# CASTLE: CA Signing in a Touch-Less Environment

Stephanos Matsumoto[†‡]
smatsumoto@cmu.edu

Samuel Steffen[‡]
steffsam@student.ethz.ch

Adrian Perrig[‡]
adrian.perrig@inf.ethz.ch

[†]Carnegie Mellon University
Pittsburgh, PA, USA

[‡]ETH Zurich
Zurich, Switzerland

## ABSTRACT

With the emergence of secure network protocols that rely on public-key certification, such as DNSSEC, BGPSEC, and future Internet architectures, ISPs and domain administrators not specialized in certification have been thrust into certificate-signing roles. These so-called conscripted CAs sign a low volume of certificates, but still face the same challenges that plague modern CAs: private signing key security, administrator authentication, and personnel and key management. We propose CA Signing in a Touch-Less Environment (CASTLE), an air-gapped and completely touchless system to enable low-volume, high-security certificate signing in conscripted CAs. We demonstrate that CASTLE's layered, defense-in-depth approach is technically and practically feasible, and that CASTLE empowers conscripted CAs to overcome challenges that even professional CAs struggle with.

## 1. INTRODUCTION

Public-Key Infrastructures (PKIs) in the current Internet provide certificates, which bind public keys to information such as DNS names, routing identifiers, or organizational identities. These certificates are issued by Certificate Authorities (CAs), whose public keys are known and trusted by their relying parties. CAs are responsible for issuing certificates with the correct information and for protecting their private signing keys.

While most CAs operate as business entities dedicated to certificate issuance and maintenance, they are not the only entities that issue certificates. For example, corporations use internal CAs and certificates as part of an enterprise PKI. In some future Internet architectures, ISPs and domains have CA responsibilities thrust upon them [10, 16, 24]. Finally, technologies such as DNS Security Extensions (DNSSEC) are causing entities such as the operators of ccTLDs to take on CA roles. Like full-time CAs, these *conscripted CAs* sign certificates, but at a much lower volume than dedicated CAs.

Moreover, even traditional CAs, who specialize in the business of signing certificates, face operational challenges in their signature process. In 2013, TURKTRUST accidentally issued several certificates that endowed the holders with CA authority, allowing those certificate holders to issue fraudulent certificates for Google [13]. In 2015, Symantec (the largest CA today) issued unauthorized certificates for organizations such as Google and Opera for test purposes without the knowledge of these organizations [22]. These CAs are thus failing to carefully check the certificates that they are issuing.

The storage of private signing keys is also a challenge, even for traditional CAs. Virtually all CAs protect their private signing keys through the use of Hardware Security Modules (HSMs), leveraging hardware-based protection to prevent the private key from falling into the wrong hands. They are used to protect high-value keys such as the DNSSEC root key [3, 21]. However, HSMs are not a cure-all solution for private key storage, as they can be circumvented by exploiting misconfigurations or software flaws, or by compromising authentication that enables HSM functionality [9, 23]. CA-grade HSMs can also cost upwards of tens of thousands of dollars [21], and create vendor lock-in due to proprietary, closed-source technologies that make switching to a new solution difficult.

Another challenge is the protection of the private signing keys against potentially malicious administrators, who may attempt to use these keys for nefarious purposes. For example, in 2015, a Chinese CA's private signing key was installed in a proxy, allowing the proxy to sign arbitrary certificates and mount on-the-fly man-in-the-middle (MitM) attacks [14]. While there may be legitimate reasons to allow administrator access to the private key storage, such as to back up or update the private key, we observe the need to enable useful management without unnecessarily exposing the private key to possibly adversarial administrators.

These challenges motivate the core questions of this paper. How can we encourage conscripted CAs to carefully check the certificates they sign? How can we protect the private keys used for signing certificates? How can we simplify management functions such as private key backup for these entities? Most importantly, if traditional CAs struggle to perform these operations securely, how can we make it easier for these non-specialized entities to do so?

In addressing the above questions, we specifically focus on conscripted CA organizations, which we characterize as having four important attributes: (1) a lack of specialized operations and administrators for CA functions, (2) a low issuance volume (on the order of one per week), (3) a reluctance to HSMs due to cost and vendor lock-in concerns, and (4) facilities and infrastructure to support physical security. To our knowledge, no current solution is well-suited to this class of CAs.

To address the problems of securing certificate signatures, private key storage, and key management in conscripted CAs, we propose CA Signing in a Touch-Less Environment (CASTLE), a CA signing system aimed at entities that sign a low volume of certificates. CASTLE leverages trusted computing on commodity

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

*ACSAC '16, December 05 - 09, 2016, Los Angeles, CA, USA*

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4771-6/16/12...$15.00

DOI: http://dx.doi.org/10.1145/2991079.2991115

hardware, highly restricted input/output (I/O) channels, and multi-administrator management protocols.

CASTLE takes a layered, defense-in-depth approach to securing certificate signing. The signing machine leverages a trusted platform module (TPM) to protect the private keys from the untrusted OS. Moreover, the signing machine is air-gapped, encased in a glass box, and completely touchless to thwart physical tampering. The signing protocol itself authenticates administrators via a PIN on a mobile device and strong one-time authentication protocols that prevent adversaries from authorizing certificate signatures Finally, a threshold protocol secures logging, key backup, and administrator management.

In summary, we make the following contributions:

- We design a CA application that leverages trusted computing, physical isolation, strong authentication, and distributed management to hinder adversaries.
- We develop protocols for logging, log auditing, key backup and rollover, and administrator management, secured by a quorum of honest administrators.
- We implement and analyze a proof-of-concept prototype of CASTLE and demonstrate that our system is both technically and practically viable.

# 2. BACKGROUND

In this section, we provide a brief overview on trusted computing, with a particular emphasis on the Flicker trusted computing infrastructure used in CASTLE.

The goal of trusted computing is to enable the verification of software that has executed on a machine. It enables "trusted" pieces of software to prove their integrity to a local or remote verifier, a process called *attestation*. Trusted computing also allows trusted software to make use of *sealed storage*, encrypting data such that only the same software can decrypt the data.

Trusted computing can be implemented based on the *trusted platform module (TPM)* [1], a commodity chip present in many systems. In particular, the TPM maintains a set of *platform configuration registers (PCRs)*, which store information of what code the system has loaded and executed. Before the CPU executes code, the TPM first *measures* or hashes the code, and then *extends* the code into a PCR by hashing the current contents of the PCR along with the measurement of the code. A PCR thus represents a chain of measurements.

Such a chain can be used in the *trusted boot* process [1], which leverages the BIOS as a *static root of trust*. Beginning with the BIOS, each module measures the next module in the boot process and extends it into the PCR to create a proof of what modules have been loaded and executed. By contrast, a *dynamic root of trust* provides a special CPU instruction to clear a set of PCRs and a extend particular code block (called the *secure loader block (SLB)* in Intel TXT [5]) into a PCR, allowing the *late launch* of code that can be verified without measuring the entire boot process.

A TPM can also make use of *sealed storage*, in which it encrypts data using a public key such that it can only be decrypted if the PCR configuration matches given values. The corresponding private key used for decryption never leaves the TPM. Sealing data can be used to make data accessible only to a trusted application. A TPM also has a small amount of non-volatile RAM (NVRAM), which can be used to store data on the TPM itself while protecting the data from applications with incorrect PCR values.

The Intel Software Guard Extensions (SGX) [8] are a relatively new technology in trusted computing allowing the creation of *enclaves*, a region of memory containing private data and/or code. These enclaves are created by a special CPU instruction, and their run-time memory is encrypted and integrity-protected. The CPU protects the data and execution of an enclave from the operating system and privileged processes such as a hypervisor, as well as from direct memory accesses. At the moment, SGX is a relatively new technology and only available on a limited subset of Intel processors. However, we envision using SGX in upcoming versions of CASTLE.

**Flicker** [15] is a trusted execution infrastructure that can execute small blocks of code called Pieces of Application Logic (PALs) in late-launch sessions. While executing a Flicker session, the OS is suspended to isolate the PAL from the OS and other applications. The attestation process allows verification of a PAL's integrity after a Flicker session. All memory used by the PAL is erased after a session and an all-zeroes hash is extended into the PCRs to prevent the data from being accessed further. The PAL output is transferred to the application that called the session.

The PAL is highly restricted in its operation. For example it cannot access any I/O devices (except a serial port for debugging) and cannot make use of dynamically linked libraries. However, by assembling all the security-critical logic of an application into a Flicker PAL, the trusted computing base (TCB) of a system can be minimized. In order to keep the TCB of CASTLE small, we make use of Flicker in the system architecture.

# 3. PROBLEM DEFINITION

In this paper, we address the problem of providing a secure CA signing solution for low-volume conscripted CAs for whom certificate signing is not a primary business. In this section, we define the problem in detail, including the desired properties and our adversary model.

## 3.1 Desired Properties

The primary function of our system is to *sign certificates*. In doing so, the system must authenticate administrators and generate signatures on certificates deemed to be authorized by an administrator. A secondary function of our system is to provide additional functions to *manage the system operation*. These functions include initializing the system, enrolling or removing administrators, logging past actions such as processed Certificate Signing Requests (CSRs), and backing up and restoring system information.

In providing the above functions, we aim to achieve several properties in our system:

1. **Security.** An adversary as described below cannot obtain an unauthorized certificate.
2. **Efficiency.** Interactions with the signer machine (in particular, the number of messages sent) should be minimal.
3. **Auditability.** All system actions should be logged. The log should be reviewable by those auditing the system, and be resistant to tampering.
4. **Cost-effectiveness.** The system should be of minimal cost; in particular, the cost should be at the level of a commodity PC system.

## 3.2 Assumptions and Adversary Model

Our adversary's goal is to undetectably obtain an unauthorized signature on a certificate. The adversary may attempt to achieve this goal by gaining access to the private signing key, compromising the certificate issuance process itself, or by leveraging one or more malicious administrators.

We assume that the adversary can control up to $k-1$ out of $m$ administrators *or* their personal verifier devices during a certificate signing session. The adversary can also control all components of the signing machine, (including the operating system and applica-

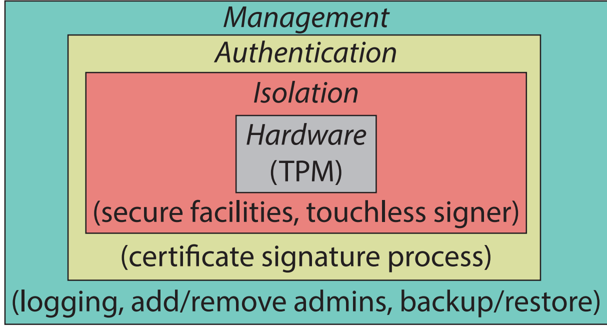


**Figure 1: Defense-in-depth layers of CASTLE.**

tions) except the CPU, TPM, the LPC bus connecting the CPU and TPM, and the memory bus on the signing machine. For management functions such as adding administrators, we assume that the adversary can control at most $u-1$ administrators or verifier devices (where $u \geq k$), as well as the signing machine as described above. In both scenarios, the compromised verifier devices can send or display arbitrary information during operation, and compromised administrators can carry out the protocol steps incorrectly and use their administrator credentials to the adversary's goal.

We assume that the adversary cannot break cryptographic primitives, i.e., public or symmetric-key cryptography or cryptographic hash functions, and that the software libraries, in particular, those used for processing QR codes and certificates, are free of exploitable bugs. We also assume that software updates are not necessary for the air-gapped component of our system.

We further assume that the conscripted CA has the means to provide and enforce physical security measures, such as security guards, surveillance cameras, restricted areas, and employee access codes. Because even conscripted CAs have facilities and resources to physically protect (particularly in enterprise environments), such as access-controlled server rooms, we find our assumptions regarding physical security to be realizable in practice.

# 4. CASTLE ARCHITECTURE

We now provide an overview of the CASTLE architecture. We begin by describing our multi-layer approach to securing certificate signatures and our architectural components. We then describe how the design of CASTLE provides security at each layer.

## 4.1 Security Layers

To secure certificate signing, CASTLE employs a four-layer defense (shown in Figure 1) as follows:

1. The **hardware layer** protects sensitive data such as the private signing key on the signing machine even from an adversary who may have compromised the machine itself. This layer consists of the TPM and hardware security mechanisms for the CPU.
2. The **isolation layer** protects the signing machine from being tampered with, both physically and at the software level. This layer consists of several forms of isolation, including the use of touchless input, air gapping, and secure physical facilities.
3. The **authentication layer** prevents the signing machine from generating signatures without the necessary authentication and authorization. This layer consists of the certificate signature process, described in more detail in Section 5.

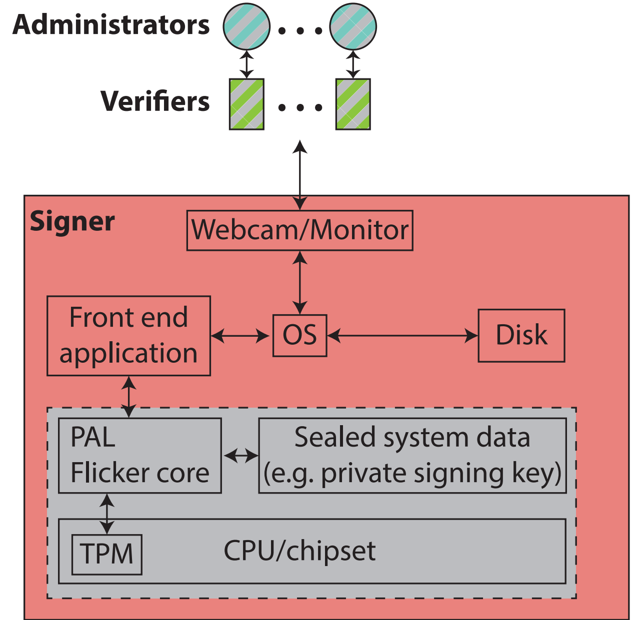


**Figure 2: CASTLE architecture. The TCB is shown in gray and in the signer is bounded by dashed lines. Gray stripes indicate partially-trusted entities.**

4. The **management layer** protects management functions, ensuring that adversaries cannot enroll themselves as administrators or gain access to private keys via the backup process. This layer consists of the log and the protocols for management operations, described in more detail in Section 6.

## 4.2 Architectural Components

As Figure 2 shows, the CASTLE architecture includes administrators, verifiers, and a signer machine.

An **administrator** interacts with CASTLE's devices in order to authenticate to the signer (see below) and authorize the signature of a certificate. Administrators also interact with each other to carry out management functions. Each administrator has a unique index $i$ in the set of administrators and a public key $K_i$.

Each administrator carries a **verifier**, a mobile device (*i.e.*, a tablet or phone) used to communicate with the signer and perform computations on behalf of the administrator. A verifier belongs to a specific administrator and stores the administrator's private key $K_i^{-1}$. The verifier must have a camera and display to communicate with the signer, an application to run the CASTLE processes, and protected storage for the private key that can only be unlocked by authenticating to the application.

The **signer** is a machine that authenticates administrators and generates a signed certificate when authorized to do so. The signer also stores the system information such as private signing keys, administrator information, and event logs, and carries out both certificate signatures and management operations by accessing this information. The signer has a commodity OS, touchless I/O interface in the form of a webcam and monitor, and a TPM.

## 4.3 Hardware Layer

The TPM, CPU, LPC bus, and memory bus on the signer are the only fully trusted pieces of hardware in CASTLE (though the

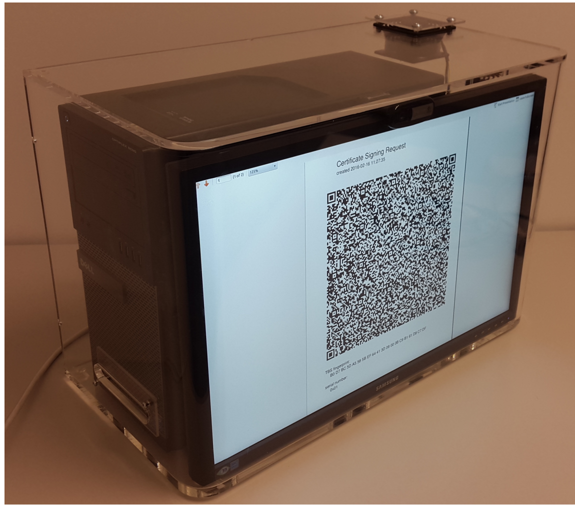


**Figure 3: CASTLE signer prototype in the glass box, displaying a QR code. Manufacturing of the glass box by Magnetron Labs Merz.**

administrators and verifiers are partially trusted. This trusted hardware comprises the hardware layer and protects information that no other component of the system should access. Protected information includes the following: (1) system private keys used in attestation, signatures, and decryption of messages from administrators, (2) public keys used to authenticate messages from administrators, (3) event logs, used to record all system operations, (4) session data used in the authentication layer, and (5) system parameters used in the management layer. The hardware layer, specifically the TPM, secures this information via sealed storage and its NVRAM. The Memoir system [18] can be used to protect the integrity and persistence of information in case of unanticipated power loss.

The protection provided at the hardware layer stems from the security of the TPM, whose security in turn relies on that of the CPU, LPC bus, and memory bus. As long as the TPM remains uncompromised, sealed storage applied properly within Flicker allows protected information to only be released to a specified application (Flicker running the CASTLE PAL), and NVRAM storage allows particularly sensitive information to only be accessible or modifiable by a specified application. Since we assume that the TPM cannot be compromised, we therefore conclude that only the correct CASTLE PAL can access any sealed information, and read or overwrite any information stored in NVRAM.

## 4.4 Isolation Layer

The verifiers and signer are stored in secure physical facilities, and the signer is additionally air gapped and touchless. These measures comprise the isolation layer, which ensure that the devices and tokens cannot be stolen or tampered with. Verifiers can be stored in facilities such as secure lockers only accessible to their respective administrators, while the signer can be housed in a facility such as a secure room with security staff to control access to the room. By our assumptions, a conscripted CA has the means to maintain such physical security, and thus the signer and verifiers can remain physically secure.

The signer machine is protected by multiple forms of physical isolation. As Figure 3 shows, the signer is housed in a glass box, with only a power connection crossing the box's boundary.[1] All fans and vents on the box are covered with additional layers of glass, hindering adversaries from physically accessing the machine via direct hardware access. The signer also has no network capabilities, preventing any access through the network or wireless peripherals.

The signer's I/O is completely touchless: input is provided by scanning QR codes (provided by the verifier) with a webcam and output is provided through a display in the form of human-readable text and machine-readable QR codes (scanned by verifiers). This I/O system hinders adversaries by limiting their interactions with the system compared to using a keyboard or mouse.

## 4.5 Authentication Layer

A certificate signature session in CASTLE is carried out by $k$ administrators and their verifiers. The verifiers are protected by a secure PIN so that only the corresponding administrators can access them. A signing session consists of four steps:

1. **Authentication**: the administrators unlock their verifiers with a PIN, authenticate to the signer via a digital signature, and initiate the signing process with a CSR conforming to the PKCS #10 specification [17].
2. **Attestation**: the PAL checks the signatures and creates a signature on the CSR provided, attesting the received CSR.
3. **Authorization**: the administrators check the attestation signature and if correct, authorize the certificate signature via signatures with their own private keys.
4. **Signature**: the PAL checks the authorization, then signs and returns the certificate, completing the process.

During a certificate signature session, the administrators communicate with the CASTLE PAL through the untrusted signer front end. Thus the signer front end can observe and modify all messages sent between the administrators and the PAL. Each step of the session is designed to prevent a malicious front end from using these capabilities to sign an unauthorized certificate. Each message from the administrators is also temporally bound, preventing the reuse of messages from previously successful process instances (e.g., attempts to re-issue certificates that were revoked).

The verifier serves to provide strong authentication, relying on what the administrator *has* (the verifier itself) and what the administrator *knows* (the PIN for the verifier application). Moreover, even a malicious administrator cannot cause a certificate to be signed without gathering a quorum of $k$ administrators.

## 4.6 Management Layer

In order to protect management functions, the management layer primarily relies on the assumption that fewer than $u$ of $m$ administrators are malicious. The management layer thus requires the authorization of at least $u$ administrators to add or remove an administrator or to back up or restore system information, including all information secured by the hardware layer. The management layer also requires authorization from $u$ administrators to change the value of $u$.

The backup and restore process allows a system's private keys, administrator information, logs, and parameters to be transferred between two signer machines. A backup from one machine is thus equivalent to a restore to the other machine. In order to prevent exposing system information to any observer who sees the messages exchanged between the signers, the information is encrypted with the public key of the restored machine, and $u$ administrators must authorize both the backup and restore actions. If we assume that fewer than $u$ administrators are malicious, we can infer that the backup will be encrypted with a real CASTLE signer's public key

[1] Using induction coupling would avoid a physical connection into the glass box for the power line.

**Table 1: Notation used in the paper.**

| Notation | Meaning | Use |
|---|---|---|
| **A** | Attestation Signature | Attest to CSR and OTP received at PAL and to the session BLOB |
| $A_i$ | Authorization Signature | Authorize final certificate signature |
| $\mathcal{B}$ | Base BLOB | Store base key sealed to the PAL. |
| **C** | CSR | Certificate signing request to be signed |
| $\mathcal{C}$ | Configuration BLOB | Store signer information (encrypted with $K_b$) |
| $d$ | Hidden digit | Encourage administrator diligence |
| $h_i$ | Epoch identifier | Identify specific log entry |
| $\mathcal{I}$ | Initialization BLOB | Store intermediate information during system setup |
| $k$ | Signature threshold | Number of administrators needed for signature process |
| $\mathbb{K}$ | Public key set | Store administrator public keys |
| $K_b$ | Base key | Encrypt sensitive system information (symmetric key, sealed to PAL) |
| $K_c, K_c^{-1}$ | Attestation key pair | Certify/verify messages from the PAL |
| $K_\varepsilon$ | Non-migratable key pair | Authenticate TPM during initialization |
| $K_r, K_r^{-1}$ | Restoration key pair | Ensure confidentiality when restoring a backup to a target machine |
| $K_s, K_s^{-1}$ | Signature key pair | Generate/verify certificate signatures |
| $\ell$ | Log length | Identify how many events have been logged since initialization |
| $L_i$ | Log event | Record authentication, signature, or management operation |
| $m$ | Administrator set size | Number of currently enrolled admins |
| $M_i$ | Initialization nonce | Prevent replay of old setup messages |
| $R_i$ | Request signature | Authenticate administrator at start of signing session |
| **S** | Signed certificate | Certificate signed with $K_s^{-1}$ |
| $\mathcal{S}$ | Session BLOB | Store signature session information between attestation and signature steps (encrypted with $K_b$) |
| $S_i$ | Setup signature | Authorize system state generation |
| $u$ | Management threshold | Number of administrators needed for management functions |

and not with a public key created by an adversary. In Section 7, we further analyze the security of the management operations.

# 5. CASTLE CERTIFICATE SIGNING

In this section, we describe CASTLE's certificate signing process. Throughout the remainder of the paper, we use notation as listed in Table 1. We begin the section with a brief description of the signer configuration, which is required to intuitively understand some of the design decisions of CASTLE. We then describe the four steps of the process (authentication, attestation, authorization, and signature).

## 5.1 Signer Configuration

The information stored at the signer is structured as shown in Figure 4. The information stored on the signer consists of three main parts: the BLOBs, the log, and the NVRAM.

The information required during a certificate signing session is stored in BLOBs. We use a hybrid encryption scheme in which only the base key $K_b$ is sealed to the PAL. We thus only seal a single symmetric key with the TPM and call this sealed key the *base BLOB* $\mathcal{B}$. The base key is used to encrypt the larger *configuration BLOB* $\mathcal{C}$, which contains the information used during a signing session, such as private keys. Table 1 explains the use of the information (and accompanying notation) used in CASTLE (including outside of a signing session).

For simplicity of implementing the logging functionality in the TPM, the log is a hash chain of events stored on disk. Each operation carried out by the PAL is logged in the form of an event $L_i$ that contains the relevant details of the operation (see Section 6.1 for precise definitions of each log event). The hash values in the chain

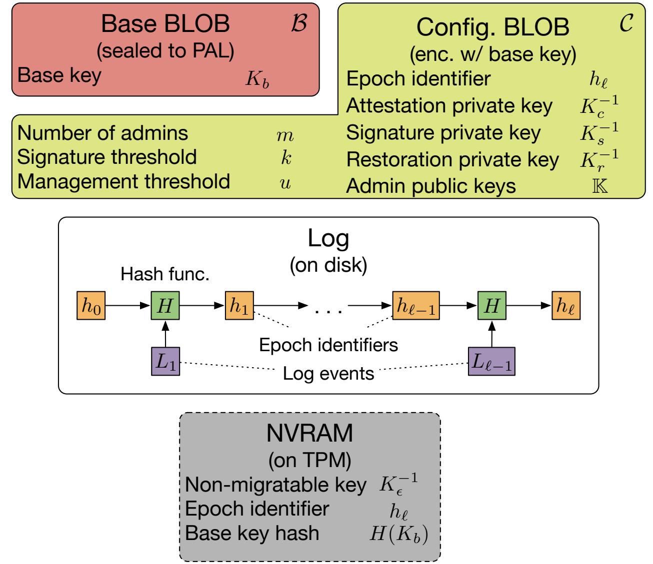


**Figure 4: Information stored by the signer.**

$h_i$ are called *epoch identifiers* and serve to link events and BLOBs temporally. We describe the logging mechanism in more detail in Section 6.1.

The NVRAM on the TPM stores a non-migratable private key $K_\varepsilon^{-1}$ (only used in the initial setup of CASTLE), as well as the current epoch identifier $h_\ell$, which prevents an adversary from undetectably replacing BLOBs with older versions or altering log entries. Moreover, in order to prevent replacement of both BLOBs, the NVRAM also stores a hash of the base key.

## 5.2 Request Step

A certificate signing session is carried out by $k$ administrators, each with their personal verifier. The first step in the session is the request, in which administrators unlock their verifiers via PIN and send the CSR and a digital signature (for authentication) to the signer. The authentication step hinders an adversary from initiating a signature process, since each request requires signatures from $k$ administrators.

To initiate the session, one administrator scans the CSR **C** at the signer's webcam, and each participating administrator creates a *request signature*

$$R_i = \mathsf{Sign}_{K_i^{-1}}(h_\ell \| \mathbf{C}) \tag{1}$$

The CSR generation process can take place on a verifier or on a separate machine, and does not need to be trusted due to the later check in the authorization step.

When the CSR is scanned, the signer displays the CSR fields in human-readable form to allow the administrators to check the CSR before authenticating to the signer. The signer then assembles and sends

$$\mathbf{C}, R_1, \ldots, R_k \tag{2}$$

to the PAL, assuming without loss of generality that the first $k$ administrators request the signing session.

## 5.3 Attestation Step

The attestation step produces a confirmation that the PAL has received the correct CSR and request signatures. The private key used for attestation is sealed and only accessible to the PAL, and

**Algorithm 1** PAL procedure during attestation, assuming the first $k$ administrators participate in the session.

1: **procedure** ATTESTATION
Input: $\mathbf{C}, R_1, \ldots, R_k, \mathcal{B}, \mathcal{C}$
2: $K_b \leftarrow$ unseal $\mathcal{B}$
3: $h_\ell$, $K_c^{-1}$, $\mathbb{K}$, $k$, $\ldots \leftarrow$ verify/decrypt $\mathcal{C}$
4: **for** $1 \le i \le k$ **do**
5: **if** $!\mathsf{Vrfy}_{K_i}(R_i, h_\ell \| \mathbf{C})$ **then**
6: $h_{\ell+1} \leftarrow H(h_\ell \| \mathtt{failed} \| \mathtt{att} \| i)$
7: $\mathcal{C}' \leftarrow$ write $h_{\ell+1}$ into $\mathcal{C}$
8: **return** $h_{\ell+1}, 0, 0, 0, \mathcal{C}'$
9: $h_{\ell+1} \leftarrow H(h_\ell \| \mathtt{success} \| \mathtt{att} \| \{1, \ldots, k\})$
10: write $h_{\ell+1}$ into NVRAM
11: $\mathcal{S} \leftarrow \mathsf{Enc}_{K_b}(h_{\ell+1} \| \mathbf{C})$
12: $\mathbf{A} \leftarrow \mathsf{Sign}_{K_c^{-1}}(h_{\ell+1} \| \mathbf{C} \| \mathcal{S})$
13: $\mathcal{C}' \leftarrow$ write $h_{\ell+1}$ into $\mathcal{C}$
14: **return** $h_{\ell+1}, \mathbf{C}, \mathcal{S}, \mathbf{A}, \mathcal{C}'$

**Algorithm 2** Administrator and verifier procedure during the authorization step.

1: **procedure** AUTHORIZATION
Input: $h_\ell, \mathbf{C}, \mathcal{S}, \mathbf{A}$
2: **if** $\mathsf{Vrfy}_{K_c}(\mathbf{A}, \mathbf{C} \| \mathcal{S})$ **then**
3: $d \leftarrow$ check human-readable CSR
4: **if** correct $d$ and correct PIN at verifier **then**
5: $A_i \leftarrow \mathsf{Sign}_{K_i^{-1}}(\mathcal{S})$
6: **return** $\mathcal{S}, A_i$
7: **else return** $0, \mathsf{Sign}_{K_i^{-1}}(h_\ell)$
8: **else return** $0, \mathsf{Sign}_{K_i^{-1}}(h_\ell)$

**Algorithm 3** PAL procedure during the signature step.

1: **procedure** SIGNATURE
Input: $\mathcal{S}, \{\mathcal{S}\}_{K_1^{-1}}, \ldots, \{\mathcal{S}\}_{K_k^{-1}}, \mathcal{B}, \mathcal{C}$
2: $K_b \leftarrow$ unseal $\mathcal{B}$
3: $h_\ell$, $K_s^{-1}$, $\mathbb{K}$, $k$, $\mathbf{C}$, $\ldots \leftarrow$ verify/decrypt $\mathcal{S}$, $\mathcal{C}$
4: **for** $1 \le i \le k$ **do**
5: **if** $!\mathsf{Vrfy}_{\mathbb{K}[i]}(\mathcal{S}, A_i)$ **then**
6: $h_{\ell+1} \leftarrow H(h_\ell \| \mathtt{failed} \| \mathtt{sig} \| i)$
7: $\mathcal{C}' \leftarrow$ write $h_{\ell+1}$ into $\mathcal{C}$
8: **return** $h_{\ell+1}, 0, \mathcal{C}'$
9: $\mathbf{S} \leftarrow \mathsf{Sign}_{K_s^{-1}}(\mathbf{C})$
10: $h_{\ell+1} \leftarrow H(h_\ell \| \mathtt{success} \| \mathtt{sig} \| \mathbf{S})$
11: write $h_{\ell+1}$ into NVRAM
12: $\mathcal{C}' \leftarrow$ write $h_{\ell+1}$ into $\mathcal{C}$
13: **return** $h_{\ell+1}, \mathbf{S}, \mathcal{C}'$

therefore the attestation signature can only be created by the PAL and only on the information that the PAL actually receives as input.

In addition to the information in Equation 2, the signer front end also sends the BLOBs $\mathcal{B}$ and $\mathcal{C}$ to the PAL, allowing the PAL to access the information it needs to complete the step. The PAL then carries out the procedure shown in Algorithm 1. In essence, the PAL checks each request signature, logging the result and producing a new epoch identifier.

If all request signatures verify successfully, the PAL creates the *session BLOB* $\mathcal{S}$, which links the new epoch identifier to the CSR. The PAL then creates an *attestation signature* $\mathbf{A}$ that contains the new epoch identifier, the CSR, and session BLOB. The PAL then updates the configuration BLOB with the new epoch identifier and returns the CSR, session BLOB, attestation signature, and updated configuration BLOB. The signer then displays a QR code encoding

$$\mathbf{C}, \mathcal{S}, \mathbf{A} \tag{3}$$

as well as the fields of $\mathbf{C}$ in human-readable form.

## 5.4 Authorization Step

The administrators scan the QR code containing Equation 3 from the signer and can carry out the next step of the session, authorization. The authorization step ensures that the administrators can check the CSR received by the PAL and then choose to authorize the signature or abort the process. The administrators must unanimously agree to authorize the signature, preventing a mistake by a fraction of the administrators from authorizing a signature.

Each verifier has a private key belonging to the administrator who is assigned to the verifier; this key is denoted $K_i^{-1}$ for the administrator $A_i$. Each administrator scans the output of the signer from the attestation step and carries out the procedure in Algorithm 2. In essence, each administrator simply verifies the attestation signature using $K_c$ to ensure that the PAL indeed received and acknowledged the correct CSR. An administrator can also check the human-readable display on the signer to confirm the CSR.

The verifier then displays the CSR fields on its display as a final confirmation of the certificate that will be signed. In order to encourage the administrator to diligently check the CSR, the verifier additionally displays a *hidden digit* $d$, which appears randomly among the CSR fields. The administrator must then find and provide $d$ as well as a PIN (established *a priori*) on the verifier. Authentication via a PIN provides access to $K_i^{-1}$ on the verifier.

If either the attestation signature check or the hidden digit and PIN check fail, the verifier returns an abort message consisting of a signature on the current epoch identifier. Otherwise, the verifier displays the session BLOB $\mathcal{S}$ (linked to the CSR $\mathbf{C}$) along with an *authorization signature*

$$A_i = \mathsf{Sign}_{K_i^{-1}}(\mathcal{S}) \tag{4}$$

which confirms $\mathcal{S}$ as the session BLOB to be passed back to the PAL. Each administrator then scans his or her respective authorization signature at the signer, which collects and sends the signatures to the PAL.

## 5.5 Signature Step

The PAL then proceeds with the final step of the session: generation of the signature. The signature step checks that the administrators have confirmed the certificate to be signed via authorization signatures, which relies on the previous steps in the session. The signature step thus ensures that a signed certificate can only be created by the PAL and only on a CSR that has been passed through the previous steps. As with the attestation private key, the signature private key is sealed and only accessible to the unmodified CASTLE PAL.

Along with the authorization signatures, the signer front end sends the BLOBs $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{S}$ to the PAL, providing the necessary information for the PAL to carry out the step. The PAL then carries out the procedure shown in Algorithm 3. The PAL first checks that the session BLOB's epoch identifier matches the value in NVRAM, preventing replays of old session BLOBs. The PAL then has the correct CSR and checks the authorization signatures. If all $k$ authorization signatures are correct, the PAL logs this event and issues the signed certificate $\mathbf{S}$. If any signature is incorrect, the PAL logs and returns an error.

The signer then displays the signed certificate $\mathbf{S}$ as a QR code.

| Operation | Information |
|---|---|
| Attestation | **C**, partic./failed admins |
| Signature | **S**, partic./failed admins |
| Key generation | $m$, $k$, $u$, $\mathbb{K}$, $K_c$, $K_s$, $K_r$ |
| Add/remove administrator | $m$, $K_m$ |
| Backup/restore | $\mathcal{C}$ |

**Table 2: Information recorded for each log event.**

The administrators can then scan this code with their verifiers to obtain, verify, and distribute the digital certificate.

# 6. DEPLOYMENT AND MANAGEMENT

While certificate signing is the main function of CASTLE, several other functions are critical to its operation in practice. In this section, we discuss four such functions: logging, initialization, adding and removing administrators, and the backup and restore process.

## 6.1 Logging

For auditability, the PAL logs all operations that modify the system state (*i.e.*, the configuration BLOB) at the signer. Each operation is recorded as a *log event* that records the attempted operation, whether or not the operation was successful, and information further describing the specifics of the operation. We thus define a log event $L_i$ as follows:

$$L_i = \texttt{<success/failure>} \| \texttt{<operation>} \| I_i \tag{5}$$

where $I_i$ for each operation is shown in Table 2.

To prevent tampering with the log, the logged events are stored in a chronologically-ordered hash chain similarly to timestamping protocols [7]. As shown in Figure 4, the epoch identifier after an event $L_i$ is

$$h_i = H(h_{i-1} \| H(L_i)) \tag{6}$$

where $h_{i-1}$ is the latest epoch identifier before the current event is logged. Each time the epoch identifier changes, the PAL writes the latest value $h_i$ into NVRAM in order to prevent replay attacks with old system state [18].

An administrator or other auditor can synchronize with the log by performing a *log check* at the PAL. The auditor provides a nonce $N_v$ to the PAL and an epoch identifier $h_v$. The PAL returns a signature on the events since $h_v$:

$$\{L_i : v < i \leq \ell\}, \{h_\ell, N_v\}_{K_c^{-1}}. \tag{7}$$

The signer then displays this information as a QR code, where it can be scanned and verified.

The latest epoch identifier is also included in the output of each PAL operation, along with a signature. This allows the administrators to keep the most recent epoch identifier, which is required to initiate a certificate signature session.

## 6.2 System Initialization

Before a CASTLE system can sign certificates, it must first have its own keys as well as those of its administrators. We achieve this via an initialization process that is carried out by the group of initial administrators. We assume that there are $m$ administrators, and that each has a key pair $K_i, K_i^{-1}$ stored on a verifier as well as a nonce $M_i$ used during setup. We also assume that the administrators have knowledge of a public key $K_\varepsilon$ whose corresponding private key $K_\varepsilon^{-1}$ is stored in and only accessible to the TPM, such as the endorsement key or attestation identity key, and agree on a threshold $k$ for authorizing certificate signatures and a threshold $u$ for management operations. Finally, we assume that the PAL code is unmodified and available to the TPM.

**Algorithm 4** PAL initialization procedure.

```
1:  procedure INIT
    Input: M_1, ..., M_m, m, k, u, K_1, ..., K_m
2:      h_0 ← generate random value
3:      K_b ← generate symmetric key
4:      write h_0 into NVRAM
5:      B ← K_b sealed to PAL
6:      write H(K_b) into NVRAM
7:      M ← ⊕_{i=1}^{m} M_i
8:      Q ← TPM quote with nonce M on PAL PCRs
9:      I ← Enc_{K_b}(h_0‖m‖k‖u‖K_1‖...‖K_m)
10:     σ ← Sign_{K_ε^{-1}}(M‖Q‖h_0‖m‖k‖u‖K_1‖...‖K_m)
11:     return M, Q, h_0, B, I, σ
```

**Algorithm 5** PAL state generation procedure.

```
1:  procedure STATE_GEN
    Input: S_1 ..., S_m, B, I
2:      K_b ← unseal B and check against NVRAM
3:      h_0, m, k, u, K_1, ..., K_m ← Dec_{K_b}(I)
4:      check h_0 against NVRAM
5:      for 1 ≤ i ≤ k do
6:          if !Vrfy_{K_i}(h_0, S_i) then
7:              return 0
8:      𝕂 ← {K_1, ..., K_m}
9:      K_c, K_c^{-1} ← generate asymmetric key pair
10:     K_s, K_s^{-1} ← generate asymmetric key pair
11:     K_r, K_r^{-1} ← generate asymmetric key pair
12:     h_1 ← H(h_0‖success‖state_gen‖m‖k‖u‖𝕂‖K_c‖K_s‖K_r)
13:     write h_1 to NVRAM
14:     C ← Enc_{K_b}(h_1‖m‖k‖u‖𝕂‖K_c^{-1}‖K_s^{-1}‖K_r^{-1})
15:     σ ← Sign_{K_ε^{-1}}(h_1‖𝕂‖K_c‖K_s‖K_r‖m‖k‖u)
16:     return h_1, K_c, K_s, K_r, C, σ
```

Each administrator uses a verifier to generate and scan a QR code encoding

$$M_i, m, k, u, K_i \tag{8}$$

at the signer. The PAL then performs the *initialization step*, which is shown in Algorithm 4. In the initialization step, the PAL creates an initial epoch identifier and base key, and writes these values into NVRAM to tamper-proof them. The PAL also seals the base key to the PCR values containing the Flicker core and PAL. To attest to the PAL that is running, the signer invokes a TPM quote on the PCRs measuring the PAL, using $\oplus_{i=1}^{m} M_i$ as a nonce (preventing any single administrator from biasing the nonce). The PAL then creates an *initialization BLOB* $\mathcal{I}$, which holds the number of administrators, thresholds for signature and management operations, and administrator public keys. The PAL then returns the above information with a signature by the non-migratable secret key $K_\varepsilon^{-1}$.

The administrators can then review this information to confirm that the PAL received the correct information. Once this check has passed, each administrator creates a *setup signature*, defined as $S_i = \mathsf{Sign}_{K_i^{-1}}(h_0)$, which authorizes the remainder of system setup. The administrators scan their setup signatures at the signer, which passes them to the PAL along with $\mathcal{B}$ and $\mathcal{I}$.

The PAL then performs *state generation*, shown in Algorithm 5. In this step, the PAL checks the setup signatures to ensure that the administrators have authorized the remainder of setup, and also en-

sures that the initial epoch identifier matches the value in NVRAM (thus preventing replays of old setup signatures). If these checks pass, the PAL generates the attestation, signature, and restoration key pairs and creates the configuration BLOB $\mathcal{C}$ with the newly-generated public keys and information from the initialization BLOB. Finally, the PAL logs and signs the public information in $\mathcal{C}$ and returns this information along with $\mathcal{C}$ (which is encrypted with $K_b$). The verifiers can use this information to check the signature $\sigma$ and store the public keys $K_c$, $K_s$, and $K_r$.

## 6.3 Administrator Management

Adding or removing an administrator requires agreement by at least $u$ administrators. The new administrator $A_{m+1}$ must have a key pair and verifier like all other administrators. We assume that the $u$ administrators all know $K_{m+1}$ and the latest epoch identifier $h_\ell$, which links the operation to a specific point in time.

The quorum of administrators each contribute a message of the form

$$\mathsf{Sign}_{K_i^{-1}}(h_\ell, \text{“add,”}\ K_{m+1}) \tag{9}$$

and scan these messages at the signer. The PAL then checks each signature, and if all signatures are correct, adds $K_{m+1}$ to $\mathbb{K}$ and logs the event, returning the new epoch identifier $h_{\ell+1}$.

The administrators can remove an existing administrator by providing a message of the form in Equation 9, but by stating “remove” instead of “add” along with the public key of an existing administrator. The PAL then removes the public key from $\mathbb{K}$. The thresholds $k$ and $u$ are also decremented along with $m$ if they are equal to the new number of administrators $m-1$.

Administrators can also initiate other operations, such as changing parameters or signing keys, by signing messages in the form of Equation 9, but with a string stating “change param/key” and identifying the parameter or key being changed, and if changing a parameter, the new value of the parameter. The PAL then checks the signatures and makes the change if there are at least $u$ valid signatures authorizing the change. If any keys, such as $K_b$, $K_c^{-1}$, or $K_s^{-1}$ are changed, the PAL generates a new key rather than having the administrators specify a new value (since these values must be known only to the PAL).

## 6.4 System Backup and Restoration

We now describe how a CASTLE system can be securely backed up and restored. In particular, we back up the information stored at the signer. The backup material consists of the base BLOB $\mathcal{B}$ and the main BLOB $\mathcal{C}$. We use the term *source machine* to refer to the signer being backed up, and the term *target machine* to refer to the new signer restoring the backup. In CASTLE, a backup from the source machine entails a restore to the target machine and requires a quorum of administrators at each machine. We use the restoration key pair of the target machine to encrypt the backup in transit and ensure that the contents of $\mathcal{B}$ and $\mathcal{C}$ are not accessible, even to the administrators that carry out the process.

Restoring a backup onto a target machine first requires the target machine to be initialized as described in Section 6.2 and an identical quorum of $u$ administrators to be registered at both machines. To achieve this in a setup where the target machine is already initialized with a different set of administrators, the target machine's administrators and threshold can be adapted to those of the source machine as described in Section 6.3 before starting the backup process. The restoration public key $K_r'$ of the target machine is publicly available to its own administrators, having been provided during the state generation step described in Algorithm 5.

The $u$ administrators initiate the backup at the source machine by

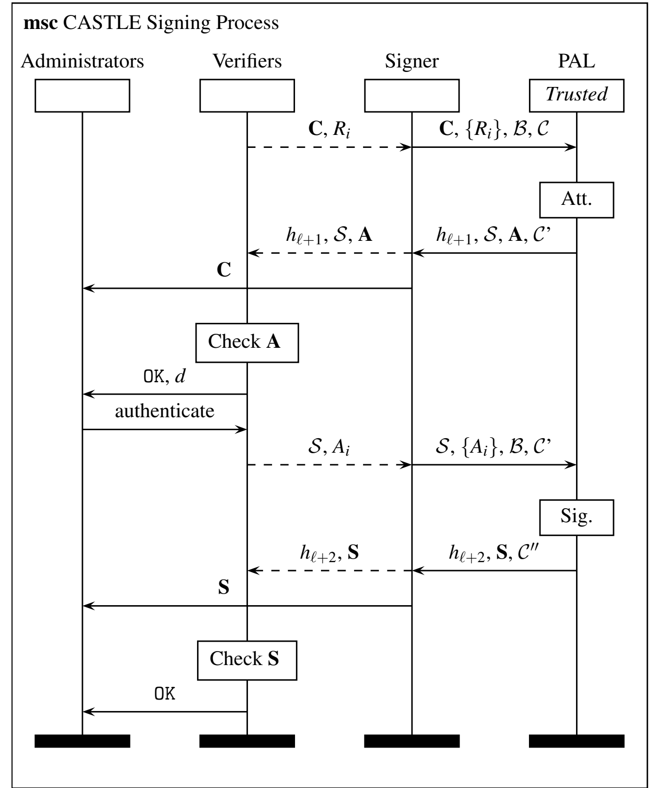


**Figure 5: Message sequence chart for a signing session. Dotted lines indicate messages encoded as QR codes.**

scanning QR codes encoding a signature on $K_r'$. Each participating administrator thus sends a message of the form $\mathsf{Sign}_{K_i^{-1}}(h_\ell, K_r')$, which the PAL responds to with the encrypted backup

$$\mathcal{E} = \mathsf{Enc}_{K_r'}(K_b, \mathcal{C}) \tag{10}$$

The PAL also returns a signature on the epoch and backup:

$$\mathsf{Sign}_{K_c^{-1}}(h_\ell, \mathcal{E}) \tag{11}$$

This information is displayed by the source signer as a QR code. Using $K_c$, administrators at both the source and target machines can verify that the signature is correct and thus that the backup was created by the source PAL. The administrators can then scan signatures of the form $\mathsf{Sign}_{K_i^{-1}}(h_\ell', \mathcal{E})$ at the target machine, which checks the $u$ signatures and overwrites its own configuration with the information provided in $\mathcal{E}$. The state of the source machine is thus copied to the target machine.

# 7. ANALYSIS

In this section, we perform an informal analysis of a certificate signing session and argue that under the limitations of the adversary model described in Section 3, the adversary cannot cause a certificate to be issued. We note that even for an adversary with capabilities beyond those listed in Section 3, the event will still be logged and can be audited in the future. We plan on conducting a formal analysis in future work.

## 7.1 Security of Certificate Signing

We now argue for the security of a certificate signing session by showing that an adversary must compromise at least $k$ administra-

tors or verifiers to issue an unauthorized certificate. Consider the messages sent by the signer as shown in Figure 5. A compromised signer can forge or replay messages, and because it directly handles the input and output of the PAL, can attempt to modify these messages in order to obtain a signed certificate from the PAL. However, we observe that without access to administrator private keys, a compromised signer cannot forge or replay any message that will be accepted by its recipient in the protocol. We assume that each administrator knows the latest epoch identifier $h_\ell$ and the set of other administrators participating in the session.

In the request step, the signer cannot send a different CSR without altering the corresponding request signatures, and cannot replay previous CSRs with their request signatures because the request signatures contain the epoch identifier at the time of the session. The signer also cannot forge BLOBs because the base key is sealed to and thus only accessible to the PAL, and cannot replay BLOBs because they too are bound to the epoch identifier in NVRAM, which is also accessible only to the PAL. Thus the signer cannot modify any messages in the request step.

After the attestation step, the signer cannot forge the new epoch identifier, since it is included in the attestation signature (which only the PAL can create). The signer also cannot replay old epoch identifiers and attestation signatures, since the administrators can easily verify the log entry, given that they know the other participating administrators by assumption, and $\mathbf{C}$ and $h_\ell$ from creating the request signature in the previous step. Thus each administrator can verify that the epoch identifier $h_{\ell+1}$, session BLOB $\mathcal{S}$, and attestation signature $\mathbf{A}$ are the correct output for the given CSR and starting epoch identifier $h_\ell$.

In the authorization step, the signer cannot forge the session, base, or configuration BLOBs because each BLOB is bound to the base key to which only the PAL has access. Similarly, the signer cannot replay the BLOBs because they are all bound to the epoch identifier $h_{\ell+1}$ written into NVRAM. Finally, the signer cannot forge the authorization signatures $A_i$ without access to administrator private keys, and cannot replay the authorization signatures because they contain the session BLOB, which cannot be forged or replayed as explained above.

Finally, after the signature step, the signer cannot forge or replay the final epoch identifier $h_{\ell+2}$ because the administrators can again check the log entry given $h_{\ell+1}$, the set of participating administrators, and the signature $\mathbf{S}$. Similarly, the signer cannot forge the signature $\mathbf{S}$ without access to the signing key $K_s^{-1}$, and cannot replay old signatures because the administrators know the CSR $\mathbf{C}$ used in the session.

We observe, however, that by compromising administrators or verifiers, the adversary can gain access to administrator private keys needed to create request and authorization signatures. In particular, with $k$ administrators or verifiers compromised, an adversary can generate the number of request signatures and authorization signatures necessary to proceed with the entire signing session for a given CSR. However, the adversary *must* compromise at least $k$ administrators or verifiers, and thus within the limitations of our adversary model, therefore cannot obtain an unauthorized signature on a certificate.

## 7.2 Security of Management Operations

For management operations, we observe that adding or removing administrators, changing parameters, or backing up or restoring a signer requires at least $u$ administrators to provide a signature authorizing the operation. Because we assume that an adversary can compromise a maximum of $u-1$ administrators and verifiers, it can only gain access to a maximum of $u-1$ administrator private keys. Thus an adversary with the limitations of our model given in Section 3 cannot authorize these management operations.

| | SLOC |
|---|---|
| Flicker kernel module | 5,521 |
| Flicker helpers | 4,789 |
| mbed TLS | 6,097 |
| PAL | 1,115 |
| **TCB Total** | 17,522 |
| mbed TLS base 64 utilities | 652 |
| Signer front-end | 2,656 |
| Crypto, I/O libraries (OpenSSL, ZBar, qrencode) | 143,621 |
| **Untrusted Total** | 146,929 |
| EJBCA Modules | 182,716 |
| EJBCA Core | 43,753 |
| **EJBCA Total** | 226,469 |

**Table 3: Source lines of TCB and untrusted code of CASTLE versus that of EJBCA.**

# 8. EVALUATION

In this section, we evaluate our prototype implementation of CASTLE. We begin by describing the implementation itself, and then discuss the performance results of the system. We then briefly discuss the cost of the prototype.

## 8.1 Implementation

We implemented our signer prototype in C. The signer application runs on a 3.6GHz Intel Core i7-4790 machine with 3GB of RAM and support for Intel's Trusted Execution Technology [5]. As verifiers will likely be mobile devices carried with administrators, we implemented the verifier in Java as an Android application. We used a variety of off-the-shelf libraries for features such as QR code detection and the GUI interface, and we note in particular that we used OpenSSL for cryptographic operations, except in the PAL, where we utilized the mbed TLS library[2] (formerly PolarSSL) for its compact code.

For the purposes of the prototype, some features were simplified or omitted. For example, CSRs are passed as hashes to be signed rather than in PEM format, and signer parameter changes were not implemented. However, sealed storage and NVRAM are fully implemented and our prototype works with X.509v3 [2] certificates (though it is not fully compatible with the standard). We will make our code open-source so that existing CAs can begin deploying CASTLE.

We measured the source lines of code of our prototype implementation, and the results are shown in Table 3, separated into code that is part of the TCB and code that is not. We further compare this to the EJBCA open-source CA code base [19], developed by PrimeKey. We measured the source lines of code in EJBCA Community 6.3.1.1, and also present these results in the table.

We observe that a majority of the CASTLE TCB consists of Flicker and the mbed TLS library. In fact, the PAL itself is a small part, comprising only 6.4% of the TCB. Similarly, the size of the signer front-end is dwarfed by the off-the-shelf libraries, particularly OpenSSL. Further optimization or a different choice of libraries could shrink the size of both the TCB and untrusted portions, but the CASTLE portion of the code base will likely remain below 10,000 lines of code, even with future extensions.

When compared to EJBCA's code base, we observe that CASTLE takes fewer lines of code than the EJBCA implementation does. Moreover, the CASTLE implementation has a limited TCB,

[2] `https://tls.mbed.org/`

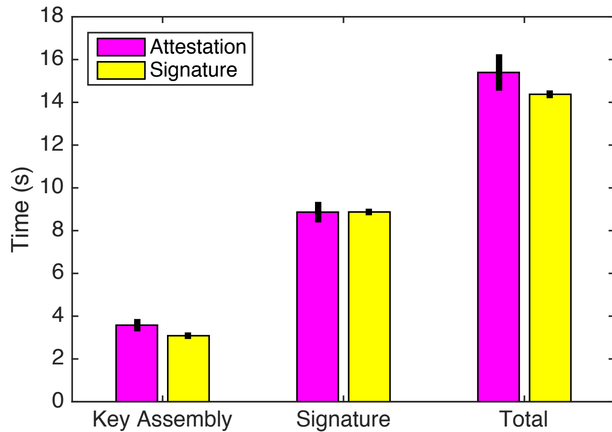


**Figure 6: Running time for attestation and signature functions. Key assembly corresponds to the time required to calculate the private key description according to PKCS#1 [11].**

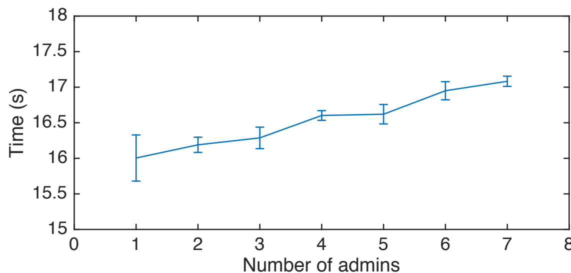


**Figure 7: PAL run times for initialization given the number of participating administrators. The error bars are standard error.**

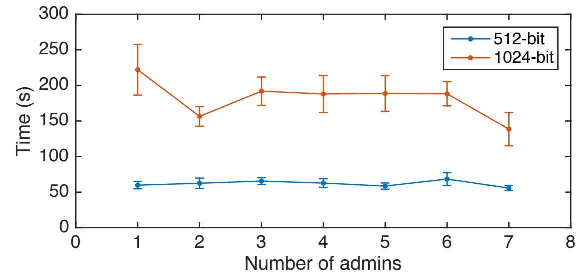


**Figure 8: PAL run times for state generation given the number of participating administrators and the key size. State generation for 2048-bit keys took around 40 minutes and are thus not shown here. The error bars are standard error.**

whereas in EJBCA all code must be trusted. We note that simply comparing lines of code does not necessarily denote an advantage of CASTLE over EJBCA (in particular, EJBCA is implemented in Java while CASTLE is implemented in C), but unless EJBCA is used in conjunction with an HSM, it cannot offer similar protection for its operations and private signing keys as CASTLE does.

## 8.2 PAL performance

To evaluate the performance of the Flicker sessions, we measured the running time of the attestation, signature, and initialization processes. For each PAL function, we measured 10 runs of execution. For the attestation and signature, we used a single administrator but measured the relative times of the key assembly and the creation of the attestation or signature, respectively. For initialization and state generation, we measured the effect of administrators on the running time. Figure 6 shows the results for the attestation and signature functions, Figure 7 shows the results for initialization, and Figure 8 shows the results for state generation.

Attestation took longer than signature on average, likely due to the extra operation of creating the session BLOB. We also observe that the majority of execution time consists of key assembly (calculating large helper numbers) along with the attestation signature or certificate signature. This indicates that the overhead of starting and ending the Flicker sessions and the remainder of the PAL logic is a small portion of the run time (around 2.5s for each mode). In initialization, we observed that there was a small effect of the number of administrators on the running time. In state generation, we observed that the number of administrators has no significant effect on the running time. This is due in part to the fact that state generation, which must obtain randomness from the TPM to generate three RSA public keys, has a running time on the order of minutes as opposed to seconds for the other operations.

We acknowledge that the duration of the Flicker sessions are long compared to standard signature processes, sometimes taking longer than 15 seconds. We could reduce the running time of the PAL by storing the private key descriptions in PKCS #1 format in $\mathcal{S}$, eliminating the key assembly overhead, and by optimizing the mbed TLS library for performance, reducing the time required for an attestation or certificate signature. However, we emphasize that the system is already usable in its current instantiation, and the few seconds of waiting time are acceptable for conscripted CAs, who issue certificates at a much lower volume than commercial CAs.

## 8.3 Signature Session Performance

To estimate the running time of a signature session with trained administrators, we performed several signature sessions. In a certificate signing session, an administrator must scan a QR code at the request, attestation, authorization, and signature steps. Thus each administrator scans or displays a total of 4 QR codes over the course of a session. However, there are $k$ administrators for each session, and QR codes must be scanned sequentially to or from the signer. Thus a total of $4k$ QR codes are exchanged between the signer and administrators.

At the request step (and in each step where sequential scanning of QR codes is required), sequentially scanning QR codes took approximately 5 seconds per administrator on average, and as shown above, attestation took about 15 seconds on average. During authorization, finding the hidden digit and authenticating took 1.5 minutes on average, assuming careful checking of the CSR contents. Finally, the signature took about 14 seconds on average.

Scanning the certificate signature and exporting the certificate took 30 seconds on average. Thus a full signature procedure took an average of approximately $150 + 20k$ seconds. Even with careful checking in a production environment, we do not expect a full session to take more than 5 minutes. For high-value certificates, we argue that such a latency is reasonable, considering that EV certificates require an in-person meeting and can take days.

## 8.4 Cost

The physical construction of the machine, monitor, webcam, and

glass box was around US$2000, where the manufacturing of the glass box accounts for roughly $1000. We anticipate that these costs can be reduced in a larger-scale production, but even so, a cost of $2000 should be within CA means, given that purchasing a certificate costs on the order of $1000.

We anticipate that the majority of the cost of deploying CASTLE will stem from the training and salary of administrators. Due to the low volume of certificate issuances we expect from conscripted CAs, existing administrators could take on signing session duties in CASTLE. We note that the protocol requires at least two (and ideally three) administrators to secure the signature and management operations against a single misbehaving administrator.

# 9. DISCUSSION

In this section, we briefly discuss several important aspects of CASTLE. In particular, we address several design alternatives for CASTLE in practice. We then acknowledge limitations of CASTLE and outline our future work.

**Certificate Revocation.** The current version of CASTLE does not handle certificate revocation. CASTLE can support various revocation systems such as CRLs [2] or OCSP [20], but the use of an air-gapped signer would limit the frequency of interactions with the signer machine to authorize a revocation. We leave the detailed design and implementation of such a mechanism to future work.

**Limitations.** One limitation of CASTLE is that BLOBs and log entries can be modified (in encrypted form) or destroyed (though detectably so). To address this weakness, we could add an extra step to each signature operation that requires the administrator to confirm that other administrators have received a record of the signature before the final certificate signature is provided. Another limitation is the lack of formal verification of our protocols. We plan to address this limitation in future work.

**Deploying CASTLE.** In future work, we plan to conduct additional work to assess and improve the operation of CASTLE in practice. In particular, we plan to carry out a survey among both full-time and conscripted CAs to determine the relative costs of administrators, types of hardware and software, security of their physical facilities, and history of certificate misissuance. The results of this survey would provide us with an overview of the causes of certificate misissuance at both full-time and conscripted CAs, and provide us with a realistic estimate of the cost of deploying CASTLE in a conscripted CAs.

We also plan to conduct a comprehensive usability test of the CASTLE software with domain experts (i.e., administrators at conscripted CAs). This testing will allow us to improve the ease of use of CASTLE for administrators, and we anticipate that such improvements will lead to fewer operational errors on the administrators' part. Finally, we plan to perform further optimizations in the code to improve performance and security. In particular, we plan to harden the QR code and certificate processing libraries, which are critical pieces of our current signer prototype.

**Related Work.** Little related work on hardware-secured CA signing exists besides HSMs, though some proposals leverage trusted computing for authentication [4, 12], key management [25], and replay protection [18]. These offer similar functionality to that of CASTLE, but often with a larger TCB. For example, KISS [25] uses devices carried by administrators similar to verifiers, but all of these devices must be trusted. Other work has attempted to simplify the PKI signing process [6], but for end-users rather than CAs.

Several open-source projects offer code for different CA functionality. For example, OpenCA[3] offers code for an OCSP responder called OCSPD, while PrimeKey's EJBCA [19] offers a full CA application. EJBCA can be run in a virtual machine or make use of an HSM. EJBCA includes a CA, validation authority (to validate certificates), and an OCSP responder. However, while EJBCA offers the ability to use secure hardware such as HSMs and smart cards, CAs must still purchase the secure hardware and design their administrative processes.

# 10. CONCLUSION

Our layered defense-in-depth design for CASTLE shows that we can leverage a diverse arsenal of defenses to secure certificate signing and management for low-volume conscripted CAs. CASTLE is easy to use for entities who can follow operating procedures and provide physical security, and thus provides a much-needed step towards improving security, ease of use, and economic operation for conscripted CAs.

# Acknowledgments

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013), ERC grant agreement 617605, and the National Science Foundation, Grant DGS1252522. We also gratefully acknowledge support from ETH Zurich and from the Zurich Information Security and Privacy Center (ZISC).

We graciously thank Magnetron Labs Merz for production of the glass box prototype. We also thank David Barrera and Daniele Asoni, who provided feedback on drafts of the paper, and the anonymous reviewers, whose feedback helped to improve the paper.

# 11. REFERENCES

[1] TPM main specification level 2 version 1.2, revision 116. Trusted Computing Group (March 2011)

[2] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., Polk, T.: Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. RFC 5280 (May 2008)

[3] Dillow, C.: An order of seven global cyber-guardians now hold keys to the Internet. http://www.popsci.com/technology/article/2010-07/order-seven-cyber-guardians-around-world-now-hold-keys-internet (July 2010)

[4] Gajek, S., Löhr, H., Sadeghi, A.R., Winandy, M.: Truwallet: trustworthy and migratable wallet-based web authentication. In: ACM Workshop on Scalable Trusted Computing (STC). pp. 19–28. ACM (2009)

[5] Greene, J.: Intel trusted execution technology. White paper (2012)

[6] Gutmann, P.: Plug-and-play PKI: A PKI your mother can use. In: 12th USENIX Security Symposium. USENIX (2003)

[7] Haber, S., Stornetta, W.: How to time-stamp a digital document. Journal of Cryptology 3(2), 99–111 (1991), http://dx.doi.org/10.1007/BF00196791

[8] Hoekstra, M.: Intel SGX for dummies (Intel SGX design objectives). https://software.intel.com/en-us/blogs/2013/09/26/protecting-application-secrets-with-intel-sgx (September 2013)

[9] Hoogstraaten, H., Prins, R., Niggebrugge, D., Heppener, D., Groenewegen, F., Wettink, J., Strooy, K., Arends, P., Pols, P., Kouprie, R., Moorrees, S., van Pelt, X., Hu, Y.Z.: Black

[3] `https://www.openca.org/`

Tulip: Report of the investigation into the DigiNotar certificate authority breach. www.rijksoverheid.nl/bestanden/documenten-en-publicaties/rapporten/2012/08/13/black-tulip-update/black-tulip-update.pdf (August 2012)
[10] Jacobson, V., Smetters, D.K., Thornton, J.D., Plass, M.F., Briggs, N.H., Braynard, R.L.: Networking named content. In: ACM CoNEXT (December 2009)
[11] Jonsson, J., Kaliski, B.: PKCS #1: RSA cryptography specifications version 2.1. RFC 3447 (February 2003)
[12] Kostiainen, K., Ekberg, J.E., Asokan, N., Rantala, A.: On-board credentials with open provisioning. In: 4th International Symposium on Information, Computer, and Communications Security (ASIACCA). pp. 104–115. ACM (2009)
[13] Langley, A.: Enhancing digital certificate security. http://googleonlinesecurity.blogspot.ch/2013/01/enhancing-digital-certificate-security.html (January 2013)
[14] Langley, A.: Maintaining digital certificate security. http://googleonlinesecurity.blogspot.co.uk/2015/03/maintaining-digital-certificate-security.html (March 2015)
[15] McCune, J.M., Parno, B.J., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: An execution infrastructure for TCB minimization. In: ACM SIGOPS Operating Systems Review. vol. 42, pp. 315–328. ACM (2008)
[16] Naylor, D., Mukerjee, M.K., Agyapong, P., Grandl, R., Kang, R., Machado, M.: XIA: Architecting a more trustworthy and evolvable Internet. In: ACM SIGCOMM Computer Communication Review (CCR). vol. 44. ACM (July 2014)
[17] Nystrom, M., Kaliski, B.: PKCS #10: Certification request syntax specification. RFC 2986 (November 2000)
[18] Parno, B., Lorch, J.R., Douceur, J.R., Mickens, J., McCune, J.M.: Memoir: Practical state continuity for protected modules. In: IEEE Symposium on Security and Privacy (SP). pp. 379–394. IEEE (2011)
[19] PKI, P.: EJBCA PKI CA. https://www.ejbca.org/ (June 2015)
[20] Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., Adams, C.: X.509 Internet public key infrastructure online certificate status protocol - OCSP. RFC 6960 (June 2013)
[21] Sara Dickinson, R.v.R.: HSM buyers' guide. https://wiki.opendnssec.org/display/DOCREF/HSM+Buyers%27+Guide (August 2012)
[22] Sleevi, R.: Sustaining digital certificate security. https://googleonlinesecurity.blogspot.com/2015/10/sustaining-digital-certificate-security.html (October 2015)
[23] Zetter, K.: PIN crackers nab holy grail of bank card security. http://www.wired.com/2009/04/pins/ (April 2009)
[24] Zhang, X., Hsiao, H.C., Hasker, G., Chan, H., Perrig, A., Andersen, D.G.: SCION: Scalability, control, and isolation on next-generation networks. In: Security and Privacy (SP), 2011 IEEE Symposium on. pp. 212–227. IEEE (May 2011)
[25] Zhou, Z., Han, J., Lin, Y.H., Perrig, A., Gligor, V.: KISS: "Key It Simple and Secure" corporate key management. In: Trust and Trustworthy Computing, pp. 1–18. Springer (2013)