# Architecture Considerations for Massively Parallel Hardware Security Platform

## Building a Workhorse for Cryptography as a Service

Dan Cvrček[1] and Petr Švenda[2]
dan@enigmabridge.com svenda@fi.muni.cz

[1] Enigma Bridge, Cambridge, Great Britain
[2] Masaryk University, Faculty of Informatics, Czech Republic

**Abstract.** Cryptography as a service (CaaS) provides means for executing sensitive cryptographic operations when the primary computing platform does not offer the required level of trust and security. Instead of executing operations like document signing directly by an application running in untrusted environment, the operation keys are only present in trusted environment used by CaaS. Once the operation keys are put in place, the applications use a CaaS interface to obtain results of sensitive operations - document signatures - executed by CaaS. A typical scenario is the use of virtual computing platform in the cloud. Use of CaaS reduces impact of the potential compromise of this virtual platform and simplifies subsequent recovery. The attacker will not learn the value of sensitive keys (e.g., signing keys) and is only able to use the keys for a limited time. The CaaS is enabling technology for a large number of use cases where security is important. The concept of scalable and universally available CaaS has also far-reaching usability, security, legal, and economics consequences of cloud use. In this position paper, we focus on requirements for building a CaaS platform – what are the options and challenges to build hardware and software components for CaaS suitable for usage scenarios with different load patterns and user requirements. We propose a suitable architecture for CaaS that can be shared by a large number of concurrent users, i.e., providing access to a large number of cryptographic keys. We also provide practical results from our prototype implementation[3].

## 1 Introduction

There is a strong demand for a secure cryptographic platform for the cloud and mobile computing to support a variety of sensitive applications. When used in large scale distributed environments, one of the options is to provide cryptographic operations as a service (CaaS) instead of implementing sensitive computations on end-user device. There are several advantages of this approach,

[3] Full details and paper's supplementary material can be found at http://crcs.cz/papers/space2015.

as recognized in [17], particularly, end-user device might be more vulnerable to compromise or lack of entropy source for key generation.

When CaaS is discussed in research literature, performance considerations are often omitted or neglected. One of such assumptions is that when a CaaS provider is fully trusted by users, it can have unlimited access to cryptographic keys. This assumption allows the provider easy scaling of computation power for cryptographic operations, as there are no security constraints. But the performance becomes quickly an issue when the CaaS provider is untrusted - it can execute cryptographic operations, but it cannot access keys directly and it becomes subject of constraints introduced by the API providing access to keys. Execution of sensitive operations is in this case provided by a specialized trusted hardware module (HSM) that ensures the cryptographic material cannot be accessed directly. Current HSMs provide reasonable computational performance for closed, centralized systems (high-end HSMs can perform up to 9,000 RSA 1024b signatures per second) under certain conditions. But implementing scalable CaaS supporting a range of operations and concurrent use of a large number of keys of CaaS users poses a number of challenges.

So far, high-performance cryptographic hardware platform, providing high level of shareability was not discussed in details in research literature. In this work, we summarize existing challenges and introduce open research questions for alternative architectures capable to host a large number of applications, cryptographic material and concurrent users. Our paper provides considerations and proposes a suitable architecture for CaaS supporting many users and many key scenarios, architecture based on secure processors with protected between processors. The experience obtained from building such a platform will also be discussed.

The paper is organized as follows: The next section provides a short introduction to cryptography as a service (CaaS) and defines main usage scenarios with related requirements on the high-performance cloud-based CaaS platform. Section 3 reviews different hardware options available and challenges present to provide high-performance trustworthy computation CaaS platform. The proposed architecture using high number of parallel secure processors connected with secure channels is described. Section 4 presents case-study of HMAC-based one-time password provided via proposed architecture together with practical results from prototype build. Possible future directions are summarized in Section 5 with conclusions given in Section 6.

## 2 Cryptography as a Service (CaaS)

Information systems face many security threats. Some of them are almost universal and all systems and business applications have to deal with them, some are specific. Every designer has to assess risks of their existing or new application and consider methods to mitigate those risks.

Running applications in the cloud introduces a number of universal threats that one does not have to think about while he/she runs their applications

from own servers. All those new threats are related to the fact that the cloud introduces new entities into the system model of cloud applications - the cloud provider.

Threat modelling is is a subject on its own (refer to [6] for foundations of cloud security) but we need to introduce some initial assumptions that we use below for reasoning about security of CaaS and definitions of security levels. Our initial classification is based on system components.
The list of components is as follows:

*Application* – the software application providing beneficial functionality for Client.
*Application Owner* – an entity that develops the application itself and is responsible for its correct operation. In many cases the application owner will be the client of the application as well.
*Client* – user of the application; when client is different from application owner the client would expect the application to provide certain business functionality. Security aspects may be still an issue though as secure processing and storage of data is hard to verify through the business functionality provided by the application.
*CaaS Provider* – an entity that provides functionality of CaaS including support and management of CaaS, e.g., system updates. While CaaS provider would be typically independent of Cloud Provider but it may be Application Owner.
*Cloud Provider* – the entity that controls the physical platform on which the application runs. The platform has several layers of components with each layer potentially provided a different entity.
*Internet* – communication between entities of the system (e.g., via web services API).

### 2.1 Levels of trust and security

We are able to define the following levels of trust for CaaS – using a system architecture with components defined in the previous section. Let us first assume different levels of trust in CaaS.

- *Client Trust* – users may or may not trust CaaS provider directly. The trust, if it exists, may be complete or based on an assumption of split control between CaaS provider and Application Owner – an assumption that data of Client can only be compromised if both parties cooperate. Based on our empirical experience, CaaS should be trusted more than Application Owner as CaaS would provide security as its main business and as such have more expertise needed for implementing security measures correctly.
- *Application Owner Trust* – if Application Owner trusts CaaS, it can use its relationship with Clients to leverage own trust in CaaS for persuading Clients that the use of CaaS increases the security of their data.
- *Cloud Provider Trust* – from practical point of view it is irrelevant whether Cloud Provider trusts or distrusts CaaS. It has, however, means to disable access to CaaS from applications using its platform.

Use of CaaS can be either enforced by compliance requirements or by concerns of Clients. The former will require Application Owner Trust that may be based on external validations of CaaS. Such validations would have to be sufficient for compliance. Concerns of Clients may either prompt Application Owner to use CaaS or find a way to use CaaS on top of the application.
In terms of dataflows, Client may trust parts of dataflows involving its data:

- *Client* – computers and/or networks under the control of Client.
- *CaaS* – systems that implement cryptographic functions for an application must be trusted by Client and usually by Application Owner as well.
- *Application Owner* – trust in systems of Application Owner would be limited – either by Clients or even by Application Owner itself.
- *Internet* – it is generally untrusted; it is possible to relax requirements on data protection only if the data itself are not confidential or Client's security requirements allow for some security properties to be ignored.

The trust balance between Client systems and CaaS is important in terms of the CaaS setup for Application. In general, we can assume that CaaS will provide full life-cycle support for applications but Client may decide not to fully trust CaaS and keep some aspects of cryptography under its own control.

If Client trusts its computers and/or information systems, it can use it for enrolment or other bootstrapping operations that are otherwise manageable for it information systems. A generation of application keys may be one of such examples.

### 2.2 Usage scenarios

The typical usage scenario influences significantly properties required for a CaaS platform as well as imposing restrictions and limitations of the hardware/software architecture behind the CaaS platform. We will discuss possible usage scenarios with respect to a number of parallel users and a number of distinct cryptographic keys used by every user. Note that other classifications are possible, e.g., w.r.t. the amount of transmitted data (short packets vs. long data streams), number of messages to finalize a single logical operation (e.g., decryption of single packet vs. multi-packet challenge-response protocol) or list of required cryptographic algorithms to name a few. We choose a number of users, and distinct keys because these factors are the most specific for situation where a CaaS platform is significantly shared between a number of different entities – a typical "cloud" scenario that already proved its viability for general purpose computing (but we will not limit our description only to such scenario).

Note that in the following classification, we will talk about a *service* rather than a CaaS platform, as some categories would not classify as CaaS as commonly defined, but make sense to list them because of distinct features and security/performance considerations introduced.

We also use count quantifiers 1, few (M) and many (N) to describe concept categories, e.g., *many users* or *a few keys per user* as different hardware devices

used to facilitate service are capable to store and handle different numbers of cryptographic contexts of a particular type (e.g., AES or RSA). When more contexts than a device is able to hold internally are required, contexts must be offloaded from device when not used and load in again later, introducing potential delay and additional requirements like the need for out-of-device secure storage or key wrapping. A provider of a service can utilize more hardware devices to linearly increase the number of contexts that can be maintained at the same time. When we use *many* keyword, its strictly more than number of contexts that can be fit into available hardware device(s).

**S1: One user, few keys (1:M)** – no sharing of the target service, as only a single user with a single key (or very few keys) is using it. Because of exclusive use and a small number of keys, there is no need to switch cryptographic contexts (pre-scheduled keys, initialized cryptographic engines...) before serving a new request and the whole cryptographic context can reside directly inside a service computational device(s). There is little need for CaaS to provide scalability, while secure remote access, use in virtualized environments and suitable API (e.g., application oriented rather than low-level PKCS#11) requirements remain. Example: a payment card physically owned by the user with a payment authentication key or an HTTPS TLS accelerator with one private key.

**S2: One user, many keys (1:N)** - this use case does not require service sharing, but it does imply frequent changes of cryptographic contexts because of a high number of keys involved. If the number of used keys is significantly higher than the number of contexts that can fit into the underlying hardware, then cryptographic contexts may need to be changed even with every request. Context loading, cryptographic engine initialization and key scheduling may significantly contribute to the overall time required to complete a requested operation. Subsequent performance degradation with the factor of 2-5 is well known from benchmarks for encryption throughput measuring the effects of varying message length (bulk encryption vs. small messages). The performance degradation is even more severe in the case of secure switching of cryptographic contexts between the secure processor and untrusted memory. Use considerations of CaaS are similar to the 1:M scenario. However, as the higher number of keys amplifies generic key management issues, CaaS may offer better overall usability. Example: PIN verification procedure performed inside an HSM on behalf of card issuing bank.

**S3: Few users, few keys (M:M)** – limited sharing of the target service while every user uses only a few keys. This is the first use case where service's computational devices are shared by mutually distrusting users. An additional overhead is introduced due to a need to securely erase sensitive values before the context switch between users may occur. As only few keys exist in the system overall, there is no need to offload cryptographic contexts. Example: Amazon CloudHSM [4] where a small number of users (e.g., 16) is sharing same physical hardware device performing cryptographic operations.

**S4: Few users, many keys (M:N)** - an extension of *S3: Few users, few keys (M:M)* scenario with a need to perform cryptographic context offload due to a high number of used keys. As only a few users are present, secure offload can be done relatively quickly as wrapping and unwrapping engines on service devices can be left initialized and ready to process next request at all times.

**S5: Many users, many keys (N:N)** – CaaS service serves many users, each of them with few keys, resulting in many keys in total. High-level of sharing of hardware resources of service. Includes also scenarios where primary keys from many users are used to derive and use new (session) keys based on a user input. Example: TLS accelerator with different session keys established for every different user after an initial TLS handshake with the server's private key. Note that number of keys can be further amplified if TLS accelerator is shared as CaaS service between multiple web servers with different private keys. See Section 4 for details of another example providing HMAC-based one-time password verification.

### 2.3 Typical operations needed for CaaS

Although many different algorithms and protocols can be implemented and provided by CaaS, we can identify a short list of common generic operations:

1. **Generate/derive new key** – new key $K$ (symmetric, asymmetric or other secret) is generated by CaaS service. The key then either never leaves the CaaS service (analogy with on-card non-exportable private key for digital signatures) or alternatively can be exported back to the Client (e.g., in encrypted blob).
2. **Import new key** – Client provides a key $K$ to be imported and later used by CaaS. Transfer of a key $K$ can be protected for confidentiality, integrity and freshness.
3. **Process input data** – Client provides input data $M$, processed inside a CaaS service by a key $K$ and cryptographic algorithm $F$, where $C = F(M, K)$. Input data $M$ and output data $C$ (returned to a Client) may be protected for confidentiality, integrity and freshness.
4. **Obtain usage statistics** – how many times was a particular key $K$ used? Requires authorization of process input data requests and protection of usage data.
5. **Remove key from service** – when Client doesn't need to use the key $K$ any more, key is removed. Key removal might be on a Client request, automatic (time-limited exposure) or as a result of compliance requirements (e.g., reset the device at least once every 24 hours if it contains Client's keys)

### 2.4 Preferred properties of cloud-based high-performance CaaS

There is no single unified CaaS architecture which would ideally fit all scenarios described in Section 2.2. In this paper, we focus on scenario *S5: Many users,*

*many keys* as we believe this scenario is difficult to support with current technology on a sufficient level of security. We believe that the most important principles for a secure and scalable CaaS platform are as follows:

**P1: Untrusted CaaS provider** for handling of cryptographic secrets – if the provider of CaaS doesn't need to be trusted to preserve secrecy of cryptographic material, the attack surface is significantly reduced (highlight for the Client). Provider itself will not be subject of internal and external attacks due to its low impact on CaaS security (highlight for provider). Some system designs require to trust only provider with physical access to CaaS as they mitigate threats of corrupted operator with only logical access [5]. If operator can't compromise security of CaaS and its cryptographic material, we can achieve higher level of overall security. Note that this principle is also beneficial for CaaS provider as it decrease its attractivity as a target for compromise.

**P2: Easy to use API** – because platform will be used as a service, a well-defined and simple interface is vital for fast adoption. Care should be taken to provide API not only easy to integrate, but also easy to use securely [8].

**P3: Secure import of cryptographic material** – secure way to import initial cryptographic material is required in majority of use case scenarios. Even when a key is generated directly inside the CaaS service as a result of Client request, additional shared keys are usually required to authenticate subsequent process input requests.

**P4: Low latency of responses** in the presence of many requests from many parallel user – as a platform will be significantly shared, low latency should not deteriorate even when many parallel requests are served. The tolerable latency range is specific to the particular usage scenario and in turn affects limits on the sharing of a given platform.

**P5: High performance in the presence of frequent key change** – significant level of sharing between many users, everyone with potentially distinct cryptographic keys introduces a high number of expected key scheduling before request can be processed. In an extreme (but not uncommon) scenario, every request may cause initialization and key scheduling of several cryptographic engines. Overall platform performance is expected to decrease with more users/requests scale reasonably.

**P6: Authentication of input/output requests** – once cryptographic material is (securely) imported into a CaaS platform, actual use of the imported key should be authorized by the Client and performed only on data provided by authorized Client. Verification of authorization itself should not significantly impact platform performance. Usually achieved by requests authorization by separate shared request authorization keys (commonly called "API keys"). An output data provided by service back to the Client should be authenticated as well to provide strong assurance that Client's original request was really processed by the imported key.

**P7: Confidentiality of input and output data** – if sensitive data are transmitted as part of request and corresponding response, confidentiality should be protected (again, "API keys" can be used).

**P8: Easy recovery from client-side compromise** – as Client can be compromised with the assumption that the key imported to CaaS was not, procedure to recover from a compromise should be easy to perform (e.g., fresh re-installation of client environment and transparent change of request authorization keys with perfect forward secrecy property). Eventually, frequent automatic recovery process can be executed as a preventive measure for undetected compromises.

**P9: Robust audit trail of key usages** – because CaaS is offered as a service, pay-per-use model may be utilized and Client should be provided with robust audit trail how often imported key was used. Another important reason for audit trails steams from potential compromise of client software together with authorization keys for requests. An attacker can then use an imported key without a user's consent. Once the compromise is detected, the user might be interested in realizing an exact extend of service usage during the compromise period.

**P10: Limit on maximum key usages** (before re-authorization) – once a key is imported and request authorization keys are compromised, an attacker can issue a large number of requests unless limited by another factor. To limit an extend of expected malicious usages of imported key, a Client can import key together with a number of "credits" limiting the maximum number of requests which can be served by service. Again, the provider of the service should not be able to manipulate with credits already used.

**P11: Tolerance to occasional hardware/software failures** – large level of sharing and high number of requests will inevitably result in occasional failures of the platform components, which should not impact other parallel users significantly. Natural requirement, but might be harder to achieve, if CaaS provider is not trusted and thus cannot inspect the full results of operation for errors itself. Also, move of Client request from failed to functioning device is more difficult if relevant contexts are cryptography bound to a single device.

## 3 Building hardware for CaaS back-end

In this section, we will discuss various options for building hardware platform which will satisfy principles described in Section 2.4. Different architectures are discussed both from performance and security perspective with the focus on *S5: Many users, many keys* scenario.

### 3.1 Designing CaaS

There are many ways how to build computational platform for CaaS. The following list shows some of the more obvious options:

1. *Use of general-purpose hardware*, e.g., high-performance multi-core server processor and implementation of the required cryptographic functionality in software. The advantage is fast development and deployment with existing cryptographic libraries like OpenSSL [2] or cryptlib++ [1] and medium expected performance. The main disadvantage is need to trust CaaS provider

as all cryptographic secrets and input/output data are easily accessible inside the CaaS implementation. Note that the level of trust to provider with logical-only access can be limited by a combination of virtualization and trusted computing [5]. In this particular case, a modified Xen hypervisor is used to make standard TPM available for secret-less virtual machine resulting in significant decrease in the size of trusted computational base (TCB).

2. *Use of generic programable hardware* (e.g., Field-programmable gate array (FPGA) or Graphics processing unit (GPU)) with cryptographic operations accelerated by programmable hardware with advantage of higher performance. The disadvantage is increased difficulty of implementation and deployment due to lower number of readily available cryptographic implementations. Note that GPU architectures like nVidia CUDA [3] provides top throughput only when the same program (including data-dependent branching) is executed over multiple input data blocks in parallel. As selected cryptographic operations are heavily data/key dependent (e.g., public-key algorithms based on modular multiplication like RSA) performance gain may be more difficult to achieve [11]. The need for trust to provider is still present although more advanced skills may be required to extract cryptographic secrets from less common architectures, possibly via side-channel attacks. Additionally, a more complex architecture makes more difficult evaluation of security assurances when a single device is shared by mutually distrusting Clients.
3. *Use of dedicated cryptographic circuits*, e.g., application-specific integrated circuit (ASIC) can provide very high performance implementation for selected cryptographic algorithms. The disadvantage is a significant increase in the cost of design and development if required circuits are not readily available. High-speed cryptographic circuits were proposed and sometimes built for brute-force cracking of algorithms with insufficient length of key of used password like Copacobana (based on FPGA) [12]. Note that brute-force cracking architectures are usually not designed to handle high input/output traffic. If cryptographic circuit is not additionally protected, trust to provider is still required.
4. *Use of secure processors*, e.g., cryptographic smart cards or hardware security modules (HSM) can significantly limit level of trust put on CaaS provider. HSMs are able to provide high performance for certain use-cases (see 2.2) while providing good security for cryptographic keys even for attackers with physical access to CaaS. Use from virtualized environments is also possible – Virtual HSM project [15] provides remote physical HSM via PKCS#11 API.
5. *Use of fully homomorphic encryption* (FHE) – all architectures mentioned so far except secure processors required trust to CaaS provider. Fully homomorphic encryption [10] provides a way to perform sensitive computations on untrusted platform – a feature well suited for CaaS as well as cloud-based computations in general. While performance of FHE schemes has been significantly improving in recent years [13], including highly optimized implementations for FPGAs [7] , the overall performance is still several orders of magnitude slower when compared to unprotected implementations.

### 3.2 The proposed design

As discussed in previous section, dedicated high-performance hardware offers the best overall performance, but also comes with a high additional cost to verify required security properties in an auditable manner.

We instead propose to build CaaS from simple and small secure processing units that are easier to test for security assurances. A large number ($10^2 - 10^4$) of these secure processing units are connected in a massively parallel multi-processor device[4]. Every secure processor has limited persistent storage and may provide acceleration of some cryptographic operations. The design has to take care of all communication between secure processors if needed and to provide data confidentiality, integrity and freshness with the use of secure processors. Due to limited computational resources of secure processors, CaaS design will have to carefully separate untrusted storage for secure off-loading of sensitive dat from secure processors, provide untrusted connectivity of CaaS components, allocate of secure processors to tasks, and so on. Overall resiliency of CaaS can be high if failed or malfunctioning secure processing units are quickly and efficiently isolated.

In the rest of the section, we will describe how the proposed design can be implemented from a large number of modern cryptographic smart cards (secure processors) and how the principles laid out in Section 2.4 can be achieved with an example test application for computing OATH HOTP values [9] provided in Section 4.
The proposed architecture has the following key properties:

1. **High number of secure processors** – depending on the required performance, at least $10^2 - 10^4$ processors. Each processor is able to withstand focused physical and logical attacks as required by FIPS140-2 Level 3 or 4, CC EAL 4+, or similar. Attacker should not be able to learn any cryptographic secrets stored inside secure processors, read any sensitive input/output data even with direct physical access or modify applications running inside secure processors.
2. **Small trusted computing base** – every secure processor contains a small application capable of processing requests coming from Client using previously imported cryptographic secrets.
3. **Untrusted controller** – software responsible for efficient distribution of Client requests and storage of data offloaded from secure processors. The controller is untrusted, i.e., it must not be able to access plain values of any cryptographic material or input/output data supplied by Client for processing. If the controller is compromised, no secrets are revealed.

[4] Note that analogy with the current multi-core graphic processing units (GPUs) ends with the high number of cores. Parallel cores of GPUs are not designed for use as secure processors (both for performance and cost reasons). GPU cores share both memory and program's instructions. Also, the GPU is not specifically built to accelerate cryptographic operations (although high-speed encryption, etc. is possible – especially when only single key and large data are processed).

4. **Secure channels between secure processors** – if sensitive data is to be transferred between secure processors, end-to-end secure channels have to be established and used. A secure channel should be as lightweight as possible, yet able to withstand common attacks on network layer, such as packet replay.
5. **High-speed I/O data interface** – large number of requests imply significant volumes of data traffic in the order of gigabits per second. Note that because of a high number of parallel processors, it would be natural to create logical or physical clusters of processors with dedicated I/O interface to keep traffic volume within current technology capabilities.
6. **Initialization phase** – before a CaaS device is ready for operational use, it has to be securely initialised. This includes bare hardware and other components' configuration, upload and installation of verified application packages, exchange of initial secrets needed for secure processors' communication, generation and certification of keys and public keys. Initialisation phase is the single most critical operation of any CaaS device and its correct and secure execution must be independently verifiable at any time afterwards.
7. **Operational phase** – after a trusted initialization, a CaaS device is switched into operational mode and starts serving Client requests. Only code inside secure processors has to be trusted for processing Client's cryptographic secrets and data, once in operation mode.
8. **Restricting use and audit trail** – trust is the single most important aspect of CaaS. While CaaS has to provide maximum security, it must also offer means to audit and verify its operation. One of the approaches is to use authorisation tokens that has to be regularly, or on demand, re-issued. Client cryptographic secrets are then imported together with authorisation tokens limited use of secrets. Issued tokens can be then matched against a trusted audit trail produced by secure cryptographic processors. This not only allows independent verification of CaaS operation but it also gives Clients an efficient way to disable or even remove their secrets from CaaS – simply by not refreshing authorisation tokens.

### 3.3 Why smart cards?

Cryptographic smart cards [16] were designed to withstand attacks in completely hostile environment under full control of attackers. Cryptographic smart cards have following significant advantages in comparison to common CPU: 1) Secure runtime environment (an attacker cannot directly inspect executed code or manipulated data values including cryptographic keys); 2) Dedicated cryptographic coprocessors to speedup operations (especially relevant for asymmetric cryptography); 3) Secure on-card TRNG generator (usable for on-card keys generation); 4) Secure on-card storage (but limited in size); 5) Reasonable price per unit (when bought in larger quantities).

But smart cards are generally perceived as being quite slow and usable only for a single holder (user), not as a potential component for high-performance computation. Although it might be true when one compares single card with

a performance of desktop CPU, small size, low energy consumption, relatively low price and inherent advantage of secure contained environment make smart cards good candidate for powerful, yet secure computational device following principles defined in Section 2.4 – if a large number of cards can be utilized as array of secure processors.

In Table 1, raw performances of selected cryptographic algorithms are presented[5], showing that especially for RSA algorithm, smart cards have decent performance on its own. If an array of hundreds to thousands of smart cards can be run in parallel, high-performance composite device can be obtained.

| Card type | AES-128 CBC encrypt | RSA-1024 sign | RSA-2048 sign |
|---|---|---|---|
| **NXP CJ2A081 (2014)** | 36.5kB/sec | 10.5 signs/sec | 2.3 signs/sec |
| **NXP CJ3A080 v2.4.1 (2013)** | 17.6kB/sec | 6.3 signs/sec | 1.6 signs/sec |
| **Gemalto GXP R4 72K (2008)** | 10.8kB/sec | 2.5 signs/sec | 0.6 signs/sec |
| **NXP JCOP4.1 v2.2.1 72K (2008)** | N/A | 9.3 signs/sec | 1.6 signs/sec |

Table 1: The raw performance of Cipher engine with AES-128 key in CBC encryption mode and RSA-1024/2048 in PKCS1 sign mode with SHA-1 hash function. The raw performance is performance achievable when only time spend inside cryptographic coprocessor itself is assumed – no transfer of input data to card, key scheduling, engine init and startup etc. For AES algorithm, raw performance was computed from the difference between an encryption time for 512 and 256 bytes. For RSA algorithm, sole time to execute single sign operation on-card was measured.

## 4 The case study: HMAC-based one-time password

HMAC-based one-time password protocol (HOTP) [14] is widely used algorithm for generation of one-time passwords for an authentication. HOTP authentication code is based on a secret key shared between an authentication server and user and changing counter value incremented after every one-time password generation. HOTP is widely used, e.g., as a basic building block for Initiative For Open Authentication (OATH) [9]. We selected HOTP as example which involves not only single operation (e.g., RSA signature), but also maintenance of updated state (which must be offloaded outside physical card) and need for protected input and output from the Client (authentication server in this case) of a CaaS service.

[5] Note that provided comparison is meant only to demonstrate achievable level of performance and not as the exact comparison between various cards (there are differences between batches of cards). The more detailed comparison is provided in [18].

HOTP algorithm (RFC4226 [14]) is defined as sequence of four logical steps:

1. $HMAC(K, C) = SHA1(K \oplus 0x5c5c \ldots | SHA1(K \oplus 0x3636 \ldots | C))$, where $K$ is a secret key shared between user and authentication server, $C$ is counter incremented after every authentication attempt, HMAC is construction defined in RFC2104, and SHA1 is a cryptographic hash function.
2. $HOTP(K, C) = Truncate(HMAC(K, C)) \,\&\, 0x7FFFFFFF$, where *Truncate function* selects 4 bytes in a deterministic way from HMAC output.
3. $HOTP - Code = HOTP(K, C) \, mod \, 10^d$ where $d$ is desired number of digits of resulting code (system parameter).
4. $HOTP - Code$ generated by user is compared with expected $HOTP - Code$ generated by the authentication server.

### 4.1 Why would HOTP will benefit from CaaS?

Because both server and user need to store and use same secret key value $K$ used during every authentication attempt, not only the user, but also server becomes a plausible attacker's target when the value of the secret key is of interest. When an authentication server is temporarily compromised, an attacker can learn the secret keys for all of its users – or at least for those authenticated during the compromise period. An attacker can then use obtained secret keys to impersonate legitimate users later. To mitigate this threat, authentication server can utilize CaaS for HOTP code verification instead of computing expected HOTP code on its own. When a user provides HOTP code, authentication server asks CaaS service to compute expected code and verify it against supplied user code. An authentication server is then just notified about the verification result and does not need to be able to compute expected HOTP code itself. Even when authentication server is temporarily compromised, an attacker will not learn used secret key(s) (although may issue requests to CaaS on behalf of compromised server).

To learn a secret key, an attacker needs to attack CaaS platform, which can utilize secure hardware (e.g., HSM) to protect manipulated secrets. Authentication server can also utilize secure hardware itself – but because of associated upfront costs and management issues, only some will do while others stay with computation of HOTP in software. Additionally, when the authentication server runs as a virtual image in a public cloud environment, options to connect own secure hardware into datacenter are limited or not available at all.

### 4.2 Moving HOTP into CaaS

Four main operations are required to facilitate HOTP as a CaaS:

1. **Import new server's context** – done once for every authentication server. Contains keys used to protect user states and authenticate requests from authentication server to CaaS.

2. **Generation of initial, wrapped user state** – done once for every user of a particular authentication server. Contains HOTP specific state for given user including key $K$ and initial value of counter $C$.
3. **Verification of user-supplied HOTP code** – done for every user authentication request. Generates and compares expected and supplied HOTP code.
4. **Establishment and use of secure channels** – used to facilitate distribution of secrets and authorizations inside CaaS itself. Necessary to limit the overall number of HOTP verifications and provide cryptographic audit trail (principles P9 and P10).

### 4.3 HOTP implementation

To measure a real cost of HOTP verification in secure hardware using proposed architecture, we implemented HOTP verification as CaaS service, including all required operations as a part of CryptoHive design described in Sections 3.2 and 4.6. Using our implementation, we measured detailed time required to perform single HOTP verification as well as performance of the whole CryptoHive prototype.

Note that we excluded overhead related to transmission of Client request to CaaS service and back as overhead values are highly dependent on platform settings (e.g., how many credits are uploaded at once before costly recharge credits operation is invoked or how often is signed audit trail for performed operations generated).

We also intentionally excluded operations performed by the authentication server (Client) as this presents no load CryptoHive. We also did not include operations related to managing user contexts in untrusted part of CryptoHive where generic computational resources can be made powerful enough to match required load.

The following data blobs are present and processed: *Initial import of authentication server* with imported communication keys (256 bytes in total), *Authentication server context* with imported keys (4x AES128b keys, unchanged during the request, stored on CaaS platform in rewrapped form after initial import, only some keys shared with the authentication server, 88 bytes in total), *user HOTP state* (updated with every request, stored but unreadable by the authentication server, 40 bytes in total), input/output data with *user HOTP code* or verification result respectively (new with every request, provided and readable by authentication server, 24 bytes in total).

Following cryptographic keys are used: 1) The communication keys for encryption $K_{commEnc}$ and integrity $K_{commMAC}$ used for authorization and protection of data exchanged between the authentication server and CaaS (generated and used by authentication server). 2) The keys for protection of user HOTP state $K_{stateEnc}$ and $K_{stateMAC}$ (generated by CryptoHive and not shared with authentication server). 3) The authentication key $K_{auth}$ for given user (stored inside user HOTP state, generated by CryptoHive and not shared with authentication server). 4) The CaaS internal keys $K_{authServerCtxEnc}$ and $K_{authServerCtxMAC}$

for protection of offloaded authentication server contexts with $K_{commEnc}$, $K_{commMAC}$, $K_{stateEnc}$ and $K_{stateMAC}$) (generated by CryptoHive and not shared with authentication server) – note that these keys can be used to protect multiple authentication server contexts[6]. 5) RSA-2048b keypair $K_{pubCG}$ and $K_{privCG}$ for import of initial import of authentication server context (generated by CryptoHive with public key $K_{pubCG}$ distributed to authentication server).

### 4.4 Performance results – a single card

At first, we provide performance results for primitive operations used as building blocks to implement whole HOTP in CaaS, followed by the discussion about possible speedups.

Table 2 provides list of times required to finish HOTP operations[7]. HOTP verification operation is measured in two settings – in the first case (called *Clean call*) verification is performed with full initialization of all keys and cryptographic engines – corresponding to the situation when a given user was not authenticated recently and no pre-initialized engines can be used. In the second case (called *Repeat call*), card already have relevant keys initialized from the previous *Clean call* – corresponding to the situation when controller was able to keep secrets on card (e.g., due to low service load or dedicated card for target authentication server and user – kind of caching).

### 4.5 Improving on expected performance

Based on the measured results we can identify the steps which consumes most of the time to process. At first, *Verify HOTP code* operation is the dominating operation as all others are executed only once for every authentication server (*Import authentication server context*) or limited number of times (*Generate HOTP state for a new user* once for every user). The *Verify HOTP code* operation can be further divided into a data transmission (about 18 %), setup and clear of cryptographic engines and key objects (about 54 %), encryption/decryption and MAC operation (about 20 %) and remaining functionality like HMAC, dynamic truncation or comparison of expected and supplied HOTP code (about 8 %).

The time required for data transmission can be significantly reduced by increase in communication speed between smart card and reader (default value negotiated is 38400bps, but some smart cards can support 307200bps or more if the capable reader or custom build reader can be used).

The largest fraction of a time on the card for HOTP verification is clearly consumed by the preparation of cryptographic engines and not by the cryptographic operation itself (confirmed also by the performance comparisons for a

[6] Unwrap keys and engines can be preinitialized as are shared between multiple contexts. Additionally, limited number of unwrapped authentication server contexts can be also left on-card to decrease latency of subsequent requests.

[7] Detailed description of measurements with results for other variants can be found at http://crcs.cz/papers/space2015

| Operation | Length (bytes) | Clean call | Repeat call |
|---|---|---|---|
| **Verify HOTP code** | **I/O:157/66B** | **288ms** | **134ms** |
| 1. Transfer authentication server context, input data and user state into card | 5+88+40+24 | 34ms | 34ms |
| 2. Unwrap authentication server context – use: $K_{authServerCtxEnc}$ and $K_{authServerCtxMAC}$ | 88 | 14ms | 14ms |
| 3. Unwrap user state (HOTP counter, failed attempts, settings, HMAC key) – prepare&use: $K_{stateEnc}$ and $K_{stateMAC}$ | 40 | 65ms | 11ms |
| 4. Unwrap input data (HOTP code provided by user) – prepare&use: $K_{commEnc}$ and $K_{commMAC}$ | 24 | 63ms | 10ms |
| 5. Compute HMAC&truncation over current value of counter obtained from user state– prepare&use: $K_{auth}$ | - | 20ms | 20ms |
| 6. Compare expected and supplied HOTP code, update failed attempts count, update counter | - | 4ms | 4ms |
| 7. Wrap output data with status of HOTP code verification (correct/incorrect) – prepare&use: $K_{commEnc}$ and $K_{commMAC}$ | 16 | 33ms | 10ms |
| 8. Wrap updated user state – prepare&use: $K_{stateEnc}$ and $K_{stateMAC}$ | 32 | 36ms | 12ms |
| 9. Transfer output data and user state outside card | 40+24+2 | 19ms | 19ms |

Table 2: The performance of operations required to complete single HOTP code verification request performed by NXP CJ2A081 smart card. The measured time is an average taken from 100 independent measurements. Note that results on different cards may differ, see [18]. Prepare&use means: prepare key object(s), initialize cryptographic engine(s) with prepared key(s) and decrypt/encrypt and sign/verify data.

| Operation | Length (bytes) | Time (ms) |
|---|---|---|
| **Import authentication server context** | **I/O:261/90B** | **534ms** |
| 1. Transfer wrapped authentication server context into card | 5+256 | 64ms |
| 2. Unwrap initial authentication server context – use: $K_{privCG}$ | 256 | 430ms |
| 3. Create internal authentication server context and generate keys $K_{stateEnc}$ and $K_{stateMAC}$ | 32+32 | 4ms |
| 4. Wrap authentication server context by internal keys – use: $K_{authServerCtxEnc}$ and $K_{authServerCtxMAC}$ | 88 | 14ms |
| 5. Transfer internal authentication server context outside card | 88+2 | 22ms |

Table 3: The performance of operations required to complete single import of authentication server context by NXP CJ2A081 smart card. Resulting rewrapped context is later used in other HOTP operations.

| Operation | Length (bytes) | Time (ms) |
|---|---|---|
| **Generate HOTP state for a new user** | **I/O:12/42B** | **70ms** |
| 1. Transfer user state information into card | 5+7 | 14ms |
| 2. Prepare new HOTP state for user and generate new $K_{auth}$ key | 28 | 3ms |
| 3. Wrap user HOTP state – prepare&use: $K_{stateEnc}$ and $K_{stateMAC}$ | 40 | 36ms |
| 4. Transfer user HOTP state outside card | 40+2 | 17ms |

Table 4: The performance of operations required to complete creation of context (HOTP state) for new user of given authentication server by NXP CJ2A081 smart card. Resulting user HOTP state is later used in *Verify HOTP code* operation.

wider range of different smart cards [18]). The fixed time required to initialize and setup the engine is especially significant when relatively short data blocks are processed. The required time can be decreased, if initialized keys and engines already present on a card are used as demonstrated by *Repeat call* measurements in Table 2 – e.g., when multiple requests for the same cryptographic context are performed in close sequence (requires proper optimization of distribution of requests to same set of cards). Design and implementation should also use lowest possible (yet secure) number of keys for different operations.

### 4.6 Performance results – network of processors

So far, we focused on performance of one secure processor – smart card. Even single card can be suitable platform for smaller uses with ability to serve more then 300,000 authentications per day. Still, we need to increase transaction rate significantly to provide CaaS service shared between many users.

We built a prototype "CryptoHive" as described in Section 3.2 to show scalability of our approach. The enclosure is a standard 1U rack-mount with a standard Intel i5 processor, 4GB of RAM, 2x 120GB SSD disk, and 2x 1Gbps ethernet interface. The first version used a set of smart cards and smart-card readers connected to this untrusted controller via USB ports. Prototype characteristics:

- standard size of 1U server, Intel i5, 4GB RAM;
- 45x NXP CJ2A081 smart cards (JavaCard 2.2.2 platform);
- Omnikey 6121 USB SIM Reader as smart card readers;
- 8x active USB hub with 7 ports - connected in a two-level tree; and
- AES128/256 CBC, CBC-MAC/ RSA 2048 as main internal cryptographic algorithms.

We have encountered several difficulties with this architecture, namely:

- only 10 or 16 card readers are detected by default on OS Windows 7/8 and Linux (Ubuntu 15.04) respectively;
- parallel requests are inherently serialized by the communication stack (i.e., PC/SC interface);

– compatibility issues with some USB hubs and selected readers;
– relatively high failure rate of smart-card readers.

We have eventually overcome these difficulties and created a functional prototype suitable for long-term tests. The experience was used for design of an improved version. Smart cards are connected via internal Ethernet hub and a custom communication layer that allow to maximise performance of smart cards plugged in the "CryptoHive" and significantly improve reliability of the whole architecture.

Even with relatively small number (45) of smart cards, the only efficient implementation of the untrusted controller is a fully asynchronous version. It turned out that 30-40 smartcards were able to serve sufficient amount of requests to create significant synchronisation bottlenecks when the system with partially synchronous implementation was used.

Asynchronous controller was able to utilise secure cryptographic processors near to physical maximum – at about 98 % of the theoretical maximum. This in effect demonstrates almost linear scalability of the computational power of the "CryptoHive".

We have introduced an additional overhead to provide auditable audit trails and key use dependent on the presence of authorisation tokens but their impact on the system throughput is in the region of 1-2% with authorisations renewed on average every 30 seconds.

## 5 Future directions

We believe that CaaS is only at its beginnings and there are a large number of research as well as engineering problems that need to be solved. Some particular issues we encountered while working on this problem include:

**Efficient secure channel context management** – the question is to find an efficient mechanism to protect and efficiently access cryptographic contexts with only a limited secure resources (computational power, memory space). Scaling of CaaS means that the number of contexts greatly exceeds the size of secure memory. This is closely related to efficient offload and restore of intermediate cryptographic context including fully scheduled keys and initialized engine. Such a feature is not currently supported by smart cards because in single holder scenario, keys are changed infrequently and will all fit into available on-card memory. Also, offloading intermediate state extends an attack surface to mount various side-channel or fault attacks. But benefits of such a would be high as preparation of cryptographic contexts accounts for more then half of total time of HOTP verification.

**Highly accessible distributed shared state with updates** – freshness of requests in highly distributed environment, where updates have to be instantly distributed to a large number of processing units. The classic option is either to limit modification of the state data only to single processor (which would decrease performance) or to combine partial state updates into a final

state later. Delayed combination of state can be performed either in secure processors (increasing latency to serve request) or in untrusted controller (which requires suitable secure scheme).

**Robust architecture tolerant to hardware/software failures** – the architecture must be be fault tolerant and be able to recover automatically; if a particular secure computational resource fails permanently, the system has to adapt to that and continue safe and secure operation.

**Encryption and authentication schemes** – establishment of cryptographic contexts is an expensive operation and ability to merge multiple atomic cryptographic operations into single invocation of cryptographic engine provides immediate computational boost. For example, it is faster to encrypt and transmit 256B of data then encrypt and then MAC only 32B of data. If some precomputation of data otherwise done by service (e.g., keystream) can be done by Client and transmitted inside encrypted request to service, performance gain can be obtained.

## 6 Conclusions

As more and more services are being moved into cloud environment, user has less control over his/her sensitive data including cryptographic keys. Cryptography as a Service (CaaS) is an attempt to offer cryptographic functions in a similar manner as a generic computation is offered in cloud. There has been little systematic discussion yet about actual user needs in such a context as well as design of new architectures able to fulfil those needs.

We described several usage scenarios of CaaS and discussed its properties. We believe that scenario with many users and many cryptographic keys fits the best situation when secure hardware is shared among many users in the cloud computing. For such *many users, many keys* scenario, we defined set of principles which should be followed by platform offering CaaS functionality. Based on these principles, we discussed various available hardware architectures which can be used to provide computational resources for CaaS.

We propose scalable secure architecture for CaaS based on large numbers of secure processors, interconnected by secure channels to facilitate information exchange via untrusted surrounding environment and provide hierarchical control yet retains high performance. The proposed architecture was implemented as a prototype called "CryptoHive" using an array of cryptographic smart cards and evaluated on HMAC-based one-time password authentication (HOTP) protocol. We identified frequent switch of the cryptographic context switching (key scheduling, cryptographic engine initialization) as the major performance impactor in the HOTP as well as other usage scenarios. Based on the practical experience, a set of tips for improving performance are discussed together with possible future directions.

## References


1. CryptLib++ project, http://www.cryptlib.com/ [12/7/2015].
2. OpenSSL project, https://openssl.org [12/7/2015].
3. NVIDIAs next generation CUDA compute architecture: Fermi. NVIDIA, 2009.
4. Amazon AWS. CloudHSM, https://aws.amazon.com/cloudhsm/ [12/7/2015].
5. Sren Bleikertz, Sven Bugiel, Hugo Ideler, Stefan Nrnberger, and Ahmad-Reza Sadeghi. Client-Controlled Cryptography-as-a-Service in the Cloud. In *Applied Cryptography and Network Security, LNCS 7954*, pages 19–36. Springer, Berlin, 2013.
6. Richard Chow, Philippe Golle, Markus Jakobsson, Elaine Shi, Jessica Staddon, Ryusuke Masuoka, and Jesus Molina. Controlling data in the Cloud: Outsourcing computation without outsourcing control. In *ACM Workshop on Cloud Computing Security (CCSW '09)*, pages 85–90. ACM, 2009.
7. Yarkin Doroz, Erdinc Ozturk, and Berk Sunar. Accelerating fully homomorphic encryption in hardware. In *IEEE Transactions on Computers*, volume 64/6, pages 1509–1521. IEEE, 2015.
8. Riccardo Focardi, Flaminia L. Luccio, and Graham Steel. An introduction to security API analysis. In *Foundations of Security Analysis and Design VI, LNCS 6858*, pages 35–65. Springer Berlin Heidelberg, 2011.
9. Initiative for open authentication (OATH). http://www.openauthentication.org/ [12/7/2015].
10. Craig Gentry. Fully homomorphic encryption using ideal lattices. In *41st ACM Symposium on Theory of Computing (STOC)*, pages 169–178. ACM, 2009.
11. Keon Jang, Sangjin Han, Seungyeop Han, Sue Moon, and Kyoungsoo Park. SSLSshader: cheap SSL acceleration with commodity processors. In *8th USENIX conference on Networked systems and implementation, NSDI11*. USENIX Association, 2011.
12. Sandeep Kumar, Christof Paar, Jan Pelzl, Gerd Pfeiffer, and Manfred Schimmler. Breaking ciphers with COPACOBANA –a cost-optimized parallel code breaker. In *Proceedings of the 8th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'06, pages 101–118. Springer-Verlag, 2006.
13. Tancrede Lepoint and Michael Naehrig. A comparison of the homomorphic encryption schemes FV and YASHE. In *AFRICACRYPT 2014, LNCS 8469*, pages 318–335. Springer, 2014.
14. D. M'Raihi, M. Bellare, F. Hoornaert, D. Naccache, and O. Ranen. HOTP: An HMAC-based one-time password algorithm. In *RFC 4226*. IETF, 2005.
15. OpenVZ. VirtualHSM project, https://openvz.org/virtual_hsm [12/7/2015].
16. W. Rankl and W. Effing. *Smart Card Handbook, ISBN 9780470856680*. Wiley, 2004.
17. Peter Robinson. Cryptography as a service. In *RSAConference Europe 2013*, 2013.
18. Petr Švenda. JCAlgTester project, http://www.fi.muni.cz/∼xsvenda/jcsupport.html [12/7/2015].