Keys in the Clouds: Auditable Multi-device Access to Cryptographic Credentials

Arseny Kurnikov
Aalto University, Finland
arseny.kurnikov@aalto.fi

Andrew Paverd
Aalto University, Finland
andrew.paverd@ieee.org

Mohammad Mannan
Concordia University, Canada
m.mannan@concordia.ca

N. Asokan
Aalto University, Finland
asokan@acm.org

ABSTRACT

Personal cryptographic keys are the foundation of many secure services, but storing these keys securely is a challenge, especially if they are used from multiple devices. Storing keys in a centralized location, like an Internet-accessible server, raises serious security concerns (e.g. server compromise). Hardware-based Trusted Execution Environments (TEEs) are a well-known solution for protecting sensitive data in untrusted environments, and are now becoming available on commodity server platforms.

Although the idea of protecting keys using a server-side TEE is straight-forward, in this paper we validate this approach and show that it enables new desirable functionality. We describe the design, implementation, and evaluation of a TEE-based Cloud Key Store (CKS), an online service for securely generating, storing, and using personal cryptographic keys. Using remote attestation, users receive strong assurance about the behaviour of the CKS, and can authenticate themselves using passwords while avoiding typical risks of password-based authentication like password theft or phishing. In addition, this design allows users to i) define policy-based access controls for keys; ii) delegate keys to other CKS users for a specified time and/or a limited number of uses; and iii) audit all key usages via a secure audit log. We have implemented a proof of concept CKS using Intel SGX and integrated this into GnuPG on Linux and OpenKeychain on Android. Our CKS implementation performs approximately 6,000 signature operations per second on a single desktop PC. The latency is in the same order of magnitude as using locally-stored keys, and 20x faster than smart cards.

CCS CONCEPTS

• **Security and privacy** → **Key management**; *Authentication*; *Trusted computing*;

KEYWORDS

Cloud, Intel SGX, Key management

1 INTRODUCTION

Personal cryptographic keys are the foundation of many secure services, such as signing or decrypting emails, signing code, authenticating to remote servers, or decrypting cloud storage. However, storing personal cryptographic keys *securely* often proves to be difficult for users, especially when keys must be usable from multiple devices (e.g. PCs, smartphones, and tablets). Although we already have various approaches for protecting personal cryptographic keys, these have various limitations:

Password only: Keys stored directly on the user's device may be compromised by malicious software on the device. Even if the keys are protected by a password, either the password or the decrypted key could be captured by malicious software (see e.g. [33]).

Device key store: Storing keys in e.g. the hardware-backed Android or iOS key store protects them from malicious software, but this type of device key store may not be available on all the user's devices.

External peripherals: Smart cards or USB tokens (e.g. Trezor [38] and YubiKey [42]) can protect keys from malicious software whilst being usable from multiple devices. However, they require additional hardware or peripherals (e.g. smart card reader), which incur costs and may be incompatible with some devices.

It is also important to consider the *availability* of the keys: keys that only exist on a single device (e.g. generated and used exclusively within a hardware-backed device key store or smart card) would become unavailable if the device/smart card is lost or damaged.

Storing personal cryptographic keys in a centralized location (e.g. an Internet-connected server) solves many of the above challenges: users can authenticate and use their keys from any Internet-connected device, and centralized servers generally have lower risk of loss or failure than individual users' devices. However, the principal challenge is how to *protect the keys* against external attackers, other users, and even malicious server administrators.

Password-based encryption: A naive approach is to encrypt keys using the user's password and download the encrypted keys to the user's devices when needed. However, since passwords are generally weak secrets [5], this does not provide sufficient protection

against an adversary who can obtain the encrypted keys (e.g. via a compromised server) and will eventually be able to guess many users' passwords. Furthermore, the decrypted keys would still be vulnerable to any malicious software on the user's device.

Key splitting: Another approach is to split a key into two (or more) shares, one of which is held by the server and another by the user's device (e.g. [2, 11, 26]). To use the key, both the server and the user's device must cooperate. Although this approach protects the keys against server compromise, it entails a relatively complicated process in order to use the key from a new device (e.g. a share of the key must be securely transferred to the new device).

Trusted Execution Environment (TEE): Hardware-enforced TEEs, like Intel SGX [20] and ARM TrustZone [36], can be used to isolate and protect a small amount of trustworthy code and data from all other software on the machine, including the OS and hypervisor. Furthermore, by using *remote attestation*, a TEE can provide strong assurance to remote parties about precisely what code it is running, and can establish a secure communication channel. Server-side TEEs are therefore a promising solution for protecting cryptographic keys on a centralized server, and are now becoming available in commodity server hardware [19].

We present the design, implementation, and evaluation of a *Cloud Key Store* (CKS) in which keys are generated, stored, and used exclusively within a server-side TEE. We refer to these as *protected keys*. Using remote attestation, the user receives strong assurance that she is communicating with a legitimate TEE, and can establish a secure communication channel directly to the TEE. The user authenticates herself to the TEE by sending her username and password via this channel. Password-based authentication is still the most widespread user authentication method [6] and can be used from any of the user's devices. By rate-limiting the authentication attempts for each username, the TEE prevents password guessing attacks, even from a compromised server. The user submits requests to the CKS, which performs the requested operations inside the TEE, and returns the results via the same channel.

In addition to storing keys securely and providing access from multiple devices, the CKS also enables the following new features:

- **Policy-based access control:** The key owner can restrict key use to a specific time period and/or number of uses.
- **Key delegation:** Users can delegate access to their keys to other users of the same CKS for either a specific time period and/or number of uses.
- **Key usage auditing:** Every operation performed using the protected keys can be logged by the CKS, and a user can audit these logs to detect any misuse of the protected keys.

Our contributions are as follows:

- We design and fully implement a Cloud Key Store (CKS) that supports password-based user authentication, secure key generation, decryption and signature operations, policy-based access control, key delegation, and key usage auditing (Sections 4 and 5). Our implementation uses off-the-shelf hardware, and is available as open source software.[1]
- We demonstrate the functionality of our CKS by integrating it into two applications: GnuPG on Linux and OpenKeychain on Android (Section 5), which we also provide as open source software.[1] The core CKS operations do not require any changes

in the applications' UI. The new CKS features (i.e. delegation and auditing) are provided via a web interface, but in future could also be integrated into the applications.

- We evaluate the security and performance of our CKS implementation, using both GnuPG on Linux and OpenKeychain on Android (Section 6). On a single desktop CPU, our CKS implementation can perform approximately 6,000 signature operations per second. The average latency is in the same order of magnitude as using an unprotected key on the local device, and 20x faster than using a smart card.
- Finally, we present a comparison of different approaches for managing personal cryptographic keys in terms of their security and functionality (Section 7). This shows that our CKS outperforms alternative approaches in multiple aspects, whilst providing additional desirable features.

2 BACKGROUND

2.1 Intel SGX

Intel Software Guard Extensions (SGX) [20] is one recent instantiation of a TEE. SGX is a set of CPU extensions that allow applications to define one or more *enclaves* that are protected from all other software on the platform, including the operating system (OS) and hypervisor. Data within an enclave can only be accessed from code running within the enclave. The CPU automatically encrypts enclave memory before it leaves the boundary of the CPU package (e.g. is written to DRAM) [18]. This protects the confidentiality and integrity of the enclave's data, even against an adversary with access to the platform hardware. An untrusted application can invoke functions within an enclave through pre-defined ECALLs. Enclaves can call functions from the untrusted application via OCALLs.

When an enclave needs to persist data, it can encrypt the data using a CPU-protected key, called a *sealing key* exclusively available to that specific enclave. This *sealed data* can be safely stored outside the enclave. SGX provides *monotonic counters* to prevent roll-back attacks on sealed data. Finally, SGX provides *remote attestation*, a process through which a remote relying party (the *verifier*) can identify the precise code running inside an enclave (the *prover*), and can establish a secure channel directly to the enclave. Since well-designed enclaves contain only the minimal amount of code necessary for their intended functionality, this code can be audited or otherwise analysed by the verifier (or a third-party of the verifier's choosing) in order to determine that the code is trustworthy.

2.2 Pretty Good Privacy (PGP)

Pretty Good Privacy (PGP) [43], and the associated OpenPGP standard (RFC4880) [8], provide a set of algorithms and procedures to encrypt and/or authenticate data. OpenPGP is the most widely used standard for encrypting and signing e-mail communications.

A major challenge in PGP is to establish a trustworthy mapping between public keys and real-world users. This is usually achieved using a PKI or a web of trust: a reputation system in which trusted users vouch for the public keys of other users. Since building up trust through this type of system takes time, revoking a key and re-establishing trust in a new key incurs a relatively high cost.
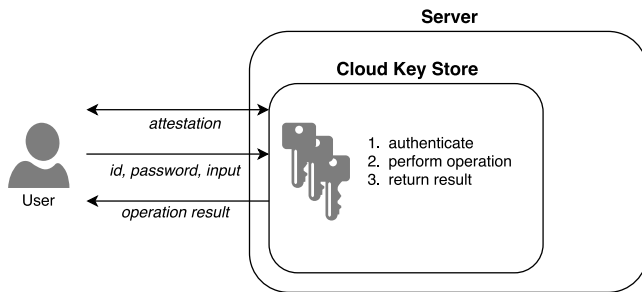
---

[1]https://github.com/cloud-key-store

Figure 1: Cloud Key Store architecture.

If an attacker gets access to the user's private key, he can decrypt past e-mails sent to the user, and impersonate the user by generating falsified signatures. The only way to prevent further damage is to revoke the key. For this reason, users are recommended to instead use subkeys for day-to-day operations, since individual subkeys can be revoked without affecting the main user identity.

GnuPG [15] is an open-source implementation of the OpenPGP standard. By default, GnuPG stores encrypted private keys on the user's local device, but it can also use keys stored on a smart card or similar device.

3 SYSTEM MODEL AND REQUIREMENTS

3.1 System Model

Figure 1 shows the abstract system model of our CKS running in a Trusted Execution Environment (TEE) on a remote server. As we discuss in Section 8, the CKS can also be distributed across multiple physical or virtual servers.

Core functionality: The CKS has the ability to generate, store, and use *protected keys* within the TEE. The user first runs a *remote attestation* protocol to ensure that the CKS is running in a genuine TEE, and to establish a secure channel to the TEE. Via the secure channel, the user submits requests to the CKS, containing her username, password, and the inputs for the cryptographic operation (e.g. data to be decrypted or signed). Upon receiving a request, the CKS checks the user's password, performs the requested operation, and returns the result via the secure channel.

Additional functionality: The CKS allows users to delegate usage rights for their keys to other users for a specific period of time and/or number of uses of the key, and to audit all operations that have been performed using their protected keys.

3.2 Adversary model

We assume a strong adversary who has the ability to run arbitrary code on the server with root privileges. In addition, the adversary has full control of the network, including the ability to monitor, drop, modify, and/or replay any network communication. We assume the adversary cannot feasibly break correctly implemented cryptographic primitives or guess strong cryptographic keys. We also assume that the adversary cannot feasibly subvert the security guarantees of hardware-based Trusted Execution Environments (although we discuss side-channel attacks against TEEs in Section 6). In reality, a well-resourced adversary may be able to subvert a

hardware-based TEE, but this requires direct physical access and would incur significant cost. Therefore, our adversary model captures various possible scenarios, including:

- An attack and elevation of privileges against the server;
- A server operator being coerced by law enforcement;
- A malicious employee of server operator (insider threat).

In this work, we assume that users do not choose particularly weak passwords (e.g. passwords are not in the list of the top 100 most common passwords). This is a reasonable assumption because i) an average password is estimated to provide the equivalent of approximately 20 bits of entropy [5], and ii) users who are using personal cryptographic keys are likely to be able to choose a password of reasonable strength.

3.3 Security requirements

The primary security goal of the CKS is to protect keys against exfiltration and/or unauthorized use. Since key use is authorized by the user's password, the CKS must ensure the security of this password. Given the adversary model above, we define the following security requirements to achieve this goal:

R1 **Authentication:** Protected keys can only be used or delegated by supplying the correct password.
R2 **Offline attacks:** Offline guessing of the password must be infeasible.
R3 **Online attacks:** Online guessing of the password must be rate limited.

3.4 Performance Goals

Ideally, the CKS should not cause a noticeable slow-down from the perspective of the user (e.g. compared to using a smart card), and should maximize the throughput per server, thus minimizing the number of servers required. We therefore define the following two performance goals:

- **Latency:** The time required to perform a cryptographic operation using a protected key must be comparable or better than that of using a key stored on a smart card.
- **Throughput:** The design should maximize the rate of cryptographic operations performed by the server.

3.5 Deployability Goals

The ability to create and restore backups of the CKS is critical for mitigating the impact of failure or theft of the physical server hardware. The ability to scale horizontally is also necessary to ensure the availability of the CKS, even during periods of peak demand. However, implemented naively, both of these capabilities could be abused by the adversary to increase his rate of password guessing, thus undermining security requirement R3. We therefore define the following deployability goals:

- **Backup and recovery:** The server should support secure backup and recovery procedures to mitigate the risk of hardware failures.
- **Scalability:** The server should support secure yet dynamic horizontal scalability (e.g. it should be load balancer friendly).

Table 1: Cloud Key Store operations

Name	Input	Output
create_user	uid, new_pswd, new_reset_pswd	–
pswd_reset	uid, reset_pswd, new_pswd	–
gen_key	uid, pswd, key_attributes	key_id
import_key	uid, pswd, key_data	key_id
pksign	uid, pswd, key_id, msg_hash	signature
pkdecrypt	uid, pswd, key_id, enc_msg	decryption
set_policy	uid, pswd, key_id, key_policy	–
delegate	uid, pswd, key_id, delegatees, policy	–
undelegate	uid, pswd, key_id, delegatees	–
audit	uid, pswd, key_id, time_period	key_log
delete_key	uid, pswd, key_id	–

Figure 2: Sequence of interactions between the user and the Cloud Key Store.

4 DESIGN

In this section we present the design of the Cloud Key Store (CKS), focusing on the challenges and our solutions. The operations that can be performed by the CKS are listed in Table 1, and an overview of the interaction between the user and the CKS is depicted in Figure 2. In reality, the human user would use some type of client-side software to interact with the CKS (e.g. GnuPG on Linux or OpenKeychain on Android). For clarity of explanation and without loss of generality, we simply use the term *user* to refer to the human user and all client-side software.

4.1 Establishing a Secure Channel

To establish trust in the CKS, the user must receive strong assurance that the CKS is running in a genuine TEE, and that only the CKS software is running in the TEE. This is achieved through remote attestation. Since we only require unilateral authentication of the TEE towards the user, we use the efficient remote attestation protocol proposed by the SafeKeeper system [24]. This allows the user to verify the TEE, establish a secure channel, and issue a request all in a single message round-trip. In this section we illustrate the protocol using a discrete logarithm Diffie-Hellman (DH) key agreement protocol, but in practice any equivalent key agreement protocol could be used (e.g. Elliptic curve DH).

Before any users connect, the TEE generates a private key a and a DH public key g^a, where g is a generator for a suitable DH group. All operations are performed in this group (i.e. $\mod n$, where n is the group size). Using the platform functionality, the TEE generates a *quote* containing this public key and full details of the software running in the TEE. This provides assurance that only the specified software has access to the private key corresponding to g^a.

When a user first connects to the CKS, she requests this quote. The user verifies the authenticity of the quote, and that the public key supplied by the CKS (or a hash thereof) is included in the quote. The user then generates her own private key b and DH public key g^b, and sends g^b to the CKS. At this point the user has established a shared secret key $K = g^{ab}$ with the CKS, and unilaterally authenticated the CKS. The user encrypts subsequent requests to the enclave with the derived key.
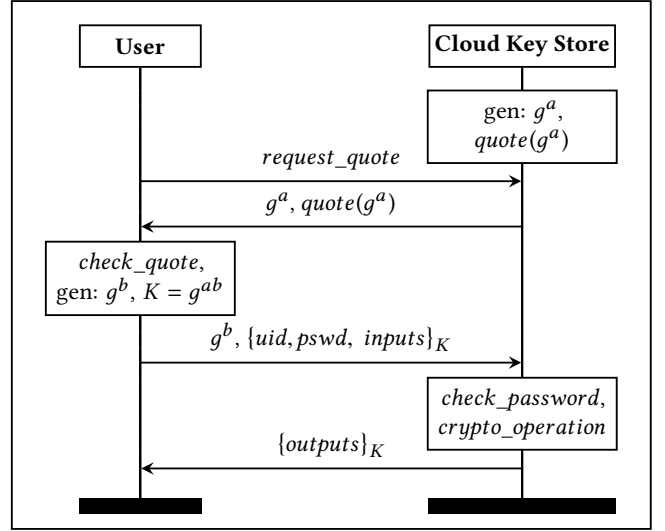
4.2 Authenticating the User

Once the secure channel has been established, the communication between the user and CKS follows a typical request-response model. When a request is received, the CKS decrypts the request, looks up the username, and compares the supplied password. To protect against online password-guessing attacks, the TEE limits the rate of authentication attempts using an exponential back-off algorithm. When the user authentication fails, the CKS temporarily disables further authentication attempts for that particular user account for a user-defined duration. If subsequent authentication attempts also fail, the duration between attempts is increased exponentially. A successful authentication resets the duration to its initial value. As discussed in Section 6.1, the rate-limiting mechanism in the TEE is primarily intended to prevent a compromised server from performing online guessing attacks against the CKS. The server operator would also implement their own rate-limiting mechanism outside the TEE to prevent external adversaries from causing a denial of service attack by abusing the CKS rate-limiting mechanism.

4.3 New Users and Password Recovery

When a user first accesses the system, she uses the `create_user` function and supplies a new username, a password, and a reset password. The username and password must be included in all subsequent requests. If the user forgets her password, or accidentally reveals it to an adversary, she can send a `pswd_reset` request using her reset password. The limitation of this mechanism is that it still requires the user to store the reset password securely. However, since this password is not used frequently, it should be feasible for the user to store it securely, e.g. written down in a safe. Other password recovery mechanisms like email may be possible, but would require the user to trust additional services.

4.4 Policy-based Access Control

If the authentication succeeds, the CKS performs the requested operation. The user can generate a new key or import a key using the `gen_key` and `import_key` functions respectively. Every key has an associated *key usage policy*, which defines:

- the username of the key's owner;
- what types of operations may be performed using the key;
- the key expiration time;
- the remaining number of operations.

The user can sign or decrypt data using the `pksign` and `pkdecrypt` functions respectively. Every cryptographic operation is checked against the key's usage policy. If the policy check succeeds, the CKS performs the cryptographic operation and returns the outputs to the user via the secure channel.

The key owner can always change a key's usage policy using the `set_policy` function, but this policy is still checked even for the key owner to prevent accidental use of the key (e.g. using a signing key for a decryption operation). The user can delete any keys she owns using the `delete_key` function.

4.5 Key Delegation

A user can delegate access to protected keys to other users using the `delegate` function. The *delegator* specifies the key to be delegated and the usernames of the *delegatees*, as well as any restrictions on the delegation (i.e. duration and/or number of uses). For each delegatee, the CKS first checks that the restrictions match the main key usage policy (i.e. a key cannot be delegated beyond its expiry date). If this check passes, the CKS adds an additional section to the key's usage policy specifying the username of the delegatee, the permitted operations, the duration of the delegation, and the remaining number of operations. The CKS may add multiple additional sections if the key is delegated to multiple users.

When the delegatee attempts to use this key, the CKS checks the operation against the additional section of the key's usage policy. The delegatee cannot modify the key's usage policy. In our current design, the delegatee also cannot delegate these keys further, however, it would be reasonably straightforward to allow this functionality through our flexible key usage policy.

4.6 Key Usage Auditing

The CKS maintains a secure log of all operations performed using protected keys. Before performing an operation, an entry is added to the log specifying the time[2] of the operation, the key used, the type of operation, the input values, and the output values. The user can obtain the log using the `audit` command, which takes a key identifier and time period as parameters, and returns all log entries for that key during the specified time period. If a user notices operations that she did not perform, she can take remedial actions such as resetting the password. Assuming the user's recovery password has not been compromised, the user does not need to revoke the key – changing the main password would be sufficient to prevent the adversary from using the key. If the adversary has delegated the key, this can be reversed using the `undelegate` function.

---

[2]This requires a TEE that provides trusted time capabilities.

5 IMPLEMENTATION

We implemented the CKS as an Intel SGX enclave running on the server, and we modified two client applications to use our CKS: GnuPG on Linux, and OpenKeychain on Android. In this section we summarize the main technical details of our implementations.

5.1 Enclave

The enclave's API consists of 3 ECALLs: `initialize`, `process`, and `shutdown`, as explained in the following subsections. The enclave maintains a *key database* containing the username and password of each user, all cryptographic keys generated or imported by the users, and the associated metadata like the key usage policies.

5.1.1 Enclave initialization. The `initialize` ECALL causes the enclave to generate a fresh DH key pair, pre-generate the information required for remote attestation, unseal the database of keys, and load this into the enclave's memory. As explained in Section 4.1, our design uses a more efficient remote attestation protocol than the default 4-message protocol provided in the SGX SDK. To ensure that it has unsealed the latest version of the database, the enclave compares the version number and version nonce in the database against the value of a hardware monotonic counter. If there is a mismatch, the enclave raises an error and aborts operation. Before the enclave begins to process requests, it increments the hardware monotonic counter to indicate that the state of the database may have changed. This prevents the adversary from crashing the enclave and attempting to restore the last-saved version of the database, since the enclave may have made changes that are not yet committed to the saved database.

5.1.2 User authentication. The `process` ECALL is used for all CKS operations. Compared to creating separate ECALLs for each CKS operation, this design means that the enclave's interface does not need to be changed, even if new CKS functions are added. This ECALL takes as input an opaque data buffer, the length of the buffer, and the public DH key of the requesting party.

When the enclave receives a request, it first completes the DH key agreement using the public key supplied by the user, and then decrypts the message. After decryption, the enclave retrieves the username and password from the map of registered users and compares them to the provided values. As explained in Section 4.2, failed authentication attempts cause the enclave to temporarily disable authentication attempts for a user-defined duration, that is exponentially-increasing with subsequent failed attempts. If the authentication succeeds, the enclave parses the request to obtain the required operation.

5.1.3 Signing and decryption. Our prototype implementation supports both signing and decryption operations using either RSA or Elliptic Curve cryptography. For RSA, we support the key size of 3072 bits. We support SHA-256 for hashing. For Elliptic Curve cryptography, we use the curve `secp256r1`. Other algorithms and key sizes can be easily added to the enclave implementation.

5.1.4 Delegation and trusted time. In addition to the username and password of the key owner, the delegation function takes as input the username of the delegatee, and the delegation policy that

can specify the duration and/or the maximum number of uses allowed. To enforce the time-limited delegation, our enclave uses the SGX Trusted Time API. The `sgx_get_trusted_time` function returns the current time relative to an arbitrary but fixed reference point [21]. This reference point is guaranteed to remain unchanged as long as the `time_source_nonce` does not change. For delegation, there is no need to synchronize SGX and wall-clock times because the enclave measures a time *period* starting from the point of processing the delegation request. The enclave stores the expected `time_source_nonce` value and checks this whenever the time is checked. If this value changes, the enclave raises an error and refuses to process requests, since an adversary could have modified the platform's hardware clock.

5.1.5 Auditing and time synchronization. To ensure that every operation is logged, the enclave creates a log entry *before* performing any cryptographic operation. In the worst case, the adversary could interrupt the enclave between creating the log entry and performing the operation, leaving an additional entry in the log. However, this can easily be detected by the user, and is no different from the scenario in which the result of the operation is dropped by the network. Each log entry contains a timestamp and the identifier of the key that was used. To audit the usage of a particular key, the user can request all log entries associated with that key for a specified time period. In this case, it is desirable to synchronize SGX time to wall-clock time because the log entries contain individual *points* in time. When the CKS is first created (i.e. before any users have been added), the server operator is given a one-off opportunity to specify a time offset value for the enclave. This is intended to synchronize SGX time to a specific wall-clock time reference point (e.g. Unix time). This is purely a convenience feature and does not have any security implications. If the server operator provides a different offset, users of this CKS will still see consistent time points relative to a different reference point.

5.1.6 Enclave shutdown/restart. The `shutdown` ECALL causes the enclave to persist all important data in preparation for the enclave to be restarted or the platform rebooted. The enclave stops processing requests and seals the current state of the key database, including a *version number* and *version nonce* obtained from one of the enclave's hardware monotonic counters. On platforms without hardware monotonic counters, a distributed counter service like ROTE [27] can be used instead. The sealed database must be provided to the enclave when it is restarted.

5.1.7 Irrecoverable errors. In the following situations, the enclave cannot recover without potentially compromising the security guarantees of the system:

- If the system's hardware clock is changed, the enclave cannot determine whether operations satisfy the key usage policy.
- If an incorrect version of the sealed database is provided during initialization, the enclave cannot safely use this because an unknown number of changes could have been lost.

In the above cases, the enclave raises an error and refuses to process any requests. In a real deployment, the CKS service would be distributed across multiple enclaves, such that failure of any single enclave does not affect the overall availability of the system (see Section 8). Note that a compromised server operator can always

perform a denial of service attack against the system by simply refusing to process network packets or run the enclave.

5.2 GnuPG integration

We integrated our CKS into GnuPG as a replacement for the original smart card daemon. GnuPG has a modular architecture in which the main process provides the user interface but contacts a GnuPG *agent* for the backend operations. The agent provides a common interface to perform cryptographic operations and key management, regardless of where the keys are stored. For example, when using a smart card, GnuPG starts the smart card daemon in the background and uses this daemon to perform the cryptographic operations. We did not extend the existing smart card daemon to communicate with the CKS, but rather we implemented our own CKS daemon, which provides the same interface as the smart card daemon. This design choice gives finer control of the daemon operation and will allow us to in future integrate the additional features offered by the CKS (e.g. key delegation, auditing) into GnuPG.

All messages exchanged between the users and the CKS are encoded using *S-expressions* [28], a flexible and efficient tree-based data structure. GnuPG uses S-expressions internally when it utilizes its cryptographic library `libgcrypt`. Additionally we implemented an HTTP transport to communicate with the CKS.

GnuPG components communicate with one another via a protocol called Assuan [40]. This is a simple text-based protocol in which each line corresponds to one request or response. The first word in the line specifies the command and the arguments follow separated by white-space. For example, when exporting the key to the smart card, the Assuan request line consists of the keyword `KEYDATA` followed by the encoded key data. Our new CKS daemon performs the following operations:

1. When started, request the quote from the remote enclave and verify this using the Intel Attestation Service (IAS).
2. Generate a new DH key pair and calculate the shared key with the enclave.
3. Start Assuan server and wait for requests from the GnuPG agent.
4. Dispatch the commands from the agent to the corresponding command handlers.
5. If the command is security sensitive, the daemon launches a PIN entry program to request the user's password.
6. Encrypt the buffer containing the operation and its arguments with the shared key established via remote attestation.
7. Send the encrypted buffer to the enclave for processing.
8. Decrypt the enclave's response and pass the result back to the agent.

Our GnuPG integration does not require any changes to the GnuPG UI. When using a CKS protected key, the user's password is requested in the same way as GnuPG would request the user's PIN when using a smart card. Thus the core CKS operations (i.e. key generation, signing, and decryption) can all be carried out from within GnuPG. For the additional CKS features (i.e. delegation and auditing), we provide a simple client-side web app that communicates with the enclave as described above and allows the user to issue delegation commands and inspect the key usage log.

5.3 OpenKeychain integration

Following a similar pattern as for GnuPG, we integrated the CKS into the OpenKeychain [32] application for Android. Although there is a version of GnuPG for Android [13], this is not being actively maintained, and has not been updated in the past three years. Therefore we decided to target OpenKeychain. By default, OpenKeychain assumes that keys will be stored locally on the device, but it also supports smart cards.

Similarly to the integration with GnuPG described above, the smart card functionality of OpenKeychain can be changed to communicate with our CKS, so we omit the details. No changes to the app's UI are required to support the core CKS functionality.

6 EVALUATION

6.1 Security

We analyse the security of our solution based on the requirements in Section 3.3, and we discuss other attacks and their mitigations.

R1 Authentication: By the design of our enclave, the protected keys can only be used if a valid password is provided. Since the user attests the enclave before she generates or imports keys, she can be sure that it will always perform this authentication step.

R2 Offline attacks: Whenever the key database leaves the boundary of the enclave, it is sealed using the enclave's sealing key. This prevents offline guessing attacks against the passwords.

R3 Online attacks: A compromised server can attempt to perform an online password guessing attack against the CKS, by pretending to be a legitimate user. Correctly guessing a user's password would allow the adversary to use that user's protected keys. The rate-limiting mechanism (Section 4.2) makes it difficult even for a compromised server to mount a successful online guessing attack.

Password compromise: Even if the adversary learns the user's password, he cannot exfiltrate keys from the CKS. He can perform operations using the keys until the user detects it via auditing, but when the key owner (i.e. the original user) resets her password, using her recovery password, there is no need to revoke her key, since the adversary will no longer be able to access this key using the password he obtained. If the adversary has delegated the key to other users, the key owner will see this from the audit log, and can revoke these delegations using the delete_delegation function.

Audit log attacks: Since the enclave creates a log entry before performing any operation, the only way for an adversary to corrupt the log is to prevent the enclave from performing the operation after the log entry was created. However, this only allows the adversary to add spurious events to the log. The user can easily detect these because he would not receive the result of the requested operation. However it is not possible for the adversary to perform any operation without entries appearing in the log.

SGX side-channel attacks: SGX is vulnerable to certain side-channel attacks, including memory access pattern attacks [37, 39, 41], cache attacks [7, 17], and branch shadowing attacks [25]. The protected keys are not vulnerable to side-channel attacks because we perform all cryptographic operations using the recommended Intel cryptographic libraries, which offer state-of-the-art protection against SGX side-channel attacks. Although the identity of the user is only included in the encrypted buffer, an adversary may still be able to infer whether two requests originated from the same user by monitoring the enclave's memory access pattern. If this information should also be protected, various mechanisms (e.g. [34, 35]) can be used to hide the enclave's memory access pattern.

Denial of service: The various protection mechanisms included in our enclave could be used by a compromised server to cause a denial of service attack against the CKS. However, this attack is not in scope because a compromised server could mount this type of attack in various other ways (e.g. dropping network packets or simply refusing to run the enclave).

6.2 Performance

We evaluate the performance of the CKS in terms of the throughput and memory consumption of the enclave, and the latency introduced from the user's perspective. The key generation and import operations are not benchmarked, as they are not executed often, so they do not have a significant impact on the user experience.

Enclave throughput: To measure the overall throughput of the CKS, we benchmarked our implementation on an Intel Core i5-6500 3.20 GHz CPU with 8GB of RAM running Ubuntu 16.04 and the Intel SGX SDK version 2.1. Over 10 experiments, the average time required to perform 10,000 signature operations using the secp256r1 curve was 1,667 ms (±91 ms). This gives an effective rate of approximately 6,000 signature operations per second, although it is likely that server-class CPUs would achieve higher throughput. Throughput for decryption operations is similar.

Enclave memory: The memory footprint depends heavily on the key policies, but a typical user record would consist of 600-700 bytes for an RSA key of length 3072 bits. So an enclave can serve 100,000 users consuming 100 Megabytes of heap memory.

Latency: To evaluate the impact on the user experience, we measured the latency of a signing operation with and without the CKS. We ran GnuPG on an Intel Core i5-6500 3.20 GHz CPU with 8 GB of RAM, running Ubuntu 16.04. We ran OpenKeychain on a Samsung Galaxy S6 Edge+ with an Exynos 7420 Octa-core CPU (4x2.1 GHz Cortex-A57 and 4x1.5 GHz Cortex-A53) and 4 GB of RAM, running Android 7.0. The CKS was run on the same machine described above, and all results are the average over 10 experiments.

As the GnuPG baseline, we measured the time required to create an RSA 3072 bit signature using a key on a GnuPG smart card, attached via a USB card reader. The average latency was 1,282 ms (±5 ms). We measured the same signature operation when using CKS from GnuPG. To remove the variability of network latency, we ran the GnuPG client and the CKS enclave on the same physical machine communicating over localhost. The average time required to create the signature was 15 ms (±1 ms).

As the OpenKeychain baseline, we measured the time required to create an RSA 3072 bit signature using an unprotected key in the device. The average latency was 37 ms (±4 ms). To measure the latency of OpenKeychain using CKS whilst removing the network latency, we connected the Android device to the PC running the CKS enclave via a USB connection. The average time required to create the signature was 24 ms (±2 ms).

All of the above measurements intentionally exclude the network latency, since this will depend on the type of network to which the user is connected. To estimate network latency, we performed an HTTP ping of Amazon Web Service (AWS) regions using the

CloudPing tool [10]. On both a wired network connection and a 4G mobile network, the latency to the nearest three AWS regions (Ireland, London, and Frankfurt) was in the order of 50ms. Assuming a network latency of 50ms, the latency of using CKS is in the same order of magnitude as using a local key on the device, and should not be noticeable to the user.

7 KEY STORAGE COMPARISON

To compare different approaches for securely storing personal cryptographic keys, we develop a comparative evaluation framework that considers aspects of security and functionality. Using this framework we analysed several common approaches, in comparison with our new CKS approach. The results are summarized in Table 2. In this table, a filled circle (•) indicates that the approach fully defends against the specified attack or fully provides the respective functionality. An unfilled circle (○) means that the approach partially supports the feature, whilst a dash (−) shows that the approach does not provide that functionality.

Table 2: Comparison of key storage approaches

	Password encrypted (local)	Password encrypted (cloud)	Smart card	Local hardware storage	Hardware token	Hardware Security Module	Cloud Key Store (CKS)
Security							
Secure environment	-	-	•	•	•	•	•
Resist offline guessing	-	-	•	•	•	•	•
Resist online guessing	-	○	•	•	•	•	•
Remote attestation	-	-	-	-	-	-	•
Functionality							
No additional hardware	•	•	-	○	-	-	•
Multi-device use	-	•	○	-	○	•	•
Recovery	-	○	-	-	-	•	•
Auditability	-	-	-	-	-	○	•
Time-bounded delegation	-	-	○	-	○	○	•
Use-bounded delegation	-	-	-	-	-	○	•

As shown in Table 2, the security considerations include: whether the key is used within a *secure environment*; and whether the stored keys are vulnerable to *offline* or *online guessing* attacks (e.g. of a password). The functionality considerations include: whether an approach requires additional hardware; whether the key can be used from *multiple devices*; whether the user can *recover* from loss of a device without having to revoke the key; whether the user can *audit* the use of the key; and whether the user can *delegate* access to the key for either a bounded duration or number of uses.

Note that in some cases, certain functionality could be provided by using application-specific mechanisms. For example, time-bounded delegation could be achieved for PGP signing keys by generating and signing a subkey for a limited duration. However, we exclude these application-specific mechanisms from the framework because they cannot be used in all cases (e.g. the subkey approach cannot be used for decryption keys).

Password encrypted (local): Keys stored on the user's local device, even if encrypted with a password, are used directly by the applications, which would not typically be considered a *secure environment*. These keys are vulnerable to offline guessing attacks against the password (online attacks are not applicable). In terms of functionality, this approach does not require any additional or specialized hardware, however, by default the key cannot be used from multiple devices, cannot be recovered if lost, and cannot be audited or delegated.

Password encrypted (cloud): The encrypted key is stored in the cloud and downloaded to users' devices when needed. Compared to local storage, the security remains mostly unchanged, except that partial protection against online guessing attacks may be provided if the cloud server is trusted. In terms of functionality, this allows the key to be used from multiple devices, and provides partial recovery capabilities if the user's device is lost but the password is not leaked. This approach does not provide centralized auditability since the key is downloaded to the user's device.

Smart card: As expected, a smart card provides significantly better security guarantees: all operations take place in a secure environment, and the smart card is designed to resist offline and online guessing of the password or PIN. However, in terms of functionality, smart cards themselves are additional hardware and also require a smart card reader. A single smart card can be used from multiple devices, provided they all have compatible readers. In general, the device and smart card must be in the same physical location, so protected keys cannot be used concurrently from two different locations. Smart cards do not provide any recovery mechanism in case of loss or damage. To some extent, smart cards can enable time-bounded delegation: the key owner can give the smart card to the delegatee, provided she receives it back at the agreed time.

Local hardware storage: Keys stored in e.g. the Android or iOS device key stores enjoy similar protection to that of a smart card. Although this approach does not require additional hardware, it is not by default possible to use the keys from multiple devices or delegate them to other users.

Hardware token: Secure hardware tokens like Trezor [38] and YubiKey [42] are very similar to smart cards. Although they are by definition additional hardware, they usually do not require readers, like with smart cards.

Hardware Security Module (HSM): A cloud-hosted HSM might offer similar features as our TEE-based CKS, but this depends on the specific design of the HSM (e.g. a general-purpose HSM is unlikely to provide advanced features like auditing and delegation). The HSM is an additional hardware requirement, and is also unlikely to provide remote attestation of its internal behaviour.

Cloud Key Store: In comparison, the CKS performs all operations within a secure environment, resists offline and online guessing attacks, and provides remote attestation (Section 6.1). By design, the CKS does not require additional hardware, and can be accessed from multiple devices concurrently. Finally, the CKS allows the key owner to audit all key usage, to recover from device loss without

revoking the key, and to delegate keys in either a time-limited or use-limited fashion (Section 4).

8 EXTENSIONS

8.1 Group keys

The user identities framework can be extended to include groups of users. Similarly to delegating keys to other users, the group user could set security policies on the group keys. The functions to add and remove group members can be implemented in the same way as the current delegation functions. Flexible per-user policies can be combined with setting up per-group key policies. Then the use of group keys can also be use-limited and time-limited.

8.2 Multiple passwords

The password authentication mechanism can be extended to support multiple passwords for each user. For example, the user can set up different passwords for each of her devices. If one of the user's devices is lost, only the password for that device needs to be reset.

8.3 Multi-factor user authentication

The user authentication mechanism can also be extended to support multi-factor authentication e.g. using the Universal 2nd Factor (U2F) standard [12] or a one-time password (OTP) algorithm [30, 31]. This would help to prevent misuse of a protected key if the user's password is compromised. However, users who activate multi-factor authentication would need to carry and use an additional authentication token or OTP generator app.

8.4 Trusted time server

In addition to the trusted time synchronization mechanism described in Section 5, it would be possible to have the enclave contact a trusted Network Time Protocol (NTP) server over a secure channel. This would allow the enclave to synchronize its local SGX time to wall-clock time from the NTP server. When attesting the enclave, users could verify that this process took place, and if the users also trust the same NTP server, they can be assured the enclave has been synchronized to the same time reference.

8.5 Backup, recovery, and scalability

Since availability is a critical concern, it should be possible to create backups of the CKS key database and restore these on another SGX-enabled machine, to mitigate against hardware failure. Additionally, in a cloud setting, it is common to replicate a service across multiple servers in order to achieve higher throughput and better load balancing. Krawiecka et al. [24] faced a similar challenge in terms of backup and scalability in the SafeKeeper system. The solution they proposed was to run the same enclave on multiple physical machines and perform mutual attestation between these enclaves. The primary enclave can then replicate the sensitive data (i.e. the CKS key database) across these different machines, without reducing the security guarantees (since the enclaves are identical). They proposed a simple consensus protocol that the enclaves could use to agree on which replicas are active and which are backups. A similar type of approach could be used to provide backup, recovery, and scalability in CKS.

8.6 User-authorized migration

In a real-world deployment, the server operator may need to upgrade to a newer version of the CKS enclave. However, in our design, any changes to the enclave's software will result in a different enclave identity (MRENCLAVE value), and thus the new enclave would not be able to unseal the database of passwords and protected keys. Allowing the server operator to upgrade the enclave's software without authorization from the users could allow a malicious or coerced operator to surreptitiously 'upgrade' to an enclave that does not adequately protect users' keys. To prevent this attack, users must explicitly authorize any enclave software upgrade. For example, this could be achieved using a new upgrade_cks command that authenticates the user via her password (as usual) and includes the identity of the new enclave, which the user has verified as being trustworthy. With this authorization, the old enclave could attest the new enclave, confirm that its identity matches the user's authorization, and then transfer the user's keys. This is similar to the credential transfer protocol proposed by Kostiainen et al. [23], but is slightly simpler because we assume both enclaves can be online simultaneously.

9 RELATED WORK

Key management services: Major cloud providers often provide key management services [1, 16, 29]. These services can be used to generate, store, and use cryptographic keys, without exposing the keys to potentially untrusted software. For example, the OpenStack Barbican key management service can store secrets in a dedicated Hardware Security Module (HSM), and more recently in Intel SGX enclaves [9]. Similarly, the Fortanix Self-Defending Key Management Service uses SGX to protect keys [14]. Both of these SGX-based approaches also use remote attestation to provide assurance to remote verifiers. However, the above solutions are designed to manage secrets used by cloud customers (e.g. secrets used by VMs running on the provider's cloud infrastructure). In contrast, our Cloud Key Store is designed to be used by end-users directly from their devices, and to provide user-centric capabilities like key usage auditing and key delegation between users. This is enabled by our use of password-based authentication of end-users within the TEE.

User authentication: Balisane et al. [4] discuss the idea of storing *authentication templates* (e.g. password databases) inside a TEE. However, in their proposed architecture, the server operator would still be able to capture a user-submitted password in transit before it is input to the TEE. To avoid this, we first establish an end-to-end secure channel between the user and the TEE, by performing a key agreement protocol authenticated via the TEE's remote attestation, as explained in Section 4.1. This channel protects the password in transit, even against a compromised server operator.

Logging and auditing: Karande et al. [22] proposed *SGX-Log*, a tool that uses SGX to protect the confidentiality and integrity of system logs. Aublin et al. [3] presented *LibSEAL*, a server-side library that creates a non-repudiable log of all requests and responses by terminating TLS connections inside an SGX enclave. Although conceptually similar to the above approaches, our key usage auditing functionality is designed to record operations performed by the enclave itself, rather than logging externally-generated events.

10 CONCLUSION AND FUTURE WORK

We presented a Cloud Key Store (CKS) – the system that enables users to use their keys from any device, without additional hardware. Using Intel SGX, our implementation of the CKS protects users' keys against even a compromised server. To demonstrate feasibility and evaluate performance, we integrated the CKS into GnuPG on Linux and OpenKeychain on Android. The CKS achieves the same strong security guarantees as smart cards or hardware tokens, whilst providing enhanced functionality like delegation and auditing. Our performance evaluation shows that the CKS throughput is high and the latency is in the same order of magnitude as local key storage. As future work we plan to investigate new additional features that could be realized as part of the CKS. For example, if a key has been generated within the CKS, the user could be given the ability to provably but temporarily *lock* the key, thus preventing even herself from using the key during the specified time period. This could be useful from a legal perspective since a user can prove that she will not have access to the key until a specified time point in the future (e.g. preventing decryption of data encrypted under that key until a future date).

ACKNOWLEDGMENTS

REFERENCES

[1] Amazon. 2017. AWS CloudHSM. (2017). https://aws.amazon.com/cloudhsm/.
[2] Erinn Atwater and Urs Hengartner. 2016. Shatter: Using Threshold Cryptography to Protect Single Users with Multiple Devices. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec '16)*. 91–102. https://doi.org/10.1145/2939918.2939932
[3] Pierre-Louis Aublin, Florian Kelbert, Dan O'Keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. 2018. LibSEAL: Revealing Service Integrity Violations Using Trusted Execution. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. https://doi.org/10.1145/3190508.3190547
[4] Ranjbar A. Balisane, Ravishankar Borgaonkar, Ahmad Atamli-Reineh, and Andrew Martin. 2017. Architectures for Enhancing Authentication Privacy and Security Using Trusted Computing. In *2nd Workshop on System Software for Trusted Execution (SysTEX'17)*. https://doi.org/10.1145/3152701.3152713
[5] Joseph Bonneau. 2012. The science of guessing: analyzing an anonymized corpus of 70 million passwords. In *IEEE Symposium on Security and Privacy*.
[6] Joseph Bonneau, Cormac Herley, Paul C Van Oorschot, and Frank Stajano. 2012. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *IEEE Symposium on Security and Privacy*.
[7] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *USENIX Conference on Offensive Technologies*.
[8] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. 2007. RFC4880: OpenPGP Message Format. (2007). https://tools.ietf.org/html/rfc4880.
[9] Somnath Chakrabarti, Brandon Baker, and Mona Vij. 2017. Intel SGX Enabled Key Manager Service with OpenStack Barbican. (2017). https://arxiv.org/abs/1712.07694.
[10] CloudPing. 2018. (2018). http://www.cloudping.info/.
[11] Clifford Cocks. 1998. Split generation of RSA parameters with multiple participants. (1998). https://www.gchq.gov.uk/split-generation-rsa-parameters-multiple-participants.
[12] FIDO Alliance. 2017. Universal 2nd Factor (U2F) 1.2 Specifications. (2017). https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411.
[13] Gnu Privacy Guard for Android. 2018. (2018). https://github.com/guardianproject/gnupg-for-android.
[14] Fortanix. 2017. Self-Defending Key Management Service. (2017). https://www.fortanix.com/products/sdkms.
[15] The Gnu Privacy Guard (GnuPG). 2018. (2018). https://www.gnupg.org/.
[16] Google. 2017. Google Cloud Key Management Service. (2017). https://cloud.google.com/kms/.
[17] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *10th European Workshop on Systems Security*. https://doi.org/10.1145/3065913.3065915
[18] Shay Gueron. 2016. Memory Encryption for General-Purpose Processors. *IEEE Security Privacy* 14 (2016), 54–62.
[19] Intel. 2017. 1U System Delivering Cryptographic Isolation Technology. (2017). https://www.intel.com/content/www/us/en/data-center-blocks/business/secure-enclaves-blocks.html.
[20] Intel. 2017. Software Guard Extensions (Intel SGX). (2017). https://software.intel.com/en-us/sgx.
[21] Intel. 2018. SGX Reference: sgx_get_trusted_time. (2018). https://software.intel.com/en-us/node/709158.
[22] Vishal Karande, Erick Bauman, Zhiqiang Lin, and Latifur Khan. 2017. SGX-Log: Securing System Logs With SGX. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS '17)*. https://doi.org/10.1145/3052973.3053034
[23] Kari Kostiainen, N Asokan, and Alexandra Afanasyeva. 2011. Towards user-friendly credential transfer on open credential platforms. In *International conference on Applied cryptography and network security*.
[24] Klaudia Krawiecka, Arseny Kurnikov, Andrew Paverd, Mohammad Mannan, and N. Asokan. 2018. SafeKeeper: Protecting Web Passwords Using Trusted Execution Environments. In *Proceedings of the 2018 World Wide Web Conference (WWW '18)*. https://doi.org/10.1145/3178876.3186101
[25] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security Symposium*.
[26] Phillip MacKenzie and Michael. K. Reiter. 2001. Networked Cryptographic Devices Resilient to Capture. In *IEEE Symposium on Security and Privacy*. https://doi.org/10.1109/SECPRI.2001.924284
[27] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback Protection for Trusted Execution. In *USENIX Security*. https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-matetic.pdf
[28] John McCarthy. 1960. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM* 3, 4 (1960), 184–195.
[29] Microsoft. 2017. Azure Key Vault. (2017). https://azure.microsoft.com/en-us/services/key-vault/.
[30] D. M'Raihi, M. Bellare, F. Hoornaert, D. Naccache, and O. Ranen. 2005. RFC4226: HOTP: An HMAC-Based One-Time Password Algorithm. (2005). https://tools.ietf.org/html/rfc4226.
[31] D. M'Raihi, S. Machani, M. Pei, and J. Rydell. 2011. RFC6238: TOTP: Time-Based One-Time Password Algorithm. (2011). https://tools.ietf.org/html/rfc6238.
[32] OpenKeychain. 2018. (2018). https://www.openkeychain.org/.
[33] Joanna Rutkowska. 2018. Invisible Things Blog. (2018). https://blog.invisiblethings.org/keys/.
[34] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *NDSS*. https://doi.org/10.14722/ndss.2017.23193
[35] Sandeep Tamrakar, Jian Liu, Andrew Paverd, Jan-Erik Ekberg, Benny Pinkas, and N. Asokan. 2017. The Circle Game: Scalable Private Membership Test Using Trusted Hardware. In *ACM Asia Conference on Computer and Communications Security*. https://doi.org/10.1145/3052973.3053006
[36] ARM TrustZone. 2018. (2018). https://www.arm.com/products/security-on-arm/trustzone.
[37] J Van Bulck, N Weichbrodt, R Kapitza, F Piessens, and R Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security Symposium*.
[38] Trezor Bitcoin wallet. 2018. (2018). https://trezor.io.
[39] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. 2017. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *ACM SIGSAC Conference on Computer and Communications Security*.
[40] Developing with Assuan. 2018. (2018). https://www.gnupg.org/documentation/manuals/assuan/.
[41] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Symposium on Security and Privacy*.
[42] YubiKey. 2018. (2018). https://www.yubico.com/start/.
[43] Phil Zimmermann. 1994. PGP User's Guide. (1994).