The Trusted Execution Module: Commodity General-Purpose Trusted Computing

Victor Costan, Luis F. G. Sarmenta, Marten van Dijk, and Srinivas Devadas

MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, USA
victor@costan.us, {lfgs,marten,devadas}@mit.edu

Abstract. This paper introduces the Trusted Execution Module (TEM); a high-level specification for a commodity chip that can execute user-supplied procedures in a trusted environment. The TEM is capable of securely executing partially-encrypted procedures/closures expressing arbitrary computation. These closures can be generated by any (potentially untrusted) party who knows the TEM's public encryption key. Compared to a conventional smartcard, which is typically used by pre-programming a limited set of domain- or application- specific commands onto the smartcard, and compared to the Trusted Platform Module (TPM), which is limited to a fixed set of cryptographic functions that cannot be combined to provide general-purpose trusted computing, the TEM is significantly more flexible. Yet we present a working implementation using existing inexpensive Javacard smartcards that does not require any export-restricted technology. The TEM's design enables a new style of programming, which in turn enables new applications. We show that the TEM's guarantees of secure execution enable exciting applications that include, but are not limited to, mobile agents, peer-to-peer multiplayer online games, and anonymous offline payments.

1 Introduction

The Trusted Execution Module (TEM) is a Trusted Computing Base (TCB) designed for the low-resource environments of inexpensive commercially-available secure chips. The TEM can securely execute small computations expressed as partially-encrypted compiled closures. The TEM guarantees the confidentiality and integrity of both the computation process, and the information it consumes and produces. The TEM's guarantees hold even if the compiled closure author and the TEM owner do not trust each other. That is, the TEM will protect the closure's integrity and confidentiality against attacks by its owner, and will protect itself against attacks by malicious closure authors. The TEM does not trust the authors of the programs it runs. A malicious closure cannot negatively impact the TEM it runs on, and it cannot maliciously interfere with the result of closures written by other authors. This implies that there is no need for a program certification system.

The TEM executes compiled closures in sequential order, in a tamper-resistant environment. The execution environment offered by the TEM consists of a virtual machine interpreter with a stack-based instruction set, and a single flat

memory space that contains executable instructions and temporary variables. The environment is augmented with a cryptographic engine providing standard primitives and secure key storage, and with a persistent store designed to guarantee the integrity and confidentiality of the variables whose values must persist across closure executions. The persistent store is designed to use external untrusted memory, so its capacity is not limited by the small amounts of trusted memory available on inexpensive secure hardware.

The TEM's design focuses on offering elegance and simplicity to the software developer (the closure author). The instruction set is small and consistent, the memory model is easy to understand, and the persistent store has the minimal interface of an associative memory. The breakthrough provided by the TEM is the capability to execute user-provided procedures in a trusted environment, for the low price of a commodity chip.

We have implemented the TEM on an inexpensive, commercially-available JavaCard. The TEM's prototype implementation shows that the design is practical and economical. The research code implements a full stack of TEM software: firmware for the smart card, a Ruby extension for accessing PC/SC smart card readers, a TEM driver, and demo software that uses the driver. The prototype implementation leverages the advanced features of the Ruby language to provide a state of the art assembler which makes writing compiled closures for the TEM very convenient.

The TEM enables new applications by combining the flexibility of a virtual machine guaranteeing trusted execution with the pervasiveness of inexpensive secure chips. For example, the TEM can be used to provide provably secure, simple, and general solutions to mobile agents, peer-to-peer multiplayer online games, and anonymous offline payments.

The outline of this paper is as follows. We start with related work in section 2 which compares TEM with existing approaches towards trusted computing. The main concepts used in TEM are introduced in section 3. Section 4 details the architecture of TEM and section 5 discusses its implementation together with timing results from experiments. Section 6 explains how to program closures that can securely migrate between TEMs. This mechanism can be used in applications such as secure mobile agents and secure peer-to-peer multiplayer online games. For a more detailed discussion on the TEM and downloadable source code, the reader is refered to the first author's Master's thesis [**?**].

2 Landscape

Large TCBs: Trusted Modules and Processors. Early solutions to secure platforms were supplied, most notably by IBM, as tamper-resistant assemblies that can operate either autonomously or as coprocessors for high-end systems. A recent representative of these systems is the IBM 4764 co-processor [**?**], which is only available for IBM servers under custom contracts.

Secure processors represent a different approach to trusted platforms. A secure processor costs less than a trusted platform because the secure envelope

only contains the logic found inside CPUs. The AEGIS [**?**] design provides a cost-effective method for implementing a secure processor (that embeds an integrity checking interface to untrusted memory). Compared to the smart card chips, secure processors deliver higher performance and readily support large applications, but are much more expensive.

Embedded TCBs: Smart cards. Smart cards [**?**] are secure platforms embedded in thumb-sized chips. For handling convenience, the chips are usually embedded in plastic sheets that have the same dimensions as credit cards. The same chip are used as Subscriber Identity Modules (SIMs) in GSM cellphones. Smart cards have become pervasive, by offering a secure platform at a low cost.

The ISO 7816 standard regulates the low-level aspects of smart cards [**?**]. Platforms such as MultOS [**?**] and JavaCard [**?**] provide a common infrastructure for speeding up application development, and allow multiple applications from different vendors to coexist securely. However, both of these platforms were intended for monolithic applications that are contained and executed completely on the smartcard. Applications on a card can only be installed or updated if they are certified by a trusted entity, which is either the smart-card emitter, or the platform vendor.

The TEM design makes large applications easier to develop, because it loads one closure at a time into the secure environment, as opposed to smart cards that load all the applications at once. The TEM makes update deployment easier, because open classes are naturally implemented with closures. A TEM is also more flexible, as it allows its owner to execute any vendor's applications.

Fixed-Function TCBs: TPMs. The TPM is a fixed-function unit, which means it defines a limited set of entities (such as shielded locations holding cryptographic keys or hashes), as well as a closed set of operations that can be performed with primitives (such as using a key to unwrap another key or to sign a piece of data). The operations are not sufficient for performing arbitrary computation on the TPM. Instead, the TPM was envisioned to attest that the host it is attached to, a general-purpose computer, runs a trusted software stack. The strength of the bond between the TEM and its host determines the security of the entire system, since an attacker that compromises the bond can spoof the attestation system [**?**], [**?**]. The TCB on the TPM's host computer includes a secure boot loader, an operating system, and drivers. Such a TCB does not exist, because it is impractical to analyze and certify the large codebases of modern operating systems, together with their frequent updates. In practice, TPM applications (e.g., [**?**]) do not assume a TCB on the host. Thus, they are not capable of performing trusted arbitrary computation.

The TEM does not require trusted software on its host to perform arbitrary computation, and does not need to be securely bound to its host. This means that a TEM can cost less than a TPM, and that existing computers can be enhanced with TEMs via standard extension buses, like the USB.

3 Concepts

3.1 Trust Chain

The method used to attest that a platform offers the security guarantees of a TEM is a simplification of the TPM's chain of trust for platform attestation [**?**]. The root of trust is the hardware manufacturer (such as Infineon or Atmel), which acts as a Certificate Authority in a public key infrastructure as defined in [**?**]. Each TEM has a unique asymmetric key. The private part of the key (PrivEK - Private Endorsement Key) is generated inside the TEM at manufacturing time, and never leaves the TEM. The public part (PubEK) is included in an Endorsement Certificate (ECert) issued by the TEM's manufacturer, attesting that PubEK corresponds to a TEM endorsement key. Since a TEM has exactly one PubEK, the PubEK can be used to identify and track the TEM, and thus its owner. This may be unacceptable in some circumstances, as it leaks information about the users' identity. By adapting the ideas of [**?**] to augment the chain of trust, the TEM can be made untraceable (see [**?**]).

3.2 Closures

The closure is the execution primitive of the TEM. This allows the use of virtually any programming paradigm with the TEM. Compiled closures (described below) can be implemented in an execution engine that is just a bit more complex than an engine designed for procedural execution. This translates into a small[1] execution engine that is suitable for implementation on embedded platforms.

A *closure* (originally defined in [**?**]) is a fragment of executable code, together with the bindings of the variables that were in scope when the closure was defined. Closures are extremely powerful, and can be used to implement most primitive structures in modern programming languages ([**?**] and [**?**]). To provide immediate assurance to readers, listing 1.1 uses the approach employed by ECMAscript [**?**] (also known as JavaScript) to implement Object-Oriented Programming [**?**] objects featuring encapsulation.

```
1 def bank_account(account_number)
2   balance = 0
3   account = Hash.new
4   account[:balance] = lambda { balance }
5   account[:number] = lambda { account_number }
6   account[:deposit] = lambda { |amount| balance += amount }
7   account[:withdraw] = lambda { |amount| balance -= amount }
8 end
```

Listing 1.1. Bank Account object implemented with closures (functional Ruby)

Figure 1 shows the stucture of the closures created by executing listing 1.1. Each object method becomes a closure that contains a sequence of executable code, and a binding table that associates variable names with pointers to memory cells storing the variables' values.

[1] as compared to the Java Virtual Machine [**?**]

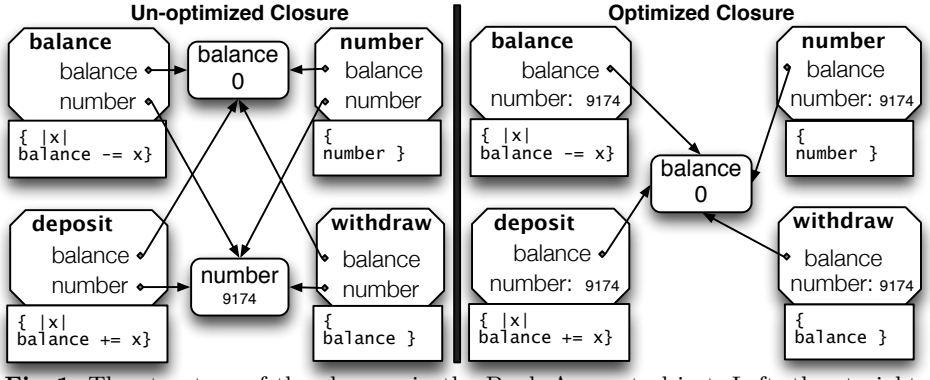

Fig. 1. The structure of the closures in the Bank Account object. Left: the straightforward result. Right: result after de-facto immutable variable optimization.

Compiled Closures. The TEM design targets embedded chips, where persistent variables are expensive[2]. The following optimization, inspired by [**?**], helps to reduce the amount of shared memory cells used by a closure. Some of the variable bindings are de-facto immutable (constant). That is, their values will never be modified throughout the lifetime of the closure. Thus, the constant value can be stored directly in each closure's binding table. [**?**] uses this mechanism to decide whether frames will be allocated on the stack or in the heap.

For example, it is easy to see that `number` in listing 1 is de-facto immutable, and `balance` is not. So the closures' binding tables can be optimized to use one shared memory location instead of two, as illustrated in Fig. 1.

The result in the right of Fig. 1 is further amenable to well-known optimizations, such as removing unreferenced variables from the binding table. For instance, the variable `number` is not used at all in the closures `balance`, `deposit`, and `withdraw`, so it can be removed from their binding tables.

A **compiled closure** is a closure that has been fully optimized for the computer that is intended to execute it. A compiled closure consists of the following:

- the computation to be performed, expressed as executable instructions that can be interpreted by the target computer,
- a binding table that contains all the non-local variables,
- values for the non-local variables that are de-facto immutable, and
- pointers to the shared memory locations holding mutable non-local variables.

SEClosures and SECpacks. A **SEClosure** (Security-Enhanced Closure) is a closure where all the information has been classified as one of the following.

- **shared**: information whose integrity must be guaranteed by the TEM. For example, executable code requires integrity in order to detect whether it has been replaced by malicious code that would output private information.

[2] Variables' values may change, therefore they would have to be stored in EEPROM. EEPROM is the slowest and most expensive type of on-chip memory.

– **private**: information that requires confidentiality guarantees from the TEM, such as a signing key or a secret algorithm. The integrity of private information must be protected by the TEM as well. Otherwise, the TEM owner could learn private information by executing closures where the private information was modified, and observing the differences in results.
– **open**: this information is not covered by any guarantee. This has to be information that the TEM's owner supplies, as the owner is the only one who does not need any proof of integrity or confidentiality.

For simplicity, it seems appealing to remove the *shared* class of information, and specify that all the information not provided by the closure's owner is private. However, SEClosures need to have *shared* information to allow the TEM owner to assert certain facts about the computation expressed in the closure.

SEClosures are compiled into a format that the TEM can easily process. A compiled SEClosure, called a **SECpack**, has all the shared, private, and open data grouped together.

Bound SECpacks. Before a SECpack is given to the TEM's owner, its content is partially encrypted with the TEM's PubEK, to protect the private and shared information. The encryption result is a **bound SECpack**, containing the same information as the original SECpack, but in a form that enforces the confidentiality and integrity of the enclosed information. A *bound* SECpack can only be executed by a platform possessing the PrivEK corresponding to the PubEK used for encryption.

Binding (explained below and summarized in Fig. 2) assumes that the TEM's PubEK is known and was validated against the TEM's ECert. The scheme is inspired by the TPM [**?**].

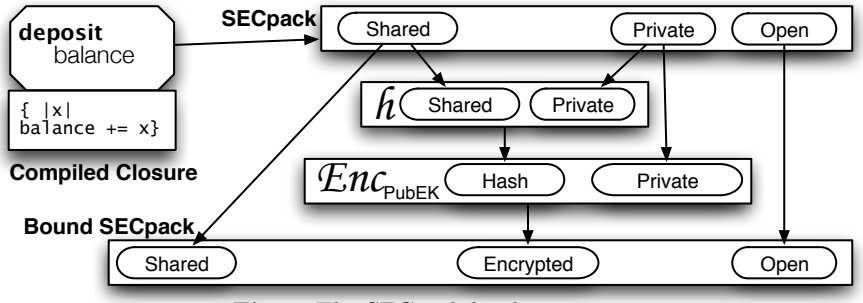

Fig. 2. The SECpack binding process

Binding is performed using the following steps:

1. Let $\mathcal{P}$ be the private data, $\mathcal{S}$ be the shared data, and $\mathcal{O}$ be the open data.
2. Use a cryptographic hashing function h (e.g., SHA1 [**?**]) to compute a digest $\mathcal{H} = h(\mathcal{P}||\mathcal{S})$ of the concatenation of the private and the shared information.
3. Use the TEM's Public Endorsement Key to encrypt the private information together with the digest; $\mathcal{E} = Enc_{\text{PubEK}}\Big(\mathcal{P}||\mathcal{H}\Big)$.
4. The bound SECpack is the concatenation $(\mathcal{S}||\mathcal{E}||\mathcal{O})$ of the encryption result $\mathcal{E}$, the shared information $\mathcal{S}$, and the open information $\mathcal{O}$.

3.3 Persistent Store

The values of mutable variables are stored inside a secured global persistent store (Fig. 3), indexed by addresses that are at least as large as cryptographic hashes. An address identifies a value, and at the same time shows proof of authorization to access that value. The information in the persistent store is stored in a way that prevents any accesses that would bypass the associative memory abstraction. The values stored in the persistent store have the same size as the addresses, to avoid memory waste. The associations can be stored in untrusted memory on the TEM's host, using the architecture in section 4.3.

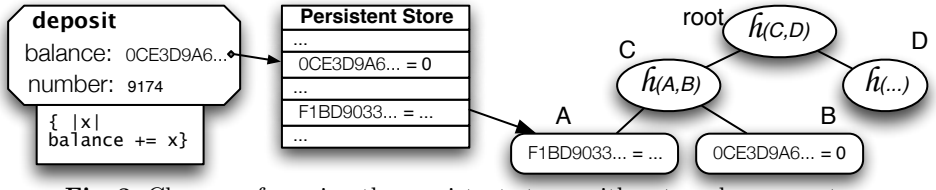

Fig. 3. Closure referencing the persistent store, with external memory tree

Assigning Persistent Store Addresses. A mutable variable uses the same persistent store address on all the TEMs it exists on. Addresses are assigned during closure compilation, using a random number generator. This simplifies deployment, because updates to a system can be easily implemented as new closures that use existing variables (in OOP, this is called open classes). Closures can also be easily migrated by re-binding the SECpacks to a different PubEK.

The probability of two different variables colliding on a TEM is at least as low as for Universally Unique IDs [**?**]. The probability of an attacker compromising a variable by guessing its address is at least as low as the probability that the attacker will be able to forge the signature on an Endorsement Certificate and break the chain of trust directly.

4 Architecture

The main components of the TEM architecture (block diagram in Fig. 4) are the execution engine, the cryptographic engine, and the persistent store.

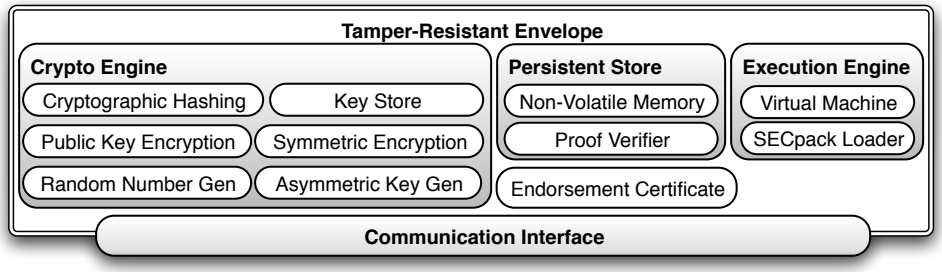

Fig. 4. High-Level TEM Block Diagram

The TEM also contains a communication interface that is mainly a transceiver for the communication channel between the TEM and its owner. If the communication between the TEM and its owner occurs via an untrusted channel, the

interface establishes a secure session using a mechanism similar to SSL [**?**], and relying on the TEM's ECert and PubEK to bootstrap the session.

4.1 Key Store

The key store (illustrated in Fig. 5) implements secure key storage, accessible as an array of key slots. A slot can contain a symmetric key, or the public or private part of an asymmetric key. This was inspired from the `javacardx.crypto` API [**?**], and chosen because each slot has well-defined `encrypt` and `decrypt` operations.

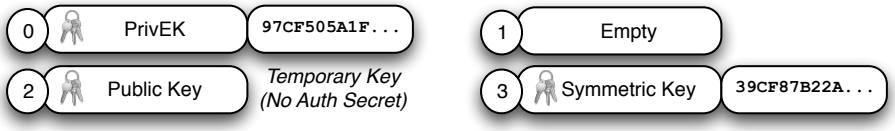

Fig. 5. Snapshot of a TEM's Key Store

A key created during SEClosure execution is temporary by default, and is released when the closure ends executing. A key becomes persistent when it is given an authorization secret. Secrets have the same size as a cryptographic hash. A closure gains access to a persistent key by presenting the associated secret.

PrivEK occupies the first slot in a TEM's key store, and is associated with an authorization value known only by the platform manufacturer. The manufacturer can build "privileged" SEClosures, which use a TEM's PrivEK to offer functionality on top of the basic model, such as TPM emulation.

The TEM owner can remove any key from the store, via driver commands. This prevents the possibility of denial of service by filling up the key store.

4.2 Virtual Machine Environment

The computation inside a SECpack is expressed as microinstructions for a **stack-based virtual machine** (VM). The entire VM interpreter state consists of an instruction pointer (IP) and a stack pointer (SP). Instructions are encoded as a 1-byte operation code (opcode), optionally followed by immediate data. The stack consists of machine word-sized entries.

Using a virtual machine makes the executable code in a SECpack universal. The alternative of having SECpack executable code target specific hardware would introduce complexity (multiple target platforms) for the closure compiler, and would greatly complicate migrating SECpacks among TEMs.

The performance degradation introduced by the virtual machine has no significant impact on most TEM applications. Assuming a reasonable VM implementation, the time cost of one asymmetric key encryption operation dwarfs the cost of interpreting thousands of VM instructions. Asymmetric key decryption is invoked for every bound SECpack, because a part of the SECpack must be decrypted with the TEM's Private Endorsement Key.

Stack-based instruction sets are easy to interpret and generate from ASTs (Abstract Syntax Trees), and are used in recent VMs for both medium- and high-level languages (e.g., the Java VM [**?**], the Ruby 1.9 VM [**?**]).

The TEM's execution environment has a **single, flat, RAM-backed memory space** that contains executable instructions, values of local and de-facto immutable non-local variables, and the virtual machine's stack. Closures have direct access to the memory space, for maximum flexibility and performance (e.g., self-modifying code).

Closures return data by placing it in the **output buffer**, an append-only memory zone. If a closure's execution is aborted, the output buffer is discarded and nothing is returned. This mechanism simplifies building closures that don't leak information.

The VM contains special-purpose instructions accessing the functions of the cryptographic engine. A closure is automatically authorized to use all the encryption keys that it creates. The SEClosure can also use keys that are already loaded in the cryptographic engine, once it demonstrates knowledge of their authorization secret. The execution engine enforces these restrictions, and aborts closures that attempt to use a key before gaining authorization.

The contents of mutable non-local variables are stored in the persistent store (sections 3.3 and 4.3) between executions. Addresses are stored in memory space, and values are transferred between the memory space and the persistent store. If a closure is aborted, all its persistent store updates are rolled back.

The execution engine design completely discards any possibility of concurrent execution. This makes security guarantees easier to prove. A multi-core TEM can be modeled as multiple execution engines that share a persistent store where each SECpack execution is treated as a transaction.

SECpack Contents and Loading. A SECpack consists of a snapshot of the initial state of the virtual machine's memory space, together with a header containing a magic value, the initial IP and SP values, and the sizes of $\mathcal{S}$ and $\mathcal{P}$ (needed to decrypt a bound SECpack). Unbound SECpacks have an empty $\mathcal{P}$. This makes virtual machine setup trivial, given an unbound SECpack.

A bound SECpack requires that the loader decrypts the private information $\mathcal{P}$ and verifies the integrity of the shared information $\mathcal{S}$. SECpacks that fail the integrity check $\mathcal{H} = h(\mathcal{S}||\mathcal{P})$ are rejected.

The loader can use any key in the store (section 4.1) to decrypt a bound SECpack. This allows for speed optimizations and extensions to the TEM's chain of trust. For example, an often-used set of SECpacks can be bound using a symmetric key instead of PubEK, if the key is somehow transmitted securely to the TEM. This can dramatically reduce execution time by avoiding asymmetric-key operations when SECpacks are loaded.

Considerations in the Design of the Instruction Set. The standard instructions are heavily inspired by the Java Virtual Machine [**?**]. The TEM-specific instructions (cryptography and persistent store) have been developed while aiming to adhere to the same principles.

The instruction set tries to strike a balance between enabling small SECpacks and keeping the VM interpreter simple. For example, most instructions operating on memory blocks have two variants. The fixed block variant (instructions ending in `fxb`) receives the information about the blocks (address, and optionally length) as immediate data. The variable block variants (instructions ending in `vb`) pop the block information off the stack. The fixed block variant takes up less space in a secpack, while the variable block form provides maximum efficiency when working with variable-length memory structures.

Exceptions were made for instructions that would not occur often in a SECpack (e.g., `rnd`), so the space savings do not warrant the extra complexity in the interpreter. The instruction set aims for consistency with respect to mnemonics and order of parameters.

4.3 Persistent Store Architecture

Backing the Persistent Store by Untrusted Memory. Like AEGIS [**?**], the persistent store relies on building a Merkle tree [**?**] (Fig. 3) where the leaves store the actual associations, and internal nodes store a cryptographic hash of their children. The TEM stores the tree's root in NVRAM but all the other nodes can be stored in untrusted memory. A symmetric encryption key[3] inside the TEM[4] is used to encrypt the two parts (address and value) of each association individually, so neither part is stored "in the clear" in untrusted memory, and the TEM can later ask for an association by its encrypted address. The internal nodes hash the external representation of their children.

The TEM's host maintains the tree structure. When a closure `read`s from the persistent store, the TEM communicates the encrypted address to the host. The host responds with the encrypted value at the address, and a correctness proof consisting of all the nodes on the path to the leaf. `write`s work similarly, but the correctness proof also describes the updates to be performed on the tree. Rolling back an aborted closure is done by rolling back the tree root on the TEM.

The external memory tree exhibits **amnesia** if TEM's host, who is managing the tree, can lie by telling the TEM that no association exists for an address, when in fact an association does exist. Undetected amnesia during a `read` can lead a SEClosure to assume that a variable has never been assigned on the TEM, and that it has its default value. During a `write`, undetected amnesia can lead the persistent store to create a duplicate leaf for an association, instead of updating the existing one. The TEM's host can then return the (old) duplicate leaf for `read`s, effectively making the persistent store "forget" updates to a variable.

Amnesia is **detectable** if the lies of the TEM's host can be recognized and stopped from propagating into incorrect SEClosure execution. For example, amnesia is detectable if, for every persistent store operation, it is known in advance if the persistent store already contains an association for the requested address.

[3] If the law does not allow symmetric encryption, an asymmetric key can be used instead, at a large performance cost.

[4] This key never leaves the TEM, and can be generated cheaply by a PUF [**?**].

If amnesia is not detectable, the correctness proofs given by the TEM's host must ensure that amnesia cannot occur. In particular, the host must be able to prove efficiently that the tree does not contain any occurence of an address. This requires external trees that are sub-optimal for a sparse address space (e.g., trees with a fixed branching factor), or complex to implement (e.g., binary search trees).

In comparison, if amnesia can be recognized, the TEM's host does not need to supply any proof when it states that an address does not exist in a tree. The requirement that a proof is given when addresses are found is sufficient to ensure correct operation under detectable amnesia. So the external tree structure can be chosen in a way that maximizes the efficiency and simplicty of the proof validating process that is performed by the TEM.

The Lifetime of Persistent Store Variables. In order to avoid a complex tree structure that may require non-trivial resources, SEClosures must manage the lifetime of their persistent store variables. The scheme must detect amnesia in the external tree, and also be able to distinguish between the case when a variable has never been used on a TEM, and the case when the variable has been assigned a value, but the corresponding association has been `removed` from the persistent store. The possibility of confusing the two cases can be exploited by replay attacks. The method below achieves these requirements.

Let an **object**[5] be a group of SEClosures that use the same mutable non-local variables. For convenience, an object's **fields** shall be all the mutable non-local variables used by the SEClosures in the object. For the example in section 3.2, an individual bank account is an object consisting of the SEClosures `withdraw`, `deposit`, `balance`, and `number`. The object has one field, the variable `balance`.

To reduce complexity, all the fields of an object are managed as a whole, following the same principles as constructors and destructors (also named finalizers). Namely, an object is **constructed** on a TEM by creating persistent store associations for all its fields. An object is **destroyed** by `remove`-ing the persistent store associations. An object's SEClosures abort execution if any of the fields they reference do not have a value in the persistent store, so the SEClosures are only usable between the object's construction and destruction.

The process uses a single monotonic counter, $\mathcal{M}_C$, that is a mutable variable for the privileged SEClosures involved in object construction. The TEM owner is given a SEClosure that signs $\mathcal{M}_C$ with PrivEK, so the read can be verified by anyone who has the TEM's ECert.

The object's owner receives a $\mathcal{M}_C$ read result, and constructs the **constructor data** for an object, which consists of the $\mathcal{M}_C$ value and the object's **constructor table**, a list of persistent store addresses and initial values for the object's fields. The constructor data is encrypted with the TEM's PubEK then given to the TEM's owner, together with the bound SECpacks for the object.

The owner runs the **constructor**, a privileged SEClosure that decrypts the construction data, and creates the associations in the **constructor table** if the

[5] Object-Oriented Programming is used to simplify the presentation. However, the mechanism presented here can be easily adapted to other programing paradigms.

$\mathcal{M}_C$ value matches. If the object is constructed, $\mathcal{M}_C$ is incremented. This way, exactly one object is constructed for a certain value of $\mathcal{M}_C$, and no constructor table is executed twice. This ensures that an object can be constructed at most once on a TEM.

The owner removes the object's fields from the persistent store by using the object's constructor data with a privileged SEClosure called the **desctructor**.

5 Implementation

The **TEM firmware** was implemented on JavaCard [**?**] smart cards, because of their widespread availability. The firmware's overall design closely reflects the TEM architecture illustrated in Fig. 4. Each component is materialized in one class with `static` fields and methods. The implementation manages its own memory buffers, to make optimum usage of the RAM and EEPROM memory on the chip, and to overcome the 255-byte limitation in APDU size on the prototype cards. The VM interpreter is implemented as one single 420-line method, and takes heavy advantage of the consistency in the instruction set.

The **TEM driver and SDK** were implemented in Ruby. Domain-Specific Languages (DSLs) [**?**] [**?**] were built for the TEM's data types and VM instructions. The SECpack assembler is also a DSL supporting comments, multiple instructions per line, named parameters, named labels, human-readable immediates, embedded Ruby code, as illustrated in listing 1.2.

```
1  def gen_bank_account(number, initial_balance, bank_key)
2    balance_address = (0...ps_addr_length).map { |i| rand(256) }
3    balance_length = 8
4
5    balance_sec = assemble do |s|
6      s.psrdfxb :addr => :balance_addr, :to => :balance
7      s.ldwc :bank_key; s.rdk
8      # will output balance + signature(nonce + balance)
9      s.dup; s.ldkl; s.ldwc balance_length; s.add; s.outnew
10     s.outfxb :from => :balance, :length => balance_length
11     s.ksfxb :from => :nonce, :length => ps_value_length +
12             balance_length, :to => 0xffff
13     s.halt
14
15     s.label :bank_key; s.immed :ubyte, bank_key.to_tem_key
16     s.label :balance_addr; s.immed :ubyte, balance_address
17     s.label :nonce; s.filler :ubyte, ps_value_length
18     s.label :balance_value; s.filler :ubyte, ps_value_length
19     s.stack; s.extra 8
20   end
21 end  # ( code for other closures and for the constructor)
```

Listing 1.2. Assembly code for `balance` in the bank account example

The prototype TEM implementation contains supplemental features to help debugging SEClosures, such as dumping the TEM state when SEClosure is

aborted. The SECpack assembler provides line-level debug information that is combined with the TEM state to pinpoint the exact line in the SECpack source code that is causing the execution to be aborted.

The TEM driver contains a full [6] suite of unit tests, covering both driver and firmware code. The unit tests automate validating both the driver and the firmware, as well as assessing the suitability of a JavaCard model as a TEM.

5.1 Performance Considerations

The table below shows the results of various tests that were run on two JavaCard models. The times are expressed in seconds. Each time is the average over multiple consecutive repetitions of an operation. The number of repetitions was chosen such that the results of running an experiment (all the consecutive repetitions of the operation) 3 times were within 1% of the mean.

Operations	**NXP 41/72k**	**Philips 21 18k**
Process APDU	0.0061 s	0.029 s
Create and release 512-byte buffer	0.084 s	0.98 s
Decrypt with PrivEK	0.76 s	1.60 s
Execute 1-op SECpack	0.16 s	0.50 s
Execute 1020-ops SECpack	0.82 s	1.99 s
Execute 1-op bound SECpack	0.89 s	1.92 s
Execute 1020-ops bound SECpack	1.53 s	3.07 s

The prototype's performance is not yet acceptable for interactive systems. This is mainly because the VM interpreter is layered on top of the JavaCard virtual machine, which introduces the overhead of interpreted versus native execution (likely on the order of 20X). In practice, the performance hit is not as dramatic (a full 20X), because a significant part of a SEClosure's execution time is spent on RSA operations on the TPM-grade 2048-bit keys. Cryptographic operations are done in hardware and do not incur the JavaCard overhead.

6 Example Application: Migratable Tokens

This section discusses the use of the TEM's architecture to implement migratable tokens (as defined in [**?**]) in an anonymous offline payments system. The technique can be reused in other applications, such as authorization tokens in personal DRM, or active items (e.g., spells) in peer-to-peer online games.

A bank emits a sum of e-money as a TEM object (section 4.3). The object contains the bank's private key, and the sum of money it represets. An object's balance is verified by executing the `balance` SEClosure that produces a signature for the current sum of money, together with a caller-supplied nonce.

Making a payment is achieved by creating a money object on the receiver's TEM. This is achieved in two steps:

[6] `rcov` indicates a line coverage of 95% or above on each Ruby source file.

– the SECpacks are migrated by a privileged SEClosure that verifies the receiver TEM's ECert, then binds each SECpack to the receiver TEM using its PubEK.
– the balance is migrated by a `transfer` SEClosure that also verifies the receiver TEM's ECert, then subtracts the required amount from the balance on the payer's TEM, and produces constructor data that will create the required balance on the destination TEM. If the payer's TEM balance reaches 0, the variables of the money object are removed.

Merchants use the same migration procedure to transfer the money to the bank. The bank then uses a `cancel` SEClosure that destroys the money object, and thus cancels the e-money it has emitted.

This scheme assumes that e-money never reaches the same TEM twice, and does not support merging e-money objects on a TEM. These restrictions can be removed by using a more complex scheme that is outside the scope of this paper.

7 Conclusion and Future Work

This paper introduces a novel approach to trusted execution. The TEM enables a new style of programming, by being capable of executing untrusted closures in a secure environment.

Understanding the applications of the TEM's execution model is a fruitful avenue for exploration. Flexible trusted execution at commodity prices should bring new life to difficult problems, such as secure mobile agents.

Improving the TEM's performance is a promising prospect, as the benchmarks in section 5.1 are close to the results needed for interactive systems. The prototype JavaCard implementation can be optimized by using code generation or bytecode generation techniques. Assuming an adequate SDK, the TEM can be implemented directly on a secure chip, and will show the needed improvements.

The TEM's design can also be modified to fit specific goals or target platforms. There are many variations on the persistent store design. The execution environment allows extensions such as parallel processing on the crypto engine.

The TEM can be easily extended to offer certified closure execution. SECpack binding assures the closure's author that a closure was executed in a secure environment. Certified execution can prove to anyone that a result was produced in the secure environment of a TEM. We are currently investigating the applications of extending the TEM with certified execution.