7-23-2007

# Secure Cryptographic Precomputation with Insecure Memory

Patrick P. Tsang
*Dartmouth College*

Sean W. Smith
*Dartmouth College*

# Secure Cryptographic Precomputation with Insecure Memory

Patrick P. Tsang and Sean W. Smith

Department of Computer Science
Dartmouth College
Hanover, NH 03755

Dartmouth Computer Science Technical Report TR2007-590

July 23, 2007

**Abstract**

Precomputation dramatically reduces the execution latency of many cryptographic algorithms. To sustain the reduced latency over time during which these algorithms are routinely invoked, however, a pool of precomputation results must be stored and be readily available. While precomputation is an old and well-known technique, how to securely and yet efficiently store these precomputation results has largely been ignored. For instance, requiring tamper-proof memory would be too expensive, if not unrealistic, for precomputation to be cost-effective.

In this paper, we propose an architecture that provides secure storage for cryptographic precomputation using only insecure memory, which may be eavesdropped or even tampered with. Specifically, we design a small tamper-resistant hardware module that we call the *Queue Security Proxy (QSP)*, which situates on the data-path between the processor and the insecure memory. Our analysis shows that our design is secure, efficient, flexible and yet inexpensive. In particular, our design's timing overhead and hardware cost are independent of the storage size.

We also discuss in this paper several interesting extensions to our proposed architecture. We plan to prototype our design assuming the scenario of precomputing DSA signatures, effectively building a cost-effective low-latency DSA signing secure coprocessor.

## 1 Introduction

precomputation is an optimization technique that reduces the execution latency of an algorithm by performing some of the algorithm's 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 one needs only to compute the remaining operations, or *post-computation* as we call it, to produce the output upon input arrival, execution latency is reduced.

**Cryptographic precomputation** In cryptography, precomputation is an old and well-known technique. For example, fixed-base modular exponentiation, an operation fundamental to almost all public-key cryptographic algorithms, can be sped up by precomputing a set of related exponentiations [BGMW92], such as using Shamir's trick [Gam85] or the sliding window method [YLL94]. As

another example, it has been well-observed that part of the generation of DSA signatures [NIS00], as is the case for ElGamal and Schnorr signatures, can be precomputed [BGMW92]. More generally, all signature schemes converted from a special type of three-move zero-knowledge proof of knowledge protocols [GMR89] known as $\Sigma$-protocols using Fiat-Shamir transformation [FS86, BR93] can benefit from precomputation quite significantly.[1] Example include many group signatures and anonymous credential systems such as [ACJT00, BBS04, CL01, TFS04]. Moreover, precomputing homomorphic encryption can speed up mix-nets [Cha81], as recently suggested in [AW07].

**Reusable v.s. consumable precomputation** precomputation can be *reusable*, i.e., a single precomputation can be reused across multiple algorithm executions, or *consumable*, i.e., a new precomputation is needed per execution. Speeding up modular exponentiation using the sliding window method 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, exponentiating a random element in DSA signature signing is a consumable precomputation—the element must be randomly drawn 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, for two reasons. First, confidentiality of the precomputed results is usually necessary in the former but not the latter. Second, the required storage space in the former grows with the expected rate of an algorithm being executed, whereas that in the latter is a constant. In this paper, we are going to overcome the bigger challenge—secure consumable precomputation—and refer to it as precomputation for simplicity's sake in the rest of the paper.

**Sustaining the low latency through buffering** Precomputation is capable of reducing latency of an algorithm execution only when a precomputed result is available upon an input's arrival. In situations when an algorithm is constantly being executed over time, computing and storing a single precomputation result would not sustain a low latency throughout the executions. One must therefore buffer precomputation results, i.e. precompute those results in advance and store in such a way that they are readily available when needed. So long as the pool is not empty, low latency can be sustained. In fact, buffering has long been in prevalent use in various fields such as networking, in which buffering helps accommodating the jitter and burstiness of traffic, and architecture, in which instructions and/or data are prefetched and cached to reduce latency.

**The need for a secure storage** The majority of cryptographic algorithms are designed and constructed without anticipating any precomputation, and thus have their security only proven assuming no precomputation. It is therefore of paramount importance, when applying precomputation to a cryptographic algorithm, to make sure that no security guarantee is violated. To this end, 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. The failing of such an assumption to hold could have devastating consequences. For example, in the case of precomputing DSA signatures, allowing an adversary to eavesdrop, overwrite, or replay only one precomputation result would be enough to leak the private signing key.

[1] In these schemes, most of the heavyweight operations, namely group exponentiation, can be precomputed.

**The challenge** Unfortunately, As the size of the storage of multiple precomputation results is significantly larger than the size of the internal states needed to be maintained for a single execution instance, it is unrealistic to assume that the whole storage would fit it a tamper-resistant module such as a hardened CPU. Tamper-proving the memory would also be too expensive and would put a upper limit on the size of the memory due to physical constraints. In this paper, we are going to overcome this challenge.

### 1.1 Our Contributions

We make the following contributions in this paper:

- We motivate that, in order to sustain the benefits of reducing latency of cryptographic algorithm execution brought by precomputation, one needs to maintain the storage of a pool of readily available precomputation results; and that the security of such storage is critical to the security of the cryptographic algorithm.
- We design an architecture that turns untrusted memory into a memory that is secure for storing precomputation results. The architecture is very efficient and has a cost independent of the size of the memory. Our analysis shows the architecture we propose is secure.
- We discuss prototyping our architecture assuming the scenario of precomputing DSA signatures, effectively resulting in an coprocessor architecture for low-latency DSA signature signing with high security assurance and yet low cost. We speculate performance figures.

The rest of this paper is organized as follows. In Section 2, we provide some background on hardware-based security and also review the cryptographic tools needed by our architecture. In Section 3, we describe our approach to overcome the challenge by giving an overview of our architectural design and explaining some of the design choices. In Section 4, we present our architectural construction in details, together with an analysis on its security and efficiency. In the same section, we also formalize the security requirements and the threat model under which these requirements must be met. In Section 5, we discuss several research directions that are worth exploring. We conclude the paper in Section 6.

## 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 physically tampering through means such as tamper-resistance, tamper-evidence, tamper-response. Deploying a hardware-based security mechanism could be very expensive, and different trade-offs between security and cost can lead to radically different paradigms.

**The IBM 4756/4764 approach** The IBM 4758 cryptographic coprocessor [SW99, DLP+01] is a secure crypto processor 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 the ATMs of banks, their high costs forbid most end-users benefit 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. Lie et al.'s eXecute-Only Memory (XOM) [LTM+00] architecture pioneered this line of research but is vulnerable to replay attacks. Suh et al. later proposed AEGIS [SCG+03a], which is immune to replay attacks and uses techniques such as Merkle-trees [Mer80] for better efficiency. The use of Counter mode for AES operations was proposed [SCG+03b, YZG03] to reduce memory-write latencies. Prediction techniques are used in [RSP05, SLG+05] hide the memory-read latencies. Other advances include using on-chip caches [GSC+03] and incremental multi-set hashes [CDvD+03] to reduce latency incurred by memory integrity checking.

The architecture we are going to propose in this paper falls under this category of hardware-based security as it has the same trust assumptions on both the processor and the memory. Our architecture deals only with precomputation rather than arbitrary software. 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 providing a secure execution environment for general-purpose computers such as x86 PCs, we gear our architecture towards 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) [TPM06]. In the last few years, major vendors of computer systems have been shipping machines that have included TPMs, with associated BIOS support. A TPM, when mounted on the motherboard of a commodity computing device such as a PC, provides internal storage space for storing, e.g., cryptographic keys, cryptographic functions for encryption/decryption, signing/verifying, as well as hardware-based random number generation. 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., [Ber07]) is starting to demonstrate some external hardware integration flaws; and the long history of low-cost physical attacks on low-cost devices (e.g., [AK96]) hasn't caught up with the TPM yet.

## 2.2 Cryptographic Tools

There are various symmetric and asymmetric cryptographic techniques that provide security notions such has confidentiality, authentication, integrity, etc. For example, the Advanced Encryption Standard [NIS01], or AES, is a block cipher that provides confidentiality of data, whereas HMAC [NIS02], is a keyed-hash message authentication that provides both message authentication and integrity. Here we review a fairly recent symmetric cryptographic tool called authenticated encryption, which effectively combines the functionality of AES and HMAC. As we will see, our architecture requires the use of it.

**Authenticated encryption** The *Advanced Encryption Standard*, or *AES*, specifies a block cipher that operates on blocks of 128-bit data. There are several modes of operation to choose from when encrypting messages of arbitrary length using AES, such as the Electronic CodeBook (ECB) mode , the Cipher-Block Chaining (CBC) mode 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.

The modes mentioned above provides data confidentiality but no data integrity. There are modes of operation, e.g. IACBC, IAPM, OCB, EAX, CWC, CCM and GCM, that provide both. When operating under these modes, AES effectively becomes authenticated encryption rather than just encryption. Among these modes, the CCM (Counter with CBC-MAC) mode and the GCM (Galois/Counter Mode) mode are particularly extractive because authentication is done without feedback of data-blocks. This means that both authentication and encryption can be parallelized to achieve extremely high throughput. We will use AES under GCM mode in our architecture.

**AES-GCM** AES-GCM has two operations, authenticated encryption AES-GCM.Enc and authenticated decryption AES-GCM.Dec. AES-GCM.Enc takes four inputs: (1) a secret key $K$, whose length is appropriate for the underlying AES; (2) an initialization vector $IV$ that can have any number of bits between 1 and $2^{64}$; (3) a plaintext $P$, which can have any number of bits between 0 and $2^{39} - 256$; and (4) an additional authenticated data (AAD) $A$, which can have any number of bits between 0 and $2^{64}$. AES-GCM.Enc outputs (1) a ciphertext $C$ whose length is the same as that of $P$ and (2) an authentication tag $T$, whose length can be any value between 64 and 128. AES-GCM.Dec takes five inputs: $K$, $IV$, $C$, $A$ and $T$. It outputs either $P$ or a special symbol `Failure` that indicates failure, i.e. the inputs are not authentic.

We refer the readers to [MV04b, MV04a] for the 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 using tamper-proof memory would not meet our design goal of cost-effectiveness, 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 both space- and computationally efficient: its size (e.g., in terms of gate-count) and the timing overhead it incurs should grow as slowly as possible with the size of the insecure memory it is securing, or even better, be constants independent of the size.

In our architecture, the MSP situates between the processor, which executes (the precomputation and post-computation of) the cryptographic algorithm, and the insecure memory. The MSP

effectively turns the insecure memory into a secure one by instrumenting the processor's read and write accesses. This most probably means that the MSP will incur timing overhead on these accesses, but it is otherwise transparent to the processor. In other words, 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 for 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 execute the cryptographic operations, possible among many other components, 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 design of such module is going to have a very small physical size, most likely just a small fraction of the size of the processor. Figure 1 depicts an architectural model in which the MSP is situated.

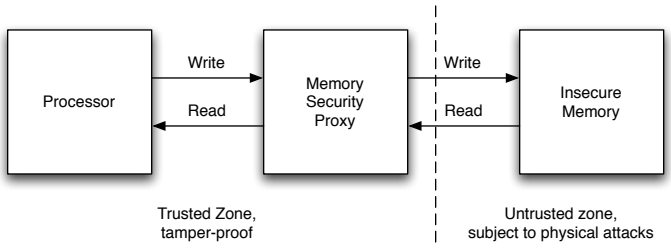


Figure 1: The architectural model

We remark that this 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 of them 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. As we will show next, we can exploit the properties of cryptographic precomputation to avoid some of the difficulties and thus achieve better results.

### 3.2 Data-Structure for precomputation Storage

The application of 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 (a.k.a. pipes), register arrays (a.k.a. Random Access Memory, or RAM) and etc., depending on the desired order of the goods being consumed (relative to them being produced).

We make the observation that using the precomputation results in the same order of their production always yields a correct execution of the cryptographic algorithms. In fact, in most cases the precomputation results are statistically uncorrelated to one another, and one may thus even use them in arbitrary order. Computing the commitment of a random value in the precomputation of DSA signatures is one example. Consequently, data-structures such as RAM and FIFO queues

and RAM are legitimate 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 two reasons.

- Securing insecure RAM efficiently has been proved 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. Our architecture requires only the "weaker" insecure FIFO queues.

From now on, we abbreviate FIFO queues as queues. Also, we call the Memory Security Proxy we are going to build, which is specialized for protecting queues, the *Queue Security Proxy (QSP)*. We end this section with some formalism of queues, which is necessary to formally 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. More rigorously, such a policy can be specified using the axiomatic approach of Larch [GH93], in which an object's sequential history is summarized by a value, which (informally speaking) reflects the object's state at the end of the history. These values are used in axioms giving the pre- and post-conditions on the objects operations. A queue implementation is said to be *correct* if the above 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. Note that correctness is defined assuming the absence of an adversary. In order to define the security of a queue implementation, we will need to decide on a threat model and specify how the implementation behaves under such a model. We leave this to the next section.

# 4 Our QSP Design

In this section, we first define the security requirements of the QSP and the threat model under which these requirements must be met. We then present our QSP design, first in the form of software pseudo-code to facilitate understanding and reasoning, then in the form of a hardware architectural design. We then analyze our architecture.

## 4.1 Security Model

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 1. Such an adversary may also probe the input-output relationship of the QSP as follows: an adversary may arbitrarily and adaptively ask the processor to enqueue or dequeue precomputation results and ask the processor to reveal a particular precomputation result. Note that in real world a precomputation result is never directly revealed to anybody but is rather used to influence the final outcome of the cryptographic algorithm. We give such a capability to the adversary so that we don't have to worry about the algorithm in question and whether it is secure or not. 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** Any QSP design must be correct: if the underlying queue 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. Nonetheless, part of the precomputation result can be made public without security breaches in certain scenarios when that part will eventually appear in the final algorithmic output, and its release a prior to the arrival of the input does not lead to security breaches. DSA signature signing precomputation results are of the form $(k^{-1}, r)$, knowing $r$, which will eventually be a part of the output signature, is fine, whereas knowing the whole result would leak to universal forgeability. Not having to encrypt the whole result may lead to better efficiency.

Now we define confidentiality. A precomputation result to be enqueued is called a data object, or simply data $D$, which is a pair $(A, P)$, where $A$ contains data that doesn't require confidentiality and $P$ is the plaintext to be encrypted by the QSP. A QSP design has confidentiality if no adversary, whose capabilities are as described above, can learn any information about $P$ in any $D$ that the adversary hasn't asked the processor to reveal.

**Integrity** Roughly speaking, a QSP with integrity is one that either behaves correctly upon access, or signals an error when having been tampered with. Note that this definition of integrity implies both data authenticity and data freshness of the queue the QSP is protecting. Specifically, if an adversary can modified a precomputation result, say the $i$-th one enqueued, without the QSP being able to detect it, 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, an adversary replaying 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 $i$ such that the precomputation results enqueued at the $i$-th differs from what is dequeued at the $i$-th time. A QSP design has integrity if no computationally bounded and physically present adversary can succeed with non-negligible probability.

### 4.2 The Construction

**The idea** The use of authenticated encryption makes our design of a QSP very simple. precomputation results (*Data_in*) generated by the processor are fed to the QSP in which the results are encrypted using AES-GCM. The initialization vector (*IV_out*) increments per encryption. This 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 is thus able to circumvent replay attacks. The output of the AES-GCM encryption (*Sec_data_out*) can then be enqueued into an insecure queue.

When the QSP is being dequeued, it in turns dequeue an entry from the insecure queue, and decrypt that entry using AES-GCM decryption. Again, the initialization vector (*IV_in*) increments per decryption. As long as the two IVs 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.

**System parameters** We provide some more details on various parameters. The key may remain constant throughout the lifetime of the QSP, or be picked uniformly at random from its space during boot-time. Note that using a new key effectively means all the old precomputation results are flushed. The two IVs are set to zero during start-up. We choose 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 use 80-bit IVs. Finally we select 96-bit as the size of the Tag.

**Software construction** Without loss of generality, we assume the underlying insecure queue provides an interface for querying whether it is full or not, and empty or not. Algorithms 1 and 2 show the software implementation of the enqueue and dequeue operations performed by the QSP respectively.

**Algorithm 1** QSP.Enqueue($Data_{in}$)

**Private Input:** $K, IV_{out}$

1: **if** $Q$.isFull() **then**
2: **return** `Error`
3: **end if**
4: $\langle P_{in}, A_{in} \rangle := Data_{in}$
5: $\langle C_{out}, T_{out} \rangle \leftarrow$ AES-GCM.Enc$(K, IV_{out}, P_{in}, A_{in})$
6: $IV_{out} \leftarrow IV_{out} + 1$
7: $Sec\_Data_{out} := \langle A_{in}, C_{out}, T_{out} \rangle$
8: $Q$.enqueue$(Sec_Data_{out})$

**Algorithm 2** QSP.Dequeue()

**Private Input:** $K, IV_{in}$

1: **if** $Q$.isEmpty() **then**
2: **return** `Error`
3: **end if**
4: $Sec\_Data_{in} \leftarrow Q$.dequeue()
5: $\langle A_{out}, C_{in}, T_{in} \rangle := Sec\_Data_{in}$
6: **res** $\leftarrow$ AES-GCM.Dec$(K, IV_{in}, C_{in}, A_{out}, T_{in})$
7: **if res**=`failure` **then**
8: **return** `Error`
9: **else**
10: $P_{out} :=$ **res**
11: $Data_{out} := \langle P_{out}, A_{out} \rangle$
12: **return** $Data_{out}$
13: **end if**

**Hardware construction** Figure 2 shows the architectural design of the QSP, with the control signals omitted for clarity. $Data\_in$ and $Data_{out}$ are the data-bus connected to the processor, $Sec\_Data_{in}$ and $Sec\_Data_{out}$ are the data-bus connected to the insecure queue. We have assumed that the underlying hardware queue memory is asynchronous, so that the enqueue and dequeue

operations can be done asynchronously. It is straightforward to modify the design if synchronous queue is used instead.

The fact that the authenticated encryption of a precomputation result is bigger in size than the result itself has the following implications. One may assume a wider data-bus for *Sec_Data* (both in and out) than that for *Data* (both in and out). However, this might not be favorable if the transparency of the QSP is critical, such as in situations where the architecture is difficult to change and adding the QSP as a "mod-chip" is the only viable solution. Another possibility is to assume that precomputation results do not use up the whole bus-width and that they still fit in the bus after the expansion due to authenticated encryption. This may require changes to the software that does the cryptographic algorithm, which could be too inconvenient to be feasible in some cases. Our recommended approach is to have the QSP split up every incoming precomputation result into two halves[2] and operate on each half as if it was a single incoming precomputation result. Similarly, when being dequeued, the QSP would in turn dequeue the insecure queue twice and combine them into a single answer. It is easy to easy that if the QSP is secure without this splitting mechanism, then the QSP remains secure with such a mechanism in place.

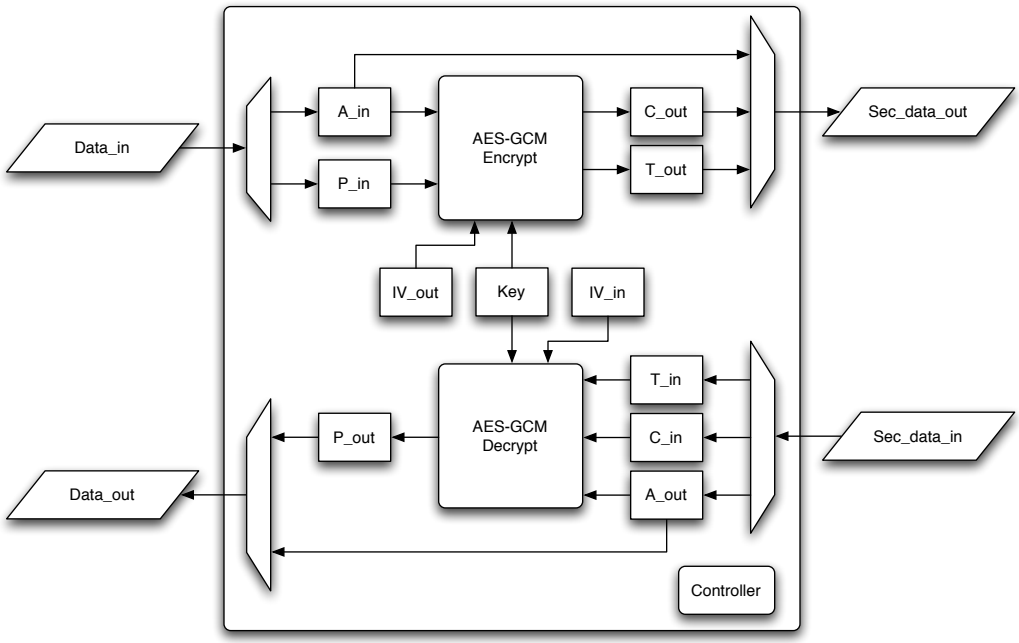


Figure 2: The architectural design of our QSP

### 4.3 Analysis

**Security** Correctness of our QSP design is a straightforward consequence of the correctness of the internal AES-GCM. Similarly, it is trivial to see that as long as the internal AES-GCM is secure in terms of confidentiality, our QSP design has confidentiality. Our QSP design has also integrity

[2]Splitting into halves works as long as a precomputation result has its bit-length greater than the bit-length of the tag, which is virtually always the case given that we picked the bit-length of the tag to be 96.

because of the following. Assume the contrary that our QSP has no integrity, then there exists an adversary whose capabilities are as described in Section 4.1 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.

**Efficiency** Let $n$ be the maximum number of precomputation results the insecure memory is expected to store. Let the bit-length of the key $K$, the IVs $IV_{in}$ and $IV_{out}$, the plaintext $P$, the additional authenticated data $A$ and the tag $T$ be $\ell_K$, $\ell_{IV}$, $\ell_P$, $\ell_A$ and $\ell_T$ respectively. Thus the bit-length of a precomputation result $\ell_D$ is $\ell_A + \ell_P$. Note that the length of the ciphertext $C$ equals that of $P$.

Now our QSP has a storage overhead of $\frac{\ell_T}{\ell_D}$. Recall that we picked $\ell_T = 96$ and this is independent of $\ell_D$. In case of DSA precomputation, $\ell_D = 320$ and thus the overhead is 30%. Space overhead is generally not a problem as insecure memory is cheap. Moreover, the figure would be a lot smaller for many group signatures, as they can easily have precomputation results comprised of 10 or so 160-bit elements.

The latency incurred by the QSP during an enqueue operation is precisely the time AES-GCM encryption takes to encrypt $\ell_D$. As discussed, we chose AES-GCM as the authenticated encryption mechanism because of its extremely low latency made possible by parallelizing the encryption and authentication so that the latency is independent of $\ell_D$. Of course, this implies to enjoy this minimal latency the gate-count of the QSP grows linearly with $\ell_D$. But again as the processor is trusted to be enable to store more or less the same amount of data anyways, this is not an issue. The actual speed and latency attained by the AES-GCM encryption and decryption depends on its implementation and the hardware it is running on. Some performance figures can be found in [MV04a, Sat07]. For example, Satoh [Sat07] has an implementation that achieves 102 Gbps throughout with 979 Kgates using 0.13-$\mu$m CMOS standard cell library.

The latency incurred by the QSP during a dequeue operation can be argued similarly. However we highlight one point in the following. Enqueue latency is usually not a concern as the operation is not on the critical path of the algorithmic execution, just like the precomputation itself. Therefore, one might not even care about speeding up QSP's enqueue operation. For example, in case of a hardware implementation, one could save cost by using less parallelization, at the expense of slower enqueuing. Nonetheless, this is not the case for the dequeue operation as it is indeed on the critical path. In the pursuit of faster dequeuing, we suggest a slight change to QSP's architecture to pre-fetch and pre-decrypt the next precomputation result stored in the insecure queue. This way, an extra of $\ell_D$-bit trusted register in the QSP is needed but precomputation results become readily available to be dequeued at the QSP when the processor wants them. Hence, our QSP provides all its security guarantees at no cost in terms of dequeue latency.

# 5 Discussion

**MSPs for other data-structures** In this paper, we focus on building a Memory Security Proxy for FIFO queues as they fit naturally for cryptographic precomputation. Plenty of work by others have looked at ways of securing RAM for general-purpose computing. It would be interesting to

build MSPs for other data-structures such as stacks, priority queues, sets, dictionaries, and etc, and find out some potential applications. For instance, Devanbu et al. suggested in [DS02] that resource-limited devices such as smart-cards and set-top boxes may offload implementations of data-structures on to hostile platforms. The authors actually proposed ways to secure stacks and queues, but only limited to protecting their integrity (i.e., no confidentiality).

**Generalizing the producer-consumer model** In our architecture, we assume that the producer and the consumer are the same entity. Alternatively, they can be two separated entities such that there is no communication channel between them except the insecure queue through which precomputation results are piped. The ability of allowing dynamic pairing between the producers and the consumers may be useful.

More interestingly, there can be an asymmetry between the number of producers and consumers, e.g. multiple consumers are coupled with only one producer, who is trusted by the consumers. Let us elaborate on this by considering a pervasive computing scenario where entities are equipped with small devices exchanging information with devices possessed by other entities, such as people carrying electronic gadgets, cars installed with sensors and electrical appliances within households. These devices sign DSA signatures on their outgoing messages for security reasons and therefore each of them requires a DSA signing engine. However, if the person (or the car, or the house) has a single trusted DSA precomputation module, then all the devices need only to do the post-computation. Since the circuitry for doing DSA precomputation is more complicated than that for doing post-computation, every device saves more than 50% in terms hardware costs needed for signing.

**A low-cost DSA signing secure coprocessor** As our next step, we plan to implement our QSP assuming the scenario of providing secure storage for DSA precomputation, effectively building a cost-effective low-latency DSA signature signing secure coprocessor. We expect such a coprocessor can benefit the information security of communication in critical infrastructures, especially those that impose stringent timing requirements on tolerable latency of message delivery such as the power grid.

## 6 Conclusions

In this paper, we have motivated the need for a secure storage in order for cryptographic precomputation to provide sustainable benefits. Our solution to the challenge is a small tamper-resistant module called Queue Security Proxy. We have formalized a threat model for cryptographic precomputation and demonstrated how the QSP can guarantee the necessary security despite hardware attacks. As our analysis has shown, our proposed design of the QSP is very efficient. This work has also inspired several interesting directions for further research.

## Acknowledgements

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.

## References

[ACJT00] Giuseppe Ateniese, Jan Camenisch, Marc Joye, and Gene Tsudik. A practical and provably secure coalition-resistant group signature scheme. In *CRYPTO*, volume 1880 of *LNCS*, pages 255–270. Springer, 2000.

[AK96] R. Anderson and M. Kuhn. Tamper Resistance—A Cautionary Note. In *Proceedings of the 2nd USENIX Workshop on Electronic Commerce*, pages 1–11, 1996.

[AW07] Ben Adida and Douglas Wikstrm. Offline/online mixing. Cryptology ePrint Archive, Report 2007/143, 2007. http://eprint.iacr.org/.

[BBS04] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In *CRYPTO*, volume 3152 of *LNCS*, pages 41–55. Springer, 2004.

[Ber07] Bernhard Kauer. OSLO: Improving the security of Trusted Computing. Technical report, Technische Universitat Dresden, Department of Computer Science, 2007.

[BGMW92] Ernest F. Brickell, Daniel M. Gordon, Kevin S. McCurley, and David Bruce Wilson. Fast exponentiation with precomputation (extended abstract). In *EUROCRYPT*, pages 200–207, 1992.

[BR93] Mihir Bellare and Phillip 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.

[CDvD+03] Dwaine E. Clarke, Srinivas Devadas, Marten van Dijk, Blaise Gassend, and G. Edward Suh. Incremental multiset hash functions and their application to memory integrity checking. In *ASIACRYPT*, volume 2894 of *Lecture Notes in Computer Science*, pages 188–207. Springer, 2003.

[Cha81] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 4(2), February 1981.

[CL01] Jan Camenisch and Anna Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In *EUROCRYPT*, volume 2045 of *LNCS*, pages 93–118. Springer, 2001.

[DLP+01] Joan G. Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer, Leendert van Doorn, Sean W. Smith, and Steve Weingart. Building the IBM 4758 secure coprocessor. *IEEE Computer*, 34(10):57–66, 2001.

[DS02] Premkumar T. Devanbu and Stuart G. Stubblebine. Stack and queue integrity on hostile platforms. *IEEE Trans. Software Eng.*, 28(1):100–108, 2002.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1986.

[Gam85] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[GH93] John V. Guttag and James J. Horning. *Larch: languages and tools for formal specification*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[GMR89] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208, 1989.

[GSC+03] Blaise Gassend, G. Edward Suh, Dwaine E. Clarke, Marten van Dijk, and Srinivas Devadas. Caches and hash trees for efficient memory integrity. In *HPCA*, pages 295–306, 2003.

[LTM+00] David Lie, Chandramohan A. Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John C. Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *ASPLOS*, pages 168–177, 2000.

[Mer80] Ralph C. Merkle. Protocols for public key cryptosystems. In *IEEE Symposium on Security and Privacy*, pages 122–134, 1980.

[MV04a] David A. McGrew and John Viega. The security and performance of the galois/counter mode (gcm) of operation. In *INDOCRYPT*, volume 3348 of *Lecture Notes in Computer Science*, pages 343–355. Springer, 2004.

[MV04b] David A. McGrew and John Viega. The Galois/Counter Mode of Operation (GCM), January 2004. Updated submission to NIST Modes of Operation Process.

[NIS00] NIST. FIPS 186-2: Digital signature standard (DSS). Technical report, National Institute of Standards and Technology (NIST), 2000. http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf.

[NIS01] NIST. FIPS 197: Announcing the advanced encryption standard (AES). Technical report, National Institute of Standards and Technology (NIST), 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[NIS02] NIST. FIPS 198: The keyed-hash message authentication code (HMAC). Technical report, National Institute of Standards and Technology (NIST), 2002. http://csrc.nist.gov/publications/fips/fips198/fips-198a.pdf.

[RSP05] Brian Rogers, Yan Solihin, and Milos Prvulovic. Memory predecryption: hiding the latency overhead of memory encryption. *SIGARCH Computer Architecture News*, 33(1):27–33, 2005.

[Sat07] Akashi Satoh. High-speed parallel hardware architecture for galois counter mode. In *ISCAS*, pages 1863–1866. IEEE, 2007.

[SCG+03a] G. Edward Suh, Dwaine E. Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *ICS*, pages 160–171. ACM, 2003.

[SCG+03b] G. Edward Suh, Dwaine E. Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Efficient memory integrity verification and encryption for secure processors. In *MICRO*, pages 339–350. ACM/IEEE, 2003.

[SLG+05] Weidong Shi, Hsien-Hsin S. Lee, Mrinmoy Ghosh, Chenghuai Lu, and Alexandra Boldyreva. High efficiency counter mode security architecture via prediction and precomputation. In *ISCA*, pages 14–24. IEEE Computer Society, 2005.

[SW99] Sean W. Smith and Steve Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(8):831–860, 1999.

[TFS04] Isamu Teranishi, Jun Furukawa, and Kazue Sako. k-times anonymous authentication (extended abstract). In *ASIACRYPT*, volume 3329 of *LNCS*, pages 308–322. Springer, 2004.

[TPM06] TPM Work Group. TCG TPM Specification Version 1.2 Revision 94. Technical report, Trusted Computing Group, 2006.

[YLL94] S.-M. Yen, C.-S. Laih, and A.K. Lenstra. Multi-exponentiation. In *IEE Proc. Computers and Digital Techniques*, volume 141, pages 325–326, 1994.

[YZG03] Jun Yang, Youtao Zhang, and Lan Gao. Fast secure processor for inhibiting software piracy and tampering. In *MICRO*, pages 351–360. ACM/IEEE, 2003.