Secure Cryptographic Precomputation with Insecure Memory[⋆]

Patrick P. Tsang and Sean W. Smith

Department of Computer Science
Dartmouth College
Hanover, NH 03755, USA
{patrick,sws}@cs.dartmouth.edu

Abstract. We propose a solution that provides secure storage for cryptographic precomputation using only insecure memory that is susceptible to eavesdropping and tampering. Specifically, we design a small tamper-resistant hardware module, the *Queue Security Proxy (QSP)*, that situates transparently on the data-path between the processor and the insecure memory. Our analysis shows that our design is secure and flexible, and yet efficient and inexpensive. In particular, both the timing overhead and the hardware cost of our solution are independent of the storage size.

1 Introduction

1.1 Precomputation

Precomputation is an optimization technique that reduces an algorithm's execution latency by performing some of its operations before knowing the input to the algorithm. The intermediate result produced by precomputation is stored and later used, when the input arrives, to compute the final output of the algorithm. As only the *post-computation*, i.e., the remaining computation that has not been precomputed, needs to be done to produce the output upon the input's arrival, execution latency is reduced.

Cryptographic precomputation In cryptography, precomputation is an old and well-known technique. For example, fixed-base modular exponentiation, which is an operation fundamental to almost all public-key cryptographic algorithms, can be sped up by precomputing a set of related exponentiations [6]. As another example, it has been widely observed that DSA signatures, as well as ElGamal and Schnorr signatures, can be precomputed [6]. More generally, all

[⋆] This work was supported in part by the U.S. Department of Homeland Security under Grant Award Number 2006-CS-001-000001, the Institute for Security Technology Studies, under Grant number 2005-DD-BX-1091 awarded by the Bureau of Justice Assistance, and the National Science Foundation, under grant CNS-0524695. The views and conclusions do not necessarily represent those of the sponsors.

signature schemes converted from Σ-protocols [15] using the Fiat-Shamir transformation [4], which include many group signatures and anonymous credential systems such as [5,7,29], can benefit from precomputation quite significantly, as most of the heavyweight operations, namely group exponentiation, can be precomputed in these schemes. Finally, precomputing homomorphic encryption can speed up mix-nets [8], as recently suggested in [1].

Reusable and consumable precomputation Precomputation can be *reusable*, i.e., a single precomputation can be reused in multiple algorithm executions, or *consumable*, i.e., a new precomputation is needed per execution. Speeding up modular exponentiation using the sliding window method [31] is an example of reusable precomputation: any modular exponentiation with the same base can be sped up by one-time precomputing a set of values for that base. On the other hand, the precomputation of DSA signatures is consumable: the group generator must be exponentiated with a new exponent for every signature to be signed.

Consumable precomputation poses a bigger challenge than reusable precomputation does when it comes to efficiently securing it against hardware attacks. Precomputation, in general, requires *integrity* of the precomputation results. Consumable precomputation usually requires *confidentiality* and *protection against replay* as well. Finally, the storage space necessary for consumable precomputation grows with the rate and burstiness of the execution of an algorithm, while it is a constant in reusable precomputation.

In this paper, we are interested in overcoming the bigger challenge, i.e., how to efficiently secure consumable precomputation. In what follows, we refer to consumable precomputation as precomputation, for simplicity's sake.

1.2 The Challenge

Precomputation is capable of reducing the execution latency of an algorithm only if a precomputed result is available upon an input's arrival. In situations where an algorithm is routinely being executed over time, computing and storing only a single precomputation result would not sustain a low latency throughout the executions; to sustain a low latency, one must therefore buffer up precomputation results, i.e., precompute those results in advance and store them in such a way that they are readily available when needed.

The need for secure storage It is therefore of paramount importance to ensure that no security breach is introduced to a cryptographic algorithm by precomputing it. The safest approach when handling the precomputation results is to treat them as internal states of the entity executing the algorithm, thereby effectively assuming them to be unobservable and unmodifiable by anyone else. Consequences could be devastating should such an assumption cease to hold. For example, when precomputing DSA signatures, allowing an adversary to eavesdrop, overwrite, or replay even only one precomputation result would leak the private signing key.

As storing multiple precomputation results requires significantly larger memory than storing only the internal states needed for a single execution, it is unre-

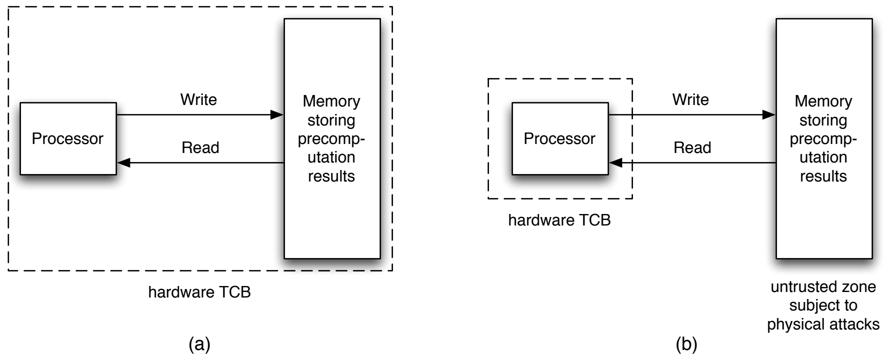
Processor
Write
Read
Memory storing precomp-utation results
hardware TCB
(a)
Processor
Write
Read
Memory storing precomp-utation results
hardware TCB
untrusted zone subject to physical attacks
(b)

Fig. 1. (a) Keeping the storage inside the hardware TCB is unrealistic; (b) putting the storage outside the hardware TCB exposes us to attacks.

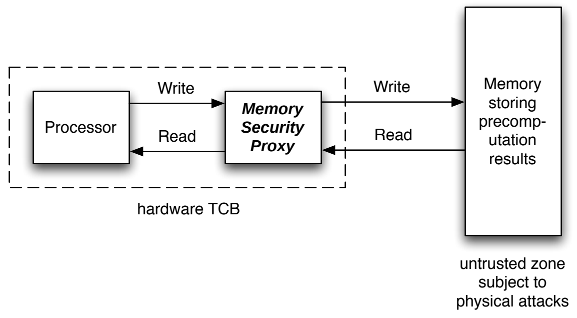
Processor
Write
Read
Memory Security Proxy
Write
Read
Memory storing precomp-utation results
hardware TCB
untrusted zone subject to physical attacks

Fig. 2. In our model, we keep the hardware TCB small by leaving the storage outside, but adding a small Memory Security Proxy.

alistic to assume that the whole storage would fit in a tamper-resistant module such as a hardened CPU (see Figure 1(a)). Leaving the memory outside (see Figure 1(b)) exposes us to the attacks discussed above. In this paper, we design a small tamper-resistant hardware module that effectively turns insecure memory into one that is secure for storing precomputation results—see Figure 2.

1.3 Our Contributions

We make the following contributions in this paper:

– We motivate that, to sustain the reduced execution latency of cryptographic algorithms made possible by precomputation, one needs to store a pool of readily available precomputation results; and that the security of such storage is critical to the security of the cryptographic algorithms.
– We design an architecture that transparently turns untrusted memory into one that is secure for storing precomputation results. The design is very

efficient and has hardware and timing costs independent of the size of the memory. Our analysis shows that the architecture we propose is secure.

The rest of this paper is organized as follows. In Section 2, we provide some background on hardware-based security and review the necessary cryptography. In Section 3, we describe our approach to overcome the posed challenge. We present our solution in detail in Section 4, and analyze its security and efficiency in Section 5. We discuss several research directions that are worth exploring in Section 6, and conclude the paper in Section 7.

2 Background

2.1 Hardware-Based Security

Designing hardware-based security mechanisms into the computing architecture is important as software security solutions are incapable of defending against hardware attacks and many software attacks. There are mechanisms that provide security against physical tampering through means such as tamper resistance, tamper evidence, and tamper detection and response. Deploying a hardware-based security mechanism could be very expensive; different trade-offs between security and cost can lead to radically different solution paradigms.

The IBM 4756 approach The IBM 4758/4764 cryptographic coprocessors [25,12] are secure crypto processors implemented on a programmable PCI board, on which components such as a microprocessor, memory, and a random number generator are housed within a tamper-responding environment. They are general-purpose x86 computers with a very high security assurance against physical attacks. While they find applications in the commerce sector such as securing bank ATMs, their high costs prevent most end-users from benefiting from them.

The hardened-CPU approach A more realistic approach to secure general-purpose computers such as today's PCs against software and even certain hardware attacks is by assuming that only CPUs are hardened. In their *Oblivious RAM (ORAM)* work, Goldreich and Ostrovsky [14] formalized a general approach to protecting the contents of exposed memory, including hiding access patterns. However, ORAM is too inefficient to use in practice. Lie et al.'s *eXecute-Only Memory (XOM)* [18] architecture built hardened-CPU prototypes but is vulnerable to replay attacks. Suh et al. later proposed *AEGIS* [26], which is immune to replay attacks and uses techniques such as Merkle-trees [20] for better efficiency. Other advances include using AES in the Counter mode to reduce memory-write latencies [27,30], prediction techniques to hide the memory-read latencies in [22,24], and on-chip caches [13] and incremental multi-set hashes [9] to reduce latency incurred by memory integrity checking.

The architecture we are going to propose in this paper falls under the hardened-CPU approach. Rather than arbitrary software, however, our architecture deals only with precomputation. We note that although architectures like AEGIS can provide the same functionality as ours, ours is simpler and more efficient due to the exploitation of properties of precomputation. In fact, while AEGIS aims to provide a secure execution environment for general-purpose computers such as x86 PCs, we gear our architecture towards a coprocessor for embedded devices.

The TCG's TPM approach The *Trusted Computing Group (TCG)* is a consortium that works towards increasing the security of standard commodity platforms. The group proposed a specification of an inexpensive micro-controller chip called the *Trusted Platform Module (TPM)* [28] to be mounted on the motherboard of commodity computing devices such as PCs. In the last few years, major vendors of computer systems have been shipping machines that have included TPMs, with associated BIOS support.

A TPM provides cryptographic functions, e.g., encryption, digital signature and hardware-based random number generation, and internal storage space for storing, e.g., cryptographic keys. TPMs act as a hardware-based root of trust and can be used to attest the initial configuration of the underlying computing platform (*attestation*), as well as to seal and bind data to a specific platform configuration (*unsealing* or *unwrapping*). The aspiration is that if the adversary compromises neither the TPM nor the BIOS, then the TPM's promises of trusted computing will be achieved. The reality is murkier: attacks on the OS can still subvert protections; recent work (e.g., [17]) is starting to demonstrate some external hardware integration flaws; and the long history of low-cost physical attacks on low-cost devices (e.g., [2]) hasn't caught up with the TPM yet.

2.2 Cryptographic Tools

Various symmetric and asymmetric cryptographic techniques provide security guarantees such as confidentiality, authentication and integrity. For example, *AES* is a block cipher that provides confidentiality of data, whereas *HMAC* is a message authentication scheme that provides both message authentication and integrity. Here we review a relatively recent cryptographic tool called *authenticated encryption*, which effectively combines the functionality of AES and HMAC. Our solution makes use of it.

Modes of operation A block cipher is a function that operates on block-length bit strings based on a key; each key determines some way of mapping block-length strings to block-length strings. Since using a block cipher on its own leads to many problems—for example, what to do with messages that are longer than a single block—*modes of operation* have been developed that describe how to extend a cipher to a secure encryption scheme.

Standard modes of operation include *Electronic Code Book (ECB)*, *Cipher-Block Chaining (CBC)*, and the *Counter (CTR)* mode. Different modes have different properties. For instance, decryption is faster than encryption under the CBC mode, which is preferable when decryption is on the critical path. The CTR mode can potentially further reduce both encryption and decryption latency by taking the AES operations away from the critical path.

Authenticated encryption Encryption constructed from block ciphers such as AES operating under any of the modes aforementioned provides data confidentiality but no data authenticity or integrity. The past decade has seen the emergence of modes of operation that efficiently provide both. When operating under these modes, a block cipher effectively becomes authenticated encryption [3], rather than just encryption alone. Among these modes, the *Counter with CBC-MAC (CCM)* mode and the *Galois/Counter (GC)* mode are particularly attractive because authentication is done without any feedback of datablocks. This means that both authentication and encryption can be parallelized to achieve extremely high throughput. We use AES under GC mode (AES-GCM) in our solution.

AES-GCM has two operations, authenticated encryption AEnc and authenticated decryption ADec. AEnc takes four inputs: (i) a *secret key* K appropriate for the underlying AES, (ii) an *initialization vector* IV, (iii) a *plaintext* P, and (iv) an *additional authenticated data (AAD)* A. Both the plaintext and the AAD will be authenticated; however, only the plaintext will be encrypted as well. AEnc outputs (i) a *ciphertext* C whose length is the same as that of P and (ii) an *authentication tag* T. The authentication tag is essentially the cryptographic checksum of the message. ADec takes five inputs: K, IV, C, A and T. It outputs either P or a special symbol that indicates failure, i.e., the inputs are not authentic. We refer the readers to [19,11] for details regarding the specification, performance and security of AES-GCM.

3 Our Approach

We now return to the challenge we discussed in Section 1: how to cost-effectively provide a secure storage for cryptographic precomputation so that it can provide a sustainable benefit without breaking the security. We introduce in this section our approach to overcome such a challenge.

3.1 Memory Security Proxy

Since putting memory within the hardware TCB would not meet our design goal of being cost-effective, in our solution insecure memory will be used. To account for the fact that insecure memory can be eavesdropped and tampered with, we augment its use with security measures enforced by a small tamper-proof hardware module through cryptographic techniques. We call such a module the *Memory Security Proxy (MSP)*. The MSP must be efficient in both space and computation: its size (e.g., in terms of gate-count) and the timing overhead it incurs should grow as slowly as possible with—or even better, be independent of—the size of the insecure memory it is securing.

In our solution, the MSP situates between the insecure memory and the processor, the latter of which executes (the pre- and post-computation of) the cryptographic algorithm. The MSP effectively turns the insecure memory into a secure one by instrumenting the processor's read and write accesses. This means that the MSP will incur timing overhead on these accesses, but should otherwise be transparent to the processor: the processor is oblivious to whether it is accessing insecure memory without protection whatsoever, memory secured through hardware tamper-proof mechanisms, or our MSP. This is an attractive property, as it simplifies designs and allows better interoperability and upgradability.

In a threat model where adversary is physically present and capable of launching hardware attacks against the computing devices, the processor that executes the cryptographic operations must be trusted to behave securely regardless of those attacks. Similarly, we rely on the MSP to be as trustworthy as the main processor. This is a realistic assumption because any satisfactory MSP design is going to have a very small physical size, most likely just a small fraction of the size of the processor. Figure 2 depicts our architectural model.

We point out that the above model is not a new one; rather, all architectures we reviewed in Section 2 that take the hardened-CPU approach follow this model. However, all those architectures focus on securing general-purpose computers such as PCs, in which the insecure memory being protected is randomly-accessible. It has been proven in these works that it is very difficult to efficiently secure insecure RAM. However, we only need to protect a storage of precomputed results. As we will show next, we can exploit the properties of cryptographic precomputation to avoid some of the difficulties and thus achieve better performance results.

3.2 Data Structure for Precomputation Storage

Applying precomputation to a cryptographic algorithm turns the algorithm's execution into a process that follows the *producer-consumer model*. Under such a model, the producer produces, through precomputation, the goods (i.e., the precomputation results) that are later consumed by the consumer, through post-computation, upon the arrival of algorithmic inputs. The asynchronous communication channel between the producer and the consumer may be implemented using data structures such as stacks, *First-In-First-Out (FIFO)* queues, register arrays (a.k.a. RAM), depending on the desired order of the goods being consumed (relative to them being produced).

We make the observation that using precomputation results in the same order as they were produced always yields a correct execution of the cryptographic algorithms. In fact, in many cases precomputation results are statistically uncorrelated to one another, and one may thus even use them in arbitrary order. The precomputation of DSA signatures is one example. Consequently, data structures such as RAM and FIFO queues are legitimate candidates for storing cryptographic precomputation results. In our design to be presented in the next section, we use FIFO queues rather than RAM because of the following reasons:

– Securing insecure RAM efficiently has been proven to be difficult. On the contrary, as we will see, our securing insecure FIFO queues is very efficient.
– Insecure FIFO queues can be efficiently implemented using insecure RAM (in both software and hardware) but not the other way round. Hence, our solution requires only the "weaker" data structure.

From now on, we abbreviate FIFO queues as queues. Also, we call the Memory Security Proxy we are going to build the *Queue Security Proxy (QSP)*, as it is specialized for protecting queues. Next, we provide some formalism of queues, which is necessary to reason about the correctness and security of our design.

Queues A queue is a data structure that implements a First-In, First-Out (FIFO) policy by supporting two operations, namely Enqueue and Dequeue, which inserts and deletes elements from the data structure respectively. Such a policy can be more rigorously specified using the axiomatic approach of Larch [16], in which an object's sequential history is summarized by a value that reflects the object's state at the end of the history. A queue implementation is said to be *correct* if the policy is satisfied. For simplicity's sake, we also use the following verbal definition of correctness: A queue is correct if the i-th item dequeued has the same value as the i-th item enqueued for all $i \in \mathbb{N}$. Note that correctness is defined assuming the absence of an adversary.

3.3 Threat Model

We now define the security requirements of the QSP and the threat model under which these requirements must be met. To attack the security of the QSP, a computationally bounded but physically present adversary may launch physical attacks on the untrusted zone of the architecture, as illustrated in Figure 2. Such an adversary may also probe the input-output relationship of the QSP as follows. She may arbitrarily and adaptively ask the processor to produce a precomputation result and enqueue it to the QSP, as well as to dequeue a precomputation result from the QSP and reveal it to her. Note that in reality a precomputation result is never directly revealed to anybody. We give such a capability to the adversary so that the security of the QSP can be agnostic to the cryptographic algorithm being precomputed. Such a modeling provides a confidentiality guarantee at least as strong as needed.

We regard a QSP design as secure if it has correctness, confidentiality and integrity, defined as follows.

Correctness If the underlying queue that a QSP is protecting is correct, then the QSP behaves correctly as a queue in the absence of an adversary.

Confidentiality We hinted earlier that the safest strategy to take when securing precomputation storage is to assume the entire precomputation result to be as private as any algorithmic internal states. In certain scenarios, however, part of the precomputation result can be made public. One such scenario is when that part will eventually appear in the final algorithmic output, and its release prior to the arrival of the input does not lead to security breaches. Not having to encrypt the whole result improves efficiency.

As an example, the precomputation result of DSA signing is of the form (r, k^{-1}) (as defined in [21]). The knowledge of r alone, which will eventually be a part of the output signature, does not give any extra information to any computationally bounded adversary, whereas knowing k^{-1} would allow the extraction of the secret key and thus lead to universal compromise.

If we denote a precomputation result to be enqueued as a data object $D = (A, P)$, where confidentiality is necessary for P, but not A, then a QSP design has confidentiality if no adversary, whose capabilities are as described above, can learn, with non-negligible probability, any information about the P in any D that the adversary did not ask the processor to reveal.

Integrity Roughly speaking, a QSP with integrity is one that either responds to accesses correctly as a queue, or signals an error when having been tampered with. Note that this definition of integrity implies both authenticity and freshness of precomputation results. Specifically, if an adversary can modify a precomputation result, say the i-th one enqueued, without the QSP being able to detect it no latter than it is being dequeued, then the QSP would not be correct as what is dequeued at the i-th time is different from what was enqueued at the i-th time. Similarly, the ability to replay an old result leads to the same violation of integrity.

More formally, an adversary is successful in attacking the integrity of a QSP when there exists an $i \in \mathbb{N}$ such that the precomputation results enqueued at the i-th time differs from what is dequeued at the i-th time. A QSP design has integrity if no adversary with capabilities described above can succeed with non-negligible probability.

4 Our QSP Design

We first give the solution idea behind our QSP design. We then present the actual design, first in the form of software pseudo-code to facilitate understanding of the design and reasoning about its correctness and security, then in the form of a hardware architectural design.

4.1 Solution Idea

The use of authenticated encryption makes our QSP design very simple. Precomputation results (*Data_in*) generated by the processor are fed to the QSP. The QSP encrypts these results with AES-GCM, incrementing the initialization vector (*IV_out*) per encryption. Incrementing the IV serves two purposes. First, for AES-GCM to be secure, the IV should never be reused under the same key. Second, the IV serves as a counter that gives a sequential and consecutive ordering to the precomputation results being operated on, and hence helps defending against replay attacks. The output of the AES-GCM encryption (*SecData_out*) can then be enqueued into an insecure queue.

When the processor asks for a precomputation result from storage, the QSP dequeues an entry from the insecure queue and decrypts that entry using AES-GCM. Again, the initialization vector (*IV_in*) increments per decryption. As long as *IV_in* and *IV_out* are set to the same value (e.g. zero) during start-up before any enqueue or dequeue operations, the precomputation result encrypted with an IV value will eventually be decrypted with the same IV value.

4.2 The Construction

System parameters Keys in the QSP may remain constant throughout the lifetime of the QSP, or be randomly generated during boot-time. Using a new set of keys effectively means all the old precomputation results are flushed. The two IVs are set to zero during start-up. We chose AES-128 for AES-GCM. This means a key size of 128-bit. The size of the IVs has to be such that the IVs never overflow. We used 80-bit IVs. Finally we selected 96-bit as the size of the Tag.

Software Construction Without loss of generality, we assume that the underlying insecure queue, denoted by Q, provides an interface for querying whether it is full or not, and empty or not. Algorithms below show the software implementation of the enqueue and dequeue operations performed by the QSP.

```
Algorithm QSP.Enqueue($Data_{in}$)
private input: $K, IV_{out}, Q$
1: if $Q$.isFull() then
2:     return error
3: $\langle P_{in}, A_{in} \rangle \leftarrow Data_{in}$
4: $\langle C_{out}, T_{out} \rangle \leftarrow$ AEnc($K, IV_{out}, P_{in}, A_{in}$)
5: $IV_{out} \leftarrow IV_{out} + 1$
6: $SecData_{out} \leftarrow \langle A_{in}, C_{out}, T_{out} \rangle$
7: $Q$.enqueue($SecData_{out}$)
```

```
Algorithm QSP.Dequeue()
private input: $K, IV_{in}, Q$
1: if $Q$.isEmpty() then
2:     return error
3: $SecData_{in} \leftarrow Q$.dequeue()
4: $\langle A_{out}, C_{in}, T_{in} \rangle \leftarrow SecData_{in}$
5: $res \leftarrow$ ADec($K, IV_{in}, C_{in}, A_{out}, T_{in}$)
6: if $res =$ failure then
7:     return error
8: else
9:     $IV_{in} \leftarrow IV_{in} + 1$
10:    $P_{out} \leftarrow res$
11:    return $Data_{out} \leftarrow \langle P_{out}, A_{out} \rangle$
```

Hardware Construction Figure 3 shows the architectural design of our QSP. *Data_in* and *Data_out* are connected to the processor, while *SecData_in* and *SecData_out* are connected to the insecure queue. We assume that the insecure queue is asynchronous, i.e., it supports asynchronous enqueuing and dequeuing. It is straightforward to modify the design if a synchronous queue is used instead.

Notice that the authenticated encryption of a precomputation result is bigger in size than the result itself. One may replace the data-bus for *SecData* (both in and out) with a wider one. Another possibility is to change the software that implements the cryptographic algorithm so that precomputation results do not use up the whole bus-width and that they still fit in the bus after the expansion. However, both approaches require modification to the existing hardware and/or software and thus lack transparency and interoperability.

Our recommended approach is to have the QSP split up every incoming precomputation result into two halves and enqueue each half as if it was a single

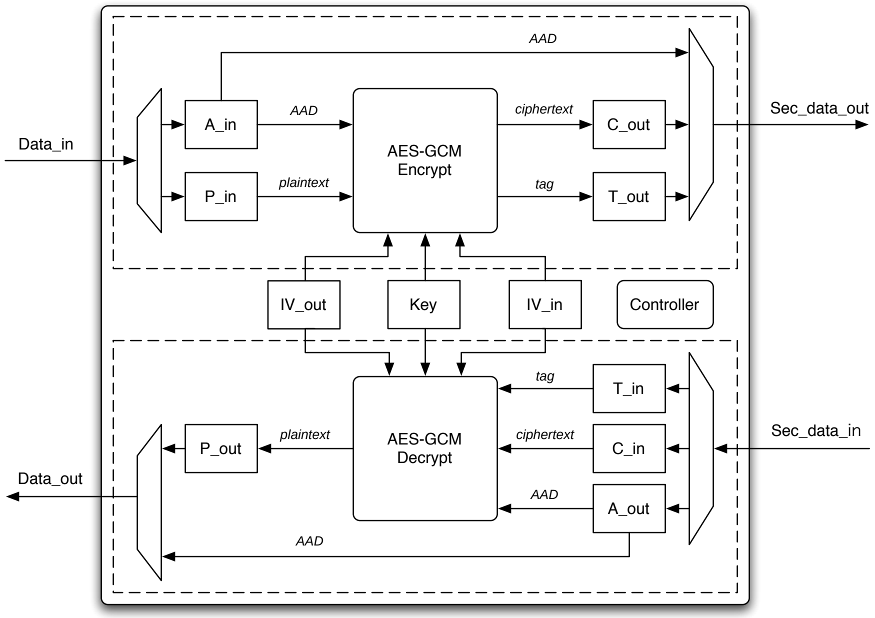

Fig. 3. The architectural design of our QSP. The upper region contributes to the QSP.Enqueue logic; the lower region contributes to the QSP.Dequeue logic. Control signals are omitted for clarity.

precomputation result. Splitting into two halves works as long as precomputation results are greater than tags in size, which is practical always the case (recall that tags are 96-bit long). Similarly, when being dequeued, the QSP dequeues the insecure queue twice and combine the two items into a single precomputation result. It is easy to see that a secure QSP remains secure with this splitting mechanism in place.

5 Solution Analysis

5.1 Security

Correctness of our QSP design is a straightforward consequence of AES-GCM's correctness. It is also trivial to see that the confidentiality of AES-GCM implies that our QSP design has confidentiality.

Our QSP design has integrity as well. We argue why below. Assume the contrary that our QSP has no integrity, then there exists an adversary whose capabilities are as described in Section 3.3 such that he, during an attack, was successful in causing the QSP to return a precomputation result D' at the i-th dequeue for some i, where D' is different from the precomputation result D given to the QSP during the i-th enqueue. If i is not unique, let i be the minimum value. Now since AES-GCM decryption did not return failure during that particular dequeue of QSP, the security of AES-GCM implies that $D = D'$, which contradicts to $D \neq D'$. Therefore our QSP has integrity.

5.2 Efficiency

Let n be the maximum number of precomputation results the insecure memory will store. Let the bit-lengths of the key K, the IVs IV_{in} and IV_{out}, the plaintext P, the AAD A and the tag T be ℓ_K, ℓ_{IV}, ℓ_P, ℓ_A and ℓ_T respectively. The bit-length of a precomputation result ℓ_D is thus $\ell_A + \ell_P$.

Space Complexity Our QSP requires a trusted storage of constant size independent of the size of the storage for precomputation results; it has a untrusted storage overhead of $\frac{\ell_T}{\ell_D}$. In case of DSA precomputation, $\ell_D = 320$ and the overhead is thus 30% (recall that we picked $\ell_T = 96$). Space overhead is generally not a problem as insecure memory is inexpensive. Moreover, the figure would be a lot smaller for many group signatures, as they can easily have precomputation results comprised of 10 or more 160-bit elements.

Time Complexity The latency incurred by our QSP during an enqueue operation is the time it takes to do one—or two, if precomputation results are split into halves as previously discussed—AES-GCM encryption. As discussed, we chose AES-GCM to implement the authenticated encryption because of its extremely low latency independent of ℓ_D, which is made possible by parallelization. The actual throughput and latency of AES-GCM operations depend on the hardware implementation. Some performance figures can be found in [19,23]. In [23], AES-GCM achieves 102 Gbps throughout with 979 Kgates using 0.13-μm CMOS standard cell library.

The latency incurred by our QSP during a dequeue operation can be argued similarly. However, we would like to highlight that enqueue latency is usually not a concern as this operation, like precomputation itself, is not on the critical path of the algorithmic execution. Therefore, one might not even care about speeding up the QSP's enqueue operation. For example, in case of a hardware implementation, one could save cost by using less or even no parallelization, at the expense of slower enqueuing.

On the other hand, since QSP dequeuing is usually on the critical path, dequeuing latency is a concern. To dequeue more quickly, we suggest slightly changing the QSP design to pre-fetch and pre-decrypt the next precomputation result stored in the insecure queue. (If splitting is employed, the next two results are pre-fetched and pre-decrypted instead.) This way, precomputation results become readily available to be dequeued at the QSP when the processor wants them. Hence, our QSP provides all its security guarantees with virtually zero timing overhead during dequeuing.

Table 1 compares the efficiency of our QSP with other approaches.

	Trusted Storage Size	Write Latency	Read Latency
Hardened RAM	$O(n)$	$O(1)$	$O(1)$
Oblivious RAM [14]	$O(\log n)$	$O(\sqrt{n}\log n)$, $O(\log^4 n)$	$O(\sqrt{n}\log n)$, $O(\log^4 n)$
AEGIS [26]	$O(1)$	$O(\log n)$	$O(\log n)$
AEGIS with prediction and caching [24,22,13]	$O(1)$	$O(\log n)$	$O(\log n)$ total, $O(1)$ non-hideable
Our QSP	$O(1)$	$O(1)$	$O(1)$

Table 1. Many existing approaches can be used to secure insecure memory for storing precomputation results, but their complexities grow with n, the number of precomputation results to be stored; our QSP requires constant trusted storage size, and incurs constant total latency when reading or writing precomputation results. (The $\log^4 n$ ORAM algorithm only becomes more efficient when $n > 2^{20}$.)

6 Discussion

A DSA signing coprocessor One can build a cost-effective low-latency DSA signature signing secure coprocessor by using our QSP to securely store DSA precomputation. Such a coprocessor can be used to secure the communications in critical infrastructures, especially those that impose stringent timing requirements on tolerable latency of message delivery such as some Supervisory Control And Data Acquisition (SCADA) systems in the power grid.

Generalizing the producer-consumer model We have assumed that the producer and the consumer of precomputation results are the same entity, namely the processor. Alternatively, they can be two separated entities such that the insecure queue through which precomputation results are piped is the only communication channel between them. This allows dynamic pairing between the producers and the consumers.

More interestingly, the number of producers and that of consumers can differ. For example, multiple consumers may be coupled with only a single producer trusted by the consumers. Consider a scenario where people carry electronic devices. For security reasons, each device signs DSA signatures on its outgoing messages when communicating with devices carried by other people and therefore requires a DSA signing engine. However, if a person has a single DSA precomputation module shared and trusted by all devices he carries, then those devices need only to do the post-computation. The hardware saving is huge since the circuitry for DSA postcomputation is a lot simpler than that for precomputation. Similar scenarios include communications among sensors installed in cars, among household electrical appliances, as well as among sensors and actuators in power substations.

MSPs for other data structures In this paper, we have focused on building a Memory Security Proxy for FIFO queues as they fit naturally for cryptographic precomputation. Some other works have looked at ways to secure RAM for general-purpose computing. MSPs for other data structures, e.g., stacks, sets and dictionaries, may also be useful. For instance, resource-limited devices such as smart cards and set-top boxes may offload implementations of data structures on to hostile platforms. As an example, Devanbu et al. [10] have proposed how to protect the integrity (but not confidentiality) of stacks and queues.

7 Conclusions

In this paper, we have motivated the need for a secure storage for cryptographic precomputation to provide sustainable benefits securely. Our solution to the challenge of how to construct such a storage is a small tamper-resistant module called the *Queue Security Proxy (QSP)*. We have demonstrated how our design can guarantee the necessary security despite hardware attacks. As our analysis has shown, our proposed design is very efficient.

In the future, we plan to prototype our QSP solution using FPGA to gain empirical figures on its performance (in terms of throughput and latency) and hardware costs (in terms of gate counts), and compare these figures with other approaches.

References

1. B. Adida and D. Wikström. Offline/Online Mixing. In *ICALP*, volume 4596 of *LNCS*, pages 484–495. Springer, 2007.
2. R. Anderson and M. Kuhn. Tamper Resistance—A Cautionary Note. In *Proceedings of the 2nd USENIX Workshop on Electronic Commerce*, pages 1–11, 1996.
3. M. Bellare and C. Namprempre. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. In *ASIACRYPT*, volume 1976 of *LNCS*, pages 531–545. Springer, 2000.
4. M. Bellare and P. Rogaway. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *Proceedings of the 1st ACM conference on Computer and communications security*, pages 62–73. ACM Press, 1993.
5. D. Boneh, X. Boyen, and H. Shacham. Short Group Signatures. In *CRYPTO*, volume 3152 of *LNCS*, pages 41–55. Springer, 2004.
6. E. F. Brickell, D. M. Gordon, K. S. McCurley, and D. B. Wilson. Fast Exponentiation with Precomputation (Extended Abstract). In *EUROCRYPT*, pages 200–207, 1992.
7. J. Camenisch and A. Lysyanskaya. An Efficient System for Non-transferable Anonymous Credentials with Optional Anonymity Revocation. In *EUROCRYPT*, volume 2045 of *LNCS*, pages 93–118. Springer, 2001.
8. D. Chaum. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Communications of the ACM*, 4(2), February 1981.
9. D. E. Clarke, S. Devadas, M. van Dijk, B. Gassend, and G. E. Suh. Incremental Multiset Hash Functions and Their Application to Memory Integrity Checking. In *ASIACRYPT*, volume 2894 of *LNCS*, pages 188–207. Springer, 2003.
10. P. T. Devanbu and S. G. Stubblebine. Stack and Queue Integrity on Hostile Platforms. *IEEE Trans. Software Eng.*, 28(1):100–108, 2002.

11. M. Dworkin. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC, June 2007.
12. J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart. Building the IBM 4758 Secure Coprocessor. *IEEE Computer*, 34(10):57–66, 2001.
13. B. Gassend, G. E. Suh, D. E. Clarke, M. van Dijk, and S. Devadas. Caches and Hash Trees for Efficient Memory Integrity. In *HPCA*, pages 295–306, 2003.
14. O. Goldreich and R. Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.
15. S. Goldwasser, S. Micali, and C. Rackoff. The Knowledge Complexity of Interactive Proof Systems. *SIAM J. Comput.*, 18(1):186–208, 1989.
16. J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.
17. B. Kauer. OSLO: Improving the Security of Trusted Computing. In *USENIX Security Symposium*, page 229237. USENIX, 2007.
18. D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *ASPLOS*, pages 168–177, 2000.
19. D. A. McGrew and J. Viega. The Security and Performance of the Galois/Counter Mode (GCM) of Operation. In *INDOCRYPT*, volume 3348 of *LNCS*, pages 343–355. Springer, 2004.
20. R. C. Merkle. Protocols for Public Key Cryptosystems. In *IEEE Symposium on Security and Privacy*, pages 122–134, 1980.
21. NIST. FIPS 186-2: Digital Signature Standard (DSS). Technical report, National Institute of Standards and Technology (NIST), 2000.
22. B. Rogers, Y. Solihin, and M. Prvulovic. Memory Predecryption: Hiding the Latency Overhead of Memory Encryption. *SIGARCH Computer Architecture News*, 33(1):27–33, 2005.
23. A. Satoh. High-Speed Parallel Hardware Architecture for Galois Counter Mode. In *ISCAS*, pages 1863–1866. IEEE, 2007.
24. W. Shi, H.-H. S. Lee, M. Ghosh, C. Lu, and A. Boldyreva. High Efficiency Counter Mode Security Architecture via Prediction and Precomputation. In *ISCA*, pages 14–24. IEEE Computer Society, 2005.
25. S. W. Smith and S. Weingart. Building a High-performance, Programmable Secure Coprocessor. *Computer Networks*, 31(8):831–860, 1999.
26. G. E. Suh, D. E. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *ICS*, pages 160–171. ACM, 2003.
27. G. E. Suh, D. E. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Efficient Memory Integrity Verification and Encryption for Secure Processors. In *MICRO*, pages 339–350. ACM/IEEE, 2003.
28. TPM Work Group. TCG TPM Specification Version 1.2 Revision 103. Technical report, Trusted Computing Group, 2007.
29. P. P. Tsang, M. H. Au, A. Kapadia, and S. W. Smith. Blacklistable Anonymous Credentials: Blocking Misbehaving Users without TTPs. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 72–81, New York, NY, USA, 2007. ACM.
30. J. Yang, Y. Zhang, and L. Gao. Fast Secure Processor for Inhibiting Software Piracy and Tampering. In *MICRO*, pages 351–360. ACM/IEEE, 2003.
31. S.-M. Yen, C.-S. Laih, and A. Lenstra. Multi-Exponentiation. In *IEE Proc. Computers and Digital Techniques*, volume 141, pages 325–326, 1994.