUF functionality in essence gives each enclave its own private PUF – and the ADSO adversary cannot circumvent the EPUF interface and break this property.

Existence of Strong PUFs. The current state-of-the-art strong silicon PUF design is the Interpose PUF (iPUF) [61]. The iPUF is a combination of two XOR Arbiter PUFs that are connected through a kind of forwarding mechanism (where the response bit of the ‘top’ XOR Arbiter PUF is inserted in the middle of the challenge for the ‘lower’ XOR Arbiter PUF). The iPUF has seen recent ML based attacks [60, 62] and calls the existence of a strong PUF into question. Even though the published attacks still seem to leave concrete parameter settings for which the iPUF attacks have not yet been demonstrated, it remains an open problem for how long such settings remain secure (and parameter settings cannot be chosen too ‘large’ as this hurts the reliability of the iPUF). It may very well be, that the iPUF design itself will need to be adapted by replacing Arbiter PUFs with Feed Forward Arbiter PUFs together with a whole new ML based security analysis.

In this paper, we assume the existence of a strong silicon PUF. We notice that if such a strong silicon PUF does not exist, then we need to implement (part of) the extended PUF interface as discussed below in hardware (to redefine and replace the EPUF functionality) and assume that its digital computation is not visible to the ADSO adversary. That is, we need to slightly weaken the ADSO adversary and assume that no side channel attack or other mechanism can be used to leak the internal digital computation in the hardware implementation of (part of) the extended PUF interface (this increases the TCB). Since the underlying PUF is now protected by the hardware implementation of (part of) the extended PUF interface, we can replace the iPUF with the original Arbiter PUF. Even though Arbiter PUFs can easily be broken using ML [63], the interface disallows direct access to its challenge response pairs and makes such ML impossible (the security is reduced to the hardness of inverting the underlying pseudo random function and solving the LPN problem on which the interface is based). See [56] as an example of how part of an extended PUF interface may be implemented in hardware (FPGA).

Notice that [56] uses a Ring Oscilator (RO) PUF which is weak in the sense that it always reconstructs the same response vector and this vector is combined with the LPN concept. In a larger TCB where we assume that the hardware implementation of the extended PUF interface does not leak digitally computed values to the adversary, we implicitly assume defense mechanisms to thwart side channel leakage. In practice one always leaks some information with every execution of the interface, hence, a single repeatedly reconstructed response vector will ultimately leak. For this reason, we propose the Arbiter PUF which has a large challenge response space allowing for fresh responses.

To correct for measurement noise, we will not want to use fuzzy extractors which use public "helper information" in the form of parity check information of an error correcting code applied to responses. Linear parity checks also leak information about reconstructed responses. And sufficient information may then be used to machine learn the Arbiter PUF. Since it is unclear whether such an attack is hard or not, we propose to use the LPN based concept which eliminates such ML modeling of the Arbiter PUF because the security of the extended PUF interface is reduced to the hardness of the LPN problem.

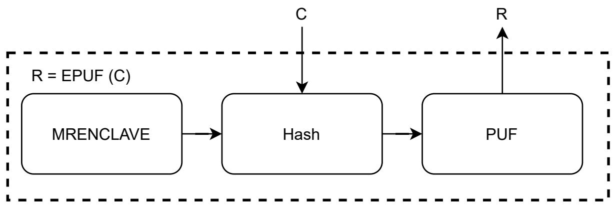

Figure 2: The EPUF call takes a challenge $C \in \{0,1\}^\lambda$ and computes a hash with the measurement of the enclave MRENCLAVE to produce a challenge for the PUF. The response $R \in \{0,1\}$ of the PUF is the output of the EPUF call.

Extended PUF Interface. We explain our extended PUF interface which calls the EPUF functionality, see Figure 2. We notice that this extended interface computes in an enclave, hence, all its intermediate computed digital values are observed by the ADSO adversary. For this reason we should not repeatedly use the same EPUF output from a remote attestation session to the next remote attestation session for bootstrapping some signing operation. This is the main reason to use a strong PUF (and not build an extended interface around a so-called weak PUF, which has a 'small' challenge response space, as in [55, 56]).

In Arbiter PUF like designs such as the iPUF, two stimuli race against each other following complementary paths indicated by a challenge c. Which stimulus arrives first determines the response bit $r \in \{0,1\}$. The difference in arrival times of the stimuli is modeled as a difference in aggregated delays that characterize each of the two paths. Without measurement noise, this is a deterministic function. With measurement noise, the arrival times may vary and as a result the response bit flips. Let p_c be the probability that the response bit flips due to measurement noise given a selected challenge c. Different challenges indicate different complementary paths over which stimuli race against each other. And for this reason the p_c are generally different for different c. Challenge c is selected uniformly from the challenge space, denoted by $\{0,1\}^\lambda$. This gives rise to a distribution of p_c with respect to the uniform distribution c over the challenge space $\{0,1\}^\lambda$. So, when letting r be a first measurement, r' be a second measurement, and $e = r + r'$ (XOR operation) represent the error between the two, then the probability $r \neq r'$ is equal to

$$Pr[e=1] = \frac{1}{2^\lambda} \sum_{c \in \{0,1\}^\lambda} p_c = \mathbb{E}[p_c]$$

(where the probability is over uniformly selected c). We will denote this probability by P and assume $P \leq 1/2$.

When generating a challenge response pair by calling GETCRP(s, x, c), see Algorithm 1 for details, we use inputs s, x, c as random seeds; proper usage of GETCRP discards and forgets s and x after obtaining the outputted challenge response pair (C, R); c will be part of C. The EPUF functionality is used to get response bits $r_{i,j}$ based on challenges $Hash(i\|j\|c)$ derived from challenge seed c. We use a 'repetition code' of length $2k+1$ to encode bits x_i as $(y_{i,1} = x_i + r_{i,1}, \ldots, y_{i,2k+1} = x_i + r_{i,2k+1})$. These repetition code words, collectively denoted by y, become part of challenge C. The repetition code will allow us to correct for measurement errors and obtain a confidence score reflecting how certain we are about whether an $x_i = 0$ or $x_i = 1$ was encoded:

After re-measuring response bits $r'_{i,j}$ in GETRESPONSE(C), see Algorithm 1 for details, we add these to the $y_{i,j}$ and this leads to a true repetition code word with errors: $(x'_{i,1} = x_i + e_{i,1}, \ldots, x'_{i,2k+1} = x_i + e_{i,2k+1})$ where $e_{i,j} = r_{i,j} + r'_{i,j}$. Decoding of the repetition code is simply by majority voting and if the vote wins by $k+1+j$ votes against $k-j$ votes, then we call j its confidence score con. The confidence score con_i of a majority vote x'_i tells us about how likely $x'_i = x_i$. (Our construction uses a repetition code, however, other error correcting codes can be used as well and may lead to an improved parameter setting.)

It turns out that in practice the EPUF functionality provides biased response bits (due to a bias in the comparator(s) used in silicon PUF designs), e.g., for 47% of all challenges the response bit is a 1 and for 53% of all challenges the response bit is a 0. This implies that $(y_{i,1} = x_i + r_{i,1}, \ldots, y_{i,2k+1} = x_i + r_{i,2k+1})$ reveals some information about x_i (the values $r_{i,j}$ can be seen as errors at error rate 0.47) and we can learn bits x_i up to some bias. The bias in x_i given information y gets larger the longer the repetition code, i.e., the larger k. If we do not mind the resulting bias, then we may replace challenge C by $(c, y, f(0\|x))$ and use $R = f(1\|x)$. This allows us to remove the matrix multiplication in GETCRP and Gaussian elimination from GETRESPONSE, especially the latter can become a bottleneck in achieving high throughput. However, in this study we do mind the bias and therefore adapt the strategy of [55, 56] in GETCRP and use the random input s together with a fixed publicly known a-priori selected random binary $\lambda \times m$ matrix A to compute $b = s \cdot A + x$. Challenge C contains vector b and we notice that knowledge of b, A, and biased vector x reduces to the Learning Parity with Noise (LPN) problem. Assuming its computational hardness shows that s remains secret. See [55, 56] for references and discussion, current state-of-the-art algorithms for solving the LPN problem in

Algorithm 1 Extended PUF Interface

$A \in \{0,1\}^{\lambda \times m}$ is a global variable a-priori chosen as a random matrix; $Hash(.)$ is a collision resistant hash function and $f(.)$ is a pseudo random function; inputs $s \in \{0,1\}^{\lambda}$, $x \in \{0,1\}^{m}$, $c \in \{0,1\}^{\lambda}$ are assumed to be (pseudo) random (based on a PRNG with random seed extracted from a TRNG); All $+$-operations are binary (XOR) except when counting confidence con_i; See Algorithm 3, ModeID indicates an enclave instantiation which calls the extended PUF interface.

```
1: procedure GETCRP($s, x, c$ )
2:     for $i \in \{1,\dots,m\}$, $j \in \{1,\dots,2k+1\}$ do
3:         $r_{i,j} = EPUF(Hash(i\|j\|c\|\text{ModeID}))$
4:         $y_{i,j} = x_i + r_{i,j}$
5:     end for
6:     $y = \{y_{i,j}\}$
7:     $b = s \cdot A + x$
8:     $C = (c, y, b, f(0\|s))$
9:     $R = f(1\|s)$
10:    return CRP pair $(C, R)$
11: end procedure
1: procedure GETRESPONSE($C$)
2:     Extract $(c, y, b, f_0) = C$
3:     $i = 1$; $SizeI = 0$; $FoundA_I$ =false
4:     while $i \le m$ and $FoundA_I$ =false do
5:         $con_i = 0$
6:         for $j \in \{1,\dots,2k+1\}$ do
7:             $r'_{i,j} = EPUF(Hash(i\|j\|c\|\text{ModeID}))$
8:             $x'_{i,j} = y_{i,j} + r'_{i,j}$
9:                                      ▷ $x'_{i,j} = x_i + e_{i,j}$ where $e_{i,j} = r_{i,j} + r'_{i,j}$
10:            $con_i = con_i + x'_{i,j}$
11:        end for
12:        if $2k + 1 - con_i \ge k + 1$ then
13:            $x'_i = 0$, $con_i = (2k + 1 - con_i) - (k + 1)$
14:        end if
15:        if $con_i \ge k + 1$ then
16:            $x'_i = 1$, $con_i = con_i - (k + 1)$
17:        end if
18:                                      ▷ Both cases: $con_i = |k + 1/2 - \sum_{j=1}^{2k+1} e_{i,j}| - 1/2$
19:                                      ▷ $x'_i$ equals the majority vote among $\{x'_{i,j}\}_{j=1}^{2k+1}$
20:                                      ▷ $x'_i = x_i$ if and only if $\sum_{j=1}^{2k+1} e_{i,j} \ge k + 1$
21:        if $con_i \ge T = k - \lceil (2k+1)P \rceil$ then
22:            Add $i$ to set $\mathcal{I}$ and $SizeI$++
23:            Let $A_{\mathcal{I}}$ be the columns of $A$ indexed by $\mathcal{I}$
24:            if $A_{\mathcal{I}}$ has full row rank then $FoundA_I$ =true
25:        end if
26:        $i$++
27:    end while
28:    if $FoundA_I$ =true then
29:        Reduce $A_{\mathcal{I}}$ to a square invertible matrix
30:        Corresponding to $A_{\mathcal{I}}$ define $b_{\mathcal{I}}$ and $x_{\mathcal{I}}$
31:        Solve $b_{\mathcal{I}} = s \cdot A_{\mathcal{I}} + x'_{\mathcal{I}}$, i.e., $s = (b_{\mathcal{I}} + x'_{\mathcal{I}}) \cdot A_{\mathcal{I}}^{-1}$
32:        if $f_0 = f(0\|s)$ then return $R = f(1\|s)$
33:    end if
34:    return false
35: end procedure
```

subexponential time $2^{\Omega(\lambda/\log\lambda)}$ assume a much larger bias and assume a sufficient number of equations $m = \Omega(\lambda^2)$, where m is the number of columns of matrix A, which is only $m = O(\lambda)$ in our extended PUF interface (see below). For this reason we assume that λ also represents the security parameter for the LPN problem to which the extended PUF interface is reduced.

Matrix A needs to have many columns (m large enough) in order to find a square invertible submatrix $A_{\mathcal{I}}$ that corresponds to columns that relate to confident x'_i. This would mean that we have correctly retrieved $x_{\mathcal{I}} = (x_i)_{i\in\mathcal{I}}$ in GETRESPONSE and are able to solve the equation $b = s \cdot A + x$ and retrieve the correct s. Of course we cannot be sure and therefore we include in C the pseudo random function value $f(0\|s)$ so that we can check whether we obtained the correct s (and, hence, our confident x'_i are indeed without error). Once s is retrieved we can compute the response $R = f(1\|s)$ in GETRESPONSE:

$$(C, R) = (\ (c, y, b, f(0\|s)),\ f(1\|s)\ ).$$

We need a pseudo random function $f(.)$ otherwise $f(0\|s)$ in challenge C reveals information about s which can possibly be used to break the LPN problem or $f(0\|s)$ may be malleable so that information about response $R = f(1\|s)$ can be computed without gaining knowledge about s itself.

Vector R has λ bits and is used to obfuscate keys in this paper. For this reason it is important to make sure the GETRESPONSE functionality does not fail and is indeed able to retrieve s with high probability. If we implement remote attestation, then we are fine with a failure probability of 0.0001 because we can simply ask for a second signature if the first one fails to be produced or verified. When the response R is used to obfuscate keys in other (crypto) protocols, then we may not get away with a failure probability of 0.0001 and we may require a much smaller 10^{-15} failure probability based on using a longer repetition code. A more simple solution is to generate multiple R, each used to mask the same key. If all, say u, masks fail (i.e., GETRESPONSE fails), then this only happens with probability 0.0001^u which becomes negligibly small for relatively small u.

Table 1: Parameters for $\lambda = 128$ and $P = 0.1$.

m	k	fail	# EPUF calls	storage chall.
560	17	$0.976 \cdot 10^{-15}$	19600	2.6 KB
392	8	$0.953 \cdot 10^{-5}$	6664	0.91 KB
374	7	$0.995 \cdot 10^{-4}$	5610	0.78 KB
356	6	$1.02 \cdot 10^{-3}$	4628	0.66 KB

Failure Probability Analysis. Appendix A proves upper bounds on the failure probability of the extended PUF interface (due to too much measurement noise). Table 1 shows for $\lambda = 128$ (this implies sufficient security with respect to the LPN problem, see [55, 56]) how the failure probability decreases as $\approx 1.0 \cdot 10^{-(3+u)}$ for $m = 356 + 18u$ and $k = 6 + u$. From these parameters we infer that

$$m(2k + 1) = (356 + 18u)(13 + 2u)$$

EPUF calls are needed in order to retrieve the responses for each extended PUF interface call. In order to store a challenge we need $2\lambda + m$ bits plus the $m(2k + 1)$ response bits.

EPUF calls take about $\sim 10\ \mu$s on a 180nm Complex Programmable Logic Device [64] (CPLDs here can be considered as small scale FPGAs): The number of EPUF calls will be our main bottleneck – depending on the application, a parallelized hardware implementation is needed to mitigate this bottleneck; in our simulations we use $2k + 1$ PUFs in parallel for retrieving all the responses of the repetition code. Another optimization is to not retrieve all x'_i and sort and choose those with highest confidences, but to first select a set corresponding to confidences $con_i \geq T$, where $T = k - \lceil (2k + 1)P \rceil$ is a threshold (the analysis of Appendix A holds for this selection strategy). As soon as $\lambda + j$ positions with confidences at least T have been found that correspond to a subset of columns of A that together have full row rank, then we can extract a column-reduced invertible submatrix $A_{\mathcal{I}}$ and continue GETRESPONSE without the need to call EPUF for the other remaining positions which have not yet been investigated. Amortized over all GETRESPONSE queries this will save a fraction of EPUF calls.

One-Time Pad (OTP) Interface. We use the extended PUF interface to have access to challenge response pairs where responses are typically $\lambda = 128$ bits. These responses can be used to One-Time Pad (OTP) sensitive information such as keys, as depicted in detail in Algorithm 2. Here, we first OTP a Key which in turn is used to encrypt a larger Vector using AES symmetric key encryption. This is because each EPUF call takes about 10 μs and the extended PUF interface needs, e.g., 5610 EPUF calls per Key for failure probability $\approx 10^{-4}$ taking much more time than one AES call. (We remark that if OTP is used for multiple vectors, then we do not need to include c as a component of challenge C in OTP; we only need to remember one mapping from which a sequence of c values can be generated. This will save some storage.)

Algorithm 2 One-Time Pad Interface

We assume a Pseudo Random Number Generator (PRNG) bootstrapped from an initial seed to generate pseudo random bit strings.

```
1: procedure OTP(Vector)
2:     $(s, x, c, \text{Key}) \leftarrow \text{PRNG}(seed)$
3:     $(C, R) \leftarrow$ GETCRP$(s, x, c)$
4:     OTPKey $\leftarrow (C, \text{Key} \oplus R)$
5:     $(K_1, \ldots, K_k) =$Vector
6:     AESVector $= (\text{AES}_{\text{Key}}(K_1), \ldots, \text{AES}_{\text{Key}}(K_k))$
7:     OTPVector $= (\text{AESVector}, \text{OTPKey})$
8:     return OTPVector
9: end procedure
1: procedure OTPINVERSE(OTPVector)
2:     $(\text{AESVector}, \text{OTPKey}) = \text{OPTVector}$
3:     $(C, z) \leftarrow$ OTPKey
4:     $R \leftarrow$ GETRESPONSE$(C)$
5:     Key $\leftarrow z \oplus R$
6:     $(C_1, \ldots, C_k) =$AESVector
7:     Vector $= (\text{AES}^{-1}_{\text{Key}}(C_1), \ldots, \text{AES}^{-1}_{\text{Key}}(C_k))$
8:     return Vector
9: end procedure
```

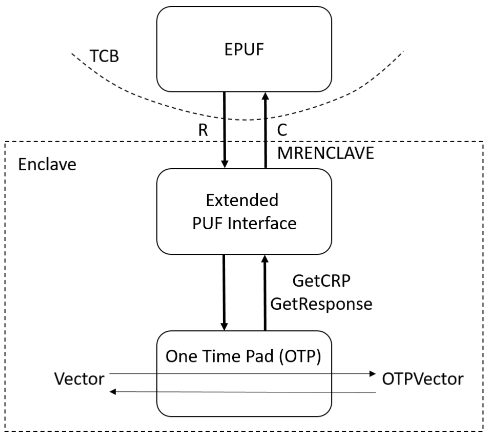

Figure 3: Key Obfuscation.

Traditionally, besides device authentication, PUFs have been used to obfuscate master keys in hardware. That is, rather than fusing a master key in hardware, a master key one-time padded with a response is fused in hardware. The circuitry uses a hard-coded challenge to extract the response from the PUF with which the master key can be retrieved. Here, only a so-called weak PUF with one (or a couple) challenge response pairs is needed. This approach means that the master key remains hidden in power-off mode. Algorithm 2 extends this traditional OTP mechanism to an interface that can obfuscate any key or data of one's choice.

Figure 3 puts all ingredients together in one larger flow diagram. The extended PUF interface and OTP reside in the enclave while the EPUF functionality is in the TCB. In the remote attestation scheme we only use OTP during initialization mode where we do not assume any adversary. Once initialization mode ends, the seed used by the PRNG in OTP must be discarded because in operation mode the ADSO adversary will be present. The next subsection explains mode selection in more detail.

2.2 On-Chip Store Interface

The proposed on-chip storage with interface in Table 2 allows each enclave to allocate persistent and tamper resistant storage (non-volatile memory) in the form of one single block of data. In the ADSO adversarial model all digital state becomes available to the adversary including the content of the on-chip store. For this reason we allow the storage to be publicly accessible in that all its content can be read using ELOADa by any enclave or (untrusted) OS. However, only the enclave can allocate its persistent on-chip storage by using EALL and only the enclave can write data into its allocated block by using ESTORE. This implies that the content of the enclave's storage can only be controlled by the enclave itself and can be used to implement for example a persistent irreversible counter (for our RA scheme), such that even the ADSO adversary cannot circumvent the access control implemented by the interface.

Command	Permission	Argument	Action
EALL	MRENCLAVE	-	Allocates and writes the pair (MRENCLAVE,nill) if a pair for MRENCLAVE does not already exist
EDALLa	Anywhere	MRENCLAVE	Resets the pair (MRENCLAVE,Data) to a default empty value if it exists
EDALL	MRENCLAVE	-	Resets the pair (MRENCLAVE,Data) to a default empty value if it exists
ELOADa	Anywhere	-	Outputs all pairs of the persistent on-chip store
ELOAD	MRENCLAVE	-	Outputs Data if pair (MRENCLAVE,Data) exists, Outputs Fail if no pair for MRENCLAVE exists
ESTORE	MRENCLAVE	DataNew	Overwrites Data with DataNew in (MRENCLAVE, Data) if it exists

Table 2: Interface to persistent on-chip store.

We have two additional commands: ELOAD allows the enclave itself to easily read its own storage (without having to read the whole content of the on-chip store as in ELOADa). EDALLa allows the OS to do memory management (in case the designer of the enclave code did not include appropriate code for de-allocation). In order for the OS to specify MRENCLAVE as an argument in EDALLa, it can use ELOADa. Since EDALLa can be called from anywhere, this is the only method for an adversary to break persistence (by removing an enclave's allocated storage).

Algorithm 3 On-Chip Persistent Store Interface Extension

We assume a MEMORYINTEGRITYCHECKING mechanism with a RootHash stored in the persistent on-chip store for the enclave [65]; we assume the leaves of an authenticated search tree represent pairs (ModeID,State) where ModeID represents the identity of an instantiation of the enclave's execution with State being a data block which represents the current state or progress made by this instantiation. State is generally updated by INITIALIZATIONMODE and OPERATIONMODE and as soon as an updated State is evicted from the cache, MEMORYINTEGRITYCHECKING is used to store it in off-chip untrusted memory with an updated RootHash in the persistent on-chip store for the enclave.

```
1: procedure ModeSelectionWrapper(ModeID)
2:     $x \leftarrow$ ELOAD
3:     if $x$ =Fail then
4:         EALL
5:         Use MemoryIntegrityChecking:
6:             To store (ModeID,Default) and build tree
7:         Continue InitializationMode(ModeID)
8:     else
9:         RootHash= $x$
10:        Use MemoryIntegrityChecking
11:            To find a pair (ModeID,State)
12:        if pair does not exist then
13:            Use MemoryIntegrityChecking:
14:                To store (ModeID,Default)
15:            Continue InitializationMode(ModeID)
16:        else
17:            Continue OperationMode(State)
18:                                        ▷ OperationMode is in charge of DALL
19:        end if
20:    end if
21: end procedure
```

Mode Selection Wrapper. Figure 4 depicts a flow diagram of a wrapper which is used to select when main code should start in *initialization mode* or *operation mode*. Detailed pseudo code is given in Algorithm 3. The selection is

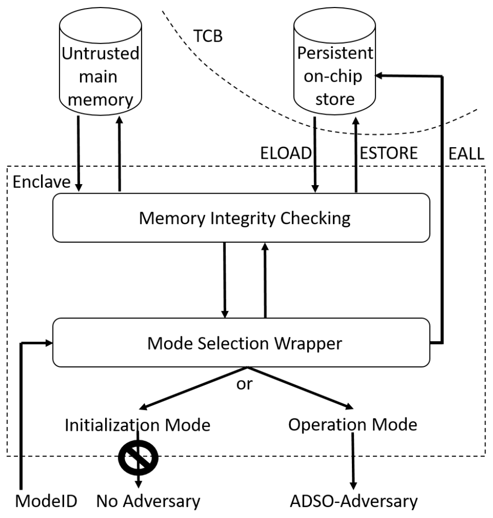

Figure 4: Mode Selection.

based on the data block stored for the enclave in the persistent on-chip store. The data block represents the state in which operation mode should continue. The wrapper takes as input a ModeID which tells the enclave which version of its execution should proceed. There may be multiple ModeID that are being executed since multiple copies of the same enclave may have been created in parallel, each executing for a different ModeID. If the inputted ModeID does not have associated data, then the wrapper will proceed in initialization mode for the given ModeID. In particular, when the enclave is created for the first time no allocated persistent on-chip storage exists and EALL is called before continuing in initialization mode.

In order to be able to store multiple (ModeID,State) pairs, we use a memory integrity checking mechanism with its RootHash in the persistent on-chip store. The memory integrity checking mechanism calls the ELOAD and ESTORE functionality for the RootHash, while the remainder of its data structure (an authenticated search tree) is stored in ordinary main memory. The memory integrity checking mechanism uses the data structure and the RootHash to check authenticity and freshness of pairs (ModeID,State) that are loaded into the enclave. If only a small number of ModeID are expected, then the data structure can also be represented as a list and a complex authenticated search tree (ordered with respect to ModeID) is not needed.

In Algorithm 3 OPERATIONMODE is in charge of DALL. This means that if the instantiation corresponding to ModeID should be removed, then the corresponding $(\text{ModeID}, \text{State})$ is first removed from the authenticated search tree used in MEMORYINTEGRITYCHECKING, and if this results in an empty tree, then DALL is called for the enclave.

By making the input to EPUF in Algorithm 1 also dependent on ModeID we also partition the challenge-response space over different enclave instantiations. (This allows us not having to memory integrity check obfuscated keys and auxiliary information $(\{osk_{ctr,j}\}_{j\in I}, aux_{ctr})$ in OPERATIONMODERA of our remote attestation scheme, see Algorithm 4, when this is loaded from memory. If the adversary tampers with these values or replaces these with values from another ModeID', then OTPINVERSE of Algorithm 2 will not properly de-obfuscate the secret keys since this depends on GETREPONSE for ModeID, hence, a wrong signature will be generated (and therefore impersonation is not successful).

ADSO Adversary. The TCB is the persistent on-chip store, used by the memory integrity checking mechanism to verify authenticity and freshness of pairs (ModeID,State) retrieved from untrusted main memory. Based on the existence of a pair for ModeID either initialization mode or operation mode is selected. The main idea is to use initialization mode for obfuscating keys using OTP (in turn based on EPUF calls and a PRNG whose seed will be discarded as soon as initialization finishes). During this process, no ADSO adversary should be present otherwise such an adversary can observe all digital state including all the keys that have not yet been obfuscated using OTP. For this reason Figure 4 indicates that in initialization mode we assume no adversary. In operation mode we do have the ADSO adversary.

Not assuming the presence of an adversary during initialization mode means that during this mode extra care is needed. Since initialization mode is a one-time execution at a controlled pre-defined moment, it is realistic to assume that the secure processor together with a refreshed default OS (in safe mode) without adversarial footprint can be realized. Initialization mode bootstraps a 'secure' operation mode in which keys can be retrieved on a session by session basis.

During operation mode the untrusted OS a.k.a. adversary can call EDALLa. After EDALLa the mode selection wrapper directs the main code to initialization mode. This means that a new State will be build up. For example, a new public key and secret session keys will be generated for remote attestation. This implies that future signatures will be generated using these new session keys. The 'old' session keys do not leak and corresponding signatures cannot be impersonated. For the new public key to be accepted by a remote user who asks for remote attestation, a certificate for the public key given by a Certificate Authority (CA) must be available. Such a certificate is not present if the adversary has forced an initialization by calling EDALLa. Only the legitimate owner through interaction with a CA can start initialization mode in order to acquire a public key certificate from the CA which is linked to the owner's identity. Notice that the owner is in control of who is chosen as CA.

Calling EDALLa leads to a Denial-of-Service (DoS) attack since no execution of the remote attestation enclave will lead to an initialization during which the exact same public key (as before) is generated such that the RootHash may be replicated. So, any future remote attestation enclave execution will not be able to retrieve session keys belonging to the 'old' public key. The threat of a DoS attack means (as for any secure system design) that we need a proper back-up plan: For example, we have multiple initialized sequences of session keys possibly at multiple processors. Here, we may have a mechanism that allows us to use the first sequence to certify second sequences so that no additional interaction with a CA is needed. A better solution is to ask the user to give permission for executing EDALLa in the form of a password (which hash is verified against the stored hash of the password); this means that EDALLa can only be called from a specialized enclave that verifies and maintains user permissions, and users (software engineer) should only supply their password if they know for sure there is no adversary present (who can learn the inputted password).

We notice that if the remote attestation enclave exits, resuming it will context switch back in all locally used variables with access to the persistent on-chip store. After removing (and recreating) the enclave or after some power glitch or outage which messes up a proper enclave exit, the non-volatile memory in the persistent on-chip store can be read again and our mode selection wrapper will allow the enclave to start in operation mode using the last saved State. Thus a DoS is only possible by using EDALLa (in our TCB we assume that the hardware cannot be tampered with, hence, is not destroyed in order to facilitate a DoS).

2.3 Secure Enclaves

We only require a secure processor architecture which can establish a secure computing environment in which sensitive code can execute without its computed values being influenced by the untrusted OS or other computation threads. We call such an environment a secure enclave which can be created, entered, exited (by the enclave itself or asynchronously by the OS), resumed, and removed. The ADSO adversary can observe all digital state including intermediate computed values and is only restricted by the hardware isolation principles that implement access control to the underlying PUF and persistent on-chip store as well as not having access to directly modify enclave code itself. Our approach is compatible with common secure processor architectures in both industry [3, 5, 7] and academia [1, 2, 4, 6, 8, 9].

We notice that a True Random Number Generator (TRNG) must be available in order to randomly select a seed for bootstrapping a PRNG in initialization mode (so that the ADSO adversary cannot replay the PRNG). EPUF outputs random bit responses, however, these can be retrieved using corresponding challenges by the proper enclave when it is created for the first time. An ADSO adversary would be able to replay such execution and observe the generated responses which are stored as intermediate computed values in registers (digital state). Hence, also in this case a TRNG is needed to generate random challenges such that the replayed execution uses different challenges.

For completeness, the reality of the ADSO-adversary is motivated by a recent survey [10] which shows that Intel SGX has been susceptible to a wide range of attacks in the recent years [11–39]. Attacks such as Spectre [29] show how the speculative buffer can leak private information, and the secret seal keys and attestation keys from Intel signed quoting enclaves can be extracted. Similar to Spectre, Meltdown [30] exploits the out-of-order execution property of modern CPUs to leak private information. Cache-based timing attacks (such as "Sneaky Page Monitoring" [19], CacheZoom attack [21], MemJam attack [26]), exploit the cache-hierarchy system and the cache access patterns in the system state which are measurable from outside the protected application by the untrusted OS. Similarly, in [23], a malicious enclave was constructed from which a cache timing attack is mounted using Prime+Probe [66] technique against other enclaves. Dall *et al.* [25] also used Prime+Probe and attacked Intel's provisioning enclave which breaks EPID's (Intel's algorithm used for attestation while preserving privacy of the trusted system) unlinkability property.

3 Public Key Session based OTS Scheme with Secret Key Exposure

A basic building block for designing our remote attestation protocol is a new primitive, called a Public Key Session based One-Time Signature (OTS) Scheme with Secret Key Exposure, denoted OTS-SKE scheme for short. The idea is to have (1) one (universal) public key that can be used to verify all session signatures, (2) each logical session with its own secret session key generates at most one signature for which it uses a “subset" of the secret session key, and (3) this “subset" is exposed to the adversary “for free.” Despite leaking these subsets of secret session keys (for each of the sessions), the adversary cannot forge a signature for any session.

An OTS-SKE scheme $\mathcal{S}$ consists of three procedures

$$\mathcal{S} = (\textsc{KeyGen}, \textsc{Sign}, \textsc{Verify}) :$$

Key generation. Based on a security parameters λ, KEYGEN generates a public key pk together with session secret keys

$$sk_i = \{sk_{i,j}\}_{j=0}^{q-1}$$

and auxilairy variables aux_i for each session $i \in \{0, \ldots, N-1\}$ and a-priori fixed parameter q. We have

$$(pk, \{sk_i, aux_i\}_{i=0}^{N-1}) \leftarrow \textsc{KeyGen}(\lambda).$$

Sign. SIGN takes as input the session id i with session secret key sk_i and auxiliary variable aux_i together with a message $M \in \{0,1\}^n$ and produces a signature σ,

$$\sigma \leftarrow \textsc{Sign}(sk_i, aux_i; M).$$

The computation of SIGN is split in three steps:

1. We have a keyed pseudo random permutation PRP$(key; x)$ which, for each key, is a bijective mapping from strings $x \in \{0,1\}^n$ to $\{0,1\}^n$. We also have an injective mapping ϕ from $\{0,1\}^n$ to subsets of $\{0, \ldots, q-1\}$ (here, $q \geq n$). SIGN first selects a random key and computes the subset

$$I = \phi(\text{PRP}(key; M)) \subseteq \{0, \ldots, q-1\}.$$

2. SIGN extracts a corresponding subset of the i-th session secret key:

$$sk_{i,I} = \{sk_{i,j}\}_{j \in I}.$$

3. SIGN uses $sk_{i,I}$ together with aux_i and input message M to produce a signature σ'. In order to make the dependence on the subset of the session key explicit, we write

$$\sigma' \leftarrow \textsc{Sign}\text{'}(sk_{i,I}, aux_i; M).$$

SIGN returns $\sigma = (\sigma', key)$.

Verify. VERIFY outputs

$$\{\mathbf{true}, \mathbf{false}\} \leftarrow \textsc{Verify}(pk, i; \sigma, M)$$

for a signed message (σ, M) for session id i. Notice that the same public key pk is used for all sessions.

Correctness. OTS-SKE scheme $\mathcal{S}$ is correct if for all $\sigma \leftarrow \textsc{Sign}(sk_i, aux_i; M)$ we have $\mathbf{true} \leftarrow \textsc{Verify}(pk, i; \sigma, M)$.

Security. Even if an adversary has knowledge of subsets of session keys

$$\{sk_{i,I_i}\}_{i=0}^{N-1}$$

together with auxiliary information $\{aux_i\}_{i=0}^{N-1}$, the adversary cannot impersonate a signature for some session with id i^*. This security notion is formalized by GameOTS-SKE for $\mathcal{S}$ as the following security game:

- **Setup**: The challenger runs KEYGEN which returns

$$(pk, \{\{sk_{i,j}\}_{j=0}^{q-1}, aux_i\}_{i=0}^{N-1}).$$

The challenger gives pk as well as $\{aux_i\}_{i=0}^{N-1}$ to the adversary.

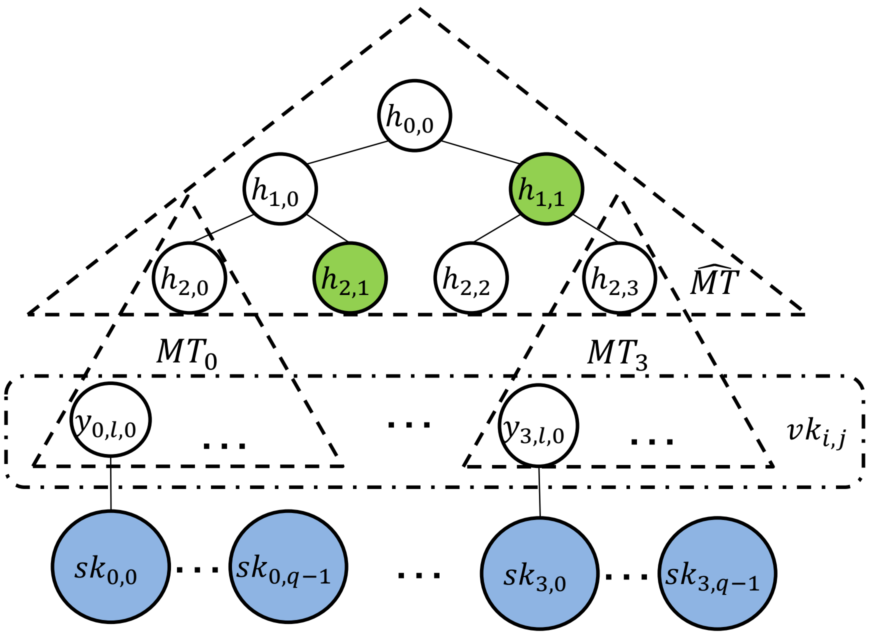

Figure 5: Construction of Hash-based OTS-SKE. It consists of verification keys $vk_{i,j} = F_{k_{i,j}}(sk_{i,j})$ for $i = 0, ...N-1$, $j = 0, ..., q-1$, and two levels of Second Pre-image Resistant (SPR) Merkle trees (MT_i and $\widehat{MT}$). The leaves of MT_i are verification keys $\{vk_{i,j}\}_{j=0}^{q-1}$, and the leaves of $\widehat{MT}$ are the root hash of $\{MT_i\}_{i=0}^{N-1}$. Taking $h_{0,0}$ as an example, the inner nodes in SPR Merkle trees are calculated as: $h_{0,0} = H_{k_{0,0}}(h_{1,0} \oplus r_{1,0}||h_{1,1} \oplus r_{1,1})$, where $r_{i,j}$ are random bit masks for every hash function call. Using random bit masks allows us to reduce the security of the whole construction to the second pre-image resistance of hash function $H_k()$, instead of collision resistance. To prove the existence of a leaf node i in $\widehat{MT}$, an authentication path is included in the signature. The two green nodes form the authentication path of leaf node $h_{2,0}$ in $\widehat{MT}$.

- **Query**: The adversary adaptively issues a sequence of messages M_i, at most one message for each session id i. The challenger computes

$$I_i = \phi(\text{PRP}(key_i; M)) \text{ and } sk_{i,I_i} = \{sk_{i,j}\}_{j \in I_i}$$

  for random key_i. The challenger gives the extracted information sk_{i,I_i} with key_i to the adversary (as soon as M_i is received).

  Notice that the adversary can use this information to sign message M_i for session i by applying SIGN'. This may lead to multiple signatures for M_i (since fresh randomness can be used for each signature generation). However, no signatures for other messages $\neq M_i$ for session id i can be forged if the following Guess does not succeed.

- **Guess**: The adversary selects a session number $i^* \in \{0, \ldots, N-1\}$ which refers to the session for which the adversary will want to forge a signature: The adversary outputs a signed message (σ, M^*) for session i^* such that $M^* \neq M_{i^*}$. The adversary wins the game if the signature verifies, that is,

$$\textbf{true} \leftarrow \text{VERIFY}(pk, i^*; \sigma, M^*).$$

In this game, $\mathcal{A}$ is called an OTS-SKE-EUF-CMA (OTS-SKE Existential UnForgeability under Chosen Message Attack) adversary.

If $\mathcal{A}$ wins GameOTS-SKE with probability $\geq \epsilon$ in time $\leq T$, then we call $\mathcal{A}$ a (T, Q_H, Q_P, ϵ)-OTS-SKE adversary for $\mathcal{S}$, where Q_H and Q_P are the maximum number of queries allowed to be made by $\mathcal{A}$ to a hash function oracle and the PRP in GameOTS-SKE. We say scheme $\mathcal{S}$ is (T, Q_H, Q_P, ϵ)-secure against OTS-SKE-EUF-CMA attacks if no (T, Q_H, Q_P, ϵ)-OTS-SKE adversary exists.

3.1 Hash based OTS-SKE

We present a post-quantum secure OTS-SKE scheme, which is solely based on secure hash functions. As illustrated in Fig. 5, our hash-based OTS-SKE scheme consists of two main building blocks: (1) Lamport one-time signature scheme [67] with the optimization introduced by Bos and Chaum [68], and (2) a second pre-image resistance (SPR) Merkle tree construction, whose security solely relies on the second pre-image resistance of the underlying hash function [57]. We also adopt the modifications of SPR Merkle tree and one-time signatures in [69] to mitigate multi-target attacks when hash functions are repeatedly used in a scheme.

The basic idea of one-time signature schemes [67] is that the signer first generates a set of random secret keys and computes the hash values (verification keys) of each secret key. Then the signer first commits the verification keys to the verifier. When it signs a message, the signer selects a subset of secret keys based on the message using a mapping function $\phi(.)$, and sends the selected secret keys to the verifier. The verifier computes the hash of the received secret keys and checks if they match the committed verification keys. The verifier also verifies whether the subset represents the associated message using $\phi(.)$. If both checks are correct, the signature is valid. Due to the one-wayness of hash functions, given verification keys, an adversary cannot recover any secret keys to forge a signature. Only the signer, who knows the original secret keys, can generate signatures. However, one of the drawbacks of this approach is that the size of verification keys sent to the verifier in the key generation phase is very large. Thus, we use a Merkle tree [57, 70] to manage and compress all the verification keys, such that only a root hash of the tree is needed to be first committed to the verifier for verifying all verification keys later. The security of the adopted SPR Merkle tree [57] is due to the second pre-image resistance of the used hash functions, i.e., no one is able to construct another SPR Merkle tree with different leaves but the same root hash.

Hash function families. Let $\mathcal{F}_\lambda = \{F_k : \{0,1\}^\lambda \to \{0,1\}^\lambda \mid k \in \{0,1\}^\lambda\}$, be a one-way function family that takes a λ-bit string and a λ-bit key k to generate a λ-bit output. Intuitively, the definition of one-way functions implies that, given a random key k and a function output $y = F_k(x)$, where $x \leftarrow \{0,1\}^\lambda$, an adversary cannot find an x^* such that $y = F_k(x^*)$ in $poly(\lambda)$ time. Let $\mathcal{H}_\lambda = \{H_k : \{0,1\}^{2\lambda} \to \{0,1\}^\lambda \mid k \in \{0,1\}^\lambda\}$ be a second pre-image resistant hash function family, taking 2λ bits as input and producing a λ-bit output. Each H_k is uniquely defined by its key $k \in \{0,1\}^\lambda$. Here, second pre-image resistance means that, given a random key k and a random input x, an adversary cannot find an $x^* \neq x$, such that $H(x) = H(x^*)$ in $poly(\lambda)$ time.

Hash-based OTS-SKE Scheme. For a scheme that signs n-bit messages, a pair of parameters (q, s) is selected, such that $\binom{q}{s} \geq 2^n$. Let $\mathbb{Q}$ denote the set {0, 1, ..., $q-1$}. Let $\phi_{n,q,s}$ be an efficiently computable injective function that maps every integer m ($0 \leq m \leq 2^n - 1$) to a unique s-element subset of set $\mathbb{Q}$ of q elements. There are many examples of functions $\phi_{n,q,s}$ in literature [68, 71]. In this work, we select the function $\phi(.)$ introduced in [68] due to its simplicity.

Below we describe the three procedures in our hash-based OTS-SKE scheme $\mathcal{S} = (\textsc{KeyGen}, \textsc{Sign}, \textsc{Verify})$:

Key Generation. The key generation procedure sets parameters of the scheme and generates a public key, all secret keys, and auxiliary information

$$(pk, \{sk_i, aux_i\}_{i=0}^{N-1}) \leftarrow \textsc{KeyGen}(\lambda)$$

as follows:

- Generate Nq λ-bit random values as session secret keys $\{sk_i = \{sk_{i,j}\}_{j=0}^{q-1}\}_{i=0}^{N-1}$.
- Compute verification keys $vk_{i,j} = F_{k_{i,j}}(sk_{i,j})$ for all combinations of i and j, where $k_{i,j}$ is randomly generated for every $F_{k_{i,j}}$ function call. We define auxiliary information $\{aux_i = \{vk_{i,j}\}_{j=0}^{q-1}\}_{i=0}^{N-1}$.
- Form SPR Merkle trees MT_i with as leaves $vk_{i,j}$, $j \in \mathbb{Q}$. Let $L = \lceil \log q \rceil$ denote the height of each SPR Merkle tree. Let t denote the position of a node on layer l from left to right, where $l \in \{0, ..., L\}$. Leaves are on layer L. The inner nodes of SPR Merkle trees are computed as $y_{i,l,t} = H_{k_{i,l,t}}((y_{i,l+1,2t} \oplus r_{i,l+1,2t}) \| (y_{i,l+1,2t+1} \oplus r_{i,l+1,2t+1}))$, where hash function keys $k_{i,l,t}$ and bit masks $r_{i,l,t}$ for $i \in \{0, ..., N-1\}$, $l \in \{0, ..., L\}$, and $t \in \{0, ..., 2^l - 1\}$ are randomly generated for every hash function call. Since $vk_{i,j}$ are leaves of the trees, $y_{i,L,j} = vk_{i,j}$ for $i \in \{0, ..., N-1\}$ and $j \in \{0, ..., q-1\}$. After constructing these SPR Merkle trees, we get their root hash $y_{i,0,0}$.
- Repeat the last step and use $\{y_{i,0,0}\}_{i=0}^{N-1}$ as leaves to construct a new SPR Merkle tree $\widehat{MT}$, and then we output the root hash $root$ of $\widehat{MT}$. Similarly to the construction of MT_i, all hash function keys and bit masks are randomly generated for every hash function call in the construction of $\widehat{MT}$.

The public key is constructed as

$$pk = (root, \{k_{i,j}\}\}_{i=0,j=0}^{N-1,q-1}, \{\{k_{i,l,t}, r_{i,l,t}\}_{t=0}^{2^l-1}\}_{i=0,l=0}^{N-1,L}).$$

Public key pk will be shared with any verifier, and the secret keys $\{sk_{i,j}\}_{i=0,j=0}^{N-1,q-1}$ are kept secret. Auxiliary information $\{vk_{i,j}\}_{i=0,j=0}^{N-1,q-1}$ is stored in a public trusted memory on the signer side. In practice, all keys $k_{i,j}$ for one-way function calls, all keys $k_{i,l,t}$ for hash function calls, and all bit masks $r_{i,l,t}$ in Merkle trees can be generated by a pseudorandom number generator, taking a seed $seed$, associated indices, and a special symbol to distinguish the generation of $k_{i,j}$, $k_{i,l,t}$, and $r_{i,l,t}$. In this case, the seed will be a part of the public key shared with verifiers, i.e., $pk = (root, seed)$. Consequently, in the security analysis, one will need to consider the success probability ϵ_{PRG} of distinguishing a pseudorandom output from a truly random output.

Signing. To sign an n-bit message M for session i, the signer uses secret keys $sk_i = \{sk_{i,j}\}_{j=0}^{q-1}$ and verification keys $\{vk_{i,j}\}_{j=0}^{q-1}$ to produce a signature σ

$$\sigma \leftarrow \text{SIGN}(\{sk_{i,j}\}_{j=0}^{q-1}, \{vk_{i,j}\}_{j=0}^{q-1}; M)$$

as follows:

- Compute $I = \phi_{n,q,s}(\text{PRP}(key; M)) \subset \mathbb{Q}$ and fetch $sk_{i,I} = \{sk_{i,j} \mid j \in I\}$ and $vk_{i,I} = \{vk_{i,j} \mid j \in \mathbb{Q} \setminus I\}$.
- Generate an authentication path AP of the SPR Merkle tree $\widehat{MT}$ for its leaf $y_{i,0,0}$. See [72] (and Fig. 5 as an example) for how to generate an authentication path for a leaf in a Merkle tree. An authentication path in a Merkle tree for a leaf node i is $\log N$ in size, and it can be used to verify the existence and position of leaf i.
- Signature $\sigma = (\sigma', key)$ with $\sigma' = (sk_{i,I}, vk_{i,I}, AP)$.

The signer sends (σ, M) to the verifier for verification.

Verify. (σ, M) for session id i, can be verified as follows:

- Compute $I = \phi_{n,q,s}(\text{PRP}(key; M)) \subseteq \mathbb{Q}$. Interpret each element in $sk_{i,I}$ as one of the $sk_{i,j}$ with $j \in I$ and each element in $vk_{i,I}$ as one of the $vk_{i,j}$ with $j \in \mathbb{Q} \setminus I$.
- Compute $vk_{i,j} = F_{k_{i,j}}(sk_{i,j})$ for $j \in I$.
- Use $\{vk_{i,j}\}_{j \in \mathbb{Q}}$ to construct the root hash $y_{i,0,0}$ of SPR Merkle tree MT_i.
- Use $y_{i,0,0}$ and AP to compute root hash $root'$ of $\widehat{MT}$.
- If the generated root hash $root'$ matches with the root hash $root$ in the public key pk, VERIFY accepts the signature σ for message M in session i and outputs $\mathtt{true}$. Otherwise, VERIFY rejects this signature and outputs **false**.

Correctness. The VERIFY procedure essentially repeats the deterministic computation in KEYGEN to regenerate the root hash $root$ of $\widehat{MT}$, using all the same random numbers shared with the verifier in pk and σ'. Hence, if sk_i corresponds to the secret keys randomly generated in KEYGEN, VERIFY will reproduce the same root hash.

Security. The security of the above OTS-SKE scheme can be reduced to the one-wayness of the one-way functions in family $\mathcal{F}_k$ and the SPR of the SPR hash functions in family $\mathcal{H}_k$. Our security analysis follows the security analysis in [67] and in [69] to analyze the security of our scheme under multi-target attacks. GameOTS-SKE for $\mathcal{S}$ is the same as the security game used in [69] (where a one-time signature of a message is a subset of secret keys). Notice that we did not leak any secret keys beyond what is exposed in the signature because all verification keys $vk_{i,j}$ are public and they can be used to reconstruct all SPR Merkle trees generated in KEYGEN. We rephrase Theorem 2 in [69] as follows.

Theorem 1 *Let $\mathcal{S}$ be the hash based OTS-SKE scheme with N sessions. Let $\mathcal{A}$ be a $(T, Q_F, Q_H, Q_P, \epsilon)$-OTS-SKE (classical or quantum) adversary for $\mathcal{S}$. Then, there exists a $(T, Q_F, Q_H, Q_P, \epsilon)$ algorithm that breaks the multi-function multi-target one-wayness of $\mathcal{F}_\lambda$ with Nq targets or the multi-function multi-target second pre-image resistance of $\mathcal{H}_k$ with $N(q-1) + (N-1)$ targets, where Q_F, Q_H, Q_P are the number of oracles queries to $\mathcal{F}_\lambda$, $\mathcal{H}_\lambda$, and the PRP used in GameOTS-SKE.*

The next subsection provides formal definitions and discussion on the multi-function multi-target security notion of hash function families.

Table 3: Cost Comparison between LOTS and WOTS. $N_w = \lceil \frac{n}{w} \rceil + \lceil \frac{\log_2(n/w)}{w} \rceil + 1$, $\binom{q}{s} \geq 2^n$, $l = \log_2 N$, and q', N'_w are the numbers of inner nodes in a Merkle tree with q, N_w leaves, respectively.

Cost	**LOTS [67, 68]**	**WOTS [73]**
KeyGen (#EPUF)	q	$N_w \cdot 2^w$
Sign (#EPUF)	s	N_w
Verify (#HASH)	$q' + l + s$	$N_w \cdot (2^{w-1}) + N'_w + l$
Storage (bits)	$3952 \cdot N \cdot q + 256 \cdot (N+1)$	$3696 \cdot N \cdot N_w \cdot 2^w + 256 \cdot (N+1)$
Comm (bits)	$256 \cdot (q + l + 1)$	$256 \cdot (N_w + l + 1)$

3.2 Additional Background and Related Work to Hash-based OTS-SKE

Multi-Function Multi-Target Security Notion. The success probability of attacking the one-wayness of a one-way function family $\mathcal{F}_\lambda$ using the generic attack is $\frac{Q_F+1}{2^\lambda}$ and $\Theta(\frac{(Q_\mathcal{F}+1)^2}{2^\lambda})$ for classical adversaries and quantum adversaries, respectively, where Q_F is the number of queries the adversaries send to functions in $\mathcal{F}_\lambda$ [69]. Similarly, for the generic attacks on the second pre-image resistance of a second pre-image resistant hash function family $\mathcal{H}_\lambda$, the success probability is $\frac{Q_H+1}{2^\lambda}$ and $\Theta(\frac{(Q_H+1)^2}{2^\lambda})$ for classical adversaries and quantum adversaries, respectively, if Q_H queries are made to functions in $\mathcal{H}_\lambda$ [69]. The above hardness regarding (second) pre-image resistance holds for both the single-function single-target (SFST) security game and the multi-function multi-target (MFMT) security game [69]. In an SFST game, the attacker is required to invert a randomly chosen function in the function family with respect to a given random target. In an MFMT game, the attacker is only required to invert one out of multiple randomly chosen functions, and each of the functions has a random target to be inverted. Concretely speaking, the security of the hash-based OTS-SKE is reduced to the MFMT case, because functions in $\mathcal{F}_\lambda$ and $\mathcal{H}_\lambda$ are used multiple times in the scheme, and each of them has a random key and a random target, due to the random bit masks and random secret keys. For completeness, we give formal definitions of MFMT one-wayness and MFMT second pre-image resistance in Definition 1 and 2.

Definition 1 *[69] The success probability of an adversary $\mathcal{A}$ against MFMT one-wayness of $\mathcal{F}_\lambda$ with N_T targets is $Succ^{OW}_{\mathcal{F}_\lambda, N_T}(\mathcal{A}) = Pr[k_i \leftarrow \{0,1\}^\lambda, x_i \leftarrow \{0,1\}^\lambda, y_i = F_{k_i}(x_i), 0 \leq i \leq N_T - 1; (j, x^*) \leftarrow \mathcal{A}((k_0, y_0), ..., (k_{N_T-1}, y_{N_T-1})) : y_j = F_{k_j}(x^*)]$*

Definition 2 *[69] The success probability of an adversary $\mathcal{A}$ against MFMT second pre-image resistance of $\mathcal{H}_\lambda$ with N_T targets is $Succ^{SPR}_{\mathcal{H}_\lambda, N_T}(\mathcal{A}) = Pr[k_i \leftarrow \{0,1\}^\lambda, x_i \leftarrow \{0,1\}^\lambda, 0 \leq i \leq N_T - 1; (j, x^*) \leftarrow \mathcal{A}((k_0, x_0), ..., (k_{N_T-1}, x_{N_T-1})) : x* \neq x_j \wedge H_{k_j}(x_j) = H_{k_j}(x^*)]$*

Comparisons with Related Hash-based Signatures. Recent developments in hash-based signatures include XMSS [73], XMSS$^{\text{MT}}$ [74], XMSS-T [69], SPHINCS [75], and SPHINCS$^+$ [76]. Among these, XMSS-T [69] is the most closely related algorithm to our construction. We both use the same construction of SPR Merkle trees to mitigate multi-target attacks in practice. However, the main difference between our hash-based OTS-SKE with the XMSS family is the choice of the underlying hash-based one-time signatures. The XMSS family [69, 73, 74] uses Winternitz One-Time Signature (WOTS), which offers a shorter signature, but the signer needs to compute signatures on the fly using its secret keys, which does not fit the adversarial model of OTS-SKE. Otherwise we have to pre-compute all possible values of sub-signatures in the key generation procedure to avoid using any secrets for signature generation in signing. A theoretical cost comparison between the Lapmport's OTS (LOTS) [68] and Winternitz OTS (WOTS) [73] under ADSO adversary is provided in Table 3. In WOTS, a n-bit message is split to $\frac{n}{w}$ w-bit chunks, and the signer needs to generate a checksum of the message using $\lceil \frac{\log_2(n/w)}{w} \rceil + 1$ w-bit chunks. Let N_w be the number of chunks needed for representing a n-bit message and its checksum. The signature size of WOTS is N_w secret keys, and therefore only N_w EPUF calls are needed in the signing procedure. If $w > 2$, the WOTS signing phase will be faster than that of LOTS. However, WOTS requires larger storage and longer key generation time. These two numbers both grow exponentially w.r.t. the increase of w. Overall, WOTS consumes more resources than LOTS, but it provides an alternative way to reduce the signing time with additional cost of storage and key generation time.

4 Autonomous Remote Attestation

Our goal is to offer secure remote attestation even when an ADSO-adversary is present. We want to show that an application enclave AppEnc can have its computed result signed for a remote user by a remote attestation enclave, called RAEnc for short. We require that the ADSO adversary, who can (1) observe all digital (intermediate) state of the AppEnc and RAEnc computations (including all digital storage) and who can (2) ask the RAEnc to sign whatever it wants, cannot forge a signature on behalf of AppEnc for the remote user. That is, even with access to all digital state, the ADSO adversary cannot forge a signature for a remote user of a malicious result R (for which AppEnc had not asked for it to be signed) such that it passes the signature verification by the remote user.

Figure 1 depicts our solution. We use the tools developed in the previous sections. Initialization mode implements KEYGEN of a OTS-SKE scheme to generate a sequence of session keys. The session keys are all obfuscated by applying OTP which in turn invokes the underlying PUF. The obfuscated session keys are stored in untrusted storage using MEMORYINTEGRITYCHECKING. KEYGEN also generates a corresponding public key pk which is bound together with the remote attestation enclave's measurement MRENCLAVE$_{ra}$ in a certificate. Producing the certificate is a result of interaction with a trusted CA. The pseudo code is given in Algorithm 4. During initialization mode we assume no observation of digital state by any adversary.

Algorithm 4 Remote Attestation

We use the mode selection wrapper (which is based on memory integrity checking with root hash in the persistent on-chip store) for (ModeID, State) where State represents the session counter ctr. We assume a OTP-SKE-EUF-CMA secure OTS-SKE scheme $\mathcal{S} = (\text{KEYGEN}, \text{SIGN}, \text{VERIFY})$, where SIGN is defined by mapping $\phi(.)$ and procedure SIGN'.

```
1:  procedure RAINITIALIZATIONMODE
2:      $(pk, \{\{sk_{i,j}\}_{j=0}^{q-1}, aux_i\}_{i=0}^{N-1}) \leftarrow \text{KEYGEN}(\lambda)$
3:      $seed \leftarrow$ TRNG;
4:      for $i \in \{0, \ldots, N-1\}$ do
5:          Apply OTP $q$ times based on PRNG($seed$)
6:              $\{osk_{i,j} = \text{OTP}(sk_{i,j})\}_{j=0}^{q-1}$
7:          Store $(\{osk_{i,j}\}_{j=0}^{q-1}, aux_i)$
8:      end for
9:      Discard $seed$
10:     Set $(\text{ModeID}, \text{State} = ctr)$ with $ctr = 0$
11:     Produce a certificate binding $pk$ and MRENCLAVE$_{ra}$
12: end procedure
1:  procedure RAOPERATIONMODE
2:      while true do
3:          if LOCALATTESTATIONQUEUE is non-empty then
4:              Pop a signing request from the queue
5:                  $(\text{MRENCLAVE}_{app}, R, \text{RemoteUser})$
6:              $M = \text{HASH}(\text{MRENCLAVE}_{app}, R)$
7:              Set $ctr = \text{State}$, where $(\text{ModeID}, \text{State})$
8:              send $ctr$ to RemoteUser
9:              Set $(\text{ModeID}, \text{State} = ctr + 1)$
10:             receive $nonce$ from RemoteUser
11:             $I = \phi(\text{HASH}(nonce, M))$
12:             Load $(\{osk_{ctr,j}\}_{j\in I}, aux_{ctr})$
13:             Apply OTPINVERSE $|I|$ times:
14:                 $\{sk_{ctr,j} = \text{OTPINVERSE}(osk_{ctr,j})\}_{j\in I}$
15:             $\sigma' = \text{SIGN'}(\{sk_{ctr,j}\}_{j\in I}, aux_{ctr}; M)$
16:             send $(\sigma', \text{MRENCLAVE}_{app}, R)$ to RemoteUser
17:         end if
18:     end while
19: end procedure
```

The core idea is that (1) the adversary can only have access to the PUF through the EPUF interface for which the implemented access control requires the hash/measurement of the calling enclave to be mixed in the challenge that is given to the PUF. This means that the challenge response space which is used for obfuscating session keys during initialization cannot be accessed by an adversary. Therefore, even though all obfuscated session keys can be read/observed by the ADSO adversary, they cannot give information about the plain session keys themselves. Second, (2) signing is implemented using our OTS-SKE scheme which only uses a subset of the session key for each session – and the subset choice is message dependent. This means that even though this subset is observed by the ADSO adversary it cannot be used to sign anything different than this message (Hash of $nonce$, MRENCLAVE$_{ra}$, and the result R).

Operation mode implements the SIGN procedure of the OTS-SKE scheme. Here, we define

$$\text{PRP}(i; M) = \text{HASH}(nonce_i, M),$$

where $nonce_i$ is the nonce received from RemoteUser for session i. Regarded as a function of M a collision resistant hash function $\text{HASH}(nonce_i, M)$ cannot be distinguished from a pseudo random permutation with non-negligible probability. Therefore, we may use this for our PRP and fit the definition of the OTS-SKE scheme.

The RA enclave receives a result R from an application enclave which needs to be signed for a remote user. RAEnc receives R from AppEnc by means of a "local attestation" primitive. We assume that local attestation is implemented as a physical authentic channel between enclaves where the channel is hardware isolated in that the messages transmitted over the channel cannot be tampered with and the source/authenticity of messages cannot be modified. Such a physical authentic channel does not exist in Intel SGX where local attestation is implemented using crypto (AES and MAC) based on a secret key; we cannot use this because our ADSO-adversary can observe digital secret keys and thus impersonate Intel SGX's local attestation. Instead, we either use Sanctum's [8] solution where the security monitor implements a physical hardware isolated channel between enclaves without needing crypto and secret keys, or we simply glue the AppEnc and RAEnc together in one single enclave such that a physical authenticated channel is inherently present (in the latter case RAEnc can be seen as a wrapper around AppEnc, and this is one the advantages of the digital autonomy our scheme offers). RAEnc will take the measurement of AppEnc with the result R to create message

$$M = \text{HASH}(\text{MRENCLAVE}_{app}, R).$$

The OTS-SKE scheme is now used to sign M for the remote user.

Algorithm 5 Request and Verification

We assume the remote user knows pk (verified by means of a certificate issued by a trusted CA) of the OTP-SKE-EUF-CMA secure OTS-SKE scheme $\mathcal{S} = (\text{KEYGEN}, \text{SIGN}, \text{VERIFY})$ used by the RA enclave.

```
 1: procedure REQUESTANDVERIFY
 2:     while true do
 3:         if receive $ctr$ from RAEnc then
 4:             send random $nonce$ to RAEnc
 5:             receive $(\sigma', \text{MRENCLAVE}_{app}, R)$ from RAEnc
 6:             $M = \text{HASH}(\text{MRENCLAVE}_{app}, R)$
 7:             $\sigma = (\sigma', nonce)$
 8:             return $\text{VERIFY}(pk, ctr; \sigma, M)$
 9:         end if
10:     end while
11: end procedure
```

In Algorithm 5 the remote user receives a request in the form of a session id ctr from RAEnc to start a session of the remote attestation protocol. The remote user knows the public key pk of RAEnc – checked by means of a certificate that binds pk with the measurement MRENCLAVE$_{ra}$ of RAEnc. The chain of trust tells the remote user that the certificate was produced as a result of the initialization mode of RAEnc during which a random pk was generated. The corresponding session signing keys were obfuscated using a processor unique PUF, and only RAEnc, executed at this processor $\mathcal{P}$, can access the challenge response space used for obfuscating the session signing keys. This allows the remote user to conclude that only the RAEnc code with MRENCLAVE$_{ra}$ could have been able to de-obfuscate the subset of the session signing key that corresponds to pk while executing at $\mathcal{P}$. (The certificate may also include the identity of the owner or serial number of $\mathcal{P}$; this gives the remote user more information about where the signed result R was computed.)

The remote user selects a nonce $nonce$. RAEnc extracts the subset of keys for session id ctr that relate to $I = \phi(\text{HASH}(nonce; M))$. This is done by using EPUF calls that only reveal the masks/OTP-layer for that subset of keys. This subset is used in SIGN' in Algorithm 4 to sign message M.

As soon as the remote attestation enclave, see Algorithm 4, has produced a signature $(\sigma', \text{MRENCLAVE}_{app}, R)$ for the i-th session with $i = ctr$, counter ctr is incremented. This implies that even if our ADSO-adversary runs the RAEnc itself, it only learns digital state from sessions with id $\leq ctr$, and in fact only learns at most one subset of keys $\{sk_{i,j}\}_{j \in I}$ with aux_i per session id i (because once a subset is used for signing, ctr is incremented). All other key information is obfuscated by the PUF functionality, that is, they are stored using OTP in initialization mode and will never be de-obfuscated using OTPINVERSE. So, the ADSO adversary finds itself exactly in the security game GameOTS-SKE and is unable to forge a signature for a new message.

If the adversary runs operation mode for message M as supplied by the remote verifier, then he learns the related subset of keys of session ctr and can create other σ' for the same message M and session id ctr. But this does not generate a signature for a new malicious message M^*. A new malicious message M^* corresponds to a different subset of the session signing key indicated by a set I^*. The security guarantee of the OTS-SKE scheme shows that the adversary cannot successfully forge a signature for M^*.

If VERIFY in Algorithm 5 verifies the signature, then the remote user may conclude that R was indeed created by AppEnc at processor $\mathcal{P}$: The chain of trust shows that the signature was created by RAEnc on $\mathcal{P}$. Its operation mode shows that it only signs messages M that are a hash of MRENCLAVE$_{app}$ together with R received by means of local attestation. Since local attestation in $\mathcal{P}$ implements a physical authenticated channel without keys, R must have been produced by EncApp. And it must have been produced at $\mathcal{P}$ because for local attestation to work EncApp must co-reside with RAEnc at the same processor. The remote user also knows that the signing request of EncApp indicated that R with its signature should be transmitted to RemoteUser. Finally, RemoteUser selected a random $nonce$, hence, the signature is fresh (and not a replay of an older signature for the same message). We conclude that under the ADSO adversary our scheme offers secure remote attestation.

Autonomous RA. The key management involved with the remote attestation scheme is out of the hands of the processor manufacturer. Certification of the public key is done during initialization mode. Any mutually trusted observer of the fact that RAEnc securely executes initialization mode can serve as a trusted authority who approves/generates a certificate that binds the newly created public key to the processor and, hence, its owner. This observer can be any trusted third party and can depend on the application which uses the remote attestation protocol. It does not have to be the Intel's provisional key management authority. Notice that multiple instances of RAEnc can run in parallel, each with its own ModeID and its own public key and sequence of session signing keys. Hence, a RAEnc instance can be made specific to an AppEnc instance together with a corresponding trusted third party in charge of certification during initialization mode. This makes our remote attestation autonomous and takes it out of the hands of processor manufacturers (which, if in hands of processor manufacturers, can only lead to conflicts of interest). Having autonomous RA also allows each government to enforce its own RA policies (for own government applications such as filing taxes, obtaining social security, and the like).

Application. An immediate application of our solution is remote attestation for individual users where the RAEnc is running on individual laptops. In this setting we are not burdened by too many signing requests and our remote attestation protocol is allowed to take say 1 second (including the communication round trip time).

Significant hardware acceleration and implementation optimization is needed if we want to apply our solution in a cloud service where high throughput is required. We will want to bring down the time needed for computing a signature in operational mode to the order of 10s of ms. One may consider pushing operational mode computation into specialized hardware.

Certificate authorities can also make use of our solution. A CA implements a scheme for generating certificates which are signatures themselves. A distributed implementation among compute nodes allows one to resist failure or compromise of a minority of compute nodes. Each node can implement its own enclave by using our method which resists the ADSO-adversary. Now we do not need to worry about compromise of compute nodes other than possibly having a denial-of-service attack; compromised compute nodes cannot help the ADSO-adversary impersonating certificates.

Multiple Levels of Trusted Execution. Ideally, we want a special "safe mode processor operation" for the initialization mode during which only the initialization code is executed and no other code, even excluding the OS (which we cannot trust due to its large code base and lack of formal verification). Such safe mode excludes all resource sharing and for this reason the performance in safe mode suffers a large impractical penalty; we do not wish to use it for normal application code. Safe mode must be used rarely and only for code snippets that require absolute hardware isolation with privacy guarantees that prevent any adversary from observing the internal computed values during safe mode. Clearly, if such safe mode exists, then it can be used to run our initialization mode. We envision an enclave being able to indicate whether to run a procedure in safe mode (by means of some EENTERSafeMode indicator) so that only after returning from the procedure call ordinary secure enclave execution progresses during which the OS is again active (and can potentially use an asynchronous exit in order to context switch computing threads for optimized resource scheduling and to resume the enclave at a later moment).

We notice that the Ascend processor architecture [77] was designed to always follow a safe mode in that all running code in Ascend is assumed non-malicious, uncompromised, and belongs solely to the legitimate party requesting its execution. Ascend hides leakage of internally computed values by guarding the boundary between the processor and it I/O to DRAM (through the memory controller). DRAM is accessed in such a way as to hide the binary content (through encryption), hide address access patterns (through Oblivious RAM), and use a memoryless time access pattern. Memory integrity checking is added in order to verify authenticity and freshness of values loaded from DRAM.

Our proposal is to allow different trust levels during enclave execution. Safe mode provides strict hardware isolation which offers strong privacy guarantees at the cost of disallowing concurrent execution and resource sharing. Normal enclave mode execution assumes the ADSO-adversary and does not offer privacy but does allow concurrent execution with resource sharing.

Recurring Initialization using Safe Mode. In practice, we want to add more session signing keys by using initialization mode when needed. If we have safe mode for initialization, then we only need to bootstrap one RAEnc during which a certificate is generated with help of a mutually trusted party. After this, more RAEnc can be created and their certificates can be signed (without needing a trusted third party) by the initially bootstrapped RAEnc as follows: Each new RAEnc instance (with its own ModeID) simply asks the initially bootstrapped RAEnc to sign a certificate (using a default $nonce$; we do not need to prove freshness), that is, the bootstrapped RAEnc functions as a CA. Even if a new instance of an RAEnc is executed in the presence of an adversary, since initialization is done in safe mode, the generated keys remain hidden from the adversary and the produced certificate indeed attests to a securely set up RAEnc instance (initialized in safe mode). The remote user needs the certificate of the bootstrapped RAEnc in combination with the certificate of the new RAEnc instance to gain trust in the public key of the new RAEnc instance.

5 Implementation and Performance Evaluation

In this section, we give the details of our implementation and performance analysis for the remote attestation scheme.

5.1 Experimental Setup

All experiments are conducted on a PC with Intel(R) Core(TM) i7-7820HQ CPU @ 2.90GHz and 32GB memory and on Windows 10 based operating system. Intel SGX is used to provide secure enclaves for the RA scheme. We have written our own C++ implementation of an EPUF-extended PUF interface simulation. For the hash-based OTS scheme implementation, we used Boost Multiprecision library[1] for big integer arithmetic. For the random number generation (RNG) in our implementation, we used a PRNG provided by MIRACL library[2], whose seed is then replaced by a random number generated from the Intel SGX's RNG.

5.2 Extended PUF Interface

For our PUF interface, we use a C++ based iPUF simulation structure with approximately 10% noise [61], which generates a 1-bit output given a $\lambda = 128$-bit input. However, in our performance evaluation, instead of reporting the time consumed by the software-based simulation, we estimated the latency of a hardware-based 128-stage iPUF to be approximately 10^{-2} milliseconds according to the physical characterization of PUFs on CPLD [64]. This shows a more realistic estimation of the performance of the RA scheme in a real system.

On top of iPUF, we implement the EPUF interface which was previously explained in detail in Section 2.1. In our configuration, we set the extended EPUF interface parameters $m = 168$, $k = 7$, and $T = 4$, see the discussion on a practical parameter setting in Appendix A. We use SHA-256 for the pseudo random function $f(.)$ in the EPUF interface.

Table 4 and Table 5 show the details of different components of the EPUF structure (based on 1 EPUF call) and an approximation of their estimated timings based on our experiments. EPUF consists of 2 components: $\text{GETCRP}(s, x, c)$ which is called during the key generation phase and $\text{GETRESPONSE}(C)$ which is called during the signing/working phase.

During each call, EPUF is called $m \times (2k + 1)$ times based on our parameter selection as given in Table 1 in Section 2.1. However, we assume using 15 parallel iPUF structures to speed up the performance. Therefore, in our timings, we average the number of iPUF calls to $m \times (2k + 1)/15$ which equals 168 for $\text{GETCRP}(.)$ and 155.6 for $\text{GETRESPONSE}(.)$.

During GETCRP, vectors s, x and c of size 128, 168 and 128, respectively, are generated using the RNG. After 168×15 EPUF calls, $b = s \cdot A + x$ is calculated. And finally challenge $C = (c, y, b, f(0\|s))$ and response $R = f(1\|s)$ are

[1]https://www.boost.org/

[2]https://github.com/miracl/MIRACL

extracted. As seen from the tables, even with possible hardware acceleration, PUF calls are the most costly operation which makes almost 75% of GETCRP's runtime.

During GETRESPONSE, given the input challenge c, EPUF calls based on input c are again made. Based on the accuracy of the responses from the PUF, confidence vector con is calculated. Based on this confidence vector, matrix $A_{\mathcal{I}}$ is extracted from matrix A based on the confidence vector. Following this, vector s is solved and 2 SHA-256 calls are performed to provide the final response (see Algorithm 1 for details).

We observe that, GETRESPONSE call is more costly than GETCRP (approx. twice more), which makes the extended EPUF interface more costly during operation mode than the initialization mode in our RA scheme. This is because choosing a submatrix $A_{\mathcal{I}}$ in GETRESPONSE is another costly operation in addition to the PUF calls.

Table 4: Cost Analysis of GetCRP of the extended EPUF interface.

Operation	Time (ms)	Repeats per call	Total (ms)
Generate s, x, c	0.00058	128+168+128	0.246
SHA-256	0.1	2	0.2
iPUF	0.01	168	1.68
Matrix Mult.	0.12	1	0.12
Total			2.24

Table 5: Cost Analysis of GetResponse of the extended EPUF interface.

Operation	Time(ms)	Repeats per call	Total (ms)
iPUF	0.01	average 155.6	1.556
Confidence Calculation	0.001	1	0.001
Choose A_I + invert	4.5	1	4.5
Solve $b_{\mathcal{I}} = s \cdot A_{\mathcal{I}} + x'_{\mathcal{I}}$	0.12	1	0.12
Majority Voting	0.5	1	0.5
SHA-256	0.1	2	0.2
Total			6.88

5.3 OTP Interface

We use AES-128 in our OTP interface implementation. Encryption is performed using Intel Advanced Encryption Standard Instructions (AES-NI).

Table 6 shows the timings of OTP and OTPINVERSE based on our implementation and experiments.

OTP initially generates the s, x, c vectors and a random encryption key of 128-bits. GETCRP(s, x, c) call is then performed to obtain a challenge and response pair (C, R). The encryption key is used to encrypt the input vector. After finishing the encryption, the encryption key is masked with the response R. The ciphertext "AESVector" and "OTPKey" are finally returned and stored.

During OTPINVERSE the previously stored AESVector and OTPKey are retrieved. Using the challenge C, response R is restored by calling GETRESPONSE(C). The previously masked encryption key is then unmasked using the response R, which is used to decrypt the vector.

The cost analysis shows that masking/unmasking keys and encryption/decryption operations are relatively fast when compared to GETCRP and GETRESPONSE; the latter become the bottleneck of the OTP interface.

5.4 Hash-based OTS-SKE

In practice, there are many different ways in choosing an efficiently implementable function ϕ [68,71]. We pick the one proposed in [68] due to its efficiency. In our implementation, both $\mathcal{F}_\lambda$ and $\mathcal{H}_\lambda$ can be instantiated with standard cryptographically secure hash functions, like SHA2 [78] and SHA3 [79], because both SHA2 and SHA3 function families satisfy both one-wayness and second preimage resistance required in $\mathcal{F}_\lambda$ and $\mathcal{H}_\lambda$. The keys of hash functions can be involved in the hash function evaluation as a prefix of hash function input. This is a common practice in literature and standards [69,73,75,76,80]. Since the complexity of a generic quantum attack on the one-wayness and second pre-image resistance of a hash function families with λ-bit output is $\Theta(2^{\lambda/2})$, our implementation using SHA-256 [78] as $\mathcal{F}_\lambda$ and $\mathcal{H}_\lambda$ has 128-bit quantum security and 256-bit classical security.

In our implementation, we choose to implement a Lamport one-time signature that supports signing a 256-bit message. Effectively, this one-time signature can sign any arbitrarily long message (random nonce), because the message

Table 6: Cost Analysis of OTP and OTPinverse.

OTP		**OTPINVERSE**	
Operation	**Time (ms)**	**Operation**	**Time (ms)**
Generate Key	0.075	**GETRESPONSE**	6.88
GETCRP	2.24	**Unmask Key**	0.0003
OTP Key (Mask Key)	0.0003	**Decryption (+AES Vector)**	0.3
Encryption (+AES Vector)	0.3		
Total	2.61	**Total**	7.18

can always be compressed to a 256-bit digest using a collision resistance hash function like SHA-256 [78]. In the implementation, we pick $q = 261$, and $s = 130$, because $\binom{q}{s} \approx 1.82 \times 10^{77} > 2^{256} \approx 1.15 \times 10^{77}$. To prevent adversaries from finding the pre-image $sk_{i,j}$ of any verification key $vk_{i,j}$, we set the length of $sk_{i,j}$ to be 256 bits. We report amortized results per session in a 1024-session setup in Table 7.

Key Generation. Initially for each session, Merkle trees MT_i are constructed from vk. After constructing MT_i (1024 trees in total), the upper level Merkle tree $\widehat{MT}$ is constructed from the roots of MT_i, which takes 1023 node calculations to calculate the very top root hash, which is essentially the public key of the hash based OTS scheme. For each node calculation in the tree, we use randomly generated 512-bit long bit-masks and 256-bit long hash-keys. Before two child nodes are hashed together to form their parent node, the hash of child nodes is XORed with random bit masks, and the hash keys are involved as a prefix in the hash computation. Therefore, in total, this computation (bit-mask and hash-key generation + node construction) is repeated $((259 \times 1024) + 1023)/1024 \approx (259 + 1)$ times per session. Overall, EPUF becomes the performance bottleneck during key generation of the hash based OTS-SKE, since it only takes 234 ms without the time consumed by PUF evaluation while it takes about 0.82 sec together with PUF.

Signing. During signing, the authentication path for Merkle tree $\widehat{MT}$ needs to be generated and forwarded to the remote user as a part of the signature. For this, $\log_2 1024 = 10$ nodes are needed in the authentication path. According to the BDS algorithm in [72], on average only 6 leaves $y_{i,0,0}$ need to be retrieved and 15 inner nodes need to be computed in every session. Notice that, the authentication path is in the Merkle tree $\widehat{MT}$, so 6 leaf accesses become constructing the root hashes of 6 Merkle trees MT_i. Constructing the lower-level Merkle tree takes about 103.8 ms each, which will become one of the main bottlenecks of signing (669.3 ms in total). Since calculation the roots is very expensive, we avoid this by storing all the root hashes of lower-level merkle trees in memory in the key generation phase, and they can be retrieved and used directly in signing. This additional storage cost will be reflected in the storage cost table (Table 8) later. In conclusion, under these settings, it takes 0.94 sec to generate one signature. Excluding the PUF operations, it takes approximately 46.55 ms.

Verification. During verification, the remote user receives the 130 secret keys used by the enclave and hashes each of them to insert them in the public merkle tree together with the rest of the $t - s = 131$ verification keys using the mapping function ϕ. After this, bit-mask and hash-keys are regenerated using the same RNG seed (sent by the enclave in the public key) for 259 nodes of the lower-level Merkle tree. Using the $\log_2 1024 = 10$ authentication nodes of the upper-level Merkle tree (sent by the enclave) and regenerating bit-mask and hash-keys for each node, the remote user calculates the root hash of the merkle tree and compares it with the pk to verify the signature. During verification, as it can be seen from the table, regeneration of the bit-masks and hash-keys becomes costly. Although, they can also be stored in memory, this would increase the memory and communication cost between the enclave and remote user since the enclave needs to forward all keys to the user each time it generates a signature.

Storage Cost. The storage cost of our scheme is presented in Table 8. Here, we assume that key generation is set up for $N = 1024$ sessions and analyze the cost based on this assumption. Note that, we do not store the whole Merkle trees in memory. For fast signing procedure, we only store the root hash of lower level Merkle trees (MT_i), which costs 3.3×10^{-2} MB for 1024 sessions. The generation of authentication paths in $\widehat{MT}$ only needs to access its leaf nodes (root hashes of MT_i), so storing the root hashes of MT_i allows us to avoid reconstructing MT_i frequently. As it can be seen from the table, this storage cost is negligible (3.3×10^{-2} MB). Besides, a 256-bit PRNG seed is stored to regenerate the bit-mask and hash-keys.

Communication. Table 9 presents the size of data transmitted between the remote user and the remote attestation enclave during the signing phase. In total, 8.5 KB data needs to be transmitted.

Table 7: Amortized Cost Analysis per Session of OTS-SKE Schemes in a 1024-session setup. q' is the number of hash calls needed for constructing a Merkle tree with q leaves. ($l = \log N$)

	Hash based OTS-SKE			
	Operation	**Time (ms)**	**Repeats**	**Total (ms)**
Key Generation	sk Gen.	0.1	261 (q)	26.1
	vk Gen.	0.1	261 (q)	26.1
	EPUF Interface	2.24	261 (q)	584.6
	Encryption	0.3	261 (q)	78.3
	Masking Encryption Key	0.0003	261 (q)	0.078
	Bit-Mask and Hash-Key Gen.	0.3	260 ($q' + \frac{N-1}{N}$)	80.7
	Merkle Tree Construction	0.1006	260 ($q' + \frac{N-1}{N}$)	26.2
	Total			***819.3***
	Total without PUF			***234.7***
Sign	EPUF Interface	6.88	130 (s)	894.4
	Unmasking Encryption Key	0.0003	130 (s)	0.039
	Decryption	0.3	130 (s)	39
	Mapping (ϕ)	1.5	1	1.5
	$\widehat{MT}$ Authentication Path Gen.	0.4006	15 ($\frac{3(l-1)}{2} + 1$)	6.01
	Total ($934.9 + 0.6 \cdot l$)			***940.9***
	Total without PUF ($40.55 + 0.6 \cdot l$)			***46.55***
Verify	Mapping (ϕ)	1.5	1	1.5
	vk Gen.	0.1	130 (s)	13
	MT_i Root Verification	0.1006	259 (q')	26
	Bit-Mask and Hash-Key Gen.	0.3	269 ($q' + l$)	80.7
	$\widehat{MT}$ Authentication Path Ver.	0.1006	10 (l)	1.006
	Total ($118.2 + 0.4 \cdot l$)			***122.2***

Table 8: Total Storage Cost of Hash based OTS-SKE during Initialization for $N = 1024$ sessions.

Component	**Size (bits)**	**Quantity**	**Total Size (MB)**
vk	256	261 × 1024 (Nq)	8.2
sk	256	261 × 1024 (Nq)	8.2
CRP	3312	261 × 1024 (Nq)	105.5
Masked AES Key	128	261 × 1024 (Nq)	4.1
RNG Seed	256	1	3.2×10^{-5}
MT_i Roots	256	1024 (N)	3.3×10^{-2}
Total ($3 \cdot 10^{-5} + 0.12 \cdot N$)			125.9

Table 9: Communication Cost between the RA Enclave and Remote User during Signing (N=1024 sessions) of Hash based OTS-SKE. (E=Sent by Enclave, U=Sent by Remote User)

Component	**Size (bits)**	**Quantity**	**Total Size (KB)**
Nonce (U)	256	1	0.031
sk in Signature (E)	256	130 (s)	4.06
vk in Signature (E)	256	131 ($q - s$)	4.09
Authentication Path (E)	256	10 (l)	0.31
Total ($8.2 + 0.03 \cdot l$)			8.5

Lines of Code (LoC) within the RA Enclave. LoC is an important indicator with respect to trust as it indicates the size of the RA enclave code which needs to be trusted not having a vulnerability which can be exploited during interactions with the outside world and which needs to be trusted to correctly compute the RA protocol. The hash based OTS-SKE scheme is very compact and only takes less than 500 lines of code.

Potential for Acceleration. Previously we have shown that the extended EPUF interface leads to a significant overhead during key generation and signing which was due to the excessive amount of PUF calls performed. Potentially, in a practical application, the EPUF interface can entirely be pushed into hardware with more circuitry and can become a component of the TCB. Depending on the availability of resources, the number of PUFs can be increased in the TCB, which may allow for more parallelism and acceleration. When the key generation time becomes critical, we can reduce it time by implementing a multi-layer signature structure as XMSS$^{\text{MT}}$ [74]. The basic idea is to initialize a hash-based OTS-SKE with a small number of sessions N, then one can always expand the structure by using a session key to sign the public key of a new hash-based OTS-SKE instance. After that, new messages can be signed by the new instance. In this way, we can support virtually unlimited number of sessions with the cost of longer signing time.

6 Conclusion

We demonstrated for the first time how to design a Remote Attestation (RA) protocol that resists an All Digital State Observing (ADSO) adversary during signing. Even with secure processor technology that implements access control using hardware isolation but without any privacy guarantees (due to the recent avalanche of attacks), our remote attestation is secure and can be used to verify computation by remote users. E.g. a remote module can verify that, rather than a powerful ADSO adversary, a critical control system provided a received command. The new RA scheme offers a first crucial level of trust for current attacked secure processor technology.

7 Acknowledgements

This research was supported by the National Science Foundation under Grants No. 1617774 and 1929261.

References

[1] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *SIGP*, 2000.

[2] G. Suh, D. Clarke, B. Gassend, Marten van Dijk, and S. Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *ICS*, 2003.

[3] ARM. Arm security technology – building a secure system using trustzone technology. *ARM Technical White Paper*, 2009.

[4] D. Champagne and R. Lee. Scalable architectural support for trusted software. *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, 2010.

[5] Rick Boivie and Peter Williams. Secureblue++: Cpu support for secure execution. *IBM, IBM Research Division, RC25287 (WAT1205-070)*, pages 1–9, 2012.

[6] Christopher W. Fletcher, Marten van Dijk, and S. Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *STC '12*, 2012.

[7] V. Costan and S. Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016:86, 2016.

[8] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, 2016.

[9] Thomas Bourgeat, I. Lebedev, A. Wright, Sizhuo Zhang, Arvind, and S. Devadas. Mi6: Secure enclaves in a speculative out-of-order processor. *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[10] A. Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. A survey of published attacks on intel sgx. *ArXiv*, abs/2006.13598, 2020.

[11] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. In Ras Bodik, editor, *Proceedings of ASPLOS 2013*, pages 253–64. ACM Press, March 2013.

[12] Jae-Hyuk Lee, J. S. Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, M. Peinado, and B. B. Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security Symposium*, 2017.

[13] Nico Weichbrodt, A. Kurmus, Peter R. Pietzuch, and R. Kapitza. Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves. In *ESORICS*, 2016.

[14] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *2nd Workshop on System Software for Trusted Execution (SysTEX)*, pages 4:1–4:6. ACM, October 2017.

[15] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *26th ACM Conference on Computer and Communications Security (CCS)*, pages 1741–1758, November 2019.

[16] Jo Van Bulck, F. Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[17] Jago Gyselinck, Jo Van Bulck, F. Piessens, and Raoul Strackx. Off-limits: Abusing legacy x86 memory segmentation to spy on enclaved execution. In *ESSoS*, 2018.

[18] Tianlin Huo, X. Meng, W. Wang, Chunliang Hao, Pei Zhao, J. Zhai, and Mingshu Li. Bluethunder: A 2-level directional predictor based side-channel attack against sgx. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020:321–347, 2020.

[19] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, X. Wang, Vincent Bindschaedler, H. Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[20] Jo Van Bulck, M. Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, F. Piessens, M. Silberstein, T. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *USENIX Security Symposium*, 2018.

[21] A. Moghimi, Gorka Irazoqui Apecechea, and Thomas Eisenbarth. Cachezoom: How sgx amplifies the power of cache attacks. *IACR Cryptol. ePrint Arch.*, 2017:618, 2017.

[22] J. Götzfried, Moritz Eckert, Sebastian Schinzel, and T. Müller. Cache attacks on intel sgx. *Proceedings of the 10th European Workshop on Systems Security*, 2017.

[23] M. Schwarz, S. Weiser, D. Gruss, Clémentine Maurice, and S. Mangard. Malware guard extension: Using sgx to conceal cache attacks. In *DIMVA*, 2017.

[24] F. Brasser, U. Müller, A. Dmitrienko, Kari Kostiainen, Srdjan Capkun, and A. Sadeghi. Software grand exposure: Sgx cache attacks are practical. *ArXiv*, abs/1702.07521, 2017.

[25] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, N. Heninger, A. Moghimi, and Yuval Yarom. Cachequote: Efficiently recovering long-term secrets of sgx epid via cache attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018:171–191, 2018.

[26] A. Moghimi, Jan Wichelmann, Thomas Eisenbarth, and B. Sunar. Memjam: A false dependency attack against constant-time crypto implementations. *International Journal of Parallel Programming*, 47:538–570, 2018.

[27] S. Lee, Ming-Wei Shih, P. Gera, Taesoo Kim, Hyesoon Kim, and M. Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. In *USENIX Security Symposium*, 2017.

[28] Dmitry Evtyushkin, Ryan Riley, N. Abu-Ghazaleh, and D. Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[29] P. Kocher, Daniel Genkin, D. Gruss, W. Haas, Michael Hamburg, Moritz Lipp, S. Mangard, Thomas Prescher, M. Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019.

[30] Moritz Lipp, M. Schwarz, D. Gruss, Thomas Prescher, W. Haas, S. Mangard, P. Kocher, Daniel Genkin, Yuval Yarom, and Michael Hamburg. Meltdown. *Science*, 283:1237 – 1237, 1999.

[31] G. Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and T. H. Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. *IEEE Security & Privacy*, 18:28–37, 2020.

[32] S. V. Schaik, M. Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. Cacheout: Leaking data on intel cpus via cache evictions. *ArXiv*, abs/2006.13353, 2020.

[33] S. V. Schaik, A. Milburn, Sebastian Österlund, Pietro Frigo, G. Maisuradze, Kaveh Razavi, H. Bos, and C. Giuffrida. Ridl: Rogue in-flight data load. *2019 IEEE Symposium on Security and Privacy (SP)*, pages 88–105, 2019.

[34] M. Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and D. Gruss. Zombieload: Cross-privilege-boundary data sampling. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[35] Hany Ragab, A. Milburn, Kaveh Razavi, H. Bos, and Cristiano Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *IEEE S&P 2021*, 2021.

[36] Yeongjin Jang, Jae-Hyuk Lee, S. Lee, and Taesoo Kim. Sgx-bomb: Locking down the processor via rowhammer attack. *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, 2017.

[37] Yoongu Kim, R. Daly, Jeremie Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372, 2014.

[38] Z. Zhang, Yueqiang Cheng, D. Liu, S. Nepal, and Zongguo Wang. Telehammer: A formal model of implicit rowhammer. *arXiv: Cryptography and Security*, 2019.

[39] Kit Murdock, D. Oswald, F. Garcia, Jo Van Bulck, D. Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1466–1482, 2020.

[40] I. Lebedev, Kyle Hogan, and S. Devadas. Invited paper: Secure boot and remote attestation in the sanctum processor. *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 46–60, 2018.

[41] Kenneth A. Goldman, R. Perez, and R. Sailer. Linking remote attestation to secure tunnel endpoints. In *STC '06*, 2006.

[42] M. Jakobsson. Secure remote attestation. *IACR Cryptol. ePrint Arch.*, 2018:31, 2018.

[43] Avani Dave, N. Banerjee, and C. Patel. Sracare: Secure remote attestation with code authentication and resilience engine. *2020 IEEE International Conference on Embedded Software and Systems (ICESS)*, pages 1–8, 2020.

[44] Edlira Dushku, M. M. Rabbani, M. Conti, L. Mancini, and Silvio Ranise. Sara: Secure asynchronous remote attestation for iot systems. *IEEE Transactions on Information Forensics and Security*, 15:3123–3136, 2020.

[45] Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Gene Tsudik. Aid: autonomous attestation of iot devices. In *SRDS*, 2018.

[46] Florian Kohnhäuser, Niklas Büscher, and S. Katzenbeisser. A practical attestation protocol for autonomous embedded systems. *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 263–278, 2019.

[47] Gene Itkis. Forward security, adaptive cryptography: Time evolution , 2004.

[48] Yevgeniy Dodis, Jonathan Katz, Shouhuai Xu, and Moti Yung. Key-Insulated Public Key Cryptosystems. In *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*, pages 65–82, 2002.

[49] Yevgeniy Dodis, Jonathan Katz, Shouhuai Xu, and Moti Yung. Strong Key-Insulated Signature Schemes. In *Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography, Miami, FL, USA, January 6-8, 2003, Proceedings*, pages 130–144, 2003.

[50] Yumiko Hanaoka, Goichiro Hanaoka, Junji Shikata, and Hideki Imai. Identity-Based Hierarchical Strongly Key-Insulated Encryption and Its Application. In *Advances in Cryptology - ASIACRYPT 2005, 11th International Conference on the Theory and Application of Cryptology and Information Security, Chennai, India, December 4-8, 2005, Proceedings*, pages 495–514, 2005.

[51] Matthew K. Franklin. A survey of key evolving cryptosystems. *IJSN*, 1(1/2):46–53, 2006.

[52] Yohei Watanabe and Junji Shikata. Identity-Based Hierarchical Key-Insulated Encryption Without Random Oracles. In *Public-Key Cryptography - PKC 2016 - 19th IACR International Conference on Practice and Theory in Public-Key Cryptography, Taipei, Taiwan, March 6-9, 2016, Proceedings, Part I*, pages 255–279, 2016.

[53] Yevgeniy Dodis, Matthew K. Franklin, Jonathan Katz, Atsuko Miyaji, and Moti Yung. Intrusion-Resilient Public-Key Encryption. In *Topics in Cryptology - CT-RSA 2003, The Cryptographers' Track at the RSA Conference 2003, San Francisco, CA, USA, April 13-17, 2003, Proceedings*, pages 19–32, 2003.

[54] Yevgeniy Dodis, Matthew K. Franklin, Jonathan Katz, Atsuko Miyaji, and Moti Yung. A Generic Construction for Intrusion-Resilient Public-Key Encryption. In *Topics in Cryptology - CT-RSA 2004, The Cryptographers' Track at the RSA Conference 2004, San Francisco, CA, USA, February 23-27, 2004, Proceedings*, pages 81–98, 2004.

[55] Charles Herder, Ling Ren, Marten Van Dijk, Meng-Day Yu, and Srinivas Devadas. Trapdoor computational fuzzy extractors and stateless cryptographically-secure physical unclonable functions. *IEEE Transactions on Dependable and Secure Computing*, 14(1):65–82, 2016.

[56] Chenglu Jin, Charles Herder, Ling Ren, Phuong Ha Nguyen, Benjamin Fuller, S. Devadas, and Marten van Dijk. Fpga implementation of a cryptographically-secure puf based on learning parity with noise. *Cryptogr.*, 1:23, 2017.

[57] Erik Dahmen, Katsuyuki Okeya, Tsuyoshi Takagi, and Camille Vuillaume. Digital Signatures Out of Second-Preimage Resistant Hash Functions. In Johannes Buchmann and Jintai Ding, editors, *Post-Quantum Cryptography*, volume 5299, pages 109–123. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[58] Chenglu Jin, Wayne Burleson, Marten van Dijk, and Ulrich Rührmair. Erasable pufs: Formal treatment and generic design. In *Proceedings of the 4th ACM Workshop on Attacks and Solutions in Hardware Security*, pages 21–33, 2020.

[59] Georg T Becker. The gap between promise and reality: On the insecurity of xor arbiter pufs. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 535–555. Springer, 2015.

[60] Johannes Tobisch, Anita Aghaie, and Georg T Becker. Combining optimization objectives: New modeling attacks on strong pufs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 357–389, 2021.

[61] Phuong Ha Nguyen, Durga Prasad Sahoo, Chenglu Jin, Kaleel Mahmood, Ulrich Rührmair, and Marten van Dijk. The interpose puf: Secure puf design against state-of-the-art machine learning attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 243–290, 2019.

[62] Nils Wisiol, Christopher Mühl, Niklas Pirnay, Phuong Ha Nguyen, Marian Margraf, Jean-Pierre Seifert, Marten van Dijk, and Ulrich Rührmair. Splitting the interpose puf: A novel modeling attack strategy. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 97–120, 2020.

[63] Ulrich Rührmair, Frank Sehnke, Jan Sölter, Gideon Dror, Srinivas Devadas, and Jürgen Schmidhuber. Modeling attacks on physical unclonable functions. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 237–249, 2010.

[64] Shahin Tajik, Enrico Dietz, Sven Frohmann, Jean-Pierre Seifert, Dmitry Nedospasov, Clemens Helfmeier, Christian Boit, and Helmar Dittrich. Physical characterization of arbiter pufs. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 493–509. Springer, 2014.

[65] R. Elbaz, D. Champagne, C. Gebotys, R. Lee, N. Potlapally, and L. Torres. Hardware mechanisms for memory authentication: A survey of existing techniques and engines. *Trans. Comput. Sci.*, 4:1–22, 2009.

[66] Dag Arne Osvik, A. Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In *CT-RSA*, 2006.

[67] Leslie Lamport. Constructing digital signatures from a one-way function. Technical report, Technical Report CSL-98, SRI International, 1979.

[68] Jurjen NE Bos and David Chaum. Provably unforgeable signatures. In *Annual International Cryptology Conference*, pages 1–14. Springer, 1992.

[69] Andreas Hülsing, Joost Rijneveld, and Fang Song. Mitigating Multi-target Attacks in Hash-Based Signatures. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *Public-Key Cryptography – PKC 2016*, volume 9614, pages 387–416. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[70] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

[71] Leonid Reyzin and Natan Reyzin. Better than BiBa: Short One-Time Signatures with Fast Signing and Verifying. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Lynn Batten, and Jennifer Seberry, editors, *Information Security and Privacy*, volume 2384, pages 144–153. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[72] Johannes Buchmann, Erik Dahmen, and Michael Schneider. Merkle Tree Traversal Revisited. In Johannes Buchmann and Jintai Ding, editors, *Post-Quantum Cryptography*, volume 5299, pages 63–78. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[73] 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, pages 117–129. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[74] Andreas Hülsing, Lea Rausch, and Johannes Buchmann. Optimal Parameters for XMSS MT. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Alfredo Cuzzocrea, Christian Kittl, Dimitris E. Simos, Edgar Weippl, and Lida Xu, editors, *Security Engineering and Intelligence Informatics*, volume 8128, pages 194–208. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[75] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: Practical Stateless Hash-Based Signatures. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, volume 9056, pages 368–397. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[76] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The SPHINCS $^+$ Signature Framework. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2129–2146, London United Kingdom, November 2019. ACM.

[77] Ling Ren, Christopher W Fletcher, Albert Kwon, Marten Van Dijk, and Srinivas Devadas. Design and implementation of the ascend secure processor. *IEEE Transactions on Dependable and Secure Computing*, 16(2):204–216, 2017.

[78] Quynh H. Dang. Secure Hash Standard. Technical Report NIST FIPS 180-4, National Institute of Standards and Technology, July 2015.

[79] Morris J. Dworkin. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Technical Report NIST FIPS 202, National Institute of Standards and Technology, July 2015.

[80] A.T. Hülsing, Denise Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. Xmss: extended hash-based signatures. rfc 8391, 2018.

A Failure Rate Extended PUF Interface

In this appendix we analyse the failure probability. We select a threshold T and call positions i high confident if $con_i \geq T$ and call positions low confident if $con_i < T$; let us denote these subsets by $\mathcal{H}$ and $\mathcal{L}$ respectively. Set $\mathcal{H}$ can be split into two kinds: $\mathcal{H}^1$ are the positions with errors $e_i = 1$ and $\mathcal{H}^0$ are the positions without errors $e_i = 0$. These different cases correspond to the following probabilities:

$$\begin{aligned} Pr[i \in \mathcal{H}^0] &= \sum_{j=0}^{k-T} \binom{2k+1}{j} P^j (1-P)^{2k+1-j} \\ Pr[i \in \mathcal{L}] &= \sum_{j=k+1-T}^{k+T} \binom{2k+1}{j} P^j (1-P)^{2k+1-j}, \\ Pr[i \in \mathcal{H}^1] &= \sum_{j=k+1+T}^{2k+1} \binom{2k+1}{j} P^j (1-P)^{2k+1-j}. \end{aligned}$$

The probability of GETRESPONSE failing is at most the probability that GETRESPONSE fails where $\mathcal{I}$ is restricted to be a subset of $\mathcal{H}^0 \cup \mathcal{H}^1$ (indices in $\mathcal{I}$ may only correspond to confidences $\geq T$). Let $h_0 = |\mathcal{H}^0|$ and $h_1 = |\mathcal{H}^1|$. Then in order to succeed, the matrix $\mathcal{A}_{\mathcal{H}}$ must have full (row) rank so that an invertible submatrix $A_{\mathcal{I}}$ exists. Since A was selected a-priori at random, the probability of $\mathcal{A}_{\mathcal{H}}$ having full (row) rank is equal to

$$\prod_{i=1}^{\lambda} (1 - 2^{i-1-(h_0+h_1)})$$

for $h_0 + h_1 \geq \lambda$ (if not, then this probability is 0). GETRESPONSE selects an index set $\mathcal{I}$ of size λ and only if these positions correspond to a subset in $\mathcal{H}^0$, then all positions are without error and s can be correctly reconstructed. This happens with probability

$$\binom{h_0}{\lambda} / \binom{h_0+h_1}{\lambda} = \frac{h_0!(h_0-h_1-\lambda)!}{(h_0-\lambda)!(h_0+h_1)!}.$$

Combining the above formulas shows that the probability of success is at least

$$\sum_{h_0+h_1=\lambda}^{m} \binom{m}{h_0, h_1} Pr[i \in \mathcal{H}^0]^{h_0} Pr[i \in \mathcal{L}]^{m-h_0-h_1} \cdot$$
$$\cdot Pr[i \in \mathcal{H}^1]^{h_1} \frac{h_0!(h_0-h_1-\lambda)!}{(h_0-\lambda)!(h_0+h_1)!} \prod_{i=1}^{\lambda} (1 - 2^{i-1-(h_0+h_1)}).$$

This is in turn at least the same expression where the sum is from $h_0 + h_1 = t$ to m for $t \geq \lambda$. Therefore the probability of failure is at most

$$1 - \sum_{h_0+h_1=t}^{m} \binom{m}{h_0, h_1} Pr[i \in \mathcal{H}^0]^{h_0} Pr[i \in \mathcal{L}]^{m-h_0-h_1} \cdot$$
$$\cdot Pr[i \in \mathcal{H}^1]^{h_1} \frac{h_0!(h_0-h_1-\lambda)!}{(h_0-\lambda)!(h_0+h_1)!} \prod_{i=1}^{\lambda} (1 - 2^{i-1-(h_0+h_1)}).$$

By expressing 1 is a binomial over h_0 and h_1 we obtain that the above expression is equal to the sum of (1) and (3) given below:

$$\begin{aligned} &\sum_{h_0+h_1=0}^{t-1} \binom{m}{h_0, h_1} \begin{array}{r} Pr[i \in \mathcal{H}^0]^{h_0} Pr[i \in \mathcal{L}]^{m-h_0-h_1} \\ \cdot Pr[i \in \mathcal{H}^1]^{h_1} \end{array} \\ &= \sum_{h=0}^{t-1} \binom{m}{h} \begin{array}{r} (Pr[i \in \mathcal{H}^0] + Pr[i \in \mathcal{H}^1])^h \\ \cdot Pr[i \in \mathcal{L}]^{m-h} \end{array} && (1) \\ &\leq \exp(-2m((Pr[i \in \mathcal{H}^0] + Pr[i \in \mathcal{H}^1]) - t/m)^2) && (2) \end{aligned}$$

for $t \leq m(Pr[i \in \mathcal{H}^0] + Pr[i \in \mathcal{H}^1])$ by Hoeffding's inequality. The second part is

$$\begin{aligned}&\sum_{h_0+h_1=t}^{m} \binom{m}{h_0, h_1} \begin{array}{r} Pr[i \in \mathcal{H}^0]^{h_0} Pr[i \in \mathcal{L}]^{m-h_0-h_1} \\ \cdot Pr[i \in \mathcal{H}^1]^{h_1} \end{array} \\ &\cdot(1 - \frac{h_0!(h_0-h_1-\lambda)!}{(h_0-\lambda)!(h_0+h_1)!} \cdot \prod_{i=1}^{\lambda}(1 - 2^{i-1-(h_0+h_1)})). \qquad (3)\end{aligned}$$

Notice that

$$\begin{aligned}&\prod_{i=1}^{\lambda}(1 - 2^{i-1-(h_0+h_1)})) \geq 1 - \sum_{i=1}^{\lambda} 2^{i-1-(h_0+h_1)} \\ &= 1 - (2^{\lambda} - 1)2^{-(h_0+h_1)} \geq 1 - 2^{\lambda-(h_0+h_1)}.\end{aligned}$$

This proves that (3) is at most

$$\begin{aligned}&\sum_{h_0+h_1=t}^{m} \binom{m}{h_0, h_1} \begin{array}{r} Pr[i \in \mathcal{H}^0]^{h_0} Pr[i \in \mathcal{L}]^{m-h_0-h_1} \\ \cdot Pr[i \in \mathcal{H}^1]^{h_1} \end{array} \\ &\cdot(1 - \frac{h_0!(h_0-h_1-\lambda)!}{(h_0-\lambda)!(h_0+h_1)!} + 2^{\lambda-(h_0+h_1)}),\end{aligned}$$

which is at most

$$2^{\lambda-t} \qquad (4)$$

plus

$$\begin{aligned}&\sum_{h_0+h_1=t}^{m} \binom{m}{h_0, h_1} \begin{array}{r} Pr[i \in \mathcal{H}^0]^{h_0} Pr[i \in \mathcal{L}]^{m-h_0-h_1} \\ \cdot Pr[i \in \mathcal{H}^1]^{h_1} \end{array} \\ &\cdot(1 - \frac{h_0!(h_0-h_1-\lambda)!}{(h_0-\lambda)!(h_0+h_1)!}), \qquad (5)\end{aligned}$$

where we note that the expression in brackets equals 0 for $h_1 = 0$ and is at most 1 for $h_1 > 0$. Hence, (5) is at most

$$\begin{aligned}&\sum_{h_0=0}^{m} \binom{m}{h_0} Pr[i \in \mathcal{H}^0]^{h_0} \cdot \qquad (6) \\ &\sum_{h_1=\max\{1,t-h_0\}}^{m-h_0} \binom{m-h_0}{h_1} \begin{array}{r} Pr[i \in \mathcal{L}]^{m-h_0-h_1} \\ \cdot Pr[i \in \mathcal{H}^1]^{h_1} \end{array}\end{aligned}$$

The second sum is equal to

$$\sum_{h_1=\max\{0,t-1-h_0\}}^{m-1-h_0} \binom{m-h_0}{h_1+1} \begin{array}{r} Pr[i \in \mathcal{L}]^{m-1-h_0-h_1} \\ \cdot Pr[i \in \mathcal{H}^1]^{h_1+1} \end{array}.$$

This is in turn equal to

$$\begin{aligned}&\sum_{h_1=\max\{0,t-1-h_0\}}^{m-1-h_0} \binom{m-1-h_0}{h_1} \begin{array}{r} Pr[i \in \mathcal{L}]^{m-1-h_0-h_1} \\ \cdot Pr[i \in \mathcal{H}^1]^{h_1} \end{array} \\ &\qquad\qquad \cdot \frac{m-h_0}{h_1+1} Pr[i \in \mathcal{H}^1].\end{aligned}$$

This implies that the first sum in (6) can be reduced to

$$\begin{aligned}&\sum_{h_0=0}^{m-1} \binom{m}{h_0} Pr[i \in \mathcal{H}^0]^{h_0} \\ = &\sum_{h_0=0}^{m-1} \binom{m-1}{h_0} Pr[i \in \mathcal{H}^0]^{h_0} \cdot \frac{m}{m-h_0}.\end{aligned}$$

Combining both expressions yields the upper bound

$$mPr[i \in \mathcal{H}^1] \tag{7}$$

times

$$\begin{aligned}
&\sum_{h_0=0}^{m-1} \binom{m-1}{h_0} Pr[i \in \mathcal{H}^0]^{h_0} \cdot \\
&\sum_{h_1=\max\{0,t-1-h_0\}}^{m-h_0} \binom{m-1-h_0}{h_1} \begin{array}{r} Pr[i \in \mathcal{L}]^{m-1-h_0-h_1} \\ \cdot Pr[i \in \mathcal{H}^1]^{h_1} \end{array} \\
= &\sum_{h_0+h_1=t-1}^{m-1} \binom{m-1}{h_0, h_1} \begin{array}{r} Pr[i \in \mathcal{H}^0]^{h_0} Pr[i \in \mathcal{H}^1]^{h_1} \\ \cdot Pr[i \in \mathcal{L}]^{m-1-h_0-h_1}, \end{array}
\end{aligned}$$

which is at most 1. Summarizing, the probability of failure is at most (2)+(4)+(7) for any

$$\lambda \le t \le m(Pr[i \in \mathcal{H}^0] + Pr[i \in \mathcal{H}^1]).$$

We notice that we can achieve a possibly smaller value $m(m-1)Pr[i \in \mathcal{H}^1]^2$ in (7) if we allow one error among the t positions from which we select a subset $\mathcal{I}$ in order to construct an invertible matrix $A_{\mathcal{I}}$. However, allowing one error implies in the worst-case (if there is such an error) a search of $O(n)$ until a set $\mathcal{I}$ is found without errors such that s can be properly reconstructed. This requires a significant extra computational burden – and for this reason we want (7) to be small in the first place.

Let us further bound (2). We notice that the median of the binomial distribution that defines $Pr[i \in \mathcal{H}^0]$ is at most $\lceil (2k+1)P \rceil$. So, if we require

$$k - T \ge \lceil (2k+1)P \rceil,$$

then $Pr[i \in \mathcal{H}^0] \ge 1/2$. Together with $\lambda \le t \le m/2$ we get

$$(2) \le \exp(-2m(1/2 - t/m)^2) = \exp(-(m-2t)^2/(2m)).$$

We also notice that bound (7) is, by Hoeffding's inequality, at most

$$(7) \quad \le \quad m \cdot \exp(-2(2k+1)((1-P) - \frac{k-T}{2k+1})^2)$$

(for $k - T \le (2k+1)(1-P)$ which is true for $P \le 1/2$). This new upper bound holds for all choices satisfying $k - T \ge \lceil (2k+1)P \rceil$. By choosing

$$T = k - \lceil (2k+1)P \rceil$$

we minimize the upper bound on (2) and obtain

$$(2) \le m \cdot \exp(-2(2k+1)(1 - 2P - 1/(2k+1))^2).$$

Adding the upper bounds together achieves a failure probability of at most

$$\begin{aligned}
&\exp(-(m-2t)^2/(2m)) + 2^{\lambda - t} \\
&+ m \cdot \exp(-2(2k+1)(1 - 2P - 1/(2k+1))^2).
\end{aligned}$$

To bring the first two terms in balance we choose

$$(m-2t)^2/(2m) = (t - \lambda)\alpha \text{ with } \alpha = \ln 2,$$

that is

$$t = \frac{2+\alpha}{4}m - \frac{1}{4}\sqrt{\alpha m((4+\alpha)m - 8\lambda)}$$

which satisfies our requirment $\lambda \le t \le m/2$. This implies that the failure probability is at most (assuming $2\lambda \le m$)

$$\begin{aligned}
&2 \cdot \exp\left(-\frac{\alpha}{4}([(2+\alpha)m - 4\lambda] - \sqrt{\alpha m[(4+\alpha)m - 8\lambda]})\right) \\
&+ m \cdot \exp(-2(2k+1)(1 - 2P - 1/(2k+1))^2).
\end{aligned}$$

Table 10: Parameters for $\lambda = 256$ and $P = 0.1$.

m	k	fail	# EPUF calls	storage chall.
869	17	$0.994 \cdot 10^{-15}$	30415	4.0 KB
682	8	$1.02 \cdot 10^{-5}$	11594	1.6 KB
665	7	$0.977 \cdot 10^{-4}$	9975	1.4 KB
647	6	$1.00 \cdot 10^{-3}$	8424	1.2 KB

We may define

$$y = \alpha(2\lambda + x) + 2x \text{ with } m = 2\lambda + x$$

and write instead

$$\begin{aligned} &2 \cdot \exp\left(-\frac{\alpha}{4}\left(y - \sqrt{y^2 - 4x^2}\right)\right) \\ &+m \cdot \exp(-2(2k+1)(1-2P-1/(2k+1))^2). \end{aligned}$$

We apply this formula by choosing $\lambda = 128$, and increase x until the first exp is small enough, and next we increase k until the second exp becomes small enough, see Table 1 and 10.

Notice that one may reduce the storage of vector y if we repeat measuring $2k + 1$ responses for the same challenge. This would allows is to cut the storage of y by a factor $2k + 1$. However, the above analysis will not apply. Where we use P in our derivations we must use p_c and later take the expectation over c. This generally yields worse bounds since $F(\mathbb{E}[p_c]) \leq \mathbb{E}[F(p)]$ for $\cup$-convex $F(.)$. We assume storage is not a bottleneck on which we need to save; this allows us to use the bound proved above.

Application RA Scheme. For a single signature in remote attestation we need to mask about $2 \cdot 128$ keys of length $\lambda = 128$ during initialization mode using GETCRP, and retrieve 128 keys using GETRESPONSE when computing a signature (if we use the hash-based OTS-SKE scheme, we need 261 keys during initialization and 130 keys when computing a signature). For $k = 7$, GETRESPONSE gives a failure probability of $\approx 10^{-4}$. This translates to $\approx 128 \cdot 10^{-4} = 1.28 \cdot 10^{-2}$ failure probability when computing a signature. If a signature fails to compute, then we simply attempt to retrieve the necessary keys for the next session and inform the remote user about the new increased session counter ctr when transmitting a successfully computed signature (for the same $nonce$ supplied by the remote user for verifying freshness).

Notice that if a single successful signing key recovery for a signature takes 894.4 ms (see Table 7), then including possible failures this implies $894.4 \text{ ms} \cdot (1-0.0128) \cdot (1+2 \cdot 0.0128+3 \cdot 0.0128^2+\ldots) = 894.4 \text{ ms} \cdot (1-0.0128)/(1-0.0128)^2 = 906.0$ ms expected signature computation time (which should be discounted for in Table 7).

Practical Parameter Setting. Our lower bound on m for a given failure probability is still loose with respect to a multiplicative factor of 2. In order to achieve a failure probability of ≈ 0.0001 for GETRESPONSE in our simulations we use the following calculation: Experiments with the iPUF simulator show $P \approx 0.1099$. For $k = 7$ and $T = 4$ this gives $Pr[i \in \mathcal{H}^0] = 0.9260$ and $Pr[i \in \mathcal{H}^1] = 1.025 \cdot 10^{-9}$. The probability that there exists a high confidence position i which belongs to $\mathcal{H}^1$ is at most (there are at most m high confident positions)

$$1 - (1 - \frac{1.025 \cdot 10^{-9}}{0.9260 + 1.025 \cdot 10^{-9}})^m \leq \frac{1.025 \cdot 10^{-9}}{0.9260 + 1.025 \cdot 10^{-9}} \cdot m = 1.107 \cdot 10^{-9} \cdot m.$$

We need at least 142=128+14 highly confident positions without error so that with probability at most $2^{-14} = 0.00006$ there exists no invertible 128×128 submatrix $A_{\mathcal{I}}$. The expected number of high confident positions is $P \cdot m = 0.9260 \cdot m$. Since the Binomial distribution is well-approximated by the normal distribution, the probability that there are at least

$$0.9260 \cdot m - 3.97 \cdot \sqrt{0.9260 \cdot (1 - 0.9260) \cdot m} \tag{8}$$

high confident positions is at least 1 minus the probability of the tail of the normal distribution from 3.97 standard deviations onward, which is $1 - 0.00004$. For $m = 168$, the required 142 is equal to (8) and we conclude that with probability $\geq 1 - (1.107 \cdot 10^{-9} \cdot 168 + 0.00006 + 0.00004) = 1 - 0.0001)$ we will have 142 high confident positions without noise out of which we can choose 128 for constructing an invertible submatrix $A_{\mathcal{I}}$.

During signing we use the extended PUF interface 128 times. On average $P \cdot m = 0.9260 \cdot 168 = 155.6$ EPUF calls are needed per time.