Towards Platform-Independent Trusted Computing

Ronald Toegl
Institute for Applied Information Processing and Communications (IAIK)
Graz University of Technology
Inffeldgasse 16a
8010 Graz, Austria
ronald.toegl@iaik.tugraz.at

Thomas Winkler
Pervasive Computing /
Institute of Networked and Embedded Systems (NES)
Lakeside 802b
9020 Klagenfurt, Austria
thomas.winkler@uni-klu.ac.at

Mohammad Nauman
Institute of Management Sciences,
1-A, E/5, Phase VII,
Hayatabad Peshawar, Pakistan
nauman@imsiences.edu.pk

Theodore Hong
University of Cambridge
Computer Laboratory
William Gates Building
15 J.J. Thomson Ave.
Cambridge CB3 0FD
United Kingdom
theodore.hong@cl.cam.ac.uk

ABSTRACT
Software independence from hardware platforms is an important feature of growing significance, given the emergence of new distributed computing paradigms. It would be desirable to extend the Trusted Computing mechanisms offered by the Trusted Platform Module into the platform independent Java environment. However, there is currently no generally accepted Trusted Computing API for Java. In this paper, we describe the design of a high-level API for Trusted Computing in Java, which is set to become the new industry standard for Java applications. We describe the current state of the standardization effort being undertaken in Java Specification Request 321 (JSR 321).

Categories and Subject Descriptors
D.2.3 [Coding Tools and Techniques]: Standards; K.6.m [Miscellaneous]: Security

General Terms
Design, Security, Standardization

Keywords
Trusted Computing, Java, JSR 321

1. INTRODUCTION

Many new software paradigms such as mobile agents, grid and cloud computing are characterized by self-contained computer programs that move through the network acting on behalf of a user or other entity. Such programs, which seek to enable unique new interactive services by migrating across heterogeneous computing environments on different hardware and software platforms, are ideally supported by platform-independent runtime environments [4].

The need for security in such distributed applications is well-established [7], and Trusted Computing (TC) promises [8, 20, 1, 10, 16] to overcome most of the critical security issues. This has especially been demonstrated in many Java™-based use cases [3, 14, 20, 11, 9, 17].

Although different deployment platforms such as desktop computers, servers, professional data centers, mobile phones and embedded devices may offer compatible Java Runtime Environments, they likely employ different hardware and software mechanisms to establish trust – especially since there is currently no established standard API definition for Trusted Computing on Java platforms.

To permit independence from the actual hardware platform, whether it be a Trusted Platform Module, Mobile Trusted Module, or other vendor-specific hardware mechanism, it is necessary to provide programmers (and users) with higher abstractions over the core concepts of TC. Ultimately, this will allow mobile code to migrate between devices while maintaining trustworthiness.

In the remainder of this paper, we outline the ongoing creation of an industry standard which will provide this hardware independence in an open, transparent way. Section 2 outlines mechanisms for TC as well as the Java platform and its applicability to TC. After a review of existing approaches and related work in Section 3, we proceed to describe the design and standardization process of Java Specification Request 321 (JSR 321) in Section 4. Work in progress towards an API is presented in Section 5. We conclude in Section 6.
2. CURRENT PLATFORMS

2.1 Trusted Hardware

Trusted Computing as it exists today is defined by the specifications of the Trusted Computing Group (TCG). A hardware component, the Trusted Platform Module (TPM), is integrated into commonly available general-purpose hardware, with hundreds of millions of platforms shipped so far. Like a smart card, a TPM features cryptographic primitives, but is physically bound to its host device. It contains a tamper-resistant integrated circuit implementing public-key cryptography, key generation, secure hashing, and random-number generation. Using these components, the TPM can enforce security policies on hierarchies of secret keys to protect them from software attacks by any remote attacker. Thus, a TPM can be used to perform cryptographic signatures on user-provided data using hardware-protected private keys. However, due to limited TPM memory, keys have to be swapped out of the TPM when not in use. To protect these keys, a parent storage key specified on key creation is used to wrap the private part of the child key when it is exported from the TPM. At the top of the key hierarchy is the storage root key created when taking ownership of the TPM. Keys are assigned a user-supplied secret, which is used in several authentication protocols, and optionally a system state that has to be provided when using the key for cryptographic operations.

This state information is held in a few specially protected Platform Configuration Registers (PCRs), which can only be written via the one-way extend operation, and are used to build a chain of trust. In the simplest case, the caller computes a chain of hash values and extends a PCR with the result, starting from the BIOS, covering bootloader, kernel, and system libraries etc., up to application code, before executable code is allowed to run. A variant, which allows dynamic code measurements at runtime is available on systems with hardwired chipset-CPU-TPM cooperation.

Ultimately, the exact configuration of the platform is mapped to PCR values. In the Quote operation, the TPM signs these values together with a supplied nonce, thus enabling more complex protocols such as Remote Attestation. Here, a remote verifier can analyze the result and decide whether to trust the configuration for a given purpose or not. To protect the platform owner’s privacy in such a scenario, the unique Endorsement Key, injected by the TPM manufacturer, or created upon machine deployment, is not used for this signature. Rather, a pseudonym is used: an Attestation Identity Key (AIK). The authenticity of an AIK can be certified by an online trusted third party, called PrivacyCA or by the more complex group-signature based DAA scheme.

The TPM can also bind data to a platform by encrypting it with a non-migratable key. Such a key cannot be extracted from the TPM’s protected storage unencrypted nor can it be moved to another TPM. An extension to this is Sealing. Data may be sealed to a specific set of values of the PCRs of a specific TPM. Thus, access to the data can be restricted to a single trusted state of the TPM’s host computer.

2.2 TCG Software Architecture

The hardware resources of a TPM are manufacturer implementation specific and typically very limited. For instance, the TPM supplies only a few cryptographic key slots and thus must continually swap keys to and from external storage during operation. For later analysis of the aggregated PCR information, a Stored Measurement Log (SML) must be kept by the system software. Thus, the current TPM design establishes the need for a singleton system software component that authoritatively manages the TPM device resources and arbitrates concurrent accesses from multiple clients. To this end, the TCG specifies an architecture that implements TPM access and management, the TCG Software Stack (TSS) [15], which targets C-based systems and applications.

The lowest layer, the Trusted Device Driver Library (TDDL) abstracts the low-level hardware details into a platform independent interface that takes commands and returns responses as byte streams. Resource management is implemented in the Trusted Core Services (TCS), which runs as a singleton system service. Additional functionality provided by the TCS is persistent key storage, TPM command generation, and various communication mechanisms. The TCS event manager handles the SML. The upper layers of the software stack may access the TCS via the platform-independent Simple Object Access Protocol (SOAP) interface, a network protocol that manages multiple requests, ensuring proper synchronization. Applications can access Trusted Computing functionality by using the Trusted Service Provider (TSP) interface. This interface provides a Context object as the entry point to all other functionality such as policies and key handling, data hashing, encryption or PCR composition. In addition, mechanisms for command authorization and validation are provided. Each application dynamically uses a shared library instance of the TSP. The TSS was also designed to allow partial integration with existing high-level APIs libraries, such as PKCS #11. This enables the use of the cryptographic primitives provided by the TPM by legacy software. A limitation of this approach is that these legacy cryptographic APIs do not account for high-level TC concepts such as Sealing.

Recent years have seen the successful integration of generic TPM 1.2 hardware drivers into major operating systems. Of the C-based TSS, several commercial implementations and one open source implementation (IBM’s TrouSerS [6]) exist. To the best of our knowledge, none of these implementations completely cover the current TSS specifications.

Despite the availability of these TC-enabling software components, very few actual software products make use of the TSS to access the TPM. Indeed, a recent study [15] on the TSS concludes, that, “it is apparent that, until now, no application exists that makes use of this technology. Even the simplest applications [...] have not been applied yet.”

2.3 TC in the Java Environment

The Java programming environment has seen a broad adoption ranging from large-scale business applications hosted in dedicated data centers to resource constrained environments as found in mobile phones or Personal Digital Assistants (PDAs). Java programs are not compiled to native machine code but to a special form of intermediate code, called byte code. This byte code is then executed by a virtual machine (VM) called the Java VM. This characteristic makes Java an excellent choice for development aiming at heterogeneous environments. In contrast to conventional programming languages such as C or C++, Java is equipped

\[\text{http://www.trustedcomputinggroup.org}\]
with inherent security features supporting the development of more secure software. Among those features are automatic array-bounds checking, garbage collection and access control mechanisms. Additional aspects that distinguish Java from other environments are code-signing mechanisms and the verification of byte code when it is loaded.

Over time, Java has become one of the major development environments for business applications, especially in fields that highly depend on the security and trustworthiness of computer systems, e.g. financial service providers. This commercial business environment is one of those fields where Trusted Computing technologies are expected to see first deployments.

Another area of application is network-based software, where Java is a logical choice for highly distributed applications that are deployed in heterogeneous environments. Here also, Trusted Computing is very promising. For example, Trusted Computing is used in financial services, and the verification of byte code when it is loaded.

### 3. REVIEW OF EXISTING APPROACHES

This section presents an overview of existing libraries and APIs that provide support for Trusted Computing for Java developers. Additionally, strengths and weaknesses of the individual approaches are discussed and guidelines for a next-generation design derived.

#### 3.1 TC for the Java Platform and jTSS

The central component of the open source “Trusted Computing for the Java Platform” project, is an implementation of the Trusted Software Stack (TSS) for Java programs called jTSS [12]. It is a large library which provides the functionality to Java programs that is made available to C programs with conventional TSS stacks.

Overall, the project offers two flavors of TSS implementations:

- **jTSS Wrapper**: provides Java programs access to C-based stacks through an object-oriented API, which forwards to the TSS TSP layer using the Java Native Interface (JNI).
- **jTSS**: is a native implementation of the TCG Software Stack written completely in the Java language. It offers seamless support for Linux operating systems and Windows Vista.

The API exposed by both variants is the same, enabling Java application programmers to switch between the two seamlessly, with the choice of the back-end implementation depending on the surrounding platform [17]. While still marked as “experimental”, jTSS is one of the most widely used, supported and regularly updated TSS’s available today. It covers almost all of the functions specified by the TCG for communicating with the TPM at the fine granularity of TSS commands. As with every TSS, a complex flow of commands is required to achieve functionality such as sealing, binding, attestation and key generation for application software. The project also provides jTpmTools, which is a sample implementation of the high-level programs that can be written using the jTSS API. While jTSS significantly reduces the learning curve involved in communicating with stacks like TrouSerS for the average Java programmer, it is still a complicated API that requires a large amount of training before it can be used in large-scale projects.

#### 3.2 TPM/J

TPM/J [13] provides an high-level API that allows Java applications to communicate with the TPM. It is compatible with the Linux, Windows XP and Windows Vista and Mac OS X² operating systems thus living up to the promise of platform independence.

Strictly speaking, TPM/J is not a TSS since it intentionally deviates from the specifications of TCG’s TSS, which seems natural since the TSS specifications provide details specific to the structural programming paradigm and cannot be ported to the object-oriented perspective without major changes to the specs. A drawback is that the library is not a split design as the TSS is, therefore requiring to grant elevated rights to the JVM to access the TPM hardware resource. Moreover, a major concern for users of TPM/J is that it is not regularly maintained, thus making it unsuitable for large-scale adoption in the community.

#### 3.3 TPM4JAVA

TPM4JAVA [9] is a Java library that provides an easy-to-use API to Java programmers for communicating with the TPM. Its design is based on three levels of abstractions: (1) high-level: that allows programmer to execute often-used commands such as taking ownership, compute hashes and generate random numbers. (2) low-level: a not-so-user-friendly means of executing any TPM command allowed by the hardware. (3) backend: used internally for communicating with the TPM device driver library. While the high-level API makes several functions very easy to program, some operations, such as performing a quote during attestation, require several lines of code and a low-level understanding of the actual functioning of the TPM. This makes ‘high-level’ a misnomer. The project has not been maintained for several years. Finally, TPM4JAVA shares the limitation of all approaches of not adhering to the TCG’s specifications.

#### 3.4 Findings and Goals for a Novel API

While the aforementioned APIs all share the common goal of providing Trusted Computing functionality to Java developers, to date none of them has seen widespread adoption beyond research and academia.

One of the main reasons is that the interfaces exposed by the libraries often are difficult to learn and understand. This stems from several reasons: (1) Trusted Computing by itself is a complex technology. The specifications defining the two major components - the TPM and the TSS [19, 18] - together consist of about 1500 pages. The concepts often are not well presented for novice users and details have to be looked up in several different places. (2) Implementations like jTSS try to mimic the interface defined by the TSS specification. This interface, however, was developed for procedural programming languages like C. Even though jTSS tries to map the TSS concepts to an object oriented API, it still does not fit well into the Java ecosystem and feels unnatural to developers familiar with other Java APIs. Additionally, the amount

---

²At the time of writing, the inclusion of TPMs in Mac OS X compatible platforms had been discontinued.
of code required to set up and perform basic TC functions is relatively complex and large, because as in the original C API, methods take relatively long lists of parameters with many potentially illegal combinations. This makes the API error prone and complex to use for developers without detailed TC knowledge. (3) Implementations like TPM/J and TPM4JAV provide alternative interfaces to TC functionality. While in the first case, the interface is at a very low level, the second one offers some higher level abstraction, but is neither consistent nor complete.

Based on that situation, we propose the following design principles for a TC API for Java.

Integration with Existing TC Platforms: To the OS, the Java Virtual Machine appears just as an ordinary application. Therefore, the TPM access mechanisms need to integrate with the surrounding environment and management services [17].

Simplified Interface: To make the new API fit into the Java ecosystem, a completely new and fully object-oriented interface is to be designed. For instance, generic objects (e.g. keys) in the TSS should be replaced with instances of specific classes that represent the different types. This allows the set of offered operations to be limited to those actually applicable for a certain object type, thus furthering usability.

Reduced Overhead: The TSS API requires a substantial amount of boilerplate code for routine tasks. The proposed API should attempt to replace these lengthy code fragments with simple calls using sensible default parameters where required.

Conceptual Consistency: Names in the API should be consistent not only within the API but also with the nomenclature used by the TCG and in TC literature. This will allow users to easily switch from other environments to the proposed API. Still, naming conventions of Java must be adhered to.

Testable and Implementable Specifications: The API design should target a small core set of functionality, based on the essential use cases of Trusted Computing. This restriction in size will allow for complete implementations and functional testing thereof.

Extendability: The API should allow implementers and vendors to add functionality which is optional or dependent on the capabilities of the surrounding platform.

Standard Compliance: Having an industry-wide standard of accessing the TPM from software is indispensable for widespread use and for enabling code mobility. As the TSS API has shown to be unfit for Java environments, the newly proposed API should be based on a novel, independent industry standard.

The above approach is clearly not as flexible as a full-fledged TSS, but practical experience has shown that this flexibility (1) is not actually required for the vast majority of typical TC applications and (2) results in a very steep learning curve preventing a widespread adoption of Trusted Computing. On the other hand, these self-limitations allow to demand that implementations cover the complete API, a key requisite for true platform independence.

4. API DESIGN

We now introduce the intermediate results of the design and standardization process for the Trusted Computing API for Java, known as Java Specification Request #321 (JSR 321)\(^3\) within the Java Community Process (JCP)\(^4\). Formally, the proposed API is currently at the Early Draft Review stage.

The Java Community Process aims to produce specifications using an inclusive, consensus-based approach. It is controlled by an elected Executive Committee, which represents most major players in the Java industry. The central element of the process is to gather a group of industry experts who have a deep understanding of the technology in question and then have a technical lead work with that group to create a first draft. Consensus on the form and content of the draft is then built using an iterative review process that allows an ever-widening audience to review and comment on the document. While the JCP provides a formal framework with different phases and deliverables, an Expert Group may freely decide on its working style.

The process also demands a reference implementation. Its purpose is to show that the specified API can be implemented and is indeed viable. With the technology compatibility kit a suite of tests, tools, and documentation that is used to test implementations for compliance with the specification has to be provided as well. This enables third parties to build their own, compatible implementations. It is noteworthy that the test must achieve a 100% coverage over all the API’s method signatures.

4.1 Expected Developer Knowledge

While a major goal of the proposed JSR 321 API is to simplify Trusted Computing and make it accessible to a larger group of software developers, there are still certain requirements to be met to make full use of the API. In general, a developer using JSR 321 should be familiar with the cryptographic mechanisms provided in the Java Security Architecture. The concepts of data encryption, decryption, creation of message digests using hash algorithms should be familiar. The algorithms in particular include SHA-1 and RSA used by current TPM implementations. Moreover, a general understanding of Trusted Computing concepts and the functionality provided by a TPM is required.

TPM Lifecycle: Starting with its manufacturing, a TPM goes through a number of different states. A developer must understand this lifecycle, for instance that the TPM is shipped in an unowned state and has to be explicitly assumed ownership, activated and enabled by its owner. When the machine with the TPM reaches its end of life, the TPM may be cleared ensuring that any TPM protected data can no longer be accessed. To avoid data loss, appropriate mechanisms like key backup or migration have to be executed beforehand.

TPM Key Management: A TPM supports a range of different key types, including storage, binding and signature keys. The developer is responsible for building and maintaining a consistent hierarchy. If certain keys are created non-migratable this may rule out any backup of them.

Trusted Storage: Care must be taken when the binding and especially sealing mechanisms are applied to data or user supplied key material. Again, the problem of backup arises, especially considering state changes.

\(^3\)[http://jcp.org/en/jsr/detail?id=321]  
\(^4\)[https://jcp321.dev.java.net]  
\(^5\)[http://www.jcp.org]
Root and Chain of Trust: Ideally a consistent chain of trust would be established by the operating system. However, today’s mainstream platforms fail to do so. Developers need to take extra care to consider the security level represented by the PCR values.

Attestation: A number of different protocols have been proposed to perform attestation to a remote verifier (e.g. Trusted Network Connect). This API supplies the means to create TPM quotes, but as there is no general standard for attestation and public key infrastructures remain limited, we leave the full implementation of a communication protocol to the application developer. Also, the broader problem of how to interpret measurement hashes remains unsolved.

4.2 API Scope Considerations

It is only natural to derive a TC API from the complete TSS specifications. However, JSR 321 is not planned to fully replace the TSS in all its tasks. Instead, and as required by the nature of the JVM as a user process, it builds on and extends the TSS services offered by the operating system environment.

Also, significant differences stem from the requirements of the design processes and the targeted developer audience. In the process that had been employed to conceive the original TSS specification by the TCG, a working group devised a set of APIs to form an industry specification. It not only covers a user-oriented API (the TSPI), but also architectural and internal details clearly intended for developers who plan to build a full TSS. Still, the actual functionality is not elaborated in detail; especially the relationship of different commands on the different layers (TSP, TCS, TDDL, and TPM) is not sufficiently documented. Unfortunately, functional completeness is not required by implementations and no reference implementation is provided for. As a result, to the best knowledge of the authors, no currently available implementation covers the complete specification. Indeed, several highly complex functionalities were specified, but have never successfully been implemented or tested (i.e. DAA) in the years since the TSS standards were released. There are no test suites, compliance tests or test vectors supplied.

Quite the opposite, in the JCP it is not allowed to specify APIs and functionality without implementations and tests. As any Java integration must rely on the TSS-based services of the operating system surrounding the Java Virtual Machine (JVM), this imposes natural restrictions to the functional scope of the JSR 321 API: only those parts of the TSS spec which are available and thoroughly tested in existing TSS implementations can be used to implement JSR 321.

Thus, the major design decision is to focus on the most important core concepts of Trusted Computing. The second main goal is to provide a high usability. At the same time, the API is designed to remain modular enough to be extendable to future developments.

Also, JSR 321 will provide functionality focused on applications and middleware, rather than provide support for the low level BIOS or OS features of the TPM. This restriction matches the field of use of Java and allows to reduce the complexity significantly. JSR 321 will not duplicate elements of the Java Cryptography Architecture, thus melting in with the existing library framework.

Finally, many TSS-specified functions are simply not needed in Java APIs: Management of memory and other resources can and should be hidden from application developers. Object initialization and destruction are natural features of object-oriented languages. Cryptographic primitives like hash functions are already well-supported in the Java Cryptography Extension.

To derive the functional scope of the API, the commented complete list of TCG-specified TSP functions was considered. We filtered the list of functions to the criteria and principles laid out above, so that only features that support core use cases and for which implementations are feasible remained.

5. OUTLINE OF THE API

The unique namespace officially assigned to the JSR 321 API is javax.trustedcomputing. Within this namespace, a number of packages has been specified, each representing a well defined set of functionality. These packages are:

javax.trustedcomputing.tpm This package contains all relevant functionality for connecting to a TPM. A TPM connection is represented by the central TPMContext object that acts as a factory for other objects specified by the API such as the KeyManager or the Sealer. The TPM interface also defined in this package, allows to query general TPM related information like its version and manufacturer. Additionally, it allows to read and extend PCR registers as well as performing a quote operation required for platform attestation.

javax.trustedcomputing.tpm.keys Contrary to the TSS specification, JSR 321 introduces specific interfaces for the individual key types supported by the TPM. This includes interfaces for storage, sealing and binding keys. Compared to having one generic key object, this approach reduces ambiguities in the API and allows to enforce appropriate key usage at the interface level. Using strong key types also relates well to results in formal API design and analysis research.

javax.trustedcomputing.tpm.structures This package holds data structures required for certain TPM operations. They include the PCREvent structure required for operations on the measurement log, PCRInfo used as part of platform attestation and ValidationData as returned by the TPM quote operation.

javax.trustedcomputing.tpm.tools In this package, there are interface definitions for helpers classes to perform TPM operations such as binding, sealing, signing and time stamping.

For error handling, a single TrustedComputingException covers all lower layers. It offers the original TPM/TSS error codes, but also a human readable text representation, which is a great step forward in terms of usability. Despite using only a single exception class, implementations of the API should forward as much error information as possible. For illegal inputs to the JSR 321 API, default Java runtime exceptions are used. Finally, functions offering bit-wise access to status and capability flags are replaced by specific methods that allow access to application relevant flags.

In JSR 321, the KeyManager interface defines methods to create new TPM keys. Upon creation, a secret for key usage and an optional secret for key migration have to be specified. After a key was created, the KeyManager allows to

\*http://www.trustedcomputinggroup.org/developers/trusted_network_connect}
permanently store the key, encrypted with its parent, on non-volatile storage. As required, the KeyManager allows to reload keys into the TPM, provided that the key chain up to the SRK has been established (i.e. the parent key is already loaded into the TPM). Every time a new key is created or loaded from the permanent storage a usage secret has to be provided. This secret is represented by an instance of a dedicated class Secret that is attached to the key object upon construction. Secret also encapsulates and handles details such as string encoding, which are often a source of incompatibility between different TPM-based applications.

The extendable tools package allows using various core concepts of Trusted Computing. As each tool that accesses the TPM is already linked to a TPMContext at creation, there are little or no configuration settings required before using the tool. Each tool provides a small group of methods that offer closed functionality. For example, a Binder allows to bind data under a BindingKey and a Secret, and returns the so encrypted byte array. Usage complexity is minimal as no further parameters need to be configured and the call to unbind encrypted data is completely symmetric. Besides the core set of tools (Signer, Binder, Sealer), implementors of JSR 321 may add further sets of functionality. An example might be a tool which manages TPM ownership, if the Java library is implemented on an OS without tools for doing so.

As the API is still in the Early Draft stage, the reference implementation is not yet complete, but will ultimately be released under an open source licence, providing valuable guidance and a precise mapping to underlying TPM-related command flows. As outlined in section 4 the jTSS and jTSS Wrapper libraries are well suited to support platform-independence as they cover all low-level details regarding hardware access. At the same time, their flow of command remains intelligible for programmers of any other TSS-like library, as the jTSS API is designed to stay very close to the original C-based TSS API and does not modify the logical flow of programs. This eases creation of implementations by the industry or open source community.

6. CONCLUSIONS

In this paper, we have outlined the current status of the specification process for the Java API for Trusted Computing, JSR 321. This API is intended to bring the benefits of Trusted Computing to the platform-independent Java environment. It improves on existing Java libraries for Trusted Computing by providing simplified access to high-level Trusted Computing functions while maintaining conceptual consistency and adhering to the TCG’s specifications. The API has reached the early draft stage of the specification process and the authors invite comments through the public review mechanism.

7. ACKNOWLEDGMENTS

The authors would like to thank the members of the JSR 321 expert group for their dedication bringing this project to fruition.

8. REFERENCES


