Deleting Secret Data with Public Verifiability

Feng Hao, *Member, IEEE,* Dylan Clarke, Avelino Francisco Zorzo

Abstract—Existing software-based data erasure programs can be summarized as following the same one-bit-return protocol: the deletion program performs data erasure and returns either success or failure. However, such a one-bit-return protocol turns the data deletion system into a black box – the user has to trust the outcome but cannot easily verify it. This is especially problematic when the deletion program is encapsulated within a Trusted Platform Module (TPM), and the user has no access to the code inside.

In this paper, we present a cryptographic solution that aims to make the data deletion process more transparent and verifiable. In contrast to the conventional black/white assumptions about TPM (i.e., either completely trust or distrust), we introduce a third assumption that sits in between: namely, "trust-but-verify". Our solution enables a user to verify the correct implementation of two important operations inside a TPM without accessing its source code: i.e., the correct encryption of data and the faithful deletion of the key. Finally, we present a proof-of-concept implementation of the SSE system on a resource-constrained Java card to demonstrate its practical feasibility. To our knowledge, this is the first systematic solution to the secure data deletion problem based on a "trust-but-verify" paradigm, together with a concrete prototype implementation.

1 Introduction

Secure data erasure requires permanently deleting digital data from a physical medium such that the data is irrecoverable [13]. This requirement plays a critical role in all practical data management systems, and in satisfying several government regulations on data protection [25]. For the past two decades, this subject has been extensively studied by researchers in both academia and industry, resulting in a rich body of literature [5], [7], [8], [13], [14], [17], [23], [25], [26], [28], [33], [35]. A recent survey on this topic is published in [27] .

1.1 One-bit return

To delete data securely is a non-trivial problem. It has been generally agreed that no existing software-based solutions can guarantee the complete removal of data from the storage medium [27]. To explain the context of this field, we will abstract away implementation details of existing solutions, and focus at a higher and more intuitive protocol level. Existing deletion methods can be described using essentially the same protocol, which we call the "one-bit-return" protocol. In this protocol, the user sends a command – usually through a host computer – to delete data from a storage system, and receives a one-bit reply indicating the status of the operation. The process can be summarized as follows.

User	→	Storage	:	Delete data
Storage	→	User	:	Success/Failure (1 bit)

F. Hao and D. Clarke are with the School of Computing Science, Newcastle University, UK. Email: {Feng.Hao, Dylan.Clarke}@ncl.ac.uk. A.F. Zorzo is with Pontifical Catholic University of RS, Brazil. Email: avelino.zorzo@pucrs.br. *The first author would like to acknowledge the support of EPSRC First Grant EP/J011541/1 and ERC Starting Grant No. 106591.*

Deletion by unlinking. Take the deletion in the Windows operating system as an example. When the user wishes to delete a file (say by hitting the "delete" button), the operating system removes the link of the file from the underlying file system, and returns one bit to the user: Success. However, the return of the "Success" bit can be misleading. Although the link of the file has been removed, the content of the file remains on the disk. An attacker with a forensic tool can easily recover the deleted file by scanning the disk [12]. The same problem also applies to the default deletion program bundled in other operating systems (e.g., Apple and Linux).

Deletion by overwriting. Obviously, merely unlinking the file is not sufficient. In addition, the content of the file should be overwritten with random data. This has been proposed in several papers [5], [13], [14] and specified in various standards (e.g., [18]). However, one inherent limitation with the overwriting methods is that they cannot guarantee the complete removal of data. As concluded in [13]: "it is effectively impossible to sanitize storage locations by simply overwriting them, no matter how many overwrite passes are made or what data patterns are written." The conclusion holds for not only magnetic drives [13], but also tapes [7], optical disks [14] and flash-based solid state drives [33]. In all these cases, an attacker, equipped with advanced microsoping tools, may recover overwritten data based on the physical remanence of the deleted data left on the storage medium. Therefore, although overwriting data makes the recovery harder, it does not change the basic one-bit-return protocol. Same as before, the return of "Success" cannot guarantee the actual deletion of data.

Deletion by cryptography. Boneh and Lipton [7] were among the first in proposing the use of cryptography to address the secure data erasure problem,

with a number of follow-up works [17], [20], [21], [24]–[26], [35]. In general, a cryptography-based solution works by encrypting all data before saving it to the disk, and later deleting the data by discarding the decryption key. This approach is especially desirable when duplicate copies of data are backed up in distributed locations so it becomes impossible to overwrite every copy [7]. The use of cryptography essentially changes the problem of deleting a large amount of data to that of deleting a short key (say a 128-bit AES key). Still, the fundamental question remains: how to securely delete the key?

1.2 Key management

When cryptography is used to address the data erasure problem, the key management becomes critically important. There are several approaches proposed in the past literature to manage cryptographic keys.

The first method is to simply save the key on the disk, alongside the encrypted data (typically as part of the meta data in the file header) [17], [20], [25], [26]. Deleting the data involves overwriting the disk location where the key is stored. Once the key is erased, the ciphertext immediately becomes useless [7]. This has the advantage of quickly erasing data since only a small block of data (16 bytes for AES-128) needs to be overwritten. However, if the key is saved on the disk, cryptography may not add much security in ensuring data deletion [16]. On the contrary, it may even degrade security if not handled properly – instead of recovering a large amount of overwritten data, the attacker now just needs to recover a short 128-bit key. This may significantly increase the chance of a total recovery. Once the key is restored, the deleted data will be fully recovered. (We assume the ciphertext is available to the attacker, which is usually the case.)

The second method is to use a user-defined password as the encryption key [35]. The key is derived on the fly in RAM upon the user's entry of the password so it is never saved on the disk. However, passwords are naturally bounded by low entropy (typically 20-30 bits) [3]. Hence, cryptographic keys derived from passwords are subject to brute-force attacks. As soon as the attacker has access to ciphertext data, the ciphertext becomes an oracle, against which the attacker can recover the key through the exhaustive search. Instead of directly using a password-derived encryption key, Lee *et al.* proposed to first generate a random AES key for encrypting data and then use the password to wrap the AES key and store the wrapped key on the disk [21]. This is essentially equivalent to deriving the key from the password. The wrapped key now becomes an oracle, against which the attacker can run the exhaustive search.

The third method is to store the key in a dencentralized network. Along this line, Geambasu *et. al.* propose a solution called Vanish, which generates a random key to encrypt the user's data locally and then distributes shares of the key using Shamir's secret sharing scheme to a global-sale, peer-to-peer, distributed hash tables (DHTs). The shares of the key naturally disappear (vanish), due to the fact that the DHT is constantly changing. However, Wochok *et. al.* [32] subsequently show two Sybil attacks that work by continuously crawling the DHT and recovering the stored key shared before they vanish. They conclude that the original Vanish scheme cannot guarantee the secure deletion of the key.

The fourth method is to store the key in a tamper resistant hardware module (e.g., TPM) and define the Application Programming Interface (API) to manage the stored keys. This is in line with the standard practice employed in financial industry for key management [3]. In this paper, we will adopt the same TPM-based approach. However, the main difficulty with the TPM lies in how the API should be defined. In 2005, Perlman first proposed to use a TPM for assured data deletion [24]. In her solution, data is always encrypted before being saved onto the disk. All decryption keys are stored in a tamper resistant module and do not live outside the module. Erasing the keys will effectively delete the data. To delete a key, the user simply sends a delete command to the module with a reference to that key and receives a one-bit confirmation if the operation is successful. Clearly, this design still follows the one-bit return protocol, which assumes complete trust on the correct implementation of the software inside the module.

1.3 Motivation for public verifiability

There are similar examples of black-box systems in security. For instance, as explained in [19], the Direct Recording Electronic (DRE) e-voting machines, widely used in the US between 2000 and 2004, worked like a black box. The system returns a tally at the end of the election, which the voters have to trust but cannot easily verify. The lack of verifiability had raised widespread suspicion about the integrity of the software inside the voting machine and hence the integrity of the election, eventually forcing several states in the US to abandon DRE machines. Today, the importance of having public verifiability in any e-voting system has been commonly acknowledged and progress is being made in deploying verifiable e-voting in real-world elections [2], [6].

Unfortunately, the need for public verifiability has been almost entirely neglected in the secure data erasure field. This is an important omission that we aim to address in this research work.

When a TPM is used for key management, the trust assumption about the TPM becomes a critical question. In the past literature [3], there exist two disparate assumptions about TPM: either completely trust or totally distrust. However, we find neither of

such black/white assumptions is adequate in capturing the reality. On one hand, the fact that a TPM stores cryptographic keys implies an inherent trust. But on the other hand, the encapsulated nature of a TPM prevents users from verifying the internal software, which inevitably adds distrust. These seemingly contradictory dual-facets are echoes of similar problems in e-voting, where a DRE machine is used as a trusted device to record votes, but the public have no access to its internal code. The established solution to address this dilemma is "trust-but-verify" [2], [6], [15]: i.e., demanding the voting machine to produce additional cryptographic proofs such that by verifying the correctness of those proofs a voter can gain confidence about the integrity of the internal software (this is also succinctly summarized by Ron Rivest and John Wack as the "software independence" principle).

Summary of main idea. The main idea of this work follows the same design principle based on "trust-but-verify". By applying cryptographic techniques, we allow an end user to verify the correct implementation of two important operations inside a TPM: encryption and deletion.

First, the user is able to explicitly verify that the encryption follows the correct procedure (i.e., the ciphertext is free from containing any trap-door block). By contrast, previous cryptography-based data deletion solutions only provide *implicit* assurance: by checking if the decryption produces the same original plaintext, one gains implicit assurance about the correctness of the encryption. However, we argue that such an *implicit* assurance is inadequate (in light of Snowden revelations [40]): a TPM manufacturer might be coerced by a state-funded adversary to compress a trap-door block into the ciphertext so to keep the output length the same. The user will not be able to notice any difference and the decryption can still produce the original plaintext (we will explain more details in Section 6.2.2). This issue will be addressed in our solution through the Audit function.

Second, the user is able to verify the outcome of a deletion process. Obviously, because using software means can never guarantee the complete deletion of data, verifying the successful erasure of data appears intuitively impossible. However, "you normally change the problem if you can't solve it." (David Wheeler [31]) Here, we slightly change the problem by shifting verifying the successful deletion of data to verifying the failure of that operation. The deletion process returns a digital signature, which cryptographically binds the deletion program's commitment of deleting a secret key to the outcome of that operation. In case the supposedly deleted key is recovered later, the signature can serve as publicly verifiable evidence to prove the vendor's liability. More technical details will be explained in Section 4 after we cover the related work in Section 2 and the relevant cryptographic primitives in Section 3. Section 5 explains the proof-of-concept implementation with detailed performance measurements, followed by security analysis in Section 6. Finally, Section 7 concludes the paper.

2 Related work

In this section, we review related works that discuss the importance of verifiability for secure data deletion.

In 2010, Paul and Saxena [22] aim to give users the ability to verify the outcome of secure data deletion. They propose a scheme called the "Proof of Erasability" (PoE), in which a host program deletes data by overwriting the disk with random patterns and the disk must return the same patterns as the proof of erasability. Clearly, this so-called proof is not cryptographically binding, nor publicly verifiable, since the data storage system may cheat by echoing the received patterns without actually overwriting the disk.

In ESORICS'10, Perito and Tsudik [23] study how to securely erase memory in an embedded device, as a preparatory step for updating the firmware in the device. They propose a protocol called Proofs of Secure Erasure (PoSE-s). In this protocol, the host program sends a string of random patterns to the embedded device. To prove that the memory has been securely erased, the embedded device should return the same string of patterns. It is assumed that the embedded device has limited memory - just enough to hold the received random patterns. This protocol works essentially the same way as the PoE in [22], but with an additional assumption of bounded storage.

Finally, in 2012, Wanson and Wei [34] investigate the effectiveness of the built-in data erasure mechanisms in several commercial Solid State Drives (SSDs). They discovered that the built-in "sanitize" methods in several SSD were completely ineffective due to software bugs. Based on this discovery, they stress the importance of being able to independently verify the data deletion outcome. They propose a verification method that works as follows. First of all, a series of recognizable patterns are written to the entire drive. Then, the drive is erased by calling the built-in "sanitize" command. Next, the drive is manually dismantled and a custom-built probing tool (made by the authors) is used to read raw bits from the memory in search for any unerased data. This approach can be useful for factory testing. However, it may prove difficult for ordinary users to perform.

In summary, several researchers have recognized the importance of *verifiability* in the secure data deletion process and proposed some solutions. But none of those solutions have used any cryptography. Our work differs from theirs in that we aim to provide *public verifiability* for a secure data deletion system by adopting public key cryptography.

3 CRYPTOGRAPHIC PRIMITIVES

In this section, we explain two relevant cryptographic primitives: the Diffie-Hellman Integrated Encryption Scheme (DHIES) and Chaum-Pedersen Zero Knowledge Proof.

3.1 DHIES

The DHIES is a public key encryption system adapted from the Diffie-Hellman key exchange protocol and has been included into the draft standards of ANSI X9.63 and IEEE P1363a [1]. The scheme is designed to provide security against chosen ciphertext attacks. It makes use of a finite cyclic group, which for example can be the same cyclic group used in DSA or ECDSA [29]. Here, we use the ECDSA-like group for illustration. Let E be an underlying elliptic curve for ECDSA and G be a base point on the curve with the prime order n.

Assume the user's private key is v, which is chosen at random from $[1, n-1]$. The corresponding public key is $Q_v = v \cdot G$. The encryption in DHIES works as follows. The program first generates an ephemeral public key $Q_u = u \cdot G$ where $u \in_R [1, n-1]$. It then derives a shared secret following the Diffie-Hellman protocol: $S = u \cdot Q_v$. The shared secret is then hashed through a cryptographic hash function H, and the output is split into two keys: $encKey$ and $macKey$. First, the $encKey$ key is used to encrypt a message to obtain $encM$. Then, the $macKey$ key is used to compute a MAC tag from the encrypted message $encM$. The final ciphertext consists of the ephemeral key Q_u, the MAC tag and the encrypted message $encM$. This encryption process is summarized in Figure 1.

The decryption procedure starts with checking if the ephemeral public key Q_u is a valid element in the designated group – a step commonly known as "public key validation"[1]. Next, it derives the same shared secret value following the Diffie-Hellman protocol. Based on the shared secret, a hash function is applied to derive $encKey$ and $macKey$, according to Figure 1. Upon the successful validation of the MAC tag by using the $macKey$, the encrypted message will be decrypted accordingly by using the $encKey$. More details about DHIES can be found in [1].

It is worth noting that DHIES is essentially built on the Diffie-Hellman key exchange protocol, but with adaptations to make it suitable for a secure data storage application. For example, Alice can encrypt a message under her own public key using DHIES, so that only she can decrypt the message at a later time.

1. The original DHIES paper [1] does not explicitly mandate public key validation on the ephemeral public key, but as explained by Antipa *et al.* in [4], the security proofs in DHIES [1] implicitly assume the received points must be on the valid elliptic curve; otherwise, the scheme may be subject to invalid-curve attacks. In our specification, we regard such public key validation as a mandatory step.

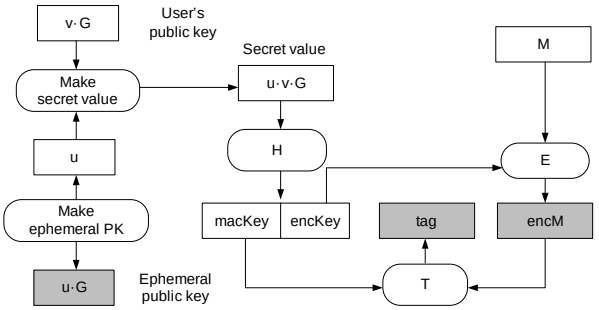


Figure 1: Encrypting with DHIES [1]. The symmetric encryption algorithm is denoted as E, the MAC algorithm as T and the hash function as H. The shaded rectangles constitute the ciphertext.

In some sense, it is like Alice securely communicating with herself in the future.

For any key exchange protocol, there is always a key confirmation step, which is either *implicit* or *explicit* [29]. The original DHIES scheme is designed to provide only *implicit* key confirmation – the key is implicitly confirmed by checking the MAC tag. However, there are two drawbacks with this approach. First, it does not distinguish two different failure modes in case the MAC verification is unsuccessful. In the first mode, wrong session keys may have been derived from the key exchange process. For example, the message had been encrypted by a different key $v' \cdot G$, $v' \neq v$. In the second mode, the encrypted message $encM$ may have been corrupted (due to storage errors or malicious tampering). It is sometimes useful for an application to be able to distinguish the two modes and handle the failure accordingly, but this is not possible in the original DHIES. The second drawback is performance. In DHIES, the latency for performing implicit key confirmation (through checking MAC) is always linear to the size of the ciphertext. However, this linear time complexity $O(n)$ can prove unnecessarily inefficient if the MAC failure was due to the derivation of wrong session keys. (We will explain more on this after we describe the Audit function in Section 4.)

We address both limitations by adding an *explicit* key confirmation step to DHIES. This change provides explicit assurance on the correct derivation of the session keys. It is consistent with the common understanding that in key exchange protocols, *explicit* key confirmation is generally considered more desirable than *implicit* key confirmation [29]. We will explain the modified DHIES in detail in Section 4.

3.2 Chaum-Pedersen protocol

Assume the same Elliptic Curve setting (E, G, n) as above. Given a tuple $(G, X, R, Z) = (G, x \cdot G, r \cdot G, x \cdot r \cdot G)$ where $x, r \in_R [1, n-1]$, the Chaum-Pedersen protocol is an honest verifier Zero-Knowledge Proof (ZKP)

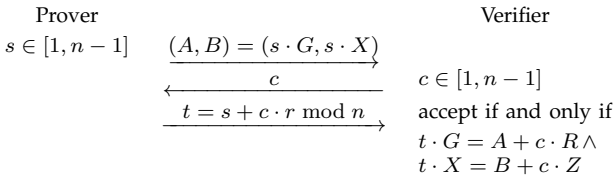


Figure 2: Chaum-Pedersen protocol [9]: a zero-knowledge proof technique to prove the statement that $(G, X, R, Z) = (G, x \cdot G, r \cdot G, x \cdot r \cdot G)$ is a DDH tuple.

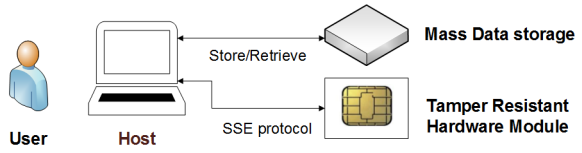


Figure 3: System overview

technique for proving that the tuple (G, X, R, Z) is a Decisional Diffie-Hellman (DDH) tuple [9]. This is equivalent to proving that $\log_G X = \log_R Z$, or alternatively, $\log_G R = \log_X Z$. For the Chaum-Pedersen protocol to work, the prover must know either the r or x value. Without loss of generality, we assume the prover knows r. The Chaum-Pedersen protocol works interactively between a prover and a verifier in three message flows, as shown in Figure 2. In our solution, we use a non-interactive variant of the Chaum-Pedersen protocol, which is realized by applying the standard Fiat-Shamir heuristics [10].

4 System Design

In this section, we will propose a Secure Storage and Erasure (SSE) system. As shown in Figure 3, in an architectural view, the system comprises three components: 1) a tamper resistant hardware module handles key management; 2) a disk drive that stores digital data; 3) a host program that controls the disk drive and communicates with the module, through a SSE protocol. In the paper, we will use TPM to refer to the tamper resistant hardware module.

One core functionality of a TPM is to be "tamper resistant", so that secrets can be safely kept inside. However, many past research works have demonstrated that it might be possible to extract secrets from a TPM in various ways, e.g., semi-invasive attacks, API attacks and side-channel attacks [3]. Hence, it is prudent not to assume the "tamper resistance" in its absolute term. Instead, we acknowledge the possibility that a TPM might be reverse-engineered and its secrets extracted. However, we assume such attacks will incur a high cost. Under this assumption, a TPM is still useful as long as the cost of reverse-engineering is significantly higher than the value of the data that the TPM protects.

4.1 Threat model

In our threat model, we will consider threats from three different angles: the data thief, the TPM provider and the user.

First, the obvious threat concerns a data thief who has captured the entire system (TPM, host and disk) and whose goal is to recover the deleted data. We assume the attacker is able to not only read all unerased data from the disk but also recover overwritten bits on the disk. However, since the data is all encrypted, the attacker must have access to the decryption keys, which are stored in the secure memory of the TPM. We assume the cost of reverse-engineering a TPM is higher than the value of the data that it protects.

The second type of threat comes from the TPM provider. In general, the TPM provider should have no business incentives to install malicious firmware in the TPM. However, two possible scenarios need to be considered. First, the firmware may contain software bugs. Second, the TPM provider might be coerced by a state-funded security agency to add trapdoors in its product[2]. In our threat model, we assume that the TPM is sold in a mass commercial market, hence any bugs or trapdoors (if any) will exist in not just one TPM, but all products in the market. In other words, we do not consider targeted attacks against a particular user.

Third, we consider the threat from a user. A user differs from a data thief in that she is the legitimate possessor of the TPM and holds the Service Level Agreement. It is expected that the user only saves the encrypted data onto the disk, so that later the data can be deleted by just erasing the key. However, a mishaving user may deviate from this expectation as follows. In parallel to saving the encrypted data onto the disk, she also backs up the plaintext data in some secret location. In that case, simply erasing the key is useless to delete the data. In our model, we do not consider this threat as it trivially breaks all cryptography-based data deletion methods. Second, a user might try to reverse-engineer the TPM and claim compensation based on the SLA. We will further analyze this scenario in Section 6 after we explain the full protocol in the next section.

4.2 SSE protocol

The TPM communicates with the host, following a Secure Storage and Erasure (SSE) protocol. This protocol is the central element in the entire system design. It operates in the same group setting as ECDSA (or DSA). Here, we choose the ECDSA setting, so it is consistent with the actual implementation of the

2. This seemingly remote threat becomes realistic in the light of the recent revelations by Edward Snowden [40]. We believe in the post-Snowden world people will take an even more critical view on TPM, and our "trust-but-verify" paradigm is one step towards addressing that concern.

protocol in a Java Card, as we will explain in Section 5. As before, let E be the underlying elliptic curve of ECDSA and G be a base point on the curve with the prime order n.

Each TPM contains a unique ECDSA signature key pair: Prv_t and Pub_t, which are generated on-board during the factory initialization stage. The ECDSA public key for every TPM is published on the TPM provider's website so that anyone can access it, while the private key is securely kept inside the TPM. As an overview, the SSE protocol specifies the following API functions:

- ***KeyGen***. To generate a random public/private key pair;
- ***Encrypt***. To encrypt data with a specified public key;
- ***Decrypt***. To decrypt data with a specified private key;
- ***Audit***. To audit if encryption was done correctly;
- ***Delete***. To delete a specified private key with a digital signature returned as a proof of deletion.

To call the above functions, the user must be authenticated first. This can be realized in several ways: for example, passwords, biometrics, etc. For simplicity, we assume the user has passed the authentication and can call the functions. Details of each API function are explained below (the notations are summarized in Table 1).

4.2.1 Key generation

$\mathsf{KeyGen}(1^k, C)$ creates an instance of the client user C. It takes as input a security parameter 1^k and the identity of the user C, generates a private key on-board $\mathrm{Prv}_{C_i} := d_{C_i} \in_R [1, n-1]$, and returns the corresponding public key $\mathrm{Pub}_{C_i} := d_{C_i} \cdot G$ and an index reference C_i to the created key pair. The user C is free to create as many instances as she wishes, subject to the constraint of the maximum persistent memory in the TPM. As an example, with 160-bit n, 32-bit index $\mathcal{C}_i$ and a TPM of 16 MB EEPROM memory (see [38]), up to 666,667 user instances can be created. The user may choose to use different instances for encrypting different types of files. The KeyGen function can be formalized as below (for simplicity, we will omit the return of error in all functions):

$$\begin{array}{lcll} \mathrm{Host} & \rightarrow & \mathrm{TPM} & : \quad 1^k,\ C \\ & & \mathrm{TPM} & : \quad \text{Generate } \mathrm{Prv}_{C_i} := d_{C_i} \\ \mathrm{TPM} & \rightarrow & \mathrm{Host} & : \quad \mathrm{Pub}_{C_i} := d_{C_i} \cdot G,\ C_i \end{array}$$

4.2.2 Encryption

$\mathsf{Encrypt}(C_i, m)$ takes as input the reference to the created user instance C_i, a message m and returns the encrypted message under the public key Pub_{C_i}. For the encryption, we adopt the Diffie-Hellman Integrated Encryption Scheme (DHIES) [1]. First, the TPM generates an ephemeral public key $Q_\eta = d_\eta \cdot G$ where $d_\eta \in_R [1, n-1]$. It then calculates two session keys, which include an encryption key $k_\eta^{enc} = H(d_{C_i} \cdot Q_\eta \,||\, 0\mathrm{x}01)$ and a MAC key $k_\eta^{mac} = H(d_{C_i} \cdot Q_\eta \,||\, 0\mathrm{x}10)$. Both keys are used to encrypt the message in an authenticated manner to obtain $E_{k_\eta}^{\mathrm{Auth}}(m)$. In addition, the TPM generates a key-confirmation key $k_c = H(d_{C_i} \cdot Q_\eta \,||\, 0\mathrm{x}11)$ and outputs a one-way hash of k_c. This is to allow explicit key confirmation during the latter decryption and audit steps. The returned ciphertext will be stored in the mass storage device. The encryption is performed inside the TPM as it involves securely generating a random factor (i.e., d_η), whose secrecy also needs to be protected. It is possible to perform public key encryption in a host computer, but the standard industry solution is to do that in a tamper resistant device so that all security-sensitive key materials are protected by the tamper resistance [3]. The Encrypt function can be formalized as:

$$\begin{array}{lcll} \mathrm{Host} & \rightarrow & \mathrm{TPM} & : \quad C_i,\ m \\ \mathrm{TPM} & \rightarrow & \mathrm{Host} & : \quad Q_\eta := d_\eta \cdot G,\ H(k_c),\ E_{k_\eta}^{\mathrm{Auth}}(m) \end{array}$$

4.2.3 Decryption

$\mathsf{Decrypt}(C_i, Q_\eta, H(k_c), E_{k_\eta}^{\mathrm{Auth}}(m))$ takes as input the reference to an existing user instance C_i, the ciphertext obtained from the earlier encryption step, and returns the decrypted message if the verifications on the key confirmation string and MAC are successful. The TPM first validates that Q_η is a valid public key on the curve. It then computes $k_c' = H(d_{C_i} \cdot Q_\eta \,||\, 0\mathrm{x}11)$ and proceeds to decryption only if $H(k_c') = H(k_c)$. The decryption procedure follows subsequently as described in DHIES [1]. Upon the successful verification of the MAC tag, the encrypted message will be decrypted and the original plaintext m will be returned. The Decrypt function can be formalized as:

$$\begin{array}{lcll} \mathrm{Host} & \rightarrow & \mathrm{TPM} & : \quad C_i,\ Q_\eta,\ H(k_c),\ E_{k_\eta}^{\mathrm{Auth}}(m) \\ \mathrm{TPM} & \rightarrow & \mathrm{Host} & : \quad m \end{array}$$

4.2.4 Audit

$\mathsf{Audit}(C_i, Q_\eta, H(k_c))$ takes as input the reference to an existing user instance C_i, the ephemeral public key Q_η and the key confirmation string $H(k_c)$, and allows the user to verify whether the earlier encryption operation was done correctly. The TPM first checks that Q_η is a valid public key on the curve, and verifies if $H(H(d_{C_i} \cdot Q_\eta \,||\, 0\mathrm{x}11)) = H(k_c)$. It then outputs $d_{C_i} \cdot Q_\eta$ and a Zero Knowledge Proof (ZKP), which proves that $\log_G d_{C_i} \cdot G = \log_{Q_\eta} d_{C_i} \cdot Q_\eta$ without leaking anything about the private key d_{C_i}. The ZKP is based on the Chaum-Pedersen protocol [9], which is made non-interactive by applying the Fiat-Shamir heuristics [10]. Because of the use of a key-confirmation string, it is unnecessary to feed in the entire encrypted message (i.e., $E_{k_\eta}^{\mathrm{Auth}}(m)$) into the audit function input. This improves the efficiency as the size of the encrypted

Notations	Meaning
Prv_t, Pub_t	A pair of unique ECDSA keys for each TPM
C	The client user
C_i	One instance of the client user
Prv_{C_i}	The private key of the client instance, $\mathrm{Prv}_{C_i} := d_{C_i}$
Pub_{C_i}	The public key of the client instance, $\mathrm{Pub}_{C_i} := d_{C_i} \cdot G$
m	An input message
Q_η	The ephemeral public key during DHIES $Q_\eta = d_\eta \cdot G$
$k_\eta^{enc}, k_\eta^{mac}$	The session keys derived from DHIES for authenticated encryption
k_c	The key-confirmation key derived from DHIES for explicit key confirmation
$E^{\mathrm{Auth}}_{k_\eta}(m)$	Authenticated encryption of m using the session keys $\{k_\eta^{enc}, k_\eta^{mac}\}$
$E(\mathrm{Pub}_{C_i}, m)$	Encryption of m under Pub_{C_i} using DHIES, $E(\mathrm{Pub}_{C_i}, m) := \{Q_\eta, H(k_c), E^{\mathrm{Auth}}_{k_\eta}(m)\}$
η	The reference to the ciphertext $E(\mathrm{Pub}_{C_i}, m)$
ZKP_η	A Zero Knowledge Proof to prove the well-formedness of ciphertext η
$\mathrm{SLA}^{\mathrm{del}}_{C_i}$	A Service Level Agreement for the deletion of client instance C_i
$Sig(...)$	A signed message using the TPM's ECDSA private key Prv_t

Table 1: Notations and meaning

message may potentially be large. With the output from the audit function, the host is able to compute the encryption and MAC keys based on $k_\eta^{enc} = H(d_{C_i} \cdot Q_\eta \,||\, 0\text{x}01)$ and $k_\eta^{mac} = H(d_{C_i} \cdot Q_\eta \,||\, 0\text{x}10)$. With these symmetric keys, the host is able to fully verify if the message was encrypted correctly using these keys. Note that this auditing only reveals the symmetric encryption and MAC keys within one DHIES session; the secrecy of the keys derived in other sessions is not affected. The Audit function can be formalized as:

$$\begin{array}{lcll} \mathrm{Host} & \rightarrow & \mathrm{TPM} & : \quad C_i,\ Q_\eta,\ H(k_c) \\ \mathrm{TPM} & \rightarrow & \mathrm{Host} & : \quad d_{C_i} \cdot Q_\eta,\ \ldots \\ & & & \quad \mathrm{ZKP}_\eta\left[\log_G d_{C_i} \cdot G = \log_{Q_\eta} d_{C_i} \cdot Q_\eta\right] \end{array}$$

4.2.5 Delete

Delete(C_i) deletes a user instance C_i by overwriting its private key d_{C_i} in the TPM's protected memory and returns $\mathrm{SLA}^{\mathrm{del}}_{C_i}$, which is a Service Level Agreement {"Delete", Pub_{C_i}} signed by the TPM's ECDSA signing key. After the erasure of the private key, all messages encrypted under Pub_{C_i} can no longer be decrypted. Assume the TPM had failed to erase the private key d_{C_i} properly and that the key is later discovered by the user. The user can present d_{C_i} together with $\mathrm{SLA}^{\mathrm{del}}_{C_i}$, as publicly verifiable evidence, that the TPM had failed to provide the secure data deletion service as promised. Based on the evidence and the terms in the Service Level Agreement, the user should be entitled to compensation (or money back). The Delete function can be formalized as:

$$\begin{array}{lcll} \mathrm{Host} & \rightarrow & \mathrm{TPM} & : \quad C_i \\ \mathrm{TPM} & \rightarrow & \mathrm{Host} & : \quad \mathrm{SLA}^{\mathrm{del}}_{C_i} := Sig(\text{“Delete”}, \mathrm{Pub}_{C_i}) \end{array}$$

5 Implementation

In this section, we will describe a full prototype implementation of the proposed SSE system, based on using a standard Java card [39] as a TPM for key management, a MacBook laptop (1.7 GHz with 4 GB memory) for the host and a standard disk drive for mass data storage. As we will show, this is a non-trivial development effort. To our knowledge, what we provide is the first public implementation of DHIES and Chaum-Pedersen ZKP on a resource constrained Java card. (The full source code for the prototype can be found at the end of the paper.)

The Java card we use has a dual interface, supporting both contact and contactless communication. We use the contactless interface for all experiments. The chip on the card has an 80 KB EEPROM for persistent storage and an 8 KB RAM for holding volatile data in memory. The card is compliant with Java Card Standard 2.2.2, but also supports some additional APIs from Java Card Standard 3.0.1. In particular, it supports *ALG_EC_SVDP_DHC_PLAIN* under the *javacard.security.KeyAgreement* interface, which allows obtaining the plain shared secret (instead of a SHA-1 hash of the secret) from the Elliptic Curve Diffie-Hellman (ECDH) key exchange protocol. This API is essential for the prototype implementation of our system.

One obstacle we encountered is that the existing Java card API standards does not support modular multiplication of big numbers (see [30], [39]). To the best of our knowledge, no Java Cards currently available in the market provide the API support to perform this basic modular operation. Therefore, we had to implement the big number modular multiplication from scratch by ourselves using the primitive arithmetic operators and byte arrays (without involving any hardware support from the low-level native C library on the card). It takes about 150 lines of Java code to execute one modular multiplication.

Regarding the elliptic curve setting, we chose the standard P-256 curve as defined in the Digital Signature Standards specification [37]. When the Java card applet is first loaded into the chip, upon initialization it generates a random ECDSA key pair over the P-256 curve. The same curve is used for the generation of all further public/private key pairs required.

In the following, we will explain the implementation details and performance measurements of all the functions specified in the SSE protocol. For each function, the latency is measured in terms of the delay in the card processing and in the card communication (via the contactless interface). We repeated the experiments thirty times and summarize the average results in Figure 4 and Table 2.

KeyGen. This function involves generating a random public/private key pair over the P-256 curve for a new user instance. The public key, along with a 16-bit unique identifier, is returned to the user. The private key (32 bytes) is stored in the TPM's EEPROM. To facilitate the encryption operation later, we also keep the public key in EEPROM. The card only supports the EC public key in the uncompressed form, so the size of the public key is 64 bytes. Given that the Java card that we use has 80 KB EEPROM in total and that the SSE program takes up 16 KB storage in EEPROM, we can create about 650 random EC public/private pairs. As shown in Table 2, this operation takes a constant 835 ms in total.

Encrypt. The function receives a plaintext file, encrypts it using DHIES and returns the ciphertext. In one DHIES session, two symmetric session keys are derived to encrypt the file in an authenticated manner (see Figure 1). In theory, there should be no limit in how long is the input file that can be encrypted under one DHIES session. However, in practice, there is an upper limit due to the constrained memory size in the Java card. (The reason shall become more evident later when we explain how the Decrypt operation works.) In our implementation, up to 2 KB data can be encrypted in one DHIES session. For a plaintext file bigger than 2 KB, the host program needs to divide the file into block with each less than 2 KB and encrypt each block in one DHIES session.

Another constraint in the implementation is the size of the APDU buffer. The card receives and sends messages through an APDU buffer, which can hold data up to 255 bytes at one time. Therefore, for a long message, the encryption cannot be done in one operation, and needs to be done in four steps. First, the card receives an instance ID that identifies the public key. Accordingly, it creates an ephemeral public key, and computes the DHIES session keys. The session keys comprise a 128-bit AES key for encrypting data and another 128-bit AES key for computing MAC. The encryption is performed in the CBC mode. A random IV for AES-CBC is generated and returned to the host in this step (this to optimize the bandwidth usage so that in the subsequent step, the plaintext data can fill up the entire APDU buffer and the returned ciphertext will occupy the whole buffer as well). Second, the message is divided into segments with each segment not more than 255 bytes. The card receives each segment in turn, performs encryption and saves the intermediate results in RAM. This step is repeated until the penultimate segment of the message. Third, it receives the last segment of the message and finalizes the encryption. Fourth, a MAC is returned, which is computed over the entire ciphertext using CBC-MAC. Full implementation details about the DHIES encryption are summarized in Algorithm 1

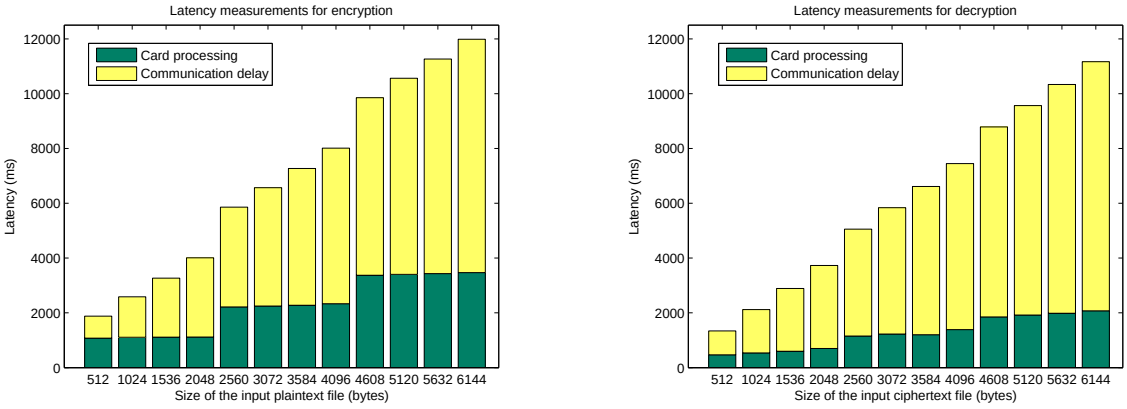
Algorithm 1 Encryption in one DHIES session

Input: User instance reference C_i, message m, elliptic curve E with generator G of order n, secure hash function H;
Output: Ephemeral public key Q_η, hashed key confirmation key $H(k_c)$, encrypted message $E^{Auth}_{k_\eta}(m)$; initialisation vector IV;

1: Client sends C_i to card;
2: Card retrieves instance private key d_{C_i} corresponding to user instance C_i;
3: Card randomly chooses $d_\eta \in [1, \dots n]$;
4: Card sets $Q_\eta = d_\eta.G$;
5: Card sets $k^{enc}_\eta = H(d_{C_i}.Q_\eta \parallel 0x01)$;
6: Card sets $k^{mac}_\eta = H(d_{C_i}.Q_\eta \parallel 0x10)$;
7: Card generates random IV and returns this to the client;
8: Client divides m into segments m_i with each segment not more than 255 bytes;
9: **for** segments m_i **do**
10: Client sends m_i to card;
11: Card generates $E_{k_\eta}(m_i)$ using AES-CBC with key k^{enc}_η;
12: Card generates MAC_i for $E_{k_\eta}(m_i)$ using AES-CBC with key k^{mac}_η and with initialisation vector set to ZERO when $i = 0$ and MAC_{i-1} when i>0;
13: Card obtains the final MAC with the entire encrypted message to give $E^{Auth}_{k_\eta}(m)$;
14: Card sets $k_c = H(d_{C_i}.Q_\eta \parallel 0x11)$;
15: Card returns to the client Q_η, $H(k_c)$, $E^{Auth}_{k_\eta}(m)$;

The latency measurements for the encryption operation are shown in Figure 4a. For each input file of different sizes, the Encrypt operation is invoked to encrypt the file and return the ciphertext. The measured total latency includes both the card processing and card communication delays. In order to obtain the communication delay, we conduct a separate experiment. We add a dummy API to the card, which works superficially similar to Encrypt in that it accepts an input file and returns an output file that has the same size as what the Encrypt API would return. However, the dummy API does not perform any processing on the input data and it immediately outputs a fixed data string that is stored in the card memory back to the host. We measure the latency of calling the dummy API and take that measurement as the communication delay. The card processing delay is obtained by subtracting the communication delay from the total latency.

As shown in Figure 4a, the card processing delay in the Encrypt operation increases with the size of the input in a step-wise manner. This is because we limit the maximum allowed plaintext data that can be encrypted within one DHIES session to be 2 KB. Hence, for the input size of less than 2 KB, the card processing cost is predominantly determined by the public key operations in DHIES to derive the session keys. The cost of the subsequent symmetric operations using the session keys is almost negligible in comparison to



(a) Performance of Encrypt

(b) Performance of Decrypt

Figure 4: Performance evaluation based on a proof-of-concept implementation using a resource-constrained Java card (5 MHz processor). The same implementation should work several hundred times faster on a high-performance Tamper Resistant Module such as IBM Storage Manager HSM (2 GHz processor) [43].

asymmetric operations. For the input size of between 2 KB and 4 KB, the card processing cost is almost doubled because the encryption involves two DHIES sessions.

Decrypt. As previously, due to the limited size of the APDU buffer, the ciphertext has to be divided into segments, with each segment not more than 255 bytes. In the implementation, this operation has five steps. First, the card receives the instance ID, the ephemeral public key, the key confirmation string and the IV (for AES-CBC decryption). After it successfully verifies the key confirmation string, the card computes a 128-bit AES encryption key and another 128-bit AES MAC key. Second, it receives each ciphertext segment in sequence, decrypts each segment using the AES encryption key derived in step one and stores the decrypted result in RAM. Meanwhile, it computes a MAC for the received ciphertext using AES-CBC. This step is repeated until receiving the penultimate segment of the ciphertext. (The computed MAC becomes the IV input for computing the next MAC using AES-CBC.) Third, it receives the last segment of the ciphertext. It decrypts the segment accordingly, saves the decrypted data to an array in RAM, and also computes the final MAC. Fourth, it receives a MAC. It checks it against the MAC that was derived in the previous step. Fifth, it returns the decrypted plaintext if the MAC was verified successfully in the previous step. The last step is called repeatedly until all plaintext data is returned. All the intermediate results during the cryptographic operations are stored in the volatile RAM. (Writing data into EEPROM is much slower, and is subject to a limited number of writing cycles, while writing data in RAM is fast and incurs no limit in the number of overwriting operations.) The decryption process is summarized in Algorithm 2.

Since the card only returns the plaintext upon successful verification of the MAC value, the maximum allowed size of the ciphertext is determined by the available RAM in the card. In the Java card that we use, the chip has 8 KB RAM, more than half of which is used to run the instance of the program. Through experiment, we found that the maximum data that the card can accommodate in RAM is 2 KB.

As shown in Figure 4b, the latency of the decryption increases with the size of the ciphertext file in a similar step-wise manner as in the encryption. As compared with the encryption that involves two scalar multiplications over the elliptic curve, the decryption only requires one. Hence, the latency of card processing in

Algorithm 2 Decryption in one DHIES session

Input: User instance reference C_i, ephemeral public key Q_η, hashed key confirmation key $H(k_c)$, encrypted message $E^{Auth}_{k_\eta}(m)$, elliptic curve E with generator G of order n, secure hash function H, initialisation vector IV;
Output: Message m or failure notification;

1: Client sends C_i, Q_η, $H(k_c)$, IV to card;
2: Card retrieves the instance private key d_{C_i} corresponding to user instance C_i;
3: Card validates Q_η is a point on E of correct order;
4: Card sets $k'_c = H(d_{C_i}.Q_\eta \parallel 0x11)$;
5: **if** $H(k'_c) = H(k_c)$ **then**
6: Card Sets $k^{enc}_\eta = H(d_{C_i}.Q_\eta \parallel 0x01)$;
7: Card sets $k^{mac}_\eta = H(d_{C_i}.Q_\eta \parallel 0x10)$;
8: Client divides $E^{Auth}_{k_\eta}(m)$ into segments M_i
9: **for** segments M_i **do**
10: Client sends M_i to card;
11: Card sets m_i to be the decryption of M_i using AES-CBC and key k^{enc}_η and IV;
12: Card generates a MAC_i for M_i using key k^{mac}_η with initialisation vector set to ZERO when $i = 0$ and MAC_{i-1} when i>0;
13: Card verifies that the final MAC generated equals the MAC included with $E^{Auth}_{k_\eta}(m)$;
14: **if** MAC verification succeeds **then**
15: Card returns all m_i to client;
16: **else**
17: Card returns failure notification to client;
18: **else**
19: Card returns failure notification to client;

decryption (Figure 4b) is about half of that in encryption (Figure 4a). The latency of card communication remains the roughly same in both cases. For an input ciphertext file of 1 KB, the decryption takes about 2 seconds in total (0.5 seconds on the card processing).

Audit. This function requires the card to prove that the two session keys in an earlier Encrypt operation had been derived correctly following the DHIES specification. The main part in the implementation is in computing a Zero Knowledge Proof to prove the equality of two discrete logarithms. The implementation needs two primitive functions. The first is to compute the scalar multiplication over the Elliptic Curve and the second is to compute a modular multiplication of two big numbers (32-byte modulus). Although the Java Card Standard 2.2.2 [39] does not provide any direct API to allow computing the scalar multiplication over the EC, it is possible to (ab)use the ECDH API as follows. First, the ECDH API is initialized with a pair of public/private keys, where the private key is the scalar. Upon receiving an ephemeral public key, the ECDH API does a scalar multiplication over the elliptic curve and returns the ECDH shared secret in the plain form. However, instead of returning a point, the API only returns the x coordinate (more specifically, the x coordinate of $B = s \cdot X$ in the first flow of the Chaum-Pedersen protocol; see Figure 2.) Hence, the host has to reconstruct the point by calculating the y coordinate from one of the two possible values. Once the whole point is reconstructed, the Zero Knowledge Proof can be verified accordingly. The limitation in the card API reduces the security level of the ZKP by exactly one bit, because it halves the search space of an exhaustive search attack.

As shown in Table 2, the audit function causes $10,594$ ms delay in the card processing. The most significant cost factor is in doing the modular multiplication (i.e., computing $t = s + c \cdot r \bmod n$ in the last step of the Chaum-Pedersen protocol; see Figure 2). It takes $9,094$ ms. This seemingly trivial calculation incurs a long delay because there is no available API in the Java card to do this operation efficiently and we had to implement it from scratch in pure software without any hardware support. We have tried our best to optimize the code and also compared with alternative methods (e.g., do a modular multiplication by abusing the RSA encryption API as described in [30]). It seems that the 9.094 seconds delay is probably best we could achieve without getting any hardware support from the card cryptographic co-processor[3]. It is worth noting that the latency in audit is a constant value. This is attributed to the use of explicit key confirmation; otherwise, with the original DHIES, we will have to feed in the entire encrypted message and the latency for auditing will have a linear time complexity $O(n)$.

3. We contacted several Java card vendors and were glad to learn from one vendor that adding native support for modular multiplication was in their development plan for future products.

Operations	Card processing	Communication	Total latency
KeyGen	782	53	835
Audit	10594	165	10759
Deletion	674	56	730

Table 2: Latency measurements (ms)

Delete. Upon receiving an index to the user instance, this function erases the private key for the specified user instance, by calling the *clearKey* method of the *javacard.security.key* interface. This follows the recommendation from the Java Card API Standard (2.2.2) that a key should be cryptographically destroyed through the clearKey method [39]. After the private key is erased, the function returns an ECDSA signature as specified in the SSE protocol. This delete operation takes about 730 ms delay in total (see Table 2).

6 Analysis

In this section, we analyze the security of the proposed system, including the API security and the threat analysis.

6.1 API security

In the SSE protocol, we have defined five API functions. The KeyGen function simply generates keys onboard. The Encrypt and Decrypt functions follow the widely standardized DHIES, which has been proven secure against chosen-ciphertext attacks [1]. We propose to add a key-confirmation string to DHIES in order to provide *explicit* key confirmation, while the original DHIES only provides *implicit* key confirmation. The key-confirmation key is derived separately from the encryption and MAC keys. This is to ensure that the encryption and MAC keys remain indistinguishable from random after the key confirmation step. Thus, the security proofs in DHIES [1] are not affected. The use of *explicit* key confirmation allows a more efficient implementation of the audit function. In the Delete function, we use the well-established ECDSA to cryptographically bind the TPM's commitment to delete a secret key with the outcome of the deletion operation. We refer the reader to [1] and [37] for the security of DHIES and ECDSA respectively. Here, we will focus on the Audit function.

The Audit function serves as an enhancement to DHIES. The aim is to allow users to verify if the encryption had been correctly implemented following the DHIES specification. To analyze the security of this function, we will consider two types of attackers: a passive attacker and an active attacker. We define a *passive* attacker as one who obtains the ciphertext only by calling the Encrypt function and subsequently feeds the obtained ciphertext into the Audit function. This is analogous to passively monitoring all inputs and outputs while the user performs the Encrypt and

Audit operations. We define an *active* attacker as one who constructs his own ciphertext and then feeds it into the Audit function.

Passive attack. First, we consider a passive attacker and make the following claim with a sketch of its proof.

Claim 1. *Under the assumption that the underlying Chaum-Pedersen ZKP is secure, the output of the audit function does not reveal any information about the private key d_{C_i} to a passive attacker.*

Proof: In the case of a passive attack, the input ciphertext will be successfully verified by the TPM since it was generated by the same TPM earlier. Given the input $\{d_\eta \cdot G, H(k_c)\}$, the audit function returns $\{d_{C_i} \cdot d_\eta \cdot G, \text{ZKP}_\eta\}$. The ZKP_η reveals nothing more than one bit information about the truth of the statement: the tuple $\{G, d_{C_i} \cdot G, d_\eta \cdot G, d_{C_i} \cdot d_\eta \cdot G\}$ is a DDH tuple[4] (see [9]). We assume that the audit function is called of an unlimited number of times. The passive attacker records every input and output, and eventually builds up a transcript of all possible tuples, each comprising $\{d_\eta \cdot G, d_{C_i} \cdot d_\eta \cdot G\}$ (recall that d_η is dynamic and d_{C_i} is static.). However, he can simulate the same transcript by generating the random values d_η by himself and computing $d_\eta \cdot d_{C_i} \cdot G$ accordingly. In conclusion, he learns nothing about d_{C_i} from the transcript that he can simulate all by himself . □

Active attack. Second, we consider an active attacker and make the following claim with a sketch of its proof.

Claim 2. *Under the assumption that the Computational Diffie-Hellman (CDH) problem in the designated group is intractable, and given that the ciphertext input, supplied by an active attacker, has passed the internal verification in the TPM, the input must have been generated with the knowledge of the ephemeral private key d_η.*

Proof: Assume the attacker has calculated the input to the audit function on his own, which includes $\{d_\eta \cdot G, H(k_c)\}$. To obtain a contradiction, we assume the attacker does not know d_η. Given the successful public key validation on $d_\eta \cdot G$, it shows that $d_\eta \cdot G$ is a valid public key in the designated group over the elliptic curve, so the discrete logarithm (i.e., the private key) with respect to the base point G must exist. In other words, the value d_η actually exists. Given the successful verification on the key confirmation, this gives the TPM explicit assurance that the supplier of the input must have obtained the same key-confirmation key k_c, which is derived from the ECDH shared plain secret through a one-way hash function: $k_c = H(d_{C_i} \cdot d_\eta \cdot G || 0x11)$. Hence, the attacker must have obtained the same ECDH shared plain secret. In summary, without knowing d_{C_i} or d_η, the attacker has computed $d_{C_i} \cdot d_\eta \cdot G$ from $\{d_{C_i} \cdot G, d_\eta \cdot G\}$. This contradicts the CDH assumption as stated in the claim. In conclusion, the active attacker must have known d_η when computing his own input to the audit function. □

Obviously, if the attacker knows d_η, he will learn nothing from the Audit function as he is able to compute the DDH tuple $\{G, d_\eta \cdot G, d_{C_i} \cdot G, d_{C_i} \cdot d_\eta \cdot G\}$ all by himself.

6.2 Threat analysis

In the threat model defined in Section 4, we have highlighted threats from three different angles. We now analyze those threats in detail.

6.2.1 Data thief

We assume the attacker has physically captured the TPM and the disk. Clearly, the attacker cannot make use of the TPM without passing the authentication mechanism. We further assume that the attacker has had the user's authentication credential, so he can invoke all API functions of the TPM. Obviously, if the keys have not been deleted, the attacker will be able to trivially decrypt the ciphertext stored on the disk. This is unstoppable as the attacker is essentially no different from a legitimate user from the system's perspective. The basic design goal of the SSE system is to prevent the attacker from recovering deleted data. Hence, before the system falls into the enemy hands, we assume that the user erases keys by calling the Delete function, or in the extreme case, physically destroying the TPM chip. The latter guarantees the complete erasure of the keys, but in our analysis we will focus on non-destructive means to delete data.

If the Delete function has been implemented correctly, the key should have been erased and its location in memory be overwritten with random data. This can prove extremely costly for the attacker to recover the deleted key; without the key, the attacker will have to do a ciphertext-only attack against DHIES, which has been proved infeasible [1].

In order to recover the deleted key, the attacker has to penetrate two layers of defence. First, he needs to bypass the physical tamper resistance, so he can gain access to the protected memory in the TPM. Second, he needs to recover the overwritten bits in the memory cells where the key was stored before the deletion. Compromising both layers is not impossible, but will incur a high cost to the attacker. This will be an arms race between defenders and attackers, but if the cost to attack is significantly higher than the value of the target data, the thief may be deterred.

6.2.2 TPM provider

As explained above, if the TPM has i) encrypted data correctly based on the DHIES algorithm, and ii) also

4. Since the non-interactive ZKP is obtained by applying the Fiat-Shamir heuristics, a random oracle model is assumed.

erased keys properly from the protected memory, it can prove prohibitively expensive for a data thief to recover the deleted data. However, we shall not take it for granted that the TPM provider must have implemented both operations correctly. Software bugs are one concern. We should also be wary of the possibility that the TPM provider might be coerced by a powerful state-funded adversary to insert a trapdoor into the products.

Instead of completely trusting the TPM, we adopt a "trust-but-verify" approach. More specifically, this "trust-but-verify" is reflected in the design of the SSE protocol in two aspects: verifiable encryption and verifiable deletion.

Verifiable encryption. First, the encryption should be verifiable. The SSE protocol allows the user to verify if the encryption has been implemented correctly following the DHIES specification. This verification is critical, because if the encryption had not been done correctly in the first place, then deleting the key will not logically lead to the deletion of data. In past work, such verification is usually done implicitly – the fact that the software program can reverse the encryption process and recover the same original plaintext gives implicit assurance that the encryption was done correctly. This kind of implicit verification is widely used in software testing to ensure the encryption and decryption are implemented correctly.

In a security-critical application, this kind of implicit assurance is insufficient, especially when the software program is encapsulated within a tamper resistant device and its source code is totally inaccessible. We provide one possible attack in Figure 5. Since the plaintext data normally contain redundancies, the TPM can compress the data first before doing encryption. The compression will create spare space to insert a trapdoor block, which is the decryption key wrapped by a trapdoor key (known to a state-funded security agency). Given that the ciphertext length remains the same and the encryption cipher is semantically secure (i.e., the output of the encryption is indistinguishable from random), users cannot distinguish the two ciphertexts in Figure 5. During the decryption, the TPM can simply ignore the trapdoor block and decrypt data as normal. This attack may be mitigated by always requiring the data compression first before encryption. However, a powerful state-funded adversary may know a compression algorithm that is more efficient than the publicly known ones. A slight advantage in the compression ratio would prove sufficient to insert a few extra bytes as the trapdoor. We assume the attacker's goal to enable mass surveillance over the Internet – once the ciphertext is sent over the Internet (say to a remote storage server), the attacker is able to trivially decrypt data without anyone being aware of it.

Our solution to the above problem is through the audit function. One trivial way to allow auditing the

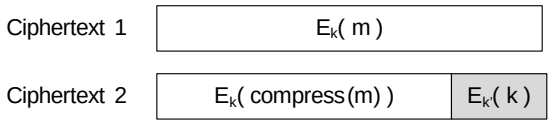


Figure 5: Ciphertext 1 is produced by an honest TPM while ciphertext 2 is by a dishonest TPM. k is an encryption key and k' is a trapdoor key (known to a state-funded security agency). Given that the encryption algorithm is semantically secure, users cannot distinguish the two ciphertexts.

encryption is to reveal the user instance's private key d_{C_i}. But the private key d_{C_i} may have been used in many DHIES sessions (each session is an invocation of the Audit function). The auditing should be limited to one specific session, but the revelation of d_{C_i} will affect the secrecy of all other sessions. This reveals too much information.

Another solution is to reveal the random factor d_η used in one DHIES session. With d_η, the two session keys can be derived and every byte in the ciphertext can be fully verified accordingly. This does not affect the secrecy of other sessions (since the random factors are all different). However, the random factor d_η is only transient in memory during the encryption process and is immediately erased once the encryption is finished.

The technique we propose has the same effect as revealing the random factor d_η, but without having to know d_η. First of all, the TPM reveals the plain ECDH shared secret: $S = d_{C_i} \cdot d_\eta \cdot G$, which can be easily computed since the TPM knows the user private key d_{C_i}. With this revealed secret S, the two session keys can be derived and every byte in the ciphertext can be verified accordingly. However, in addition to revealing S, the TPM must demonstrate that S is well-formed. In other words, if we define the tuple $\{G, C, N, S\} = \{G, d_{C_i} \cdot G, d_\eta \cdot G, d_{C_i} \cdot d_\eta \cdot G\}$, the revealed S will make the tuple form a valid DDH tuple. This is equivalent to proving either of the following two statements: 1) $\log_G C = \log_N S$; or 2) $\log_G N = \log_C S$. The choice of the statement depends on whether the prover knows either d_{C_i} or d_η. In our case, the TPM does not have d_η, but it knows d_{C_i}, hence is able to compute the ZKP based on the Chaum-Pedersen protocol.

Verifiable deletion. Second, the deletion operation should return a proof (ECDSA signature) that cryptographically binds the commitment in deleting a secret key with the outcome of that operation. If the TPM has failed to erase the key correctly, the digital signature will serve as publicly verifiable evidence to indicate the security failure. Based on the evidence and the terms in the Service Level Agreement, the user should be entitled for compensation.

Traditionally, when one (say a researcher) wants to demonstrate a security failure (or vulnerability) of

a TPM, he would need to write a technical article, post a video or do a live demo. Our protocol makes this exposure process easier and more directly: just publishing a short string of data (an ECDSA signature and the recovered key) on the internet. Anyone will be able to verify the digital signature and confirm the evidence of security failure.

6.3 User

We consider a user who is a legitimate owner of a SSE system. Depending on how the Service Level Agreement is specified, the user should be entitled to compensation (or money back) if she is able to prove that the product is faulty. However, it is possible that a user might want to profit from claiming for compensation. To prove that the system is liable for the security failure and hence claim compensation, the user needs to present an ECDSA signature together with the private key d_{C_i} (which is supposed to have been deleted[5]).

In one attack, the user can do as a data thief would do: 1) compromising the tamper resistance to gain access to the TPM's protected memory; 2) recovering the overwritten key value in the protected memory in the TPM.

However, instead of penetrating two layers of defence, the user actually just needs to compromise one layer. Once she is able to gain access to the protected memory, she can extract an existing private key d_{C_i} in memory and call the delete function to erase this key in order to obtain an ECDSA signature. Equivalently, she can extract the ECDSA private key and generate her own ECDSA signature. The evidence itself does not tell if the security failure is due to the compromise of the ECDSA signing key or due to the recovery of the allegedly deleted private key. But both keys should have been kept in the secure memory of the TPM. Hence, in any case, it should become publicly clear that the claimed "tamper resistance" has been compromised. As compared to a data thief, a user exploits a short-cut in the attack as she does not need to go further to recover the overwritten bits in the memory. This needs to be considered in the pricing strategy on determining the compensation amount in the Service Level Agreement; we will leave this a subject for future research.

7 Conclusion

While the "trust-but-verify" paradigm has been well studied and established in some fields (e.g., e-voting), it has been almost entirely neglected in the field of secure data deletion. In this paper, we initiate an investigation on how to apply the "trust-but-verify" paradigm to make the data deletion process more transparent and verifiable. We present a concrete cryptographic solution, called Secure Storage and Erasure (SSE), which enables a user to verify the correct implementation of cryptographic operations inside a TPM without having to access its internal source code. The practical feasibility of our solution is validated by a proof-of-concept implementation using a resource-contained Java card as the TPM.

Future work includes extending the "trust-but-verify" paradigm to other crypto primitives, in particular, the secure random number generator. The problem of permitting end users to audit if a random number has been generated correctly in a TPM as part of the encryption process (or a cryptographic protocol) is still largely unsolved and deserves further research.

5. The requirement of presenting the supposedly deleted private key as part of the evidence may look strong, but without it any user can arbitrarily claim fault in the product, and it will be difficult for a third party to distinguish if the product is faulty indeed or a user making a false claim.

Source Code

The source code for the Java card and host programs is publicly available at: https://github.com/SecurityResearcher/SSE). Java cards can be purchased from various sources, e.g., [41], [42].

References

[1] M. Abdalla, M. Bellare and P. Rogaway, "The Oracle Diffie-Hellman Assumptions and an Analysis of DHIES," Topics in Cryptology - CT-RSA'01, LNCS Vol. 2020, 2001.

[2] B. Adida, "Helios: Web-Based Open-Audit Voting," Proceedings of the 17th USENIX Security Symposium, pp. 335-348, 2008.

[3] R.J. Anderson, *Security Engineering : A Guide to Building Dependable Distributed Systems*, 2nd edition, New York, Wiley 2008.

[4] A. Antipa, D. Brown, A., Menezes, R. Struik, S. Vanstone, "Validation of Elliptic Curve Public Keys," Proceedings of the 6th International Workshop on Practice and Theory in Public Key Cryptography Public Key Cryptography (PKC'03), LNCS 2567, pp. 211-223, 2003.

[5] S. Bauer, N.B. Priyantha, "Secure Data Deletion for Linux File Systems," Proceedings of the 10th USENIX Security, 2001.

[6] C, Burton, C. Culnane, J.A. Heather, P.Y.A. Ryan, S. Schneider, T. Srinivasan, V. Teague, R. Wen, Z. Xia, "Using Pret a Voter in Victorian State Elections," Proceedings of the 2012 Electronic Voting Technology/Workshop on Electronic Voting (EVT/WOTE'12), 2012.

[7] D. Boneh, R. Lipton, "A Revocable Backup System," Proceedings 6th USENIX Security Conference, pp. 91-96, 1996.

[8] C. Cachin, K. Haralambiev, H.C. Hsiao, A. Sorniotti, "Policy-Based Secure Deletion," Proceedings of the 2013 ACM Conference on Computer and Communications Security (CCS'13), pp. 259-270, 2013.

[9] D. Chaum and T.P. Pedersen, "Transferred Cash Grows in Size," Proceedings of EUROCRYPT, pp. 390-407, 1993.

[10] A. Fiat, A. Shamir, "How to Prove Yourself: Practical Solution to Identication and Signature Problems," Proceedings of CRYPTO, pp. 186-189, 1987.

[11] R. Geambasu, T. Kohno, A. Levy, H.M. Levy, "Vanish: Increasing Data Privacy with Self-Destructing Data," Proceedings of the USENIX Security Symposium, 2009.

[12] S. Garfinkel, A. Shelat, "Remembrance of Data Passed: A Study of Disk Sanitization Practices," *IEEE Security & Privacy*, Vol. 1, No. 1, pp. 17-27, 2003.

[13] P. Gutmann, "Secure Deletion of Data from Magnetic and Solid-State Memory," Proceedings of the Sixth USENIX Security Symposium, pp. 22-25, 1996.

[14] P. Gutmann, "Data Remanence in Semiconductor Devices," Proceedings of the 10th conference on USENIX Security Symposium, 2001.
[15] F. Hao, M. Kreeger, B. Randell, D. Clarke, S. Shahandashti, P. Lee, "Every Vote Counts: Ensuring Integrity in Large-Scale Electronic Voting," USENIX Journal of Election Technology and Systems (JETS), Vol. 2, No. 3, 2014.
[16] N. Joukov, H. Papaxenopoulos, E. Zadok, "Secure Deletion Myths, Issues, and Solutions," Proceedings of the second ACM workshop on Storage Security and Survivability (StorageSS), pp. 61-66, 2006.
[17] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, K. Fu, "Plutus: Scalable Secure File Sharing on Untrusted Storage," Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03), pp. 29-41, 2003.
[18] R. Kissel, M. Scholl, S. Skolochenko, X. Li, "Guidelines for Media Sanitization," NIST Special Publication 800-88, 2006.
[19] T. Kohno, A. Stubblefield, A.D. Rubin, and D.S. Wallach, "Analysis of an Electronic Voting System," Proceedings of the 25th IEEE Symposium on Security and Privacy, May, 2004.
[20] J. Lee, S. Yi, J.Y. Heo, H. Park, S.Y. Shin and Y.K. Cho, "An Efficient Secure Deletion Scheme for Flash File Systems," *Journal of Information Science and Engineering*, Vol. 26, pp. 27-38, 2010.
[21] B. Lee, K. Son. D. Won, S. Kim, "Secure Data Deletion for USB Flash Memory," Journal of Information Science and Engineering, Vol. 27, pp. 933-952, 2011
[22] M. Paul, A. Saxena, "Proof Of Erasability for Ensuring Comprehensive Data Deletion in Cloud Computing," *Communications in Computer and Information Science*, Vol. 89, Part 2, pp. 340-348, 2010.
[23] D. Perito, G. Tsudik, "Secure Code Update for Embedded Devices via Proofs of Secure Erasure," Proceedings of the 15th European Conference on Research in Computer Security (ESORICS), pp. 643-662, 2010.
[24] R. Perlman, "File System Design with Assured Delete," Proceedings of the Third IEEE International Security in Storage Workshop (SISW), pp. 83-88, 2005.
[25] Z.N.J. Peterson, R. Burns, J. Herring, A. Stubblefield, A.D. Rubin, "Secure Deletion for a Versioning File System," Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies (FAST), Vol. 4, pp. 143-154, 2005.
[26] J. Reardon, S. Capkun, D. Basin, "Data Node Encrypted File System: Efficient Secure Deletion for Flash Memory," Proceedings of the 21st Usenix Symposium on Security, 2012.
[27] J. Reardon, D. Basin, S. Capkun, "SoK: Secure Data Deletion," Proceedings of the 2013 IEEE Symposium on Security and Privacy, pp. 301-315, 2013.
[28] J. Reardon, H. Ritzdorf, D. Basin and S. Capkun, "Secure Data Deletion from Persistent Media," Proceedings of the ACM Conference on Computer and Communications Security, 2013.
[29] D. Stinson, *Cryptography: Theory and Practice*, Third Edition, Chapman & Hall/CRC, 2006.
[30] H. Tews, B. Jacobs, "Performance Issues of Selective Disclosure and Blinded Issuing Protocols on Java Card," Proceedings of the 3rd IFIP WG 11.2 International Workshop on Information Security Theory and Practice (WISTP'09), 2009.
[31] D. Wheeler, "Protocols Using Keys from Faulty Data (Transcript of Discussion)," Proceedings of the 9th Security Protocols Workshop (SPW'01), LNCS No. 2467, pp. 180-187, 2002.
[32] S. Wolchok, O.S. Hofmann, N. Heninger, E.W. Felten, J.A. Halderman, C.J. Rossbach, B. Waters, and E. Witchel, "Defeating Vanish with Low-Cost Sybil Attacks Against Large DHTs," Proceedings of the 17th Network and Distributed System Security Symposium (NDSS), 2010.
[33] M. Wei, L.M. Grupp, F.E. Spada, S. Swanson, "Reliably Erasing Data From Flash-Based Solid State Drives," Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST), 2011.
[34] M. Wei, S. Swanson, "SAFE: Fast, Verifiable Sanitization for SSDs," Technical Report CS2011-0963, University of California, San Diego, 2011.
[35] C.P. Wright, M.C. Martino, E. Zadok, "NCryptfs: A Secure and Convenient Cryptographic File System," Proceedings of the 2003 USENIX Annual Technical Conference, pp. 197-210, 2003.
[36] C. Wright, D. Kleiman, S. Sundhar, "Overwriting Hard Drive Data: The Great Wiping Controversy," Proceedings of the 4th International Conference on Information Systems Security, pp. 243-257, 2008.
[37] "Digital Signature Standard (DSS)," Federal Information Processing Standards Publication, NIST FIPS Pub 186-4, July 2013.
[38] Specification of High Capacity Smart Cards, http://www.ecebs.com/high-capacity-smart-cards-a212 (accessed in Feburary, 2014)
[39] Java Card Platform Specification 2.2.2, http://www.oracle.com/technetwork/java/javacard/specs-138637.html (accessed in Feburary, 2014)
[40] The Guardian news on the Snowden documuments, September, 2013, http://www.theguardian.com/world/2013/sep/05/nsa-gchq-encryption-codes-security (accessed in Feburary, 2014)
[41] Smart Card Solution Provider, http://www.javacardsdk.com/ (accessed in Feburary, 2014)
[42] Mobile Technologies, http://www.motechno.com/ (accessed in February, 2014)
[43] IBM Tivoli Storage Manager HSM, http://www-01.ibm.com/support/docview.wss?uid=swg21319299 (accessed in May, 2014)