μTSS – A Simplified Trusted Software Stack
Abstract:

The TCG Software Stack (TSS) specifies the software layer for application developers to use functions provided by a Trusted Platform Module (TPM). However, the current TSS interface is highly complex, which makes its usage very difficult and error-prone, and the high complexity makes it unsuitable for embedded devices or security kernels. We present a simplified TSS design and implementation ($\mu$TSS) providing a lightweight and intuitive programming interface for developers based on the TPM main specification. The major principles of the $\mu$TSS design are a reduced complexity, obtaining type safety, object encapsulation, and a simple error handling. These principles ensure that the resulting $\mu$TSS is maintainable and easy to use. Moreover, the modular architecture of the $\mu$TSS allows using only a subset of the provided functionality as it is required, e.g., for embedded systems, mobile devices, or in the context of a security kernel. This paper discusses experiences with the $\mu$TSS, based on several projects such as the TCG TPM compliance test suite and a Mobile Trusted Module (MTM) implementation.
1 Motivation and Problem Description

Trusted Computing is a technology allowing parties of an electronic infrastructure to verify the integrity of remote computing platforms. The Trusted Computing Group (TCG) is an industry-consortium of important IT-enterprises that has published a list of documents specifying building blocks to realize a trusted IT-infrastructure.

The main documents include the TPM specification [19] defining a hardware module providing protected keys and cryptographic functions, the Trusted Network Connect (TNC) specification [20] defining protocols and formats on the network level, and the TCG Software Stack (TSS) specification [11] defining software layers to access the TPM. Based on these building blocks (hardware, software, network protocols), a trusted IT-infrastructure can be built where access to critical information is only permitted after a successful authentication and integrity verification of the involved computing platforms.

Trusted Computing is an evolving technology and in the near future, many new applications that are using this technology as a building block are expected to come into the market. Example applications currently under development are security kernels [12, 15], VPN gateways [5], and security add-ons such as TPM-based hard-disk encryption systems [13, 7].

However, the trust in an IT-infrastructure depends on the trust in its building blocks including the TPM, the TSS, and the used protocols. The TSS has especially to be trusted by the user, since it (i) has access to many cryptographic keys used to encrypt user-data and (ii) it can violate anonymity requirements. Moreover, the TSS specification is very complex, since it includes about 750 pages containing a huge number of structures, constants and function definitions (see Section 2 for more details) distributed over several architectural layers. This results in three main disadvantages: Firstly, the high complexity of the TSS API makes it hard for developers to profit from its security functions. Secondly, the high complexity decreases the maintainability and increases the probability of internal bugs and wrong or insecure implementations of application developers. Thirdly, the TSS specification includes functions that are not required or even not available, especially in small execution environments such as embedded systems, mobile devices, or security kernels. In some environments, for instance, persistent memory to realize an object storage is not available.

Contribution. We define requirements to be fulfilled by a software stack compatible with the TPM Main Specification and to be used in the context of small and security-critical environments such as embedded systems, security kernels, and MTM/TPM realizations. Based on these requirements, we propose an object-oriented TSS interface and the corresponding implementation providing a lightweight interface to developers that is more intuitive and easier to use than the flexible but complex interface of the existing TSS specification. The $\mu$TSS design is directly based on the TPM Specification and thus covers the full TPM functionality without adding the overhead required for a full TSS implementation. Moreover, the object-oriented design of the $\mu$TSS hides the complexity of the TPM Specification by automating functions such as key loading, unloading, or the creation of authentication sessions. Finally, the $\mu$TSS design is modular allowing the use of only a subset of the provided functionality as it is required, e.g., for embedded systems or mobile devices. The $\mu$TSS implementation has

---

1 The term complexity is used in this paper to point that something is not easy to understand and is hard to deal with. This may be substituted by the term complicated.
already been successfully used in different projects: The TCG TPM Compliance Test Suite used for certification of TPMs\(^2\), a Software-TPM as well as a Software-MTM implementation, and the TPM Manager [17], an open-source graphical TPM management tool based on the Qt widget-library\(^3\).

It is important to mention that the proposed \(\mu\)TSS design is not intended to replace the existing TSS specification. However, it is rather a suggestion of an alternative design allowing a more intuitive and easier usage of the functionality provided by the TPM. When meaningful, we decided not to support specific functions required by the TSS specification (e.g., the object storage) in favor of an easier and less complex software architecture and to make it modular enough to allow an integration into standalone binaries such as a boot loader.

**Outline:** This paper is structured as follows: The next section introduces the TSS as specified by the TCG. Section 3 outlines the deficiencies we identified during our experiences with the existing TSS and defines our requirements for our own TSS design. The design of the \(\mu\)TSS is explained in Section 4, followed by a usage example shown in Section 5. Section 6 summarizes our experiences with our \(\mu\)TSS implementation. A short reference of related work is given in Section 7. Section 8 concludes this work with a short summary.

## 2 TSS Basics

In the following, we give a short overview of the architectural layers, components, and responsibilities of the TSS as defined in its specification [11].

The main purpose of the TSS is to multiplex access to the TPM, since a TPM has limited resources and can only communicate with one client at a time. Moreover, not all functions related to Trusted Computing require TPM access. These functions are located within the TSS to allow a reduction of the complexity of the TPM.

As illustrated in Figure 1, the TSS is comprised of the three layers TSS Service Provider (TSP), TSS Core Services (TCS), and the TSS Device Driver Library (TDDL) respectively their public interfaces TSPi, TCSI, and TDDLi.

![Figure 1: Architectural layers of the TCG Software Stack (TSS).](image)

**TSP and TSPi:** An application employing TPM functionality has to use a TSP via its interface, the TSPi. Every application uses its own TSP that is loaded as a library into the application

\(^2\)http://www.trustedcomputinggroup.org/certification/
\(^3\)https://projects.sirrix.com/trac/tpmmanager
process using it. The TSP provides high-level TCG functions as well as auxiliary services such as hashing or signature verification.

One important aspect of the TSP is the TSP Context Manager (TSPCM), which provides dynamic handles allowing efficient usage of multiple TSPs and application resources. The managed context provides a connection to a TSS Core Service, which itself provides functions for resource management such as freeing of memory, creating working objects, establishing a default policy for working objects, and providing functionality to access the persistent storage database. However, providing all these functions means an enormous increase in complexity.

A second aspect is regarding the TSP Cryptographic Functions (TSPCF), which are provided to make full use of the protected functions in a TPM. In its specification, the TSPi is defined as a C interface.

TCS and TCSI: The TCS provides a common set of operations to all TSPs running on a platform. Since there is only one instance of a TCS per TPM, the main task of the TCS is to multiplex TPM access. Moreover, the TCS provides operations to store, manage, and protect keys as well as privacy-sensitive credentials associated with the platform. The TCS also includes more sophisticated functions to manage TPM resources such as key management.

TDDL and TDDLi: The TDDL, accessed via the TDDLi, provides a unique interface to different TPM driver implementations. Moreover, it provides a transition between kernel mode and user mode. The TDDL provides functions (e.g. Open(), Close(), GetStatus()) to maintain communication with the device driver. Additionally, it provides functions (e.g. GetCapability(), SetCapability()) to get and set attributes of the TSP as well as direct functions (e.g. Transmit(), Cancel()) to transmit and cancel TPM commands.

For each of these layers, the TSS specifies several data types, flags, constants, and functions. Collectively the TSS specification contains more than 716 definitions, data types, flags, constants, and 317 functions.

3 Requirements

The requirements have mainly been derived from Trusted Computing scenarios where the usage of a full TSS implementation is difficult or even impossible:

**Trusted Boot Loader:** If a trusted boot loader needs TPM commands that are not supported by the BIOS, the corresponding functions have to be integrated into the boot loader binary. In this scenario, a direct TPM access (using BIOS functionality without tcsd or external TPM driver) is required.

**Mobile and Embedded Devices:** When integrating Trusted Computing functionality such as remote attestation into a mobile phone, only a small subset of TPM commands are required and a multiplexing of different clients, as it is done by the tcsd, is not necessary.

**TPM/MTM Implementation:** By implementing a Software-TPM or a Software-MTM, the commands and structures defined by the TPM specification and not those of the TSS specification are required.

To use Trusted Computing functionality in these scenarios, a TSS is required that fulfills the following requirements:

---

4The current TSS design assumes a monolithic operating system including device drivers running in kernel mode.
1. **Compliance**: The \( \mu \)TSS should be compatible with the TCG TPM Main Specification [19]. Moreover, it would be helpful if the \( \mu \)TSS could be used as the basis of an MTM/TPM implementation.

2. **Completeness**: The \( \mu \)TSS should provide all mandatory functions defined by the TCG TPM Main Specification [19].

3. **Portability**: The \( \mu \)TSS should be usable under different operating systems such as Linux and Windows. Moreover, different hardware architectures, especially embedded platforms such as ARM architectures, should be supported.

4. **Security**: The API as well as the implementation should prevent typical implementation errors such as buffer overflows by offering intuitive interfaces and type safety.

5. **Usability**: The API should be easy and intuitive allowing application developers to use TPM features without much effort in reading and understanding the specification. Moreover, the \( \mu \)TSS interfaces should hide complexity by automating steps (e.g., key loading or opening of session) and hiding version-specific details (e.g., `TPM_CertifyKey` vs `TPM_CertifyKey2`).

6. **Maintainability**: The design should prevent code redundancies to allow an easy maintenance of the code, especially in the context of specification updates.

7. **Modularity**: Software components should be decoupled to reduce both functionality and size for usage in small and security-critical environments such as embedded systems and security kernels. Moreover, it should be possible to embed the \( \mu \)TSS, or parts of it, into single binaries without a runtime environment, e.g., in the context of a boot loader.

8. **Small code basis**: Finally, the \( \mu \)TSS should have as few dependencies as possible. This makes it possible to have a small code basis and therefore it would be suitable to be integrated in security kernels or embedded systems.

The focus of all above requirements is to prevent implementation errors of the TSS itself and of developers using the TSS in their own applications.

### 4 \( \mu \)TSS Design

In the following, we describe design and implementation aspects of the \( \mu \)TSS, an object-oriented TCG Software Stack that fulfills the requirements defined in Section 3.

We have chosen C++ as the programming language, since it allows on the one hand to realize maintainable code because, e.g., of its support for an object-oriented implementation and exceptions. On the other hand, C++ only requires a small runtime environment in contrast to higher-level languages such as Python, Java, or C#.

The \( \mu \)TSS design includes, similar to the TSS specification, the object-oriented architectural layers \( oTSP \), \( oTCS \), and \( oTDDL \) which are described in the following. However, in contrast to the TSS specification, it is an orthogonal design decision of whether these layers should all be included into one application (i.e., creating an application that is directly using the TPM device), or whether they are split into different components to allow multiplexing (i.e., implementing a `tcsd`).
4.1 oTSP Design

Similar to the TSP specification, the oTSP layer provides the high-level interface used by application developers to use TPM functions. The main goal of the oTSP is to provide an object-oriented abstraction of the TPM functionality that is more intuitive than the specified TSS interface and hides as much complexity as possible. The underlying idea behind the design is to model all objects and relations of the TPM specification that are directly visible for an application developer, such as the TPM, Keys, Counters, NV-Space, etc.

4.1.1 The TPM class

The main component of the oTSP design is the class TPM providing the main interface to the physical TPM. The class TPM includes methods to obtain general information about the physical TPM (e.g., the vendor name, version and revision) as well as the possibility to read and write TPM capabilities.

Moreover, the TPM class provides access to the TPM-internal keys, namely the Storage Root key (SRK) and the Endorsement Key (EK). Furthermore, it offers methods to manage Platform Configuration Registers (PCR), create monotonic counters, or manage non-volatile memory (see Figure 2(a)).

4.1.2 Cryptographic keys

One major task of a TPM is to manage cryptographic keys (e.g., encryption keys and signature keys). The TPM distinguishes a variety of key types (e.g., Endorsement Key, Binding Key, Legacy Key, etc.) that are directly represented by the oTSP through an assortment of C++ classes such as StorageKey, SigningKey, UnbindingKey and their public counterparts, e.g., VerificationKey and BindingKey.

Depending on the sensitivity of the key data, all oTSP keys are derived from one of the two base classes, PrivateKey and PublicKey (refer to Figure 2(b)). However, due to the fact that variant key types may implement different functions (e.g., an attestation key can be used to certify other keys and to quote, whereas a signing key can also be used to sign arbitrary data), the functionality to be realized by cryptographic keys has been split into different interfaces to be derived by concrete public key or private key instances.
Moreover, the base class `PrivateKey` hides the complexity of loading and flushing the key. The implementation ensures that the key data is loaded whenever needed, which allows a key manager to flush any key when all key slots are in use. The key data is flushed from the TPM automatically in the destructor of the `PrivateKey`.

Figure 3(a) defines the available interfaces to be used as base classes of private keys and Figure 3(b) defines the appropriate public counterparts.

Concrete TPM key types, such as `SigningKey`, SRK or `BindingKey` implement those interfaces that provide a functionality to be used with that key. For instance, the key type `PrivateLegacyKey` is derived from all private interfaces except `Storage`, since it can be used for certification, signing, quoting and unbinding but can neither seal nor unseal. Figure 4 shows two examples of concrete key instances, namely `PrivateLegacyKey` and SRK.

Other key types are constructed in a similar fashion. This type safe design prevents misuse of the keys, e.g., a `SigningKey` cannot be used by mistake for a binding operation, and the design provides a better overview of operations which the `SigningKey` can perform, namely, signing, certification and attestation.

### 4.1.3 Monotonic counters

The TPM methods `createCounter()` and `getCounter()` are used to create and instantiate monotonic counters represented by the class `Counter`. (refer to Figure 5).

The counter method `increase()` is used to increase the value of the counter, while the method `readValue()` reads the current counter value from the TPM. The method `readLastValue()` returns the last value that was read from the TPM (without querying the TPM).

### 4.2 oTCS Design

The architectural layer oTCS implements the TCS providing TPM commands, sessions, and the `CommandExecutor` interface, to execute TPM commands. Moreover, this layer defines
Figure 4: \texttt{PrivateLegacyKey} is realized by deriving from all usable private key interfaces and the \texttt{SRK} is realized by deriving from the storage interface.

Figure 5: The interface of the class \texttt{Counter} representing a monotonic counter created by a TPM.

elementary data types and structures of the TPM specification.

One of the principal design directives characterizing this layer is to realize each TPM command as a separate C++ class to allow more flexibility in supporting various specifications and minimizing the interface. By implementing this so-called command pattern [9], such a design allows the modification of single command implementations (e.g., in the case of a specification update), or the removal of all unneeded commands (e.g., to use a selection of commands in an embedded environment).

4.2.1 TPM commands

The main classes of the oTCS layer are \texttt{Command} and \texttt{Result} representing a command sent to a TPM and the result returned by a TPM, respectively (refer to Figure 6).

Figure 6: Interface of the classes \texttt{Command} and \texttt{Result} including a list of \texttt{Operand} types and \texttt{Session} types.
The method `execute()` of the class `Command` executes a TPM command and returns the associated instance of a class derived from `Result`. For example, a result of type `GetCapabilityResult` is returned by the command `GetCapability` implementing the `TPM_GetCapability` command.

Both, commands and results, consist of attributes derived from the operand base class `Operand` or `Session`. These operands represent the parameters sent to, or returned from, the TPM, while session objects are responsible for the authentication of TPM objects or roles. The inclusion of a session object in a command causes the appropriate operands (handle, nonces, etc.) to be included with the command.

Commands as well as operands implement a streaming interface allowing objects to be converted into a binary stream. The resulting binary stream can be sent directly to a TPM. This way, the complex marshalling and unmarshalling process of commands and their operands has to be implemented only once in the `Command` and `Result` base class and will be inherited by derived commands. This also includes the complex calculation of the session authentication when sending commands as well as authentication verification and consistency checks of data received from the TPM. If the TPM returns an `TPM_AUTHFAIL` error code, e.g., the `Result` automatically throws an appropriate exception of the same name to be handled by the application.

### 4.2.2 Secret Provider

The secret provider interface manages the authentication information to be used by the TPM. All authentication data, e.g., of owners, keys, or other objects are saved and managed by the `SecretProvider`. Users of the \( \mu \)TSS can derive their own implementation that, e.g., implements a secure storage, a password cache, or a GUI dialog to allow users to enter the secrets.

### 4.2.3 Sessions

Sessions are used within a TPM command to authenticate the usage of specific objects such as cryptographic keys and to authenticate a specific role, e.g., the TPM owner. Figure 7 illustrates the available session types provided by the oTCS layer.

![Figure 7: The implementation of the session types OSAP, DSAP, OIAP and Transport](image)

Depending on the specific TPM command, the number and the type of sessions may vary,
just as in the specification. For example, the TPM command `TPM_Quote` accepts one session or no session at all, and the session can be of the type OSAP or OIAP. To be able to handle this, the interface `Session` provides the public interface of all session objects.

Separate classes are dedicated to the various session types: `OIAP_Session` for OIAP sessions, `OSAP_Session` for OSAP sessions, `DSAP_Session` for DSAP sessions, `TransportSession` for transport sessions.

### 4.2.4 Command executor

The oTCS provides the `CommandExecutor` interface (see Figure Figure 8) as the central instance used to handle the execution of TPM commands as well as select the backend for execution (e.g., TDDL or `tcad`). A typical responsibility of the command executor is the handling of non-fatal errors returned by the TPM. For example, the command executor catches the non-fatal TPM errors `TPM_RETRY` and waits an appropriate amount of seconds, until it tries to resend the command to the TPM. This way, the command executor handles many exceptional TPM states that normally had to be handled by the application developer.

![Figure 8: Interface and available instances of the command executor](image)

The TCS framework allows stacking of different command executor implementations. This way, an implementation that handles certain TCS or TPM errors, or an implementation that audits the TPM commands and responses, can be added to the existing functionality.

### 4.3 oTDDL Design

The oTDDL implements the TDDL interface, i.e., the interface to the TPM device driver. As illustrated in Figure Figure 9, the singleton `TDDL` implements the `TDDLi` interface used to send TPM commands, and the associated TPM response, using the method `transmit()` based on byte streams. The static method `getInstance()` provides access to the singleton by returning a reference to the TDDL implementation.

Different back-ends implementing the `TDDLi` interface are provided. The class `SocketTDDL` uses a TCP/IP connection to send TPM commands to a remote service. The class `DeviceTDDL`, however, accesses the TPM of the local platform under different operating systems such as Linux or Windows.

#### 4.3.1 Multiplexer

To allow multiple applications to access the TPM, a `tcad`-like multiplexer has been implemented. The multiplexer acts upon the TDDL byte streams and isolates clients using the class `Context` implemented based on the TPM commands `TPM_LoadContext` and `TPM_SaveContext.`
Since the multiplexing is done on the TDDL layer, every client can use the multiplexer through the `SocketTDDL` backend.

The multiplexer itself has been implemented by only 180 code lines, including the management of different command line options.

### 5 Usage Examples

In this section we are going to give two illustrative examples to show how the \( \mu \)TSS is used.

#### 5.1 Binding

Listing 1 shows how to bind some data to the TPM and how to unbind it.

In the first line we create a random plaintext of length 204 used for encryption. Then, an unbinding key is created by the method `createUnbindingKey()` of the SRK which itself was obtained from the TPM singleton. The invoked method internally uses the TPM command `TPM_CreateWrapKey` with the correct parameters, and returns the created UnbindingKey (`ubKey`).

In order to bind the plaintext, first the method `getBindingKey()` is used to retrieve the public key part `bKey` which is then used to bind the plaintext by invoking the method `bind()`. When invoking the method `unbind()` to unbind the ciphertext, the created key data stored within the class `UnbindingKey` is automatically loaded into the TPM.

In general, the application developers neither have to care about occurring TPM or TCS errors, nor do they have to free allocated resources explicitly, because both are automatically done by the exception system in combination with appropriate destructors. For example, if the TPM returns the error `TPM_INVALIDKEYHANDLE`, because the used unbinding key is not loaded (or has been unloaded in between), this error is caught by the unbinding key which then loads the key data into the TPM and re-invokes the failed command.

For comparison, Appendix A shows an implementation of the same functionality using TrouSerS and Appendix B based on jTSS. Both examples are a copy of the standard test suite implementations and are not provided by the authors of this paper.
try {
    BYTEARRAY plain(204);
cout << "Plain: " << plain << endl;

    /// Create unbinding key
    TPM &tpm = TPM::getInstance();
    SRK srk = tpm.getSRK();
    UnbindingKey ubKey = srk.createUnbindingKey();
    /// Extract binding key from unbind key
    BindingKey bKey = ubKey.getBindingKey();

    /// Bind the data
    BYTEARRAY cipher = bKey.bind(plain);
cout << "Cipher: " << cipher << endl;

    /// Unbind the data
    BYTEARRAY result = ubKey.unbind(cipher);
cout << "Result: " << result << endl;
} catch (TPM_Error &e) {
    cerr << "TPM Error: " << e << endl;
} catch (TCS_Exception &e) {
    cerr << "TCS Exception: " << e << endl;
}

The TrouSerS example uses the TSP API and requires 146 source lines of code (LOC) versus 18 LOC using the µTSS\(^5\).

The jTSS example requires slightly more LOC, namely 30, than the µTSS-based implementation. However, it provides less type-safety for the key object. The same key data object is used for binding and unbinding, as well as for binding and signing. Thus, the wrong key usage will result in failures during runtime instead of compile time. Moreover, loading the key data has to be done explicitly and the key will not be flushed automatically after closing the application. That is, executing the existing code several times will cause a TPM_NOSPACE error.

5.2 Attestation

Listing 2 shows how to perform remote attestation and how to verify the attestation result using the µTSS.\(^5\)

First, a new Attestation Identity Key (AIK) is created by the method createAIK() of the class SRK, which itself is returned by the TPM singleton. The invoked method executes the TPM command TPM_MakeIdentity. To create the AIK, a digest of the chosen identityLabel and the privacyCA of the new TPM identity is required. In this example, we use a random digest provided by the class RND and a hard-coded key usage authentication. The owner authentication digest is provided by the SecretProvider.

After creating the AIK, a random nonce and a selection of PCRs are used as input data of the quote operation. The method quote() calls internally the TPM command TPM_Quote2 or TPM_Quote, depending on the input parameters and the used TPM. The AttestationResult keeps the returned information of the quote operation. This result can be verified by the

\(^5\) measured by sloccount http://www.dwheeler.com/sloccount/
identity object, extracted from the aiik object. The identity object exclusively possesses the
public part of the key and can perform the operation verifyQuote() to verify the attestation
result.

6 Evaluation

As shown in the examples of Section 5, using the μTSS is very easy, since it hides many details
behind simple interfaces.

A performance analysis showed that a comparison with other software solutions is irrele-
vant, due to the fact that the TPM itself is much slower than the software. However, com-
paring an implementation of the binding example explained in Section 5.1 and the binding
example explained in Appendix A shows that the implementation based on the μTSS imple-
mentation takes 1.7 seconds, while the example based on TrouSerS takes 2.00 seconds.

The μTSS is tested extensively and used in several applications. Section 6.1 gives an overview
of the most important applications that are based on the μTSS.

6.1 Applications

The following applications are implemented based on the μTSS.

To further test the extensibility and flexibility of the μTSS design, we decided to develop
a Software-TPM based on it. We hoped that the Software-TPM could be developed without
much effort, since nearly all the required data types and the specification of commands al-
ready exist. In fact our expectations have been completely satisfied. The development of the
Software-TPM is limited to the implementation of the concrete command (according to the
TPM specification), by overwriting the method execute() of each Command object using a
derived class.
Based on the Software-TPM described before, an **MTM Software Stack** and **Software-MTM** were implemented. An MTM [14] is a security extension specified by the TCG which is similar to the TPM but intended for use in embedded and mobile devices. An MTM command set is similar to the TPMs, but some differences exist: On the one hand, some TPM commands are not available on an MTM and on the other hand, the MTM has some additional commands such as `MTM.VerifyAndExtend()`. As at the moment no hardware support for MTM exists, this implementation provides a facility to experiment with the basic concepts of the MTM.

The **TrustedVPN** is a TPM-based enterprise VPN solution based on a central management server. The implementation on both the client and server side heavily utilizes TPM functionalities, e.g., to establish a PKI based on TPM-internal keys, to bind data to a platform configuration, and to realize a trusted channel [6, 16] between VPN client and management server. The actual implementation of the TrustedVPN solution is based on the μTSS, taking advantage of its easy API, its limited size and complexity due to the fact that only a subset of TPM commands are used, and the capability to use the TPM without the need to start a `tcsd`.

The μTSS has been successfully used to develop a **TPM Compliance Test Suite** including more than 650 test cases and covering nearly all TPM commands that can be tested on a standard PC platform. Here, the oTCS layer has mainly been used to implement the concrete test cases, while the high-level abstractions of the oTSP are used as helper to generate input data etc. The development of the test cases was very easy, since the μTSS hides most of the complexity of the TSS.

Another project that uses the μTSS is the **TPM Manager**, an open-source graphical TPM management tool based on the Qt widget-library. Earlier versions used the TrouSerS library as the backend, while the current developer version has been ported to the μTSS within one to two days. A specific advantage of the μTSS here is the DeviceTDDL implementation allowing an application to directly accessing the TPM driver without the need of a `tcsd` daemon. This was required to keep an `initrd` that includes a TPM Manager as small as possible.

### 6.2 Overall Evaluation Results

Although the μTSS implementation does not provide the complete functionality required by the TSS specification, our experiences show that a TSS implementation, providing a more intuitive and easier interface, helps developers to successfully use TPM functions. The object-oriented and clean design helped a stable state to be reached in a relatively small time. Finally, the command pattern [9] used to realize the TPM commands allows the easy addition, change, or removal of command implementations.

The μTSS fulfills the requirements defined in Section 3 for the following reasons:

**Compliance and Completeness:** Since the μTSS has been used to realize the TCG TPM compliance test suite, it obviously fulfills the compliance and completeness requirements. Moreover, the μTSS has been used to develop a software TPM that itself is compatible with the TPM specification.

**Portability:** The μTSS has been implemented in C++ and is in use on top of the Linux/x86, Linux/ARM, and Windows XP operating systems.

**Security:** The type safe design of the μTSS reduces the risk of runtime errors and the use of standard data containers instead of C-pointers limits the number of possible buffer and heap

overflows. However, C++ is not the ideal programming language for developing secure code but has been chosen, because it allows low-level system implementations.

**Usability:** The µTSS provides a simple API hiding most of the complexity of the TPM specification. The exception handling capability makes source code easier to read and helps developers to focus on their tasks.

**Maintainability:** The object-oriented design of the µTSS reduces code redundancies and hides implementation details behind object interfaces.

**Modularity:** The µTSS can be used in different configurations. E.g., it can directly access a TPM device, use its own TPM driver, or access a network server such as a tcsd. Moreover, it can be configured to support only a subset of TPM commands and structures.

**Small code basis:** The µTSS only includes structures, TPM commands, and logical objects of the TPM specification. While the oTCS layer is a one to one mapping of the TPM specification, the oTSP layer implements the logical TPM objects such as keys, counters, and NV-Space areas.

### 7 Related Work

In the following, we briefly discuss the related works in the context of the TCG software stack. Moreover, we analyze to which extent these implementations fulfill the requirements of Section 3. To our knowledge none of the existing implementations directly implements the TPM Main Specification [19].

**TrouSerS** is an open source TSS implementation maintained by IBM since 2004. The latest version released is stable and implements TSS version 1.1 [10] interfacing version 1.1b TPMs. Since the TCG has released a new version of the TSS specification [11] to support new functionality provided by version 1.2 TPMs, version 0.3.x of TrouSerS was developed. Most of the features are implemented. The source code itself is implemented in ANSI C. Due to the high complexity and the elaborate specification, the implementation of a TSS stack is very challenging. The usability and maintainability requirements as described in Section 3 cannot be achieved using TrouSerS.

The Trusted Platform Agent (TPA) [4] is an open source library, which is designed to minimize the effort of writing applications that use Trusted Computing technology and employ the TPM. According to the website, the TPA hides the complexity of the TSS interface. The library provides a small set of functionalities such as binding, sealing, TPM management (take-ownership and is-owned), PCR (reading and extending). The TPA library has dependencies on TrouSerS, TrouSerS tpm-tools, OpenSSL, SQLite and libcurl.

The IAIK jTSS is developed and maintained at the Institute for Applied Information Processing and Communication at Graz University of Technology [1]. The IAIK jTSS stack is a new implementation of the TCG Software Stack for the Java programming language. IAIK has initialized a Java Specification Request (JSR 321) in the Java Community Process (JCP). The current status of the specification process is outlined in the recent paper of the JSR 321 group [18]. However, the need to use a Java runtime environment for running the jTSS violates our requirement of a small code basis.

**TPM/J** [3] is an object-oriented API using Java for low-level access to the TPM. It was developed as part of the research project on Trusted Computing at MIT. TPM/J treats TPM low-level commands (i.e., the commands directly given to the TPM chip itself), and the response data structures of these commands, as first-class Java objects. The TPM/J stack does not provide
full functionality for TPM commands such as Quote2, Delegation commands, NV write commands, CertifyKey commands, CMK commands and DSAP sessions.

Another library provided by IBM [2], *libtpm*, is implemented to communicate with TPMs according to the TCG specifications. The library supports TPM v1.1b and therefore it is not fully functional with TPMs of version 1.2. For instance, `TPM_RESET` and `TPM_LOADKEY` are not working anymore. *libtpm* contains a small set of most important TPM commands such as `TPM_Seal`, `TPM_Unseal`, `TPM_Bind`, `TPM_Unbind`, and has been used in the past as basis for a security kernel [15]. As *libtpm* only provides a limited functionality and is not properly functional with TPM v1.2, it does not fulfill our requirements.

The *Minimized MRTM* [8] is a software implementation of a minimal MRTM that runs in hardware-enforced isolation inside the trusted execution environment of a Nokia N96 handset. The code is, with a few minor exceptions, compatible with the MTM v1.0 specification, and as a monolithic compilation it can execute in 20kB of RAM encompassing both code and data. This is achieved by reducing the data structures to a specification compliant minimum and by optimizing the command logic to comply with the highly specialized demands of an MRTM. However, it is not the goal of the µTSS to realize a minimal MTM. Instead, a TSS should be implemented that provides the basis for many different application scenarios including the development of a Software-MTM.

## 8 Conclusion

This paper presented the µTSS, an object-oriented TSS design and implementation providing developers an intuitive and easy to use interface. We have analyzed in Section 1 the deficiencies of a complex and difficult to understand software architecture. Based on three example scenarios, Section 3 identified requirements of a TSS to be used in specific environments such as embedded platforms or boot loaders.

The major principles behind our µTSS design are reduced complexity to simplify the interface and the implementation, type safety to prevent runtime errors and potential programming errors, object encapsulation to hide the complexity of the implementation, and meaningful error handling to prevent resource leaks. The design of the µTSS prevents code redundancies and makes maintenance of the code easier and thus the implementation more stable. The µTSS design is scalable allowing the use of only a subset of the provided functionality as it is required, e.g., for embedded systems or mobile devices. We discussed in Section 6 our positive experiences with our µTSS based on several projects such as the TCG TPM compliance test suite, the TPM Manager, and a Software-TPM as well as a Software-MTM implementation.

## A TrouSerS Example

In this Appendix we show an example of how to perform binding and unbinding operations using TrouSerS. The same example is implemented by the µTSS and can be found in Section 5.1.

In the first step of Listing 3, a context has to be created using the method `Tspi_Context_Create`, which is used to maintain a handle to the current TSP library and to be able to create new working objects within this context. The TSP Context is connected to the TCS provider by
The next step is to create a key object and data object to save the bound data using the method `Tspi_Context_CreateObject`. Then the SRK object is created using the method `Tspi_Context_LoadKeyByUUID`. `Tspi_SetAttribUint32` is used to initialize the key parameters, which is going to be created by `Tspi_Key_CreateKey`, using the SRK as the parent key. The binding key is loaded into the TPM using `Tspi_Key_LoadKey`. Subsequently, the method `Tspi_Data_Bind` is called to perform the bind operation. Afterwards, the binding result is assigned to the context using `Tspi_GetAttribData`. The unbind operation is done using `Tspi_Data_Unbind`. Finally all objects must be destroyed using the method `Tspi_Context_FreeMemory` and the context must be closed using the method `Tspi_Context_Close`.

Listing 3: Binding Example with TrouSerS

```c
TSS_HCONTEXT hContext;
TSS_HKEY hSRK;
TSS_HKEY hKey;
TSS_HPOLICY hSrkPolicy;
BYTE *prgbDataToUnBind;
TSS_HENCDATA hEncData;
UINT32 pulDataLength;
BYTE rgbDataToBind[DATA_SIZE];
BYTE *rgbEncData = NULL;
UINT32 ulDataLength = 0;
UINT32 ulEncDataLength = 0;
TSS_UUID uuid;
TSS_RESULT result;
 UINT32 exitCode;
TSS_FLAG initFlags =
    TSS_KEY_TYPE_BIND|
    TSS_KEY_VOLATILE|
    TSS_KEY_NO_AUTHORIZATION|
    TSS_KEY_NOT_MIGRATABLE;

memset(rgbDataToBind, 0x5a, DATA_SIZE);

// Create Context
result = Tspi_Context_CreateObject(
    hContext,
    TSS_OBJECT_TYPE_RSAKEY,
    initFlags,
    &hKey);
if (result != TSS_SUCCESS) {
    print("Tspi_Context_CreateObject(hKey)", result);
    Tspi_Context_FreeMemory(hContext, NULL);
    Tspi_Context_Close(hContext);
    exit(result);
}

// Create hKey
result = Tspi_Context_CreateObject(
    hContext,
    TSS_OBJECT_TYPE_ENCDATA,
    TSS_ENCDATA_BIND,
    &hEncData);
if (result != TSS_SUCCESS) {
    print("Tspi_Context_CreateObject(hEncData)",
    result);
    Tspi_Context_FreeMemory(hContext, NULL);
    Tspi_Context_Close(hContext);
    exit(result);
}

// Connect to Context
result = Tspi_Context_Connect(
    hContext,
    get_server(GLOBALSERVER));
if (result != TSS_SUCCESS) {
    print("Tspi_Context_Connect", result);
    Tspi_Context_FreeMemory(hContext, NULL);
    Tspi_Context_Close(hContext);
    exit(result);
}

// Load Key By UUID
result = Tspi_Context_LoadKeyByUUID(
    hContext,
    TSS_PS_TYPE_SYSTEM,
    SRK_UUID,
    &hSRK);
if (result != TSS_SUCCESS) {
    print("Tspi_Context_LoadKeyByUUID(hSRK)",
    result);
    Tspi_Context_FreeMemory(hContext, NULL);
}
```
B jTSS Example

In this Appendix we show an example of how to perform binding and unbinding operations using the jTSS. The same example is implemented by the µTSS and can be found in Section 5.1.
Listing 4 shows the steps required for creating a binding key and do a bind on data and afterwards unbind it using the TPM. The jTSS provides a much more simpler API for implementing the test cases, however, the jTSS does not provide type safety for the keys. In this design the binding key and unbinding key are represented by the same object, while in the example implemented using the µTSS, Listing 1, ubKey represents the UnbindingKey and contains the private portion of the key, while bKey represents binding key and contains only the public portion of the key.

Listing 4: Binding Example with jTSS

```java
try {
   // create new key
   TcIRsaKey key = context_.createRsaKeyObject(
      TcTssConstants.TSS_KEY_TYPE_BIND |
      TcTssConstants.TSS_KEY_SIZE_2048 |
      TcTssConstants.TSS_KEY_VOLATILE |
      TcTssConstants.TSS_KEY_NO_AUTHORIZATION |
      TcTssConstants.TSS_KEY_NOT_MIGRATABLE);

   keyUsgPolicy_.assignToObject(key);
   keyMigPolicy_.assignToObject(key);
   key.createKey(srk_, null);
   key.loadKey(srk_);

   // create encdata object
   TcIEncData encData = context_.createEncDataObject(
      TcTssConstants.TSS_ENCDATA_BIND);

   // bind
   TcBlobData rawData = TcBlobData.newString("Hello World from IAIK!");

   encData.bind(key, rawData);

   // get bound data
   TcBlobData boundData = encData.getAttributeData(
      TcTssConstants.TSS_TSPATTRIB_ENCDATA_BLOB,
      TcTssConstants.TSS_TSPATTRIB_ENCDATABLOB_BLOB);

   // unbind
   TcBlobData unboundData = encData.unbind(key);
}
catch (TcTssException e) {
   e.printStackTrace();
}
```
References


