# Softer Smartcards
## Usable Cryptographic Tokens with Secure Execution

Franz Ferdinand Brasser[1], Sven Bugiel[1], Atanas Filyanov[2],
Ahmad-Reza Sadeghi[1,2,3], and Steffen Schulz[1,2,4]

[1] System Security Lab, Technische Universität Darmstadt
[2] System Security Lab, Ruhr-University Bochum
[3] Fraunhofer SIT, Darmstadt
[4] Information and Network System Security, Macquarie University

**Abstract.** Cryptographic smartcards provide a standardized, interoperable way for multi-factor authentication. They bridge the gap between strong asymmetric authentication and short, user-friendly passwords (PINs) and protect long-term authentication secrets against malware and phishing attacks. However, to prevent malware from capturing entered PINs such cryptographic tokens must provide secure means for user input and output. This often makes their usage inconvenient, as dedicated input key pads and displays are expensive and do not integrate with mobile applications or public Internet terminals. The lack of user acceptance is perhaps best documented by the large variety of non-standard multi-factor authentication methods used in online banking.
In this paper, we explore a novel compromise between tokens with dedicated card reader and USB or software-based solutions. We design and implement a cryptographic token using modern secure execution technology, resulting in a flexible, cost-efficient solution that is suitable for mobile use yet secure against common malware and phishing attacks.

**Keywords:** trusted computing, security tokens, secure execution

## 1 Introduction

Although available for over a decade, cryptographic security tokens with asymmetric multi-factor authentication are still not in common use for many daily authentication procedures. Despite their significant advantages, service providers and users still prefer more usable but weak password-based authentication protocols that are highly vulnerable to simple malware and phishing attacks. Moreover, the lack of scalability in password-based authentication resulted in the widespread deployment of password managers and federal ID systems, creating single points of failures that pose a serious threat to a user's online identity, personal data and business relationships.

In contrast, cryptographic tokens allow strong user authentication and secure storage of authentication secrets. The PKCS#11 cryptographic token interface specification [26] is widely accepted and, when combined with a card reader

with dedicated user input/output pad, enables the secure authentication and authorization of transactions even on untrusted systems. However, while a clear return of investment can be identified for well-defined environments like large enterprises and organizations [10], end users face significantly higher costs for the deployment and maintenance of cryptographic token solutions. Moreover, currently available solutions still fail to meet the flexibility and security required in mobile usage scenarios. For example, the secure encryption of files and emails with smartcards is easily implemented at the work place but it is inconvenient to use a dedicated keypad in addition to a laptop or smartphone, only to secure the PIN entry process against malware attacks.

USB tokens were proposed to address the mobile usage scenario by integrating card reader and smartcard into a single USB stick [20]. However, in this case the critical PIN entry and transaction confirmation to the user is done in software on the PC and thus again vulnerable to malware and interface spoofing attacks. Alternatively, one-time password systems are sometimes deployed as a compromise between usability and security. However, such systems essentially use a symmetric authentication mechanism and do not easily scale to authenticate a user towards multiple service providers. Moreover, as demonstrated in the recent security breach at RSA Security[5], the employed centralized server-side storage of master secrets and lack of scalable revocation mechanisms represents a severe security risk. Similarly, the several proposed authentication and transaction confirmation methods for online banking (e.g., [11]) are often special-purpose solutions: Approaches that use dedicated security hardware are not easily extendible for use with multiple service providers, while software-based solutions, such as using a mobile phone to transmit a transaction confirmation number out of band (mobileTAN), are again vulnerable to malware attacks. In contrast, a secure general-purpose solution using smartcards, like "Secoder/HBCI-3"[6], again requires dedicated secure hardware, making it inconvenient for mobile use.

In recent years, consumer hardware was extended with the ability to execute programs independently from previously loaded software, including the main operating system [16,1]. These so-called Secure Execution Environment (SEE) allow the secure re-initialization of the complete software state at runtime and can be used to launch a new OS or simply execute a small security-sensitive program while the main OS is suspended. Sophisticated systems have been proposed to use these capabilities for securing online transactions, however, they require substantial modification of existing authentication procedures and software stacks. In contrast, this work presents a conservative approach that uses secure execution to implement a standards-compliant cryptographic token, thus simplifying deployment and interoperability with existing software infrastructures.

[5] A security breach of the RSA Security servers resulted in a compromise of the widely deployed SecurID tokens, incurring an estimated $66 million in replacement costs: `www.theregister.co.uk/2011/07/27/rsa_security_breach/`

[6] `http://www-ti.informatik.uni-tuebingen.de/~borchert/Troja/Online-Banking.shtml#HBCI-3`

*Contribution.* In this paper, we present the design and integration of a software security token that uses secure execution technology available in commodity PCs. Our solution achieves comparable security features to hardware tokens in face of software attacks and basic protection against common hardware attacks such as device theft and offline brute-force attacks. In contrast to previously proposed secure transaction systems using trusted computing, our solution aims for interoperability and deployment, supporting the widely accepted PKCS#11 standard. Hence, our prototype is directly usable with many existing applications, such as enterprise single sign-on solutions, authentication in VPN, e-Mail and WiFi clients and password managers. By implementing secure facilities for user input/output, we can also provide secure and convenient mechanisms for deployment, backup and migration of our software token. We implement a prototype using Flicker and OpenCryptoki, providing an easily deployable solution that can be used on many standard Laptops and PCs today.

## 2 Background and Related Work

*Cryptographic Smartcards and Tokens.* Smartcards and cryptographic tokens are used in many large organizations today. The Cryptographic Token Information Format Standard PKCS#15 [25] was developed to provide interoperability between cryptographic smartcards and is today maintained as ISO 7816-15. On a higher layer, the Cryptographic Token Interface Standard PKCS#11 specifies a logical API for accessing cryptographic tokens, such as PKCS#15-compatible smartcards or Hardware Security Modules (HSMs). Alternative standards and APIs exist to access security tokens but are outside the scope of this work.

Apart from smartcards with external card readers, security tokens are also available in form of USB tokens or microSD cards[7]. The TrouSerS [15] TCG software stack also provides a PKCS#11 interface, using the TCG TPM [31] to prevent password guessing attacks against a purely software-based security token implementation. Similar to our work, these solutions offer different kinds of compromises between security and usability. However, in contrast to the aforementioned systems our solution provides secure facilities for user input/output (trusted user I/O) and is therefore resilient against common malware attacks.

*TCG Trusted Computing.* The Trusted Computing Group (TCG) [30] published a number of specifications to extend computing systems with trusted computing. Their core component is the Trusted Platform Module (TPM), a processor designed to securely record system state changes and bind critical functions, such as decryption, to pre-defined system states [31]. For this purpose, the TPM introduces Platform Configuration Registers (PCRs) to record of state change measurements in form of SHA-1 digests. By resetting the PCRs only at system reboot and otherwise always only *extending* the current PCR value with the newly recorded state change, a chain of measurements is built. This chain can be used to securely verify the system state. The TPM then supports basic

[7] E.g., Aladdin eToken Pro, Marx CrypToken or Certgate SmartCard microSD.

mechanisms to bind cryptographic keys or data to specific system states. Most significantly for our design, keys can be *sealed* to a given system state by encrypting them together with the desired target PCR values under a storage root key (SRK)[8]. Since the SRK is only known to the TPM, it can check upon *unsealing* if the current system state recorded by the PCRs matches the desired state stored together with the sealed key, and otherwise reject the request.

A major problem of this approach is dealing with the huge amount and complexity of software in today's systems, resulting in a very large chain of measurements. Moreover, the problem of runtime compromise is currently unsolved, i.e., the TPM is only informed of explicit program startups but cannot assure that already running software was not manipulated.

*Secure Execution and Flicker.* A recent extension in many hardware platforms is the SEE, a mechanism that executes code independently from previously executed software. Essentially, the CPU is reset at runtime, so that the subsequently executed program (the payload) is not affected by previously executed, potentially malicious, software.

A major advantage of this technology is that the aforementioned chain of measurements can be reset, since previously executed software does not influence the security of the current software state anymore. For this purpose, the TPM was extended with a set of PCRs that can be reset by the CPU when entering the SEE. The CPU then measures the SEE payload and stores the measurements in the previously reset PCRs. This allows the use of shorter, more manageable measurement chains that are easier to verify. Several implementations of SEEs are available, most notably Secure Virtual Machines (SVM) [1], Trusted Execution Technology (TXT) [16] and M-Shield [4].

A flexible framework for using the SEE provided by Intel TXT and AMD SVM is Flicker [22,23]. Flicker implements a framework for executing a security critical Piece of Application Logic (PAL) by temporarily suspending the main OS and diverting execution to the SEE. In this work we use and extend Flicker to implement the security-critical cryptographic operations and secure user I/O inside the SEE.

*Secure Transaction Schemes.* Several previous proposals aim to protect user credentials, typically using either a persistent security kernel that protects the credentials [14,17,6] or relying on external trusted hardware [18]. We know of only one system that uses Flicker, aiming to provide a uni-directional trusted path [13] as an instrument for transaction confirmation. It uses Flicker to take control of devices for user I/O and then asks the user to confirm a specific transaction. Using remote attestation, a remote system can then verify that the transaction was securely confirmed by a human user. All of these approaches require substantial modification of individual applications or even the addition of a hypervisor. In contrast, our solution uses the widely established PKCS#11

[8] It is also possible to use a hierarchy of keys, however, for the purpose of this work we simply assume all sealed data to be encrypted using the SRK.

interface [26] and works seamlessly with existing applications and operating systems that make use of this interface.

The On-board Credentials (ObC) framework was introduced for mobile embedded devices [21]. ObC uses device-specific secrets of M-Shield to provide an open provisioning protocol for credentials (code and data), that are securely executed/used inside the SEE. TruWalletM [8] implements a secure user authentication for web-services based on ObC. ObC is complementary to our solution and may be used to provide a similar solution for mobile embedded devices. However, in its current state ObC supports only very limited applications due to high memory constraints.

## 3 A Softer Smartcard with Secure Execution

In the following we present the security requirements, architecture and protocols of our solution. We identify the following security requirements for a secure cryptographic token:

**Secure Token Interface:** The interface used for interaction between the (trusted) token and the (untrusted) outside world must prevent the leakage or manipulation of any secret information contained in the token, such as keys or key properties.

**Secure User I/O:** The user interface of the token solution must provide a secure mechanism for the user to (1) judge the current security status of the token, (2) infer the details of a pending transaction to be authorized and (3) input the authorization secret (Personal Identification Number, PIN).

Moreover we can identify the following functional requirements:

**Interoperability:** The token should use a standards-compliant interface to provide interoperability with existing applications, such as the PKCS#11 or Cryptographic Service Provider (CSP).

**Usability:** The token should be usable in the sense that it's use should not impose a significant burden on the user. In particular, the user should not be required to buy additional hardware and the token should be usable in mobile scenarios.

**Migration and Backup:** The token should provide facilities for secure migration between different hardware platforms or implementations, and for the creation of secure data backups.

*Adversary Model.* We assume an adversary that can compromise the user's operating system and applications (malware, trojan horse attacks) or tries to lure the user into revealing authentication secrets by imitating local applications and web services (phishing, spoofing). However, the adversary has no physical control over the user's cryptographic token and is thus restricted to the application interface of the token. The goal of the adversary is to compromise long-term secret keys, to authorize transactions or decrypt confidential data of the user.

### 3.1 System Architecture

We aim to protect the user's sensitive information by implementing the functionality of a cryptographic token in a Secure Execution Environment (SEE). The SEE manages and protects the credentials even if the underlying operating system is untrusted and potentially compromised. Hence, the secure execution environment must also establish a trusted input/output path between user and token and must be able to authenticate itself towards the user. We combine the SEE with the data sealing mechanism of the TPM to let only the locally authenticated user access the secret token state, and to assure that only a trusted token implementation running in the SEE can access the sensitive token state.

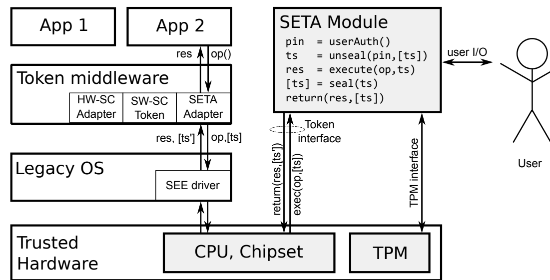



**Fig. 1.** Integration of an SEE-based cryptographic token.

The overall architecture of our SEE-based token is depicted in Figure 1. It consists of a "legacy" software stack with operating system, token middleware, and the secure execution environment on the left-hand side, which runs our Secure Execution Token Application (SETA) in parallel to the legacy operating system.

Applications must be compatible with the middleware interface to make use of cryptographic tokens. Multiple tokens can be supported by the token middleware, and in our design we add another type of token beside hardware and software tokens that calls the SEE with appropriate parameters. The call into the SEE is implemented using an SEE driver provided by the legacy OS, such as Flicker. Finally, the SETA is started as the main component of this architecture, implementing the semantics of the cryptographic token and enforcing the security-critical interfaces based on (1) isolated execution using the SEE and (2) state-specific data encryption using the TPM sealing operation (cf. Section 2). The SETA component running in the SEE supports three major interfaces, as shown in Figure 1. The user interface (*user I/O*) implements interaction with the platform peripherals for user input and output using keyboard and graphics card. The TPM interface is used for basic interaction with the TPM, specifically for the sealing, unsealing and interaction with the TPM's monotonic counters. Finally, the input and output of the SETA module, upon

entering and leaving the SEE, is used to implement the cryptographic token interface. The token interface operation *op()* that is to be executed is provided together with the encrypted token state $[ts]$. SETA must then request authorization for executing $op()$ by requesting the PIN $pin$ via user I/O, decrypt the token state $ts = \mathsf{unseal}(pin, [ts])$ using the TPM interface and compute the result $res = \mathsf{execute}(op(), ts)$ to be returned as the output of SETA.

### 3.2 Component Interaction and Protocols

Usability and flexibility are among the main concerns when using cryptographic tokens today. In the following, we present more detailed protocol flows for the main smartcard operation and caching of the user's PIN, and discuss the problems of deployment, migration and backup.

### 3.3 Deployment

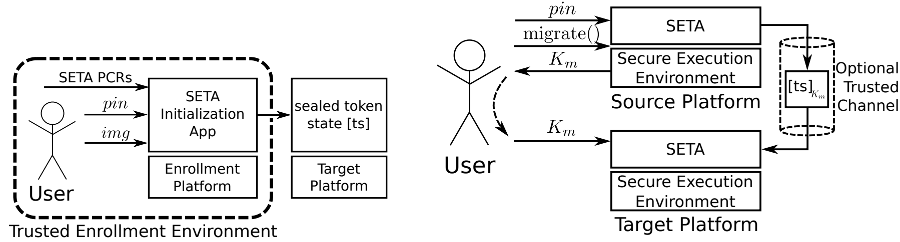
Before using the software token for the first time, the user must initialize it in a trusted enrollment environment and choose an individual picture $img$ and PIN number $pin$, as illustrated in Figure 2(a). The picture is used later on to authenticate Secure Execution Token Application (SETA) towards the user, while the PIN is used to authenticate the user towards the token. After initialization, the token state is sealed to the expected TPM PCR values of SETA, which are also supplied as input, so that only SETA can open and modify the token.

Note that if the initialization and enrollment system is different from the user's target platform, the SEE-based token can be created centrally and deployed to the user's target platform using the token migration procedure described in Section 3.4. This is particularly interesting for enterprises, which often require central enrollment and backup of authentication and encryption secrets. Hence, the overall procedure remains identical to the deployment of regular smartcards, except that the user is not required to (but could) use a physical device to transport the token state from enrollment platform to the target platform.

### 3.4 Migration and Backup

Secure backup and migration of credentials is essential for both enterprise and personal use. Note that backup and migration are very similar, since backup can be regarded as a "migration into the future", using either a dedicated backup platform as intermediate migration target or a direct migration to a target platform of yet unknown configuration.

For best usability in personal usage scenarios, we propose to realize secure migration simply based on trusted user I/O and a user passphrase $K_m$ as illustrated in Figure 3.3: To migrate the token, the user must first launch SETA and enter the correct PIN $pin$ to access the protected token state $ts$. Upon issuing the $migrate()$ command, SETA encrypts $ts$ using a symmetric randomly chosen




(a) Initialization of SEE-token state. (b) Secure migration of token state [ts] under passphrase $K_m$ and/or trusted channel.

**Fig. 2.** Procedures for the secure deployment and migration/backup of SETA.

passphrase $K_m$ and returns the encrypted token state $[ts]_{K_m}$ to the untrusted environment of the source platform. The passphrase $K_m$ is disclosed to the user via trusted I/O, enabling the owner of the token (authorized by the entered PIN) to import the encrypted token state $[ts]_{K_m}$ at the target platform.

Additionally, the migration procedure can be wrapped in a trusted channel if the target platform is known in advance and is equipped with a TPM: As proposed in [3,9], the encrypted token state $[ts]_{K_m}$ can be bound to a specific target platform's TPM and platform state before disclosing it to the source platform's untrusted environment. As a result, the protected token state can only be imported by the designated target platform, and only from within the trusted SETA environment. Token migration using trusted channels can thus effectively mitigate brute-force attacks on $K_m$ and, based on the authentication image $img$ sealed to SETA, also prevent impersonation of the importing application at the target platform. Note that if the backup system is equipped with a TPM, this extended migration protocol can also be used for secure backups.

### 3.5 Token Operation

The protocol for the regular operation of our SEE-based token is illustrated in Figure 3. As outlined in previous Section 3.1, the token middleware executes security-sensitive operations $op()$ on behalf of the applications, which in turn delegates execution of the required algorithms to SETA and returns the result $res$ at the very end of the protocol flow. For this purpose, the middleware keeps track of one or more token states $[ts]$ and their supported types of operations $op()$. Specifically, $[ts]$ consists of encrypted token state $cstate_{tk}, cstate_{pal}$ and the corresponding encryption keys $[K_{tk}], [K_{pal}]$ sealed to the SETA environment. To execute a specific operation $op()$, the middleware determines the required cryptographic token algorithm $algo$ to be executed and then asks the operating system's SEE driver to launch an instance of the SETA module, supplying the algorithm identifier $algo$, the token identifier $id$ and the respective protected token state $[ts]$ as shown in step 2.

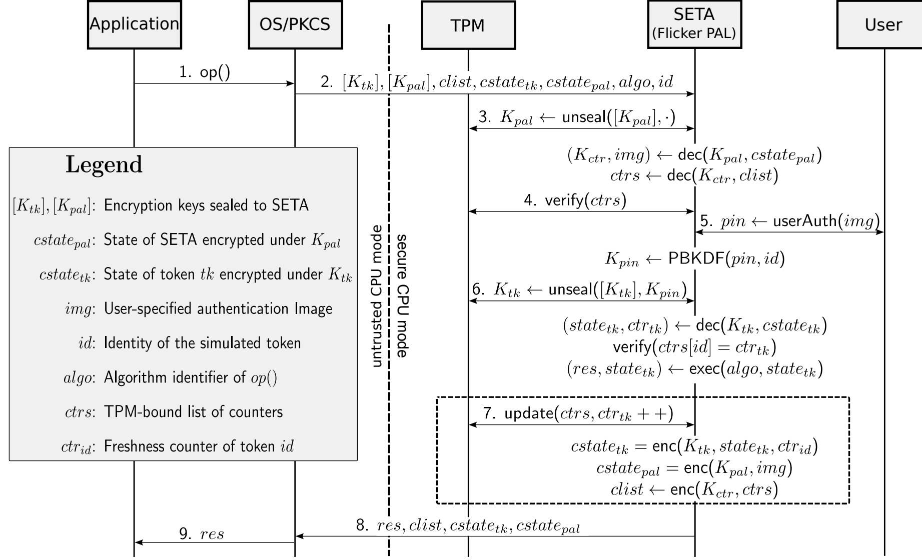



**Fig. 3.** System component interaction during normal operation.

Once launched, SETA unseals $K_{pal}$ in step 3 to decrypt the PAL meta-data state $state_{pal}$ to retrieve the master key $K_{ctr}$ of the counter list $clist$ and the secret authentication picture $img$. $K_{ctr}$ is used to decrypt $ctrs$, a list of counters that is shared by all trusted applications on the platform. In correspondence with previous work on secure virtual counters [27], the sum of the elements in $ctrs$ is compared with a predefined secure monotonic counter in the TPM in step 4. If the sum matches, $ctrs$ and thus also $cstate_{pal}$ and the individual token's counter $ctrs[id]$ and $img$ are fresh. In step 5, the picture $img$ is used to authenticate SETA to the user and retrieve the authentication PIN $pin$. The PIN and token identifier $id$ are fed into the password-based key derivation function (PBKDF2 [19]) to generate the user authentication key $K_{pin}$. This secret is in turn used in step 6 to unseal $K_{tk}$, so that $cstate_{tk}$ can be decrypted to retrieve the secret token state $state_{tk}$ and the verification counter $cntr_{tk}$. To assure the freshness of $cstate_{tk}$, SETA checks if known fresh $cntr_{id}$ is equal to $cntr_{tk}$. If successful, the actual token algorithm $algo$ can finally be executed in the context of $state_{tk}$, yielding the return value $res$. If the state $state_{tk}$ was updated during the execution of $algo$, we must increment the freshness counters $cntr_{tk}$ and $ctrs[id]$ (step 7), update the TPM hardware counter accordingly and then re-encrypt the token state $cstate_{tk}, cstate_{pal}$ (dashed box). If updated, the new states $cstate_{tk}, cstate_{pal}$ are returned together with the result $res$ in step 8. Finally, the result of the operation $op()$ can be returned to the application in step 9.

Note that even if verification of the virtual counter vector $ctrs$, which is shared together with $K_{ctr}$ among all trusted applications that require secure TPM-bound counters, is *unsuccessful*, the application can still recover the desired secret states and also determine the number of version rollbacks that have occurred as $num = cntr_{ID} - cntr_{tk}$. Hence, in case of system crashes or misbehavior of other software components SETA can inform the user and offer recovery. However, in this case the user must assure that no malicious version rollback of the sealed token state $[ts]$ took place.

While the TPM specification imposes a frequency limit on the use of the TPM's secure monotonic counters, it is unlikely that the use of SETA is affected by this limit: Most common operations carried out with the token, such as signature creation, do not modify the token state and thus do not require an update of the TPM secure counters. Moreover, common operations such as enquiring the algorithms supported by the token are not actually security sensitive and can be implemented within the token middleware's SETA adapter.

### 3.6 PIN Caching

It is often useful to cache a given user authorization for a certain time or certain number of uses. For example, in the current adoption of security tokens in health care systems it is often simply too time consuming to authorize prescriptions and other documents of minor sensitivity individually. Hence, so-called "batch signatures" were introduced that sign multiple documents at once [12]. In the following, we present an optional PIN caching mechanism for our SEE-based token that allows the authorization of multiple token operations.

Instead of requiring the PIN for each transaction, our system is able to securely cache the PIN for multiple uses. For this purpose, we seal the cached authorization secret $K_{pin}$ to the trusted system state of SETA and add a usage counter $uses$ to be maintained by the (untrusted) token middleware. We verify the value of $uses$ based on the non-invertible status of a PCR register $p$, so that the usage count can be tracked independently from the token state $cstate_{tk}, cstate_{pal}$. Another advantage of this construction is that an unexpected reset of the platform or update of the PCR $p$ does not invalidate the token state but only the cached PIN.

Figure 4 shows a modified version of the main protocol in Figure 3 to support PIN caching. When the PIN is provided for the first time and should be cached, the maximum desired number of PIN uses $uses_{max}$, the user authorization secret $K_{pin} \leftarrow \texttt{PBKDF}(pin, id)$ and a randomly chosen secret $s$ are added to the token state $cstate_{pal}$. For subsequent SETA executions with cached $K_{pin}$, the respective values are recovered from $cstate_{pal}$ as shown in Figure 4 after step 2.

In step 4, the current value $reg'$ of PCR $p$ is read in addition to the verification of $ctrs$. Due to the non-invertibility of the PCR states, this allows to verify the value of $uses$ based on the purported PCR pre-image $reg$ and the secret $s$. If this verification succeeds and $uses < uses_{max}$, the cached PIN can be used and the PCR $p$ is updated for the incremented $uses$ counter in step 5a. Otherwise, the user is asked for authorization in step 5b. After successful execution of the

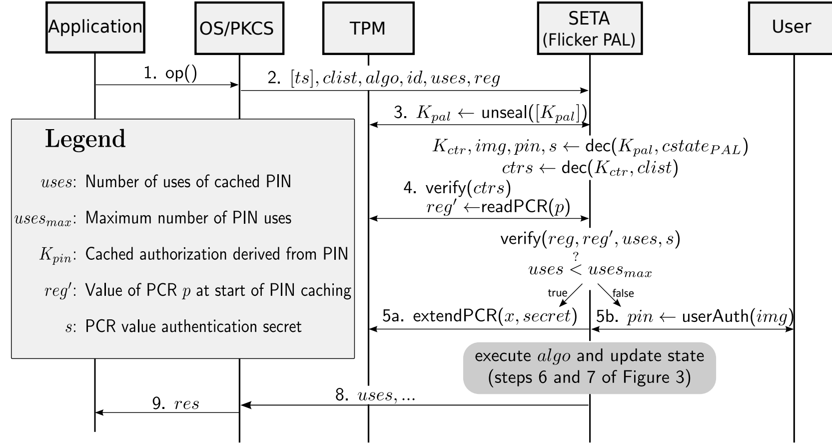



**Fig. 4.** Protocol extension for PIN-caching.

following steps 6 and 7, the result $res$ is returned together with the current usage counter $uses$ and possibly updated state $[ts]$ in step 8 and 9.

If step 6 executes successfully following step 5b, the caching state can be reset as $reg = reg', uses = 0$. Otherwise, if the PIN was (repeatedly) entered incorrectly, $K_{pin}$ should be purged from $cstate_{pal}$. As a result of this construction, the PIN caching works mostly independent from the more expensive token state updates: The values of $uses_{max}$ and $s$ can be considered relatively static and remain in $cstate_{pal}$ even if PIN caching is not used. Moreover, an unexpected modification of PCR $p$, including platform reset, only invalidates the PIN caching status, requiring only a regular user authentication to continue operation.

## 4 Implementation and Results

We implemented a proof of concept prototype of our SEE-based token based on the software token of the OpenCryptoki middleware [5]. OpenCryptoki implements the PKCS#11 interface [26], a well-known standard supported by many applications, including major web browsers, eMail clients, several VPN clients and other authentication solutions.

To build SETA, we separated the security-sensitive functions implemented by the OpenCryptoki software token into a separate software module that is then executed using the Flicker SEE driver. Additionally, we implemented basic drivers for accessing keyboard and graphics hardware from within SETA, to provide a secure user input/output interface while the untrusted OS and applications are suspended by the SEE. In this respect, we extend the work of [13] that could only provide a basic text output from within the SEE.

### 4.1 Performance Evaluation

While the processors in many hardware tokens are optimized for low cost and resistance against physical attacks, our solution benefits from the high performance of today's CPUs. The main delay for executing operations with SETA is caused by the switch into the SEE itself using Flicker, and the interaction with the (often rather slow) TPM for unsealing and counter update.

| **Existing Tokens** | **Time** |
|---|---|
| CrypToken MX2048 | 2.9s |
| eToken Pro 32k | 4.3s |
| opencryptoKi SW-Token | 0.05s |

| **Our SETA Solution** | **Time** |
|---|---|
| Switch to SEE | +1.20s |
| TPM interaction | +1.50s |
| Signing in SEE | +0.05s |
| Overall Signing in SETA | =2.75s |

**Table 1.** Speed comparison for a PKCS#11-compliant 1024 bit RSA signature.

We compared the time required for a PKCS#11 signing operation using hardware and software tokens versus using our SETA. The signature operation is perhaps the most common operation for security tokens, as it is used for authentication as well as document signing. The specific document or data length is insignificant in this case, as most applications simply hash the document themselves and only sign the hash digest, to reduce the amount of data that would otherwise have to be transferred and handled by the token.

As shown in Table 1,

Specifically, we compared an older Aladdin eToken Pro 32k and a newer MARX CrypToken MX2048 as hardware tokens against the opencryptoKi software token and our SEE-based solution on a standard Dell Optiplex 980 PC with an Intel 3.2 GHz Core i5 CPU. For the signature operation we use the PKCS#11 *C_Sign* command using RSA-1024 as the signature mechanism. As can be seen in Table 1, SETA is almost twice as fast as the older eToken Pro and still faster than the modern MX2048. As can be seen in the explicitly listed time overheads for switching to Flicker and interacting with the TPM, significant performance improvements can be expected for more complex token operations. We also expect improvements in the SEE switching and TPM interaction times once these components are relevant for daily use.

## 5 Security Considerations

The security of our overall scheme depends on the enforcement of information flow control to maintain the confidentiality and integrity of the internal token state. Specifically, our token must meet the two security requirements formulated in Section 3, (1) preventing unauthorized leakage or manipulation of the token state and (2) providing a user interface that is secure against spoofing or eavesdropping attacks by a compromised legacy operating system and that detects attempts to tamper with the SETA token implementation.

*Secure Token Interface.* Requirement (1) holds based on the assumption that the user and TPM are trusted and the PKCS#11 token interface is secure and securely implemented. The first two assumptions are standard assumptions and differ from the security of regular hardware-based cryptographic tokens mainly in that the TPM is not designed to be secure against hardware attacks.

Considering the limited security of hardware tokens against hardware attacks [2,29,24] and the prevalence of remote software attacks it is reasonable that we exclude hardware attacks in our adversary model. While some attacks on PKCS#11 implementations have been shown [7], the specification itself is considered highly mature and implementation flaws are independent from the token type.

*Secure User I/O.* Requirement (2) is met by our combination of SEE and TPM, which results in isolated execution of trusted code with full control over the local platform. The SEE suspends the legacy OS, preventing any potentially loaded malware from manipulating the execution of SETA payload and giving it full hardware access. By implementing appropriate keyboard and graphics drivers in SETA we can thus provide secure I/O on standard computing platforms. Additionally, to prevent the replacement of SETA by malicious applications, we use the TCG TPM's sealing capability to bind data to designated system states, such that a malicious SETA' $\neq$ SETA is unable to access the protected token state $[ts]$ and user authentication image $img$. Specifically, since only the pristine SETA program can access and display the secret authentication image $img$, the user can always recognize if the currently running application is the untampered SETA. A well-known open problem, in this context is that users often disregard such security indicators, allowing an adversary to spoof the user interface and intercept the secret PIN [28]. However, in our solution, an attacker that has gained knowledge of the user's PIN also requires the untampered SETA program to which the user's sealing key is bound and to which he thus has to enter the PIN (physical presence). Hence, although our solution cannot fully prevent interface spoofing against attacks against unmindful users, the attack surface is notably reduced by restricting the TPM unseal operation to pre-defined physical platforms and requiring physical presence. We suggest that enterprises monitor the migration and authorization of SETA modules for such events.

Some recent works also manage to break the rather novel SEE implementations through security bugs in BIOS and PC firmware [32,33]. The works show that PC-based SEE environments currently still require secure BIOS implementations, which can be verified using the respective TPM PCRs. Again, these vulnerabilities in the execution environment are not specific to our solution, as illustrated by recent attacks on dedicated smartcard readers [9]. However, similar to bugs in the PKCS#11 implementations such vulnerabilities are rare and usually very hard to exploit in comparison with common trojan horse or phishing attacks. Overall, SETA thus significantly improves the security of user authenti-

[9] E.g., a smartcard reader by Kobil allowed unauthorized firmware manipulation: `http://h-online.com/-1014651`

cation and transaction authorization by preventing the vast majority of malware attacks and significantly reducing the applicability of social engineering.

## 6 Conclusion and Future Work

We introduced an SEE-based PKCS#11 token that combines the flexibility of a software-based PKCS#11 token with the security of modern trusted computing technology. While our solution does achieve the same resilience against hardware attacks as some hardware cryptographic tokens, it presents a significant improvement over software-based solutions or cryptographic tokens used without dedicated keypad and display for secure PIN entry and transaction confirmation. By integrating secure user I/O with increasingly deployed SEE technology and leveraging standard cryptographic token interfaces, we can provide a secure plug-in solution that is especially attractive for today's mobile computing environments.

For future work, we aim to further reduce time delay when accessing SETA by parallelizing TPM interactions with software computations. Moreover, we aim to port our prototype to other platforms such as ObC or the Windows OS and include additional PKCS#11 functionality such as elliptic curve cryptography.

## References


1. Advanced Micro Devices (AMD): AMD64 Virtualization Codenamed "Pacifica" Technology - Secure Virtual Machine Architecture Reference Manual (2005)
2. Anderson, R.J., Kuhn, M.: Tamper resistance – a cautionary note. USENIX Workshop on Electronic Commerce. pp. 1–11. USENIX (1996)
3. Asokan, N., Ekberg, J.E., Sadeghi, A.R., Stüble, C., Wolf, M.: Enabling Fairer Digital Rights Management with Trusted Computing. Research Report HGI-TR-2007-002, Horst-Görtz-Institute for IT-Security (2007)
4. Azema, J., Fayad, G.: M-Shield$^{TM}$ mobile security technology: making wireless secure. Tech. rep., Texas Instruments (2008)
5. Bade, S., Thomas, K., Rabinovitz, D.: PKCS#11 openCryptoki for Linux (2001), IBM developerWorks
6. Balfe, S., Paterson, K.G.: e-EMV: emulating EMV for internet payments with trusted computing technologies. Workshop on Scalable Trusted Computing (STC). pp. 81–92. ACM (2008)
7. Bortolozzo, M., Centenaro, M., Focardi, R., Steel, G.: Attacking and fixing PKCS#11 security tokens. Computer and Communications Security (CCS). pp. 260–269. ACM (2010)
8. Bugiel, S., Dmitrienko, A., Kostiainen, K., Sadeghi, A.R., Winandy, M.: TruWalletM: Secure web authentication on mobile platforms. International Conference on Trusted Systems (INTRUST). LNCS, Springer (2010)
9. Catuogno, L., Dmitrienko, A., Eriksson, K., Kuhlmann, D., Ramunno, G., Sadeghi, A.R., Schulz, S., Schunter, M., Winandy, M., Zhan, J.: Trusted Virtual Domains - design, implementation and lessons learned. International Conference on Trusted Systems (INTRUST). Springer (2009)
10. Datamonitor Group: The ROI case for smart cards in the enterprise (2004)

11. Dimitriadis, C.K.: Analyzing the security of Internet banking authentication mechanisms. Information Systems Control 3 (2007)
12. Federal Office for Information Security: Batch signature with the Health Professional Card (Stapelsignatur mit dem Heilberufsausweis) (2007)
13. Filyanov, A., McCune, J.M., Sadeghi, A.R., Winandy, M.: Uni-directional trusted path: Transaction confirmation on just one device. Dependable Systems and Networks (DSN). pp. 1–12. IEEE (2011)
14. Gajek, S., Sadeghi, A.R., Stüble, C., Winandy, M.: Compartmented security for browsers - or how to thwart a phisher with trusted computing. Availability, Reliability and Security (ARES). IEEE (2007)
15. IBM: TrouSerS trusted software stack. `trousers.sourceforge.net/` (2011)
16. Intel Corp.: Intel Trusted Execution Technology MLE Developer's Guide (2009)
17. Jackson, C., Boneh, D., Mitchell, J.C.: Spyware resistant web authentication using virtual machines (2007)
18. Jammalamadaka, R.C., van der Horst, T.W., Mehrotra, S., Seamons, K.E., Venkatasubramanian, N.: Delegate: A proxy based architecture for secure website access from an untrusted machine. Annual Computer Security Applications Conference (ACSAC). IEEE (2006)
19. Kaliski, B.: PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898 (2000)
20. Kolodgy, C.: Identity management in a virtual world (2003)
21. Kostiainen, K., Ekberg, J.E., Asokan, N., Rantala, A.: On-board credentials with open provisioning. ACM Symposium on Information, Computer, and Communications Security (AsiaCCS). pp. 104–115. ACM (2009)
22. McCune, J.M., Parno, B., Perrig, A., Reiter, M.K., Seshadri, A.: Minimal TCB code execution. Research in Security and Privacy (S&P). IEEE (2007)
23. McCune, J.M., Parno, B.J., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: An execution infrastructure for TCB minimization. European Conference on Computer Systems (EuroSys). pp. 315–328. ACM (2008)
24. Nohl, K.: Reviving smart card analysis. Black Hat, Las Vegas (2011)
25. Nyström, M.: PKCS #15 - a cryptographic token information format standard. Workshop on Smartcard Technology. p. 5. USENIX (1999)
26. RSA: PKCS #11: Cryptographic token interface standard. Public-key cryptography standards (PKCS), RSA Laboratories (2009), version 2.30
27. Sarmenta, L.F.G., van Dijk, M., O'Donnell, C.W., Rhodes, J., Devadas, S.: Virtual monotonic counters and count-limited objects using a TPM without a trusted OS. Workshop on Scalable Trusted Computing (STC). pp. 27–42. ACM (2006)
28. Schechter, S.E., Dhamija, R., Ozment, A., Fischer, I.: The emperor's new security indicators – an evaluation of website authentication and the effect of role playing on usability studies. Research in Security and Privacy (S&P). IEEE (2007)
29. Tarnovsky, C.: Hacking the smartcard chip. Black Hat, DC (2010)
30. Trusted Computing Group (TCG): `www.trustedcomputinggroup.org` (2009)
31. Trusted Computing Group (TCG): TPM Main Specification, Version 1.2 (2011)
32. Wojtczuk, R., Rutkowska, J.: Attacking Intel Trusted Execution Technology. Black Hat DC (2009)
33. Wojtczuk, R., Rutkowska, J., Tereshkin, A.: Another way to circumvent Intel Trusted Execution Technology (2009)