TCG Inside? - A Note on TPM Specification Compliance*

Ahmad-Reza Sadeghi, Marcel Selhorst, Christian Stüble,
Christian Wachsmann, and Marcel Winandy

Horst Görtz Institute for IT Security
Ruhr-University Bochum
Universitätsstr. 150, D-44780 Bochum, Germany
{sadeghi,selhorst,wachsmann,winandy}@crypto.rub.de, stueble@acm.org

Abstract

The Trusted Computing Group (TCG) has addressed a new generation of computing platforms employing both supplemental hardware and software with the primary goal to improve the security and the trustworthiness of future IT systems. The core component of the TCG proposal is the Trusted Platform Module (TPM) providing certain cryptographic functions. Many vendors currently equip their platforms with a TPM claiming to be TCG compliant. However, there is no feasible way for application developers and users of TPM-enabled systems to verify this compliance. In practice, manufacturers may exploit the flexibility that the specification itself provides, or they may deviate from it by inappropriate design that might lead to security vulnerabilities. Hence, it is crucial to have an independent means for testing the compliance as well as analyzing the security of different TPMs. In this paper, we aim at making the first steps towards fulfilling this requirement: We have developed a test strategy as well as a prototype test suite for TPM compliance testing. Although our test does not cover the complete TCG specification, our test results show that many TPM implementations do not meet the TCG specification and have bugs. Moreover, we discuss that non-compliance may have crucial impact on security, and point out the corresponding security problems in case of a widespread TPM.

1 Introduction

The Trusted Computing Group (TCG)[1] aims at developing and supporting open industry specifications for trusted computing across multiple platform types. The TCG has released several specifications describing how to add trust to existing computing systems and how to extend security in certain environments, e.g., in network infrastructures. A core component the TCG specifies is the *Trusted Platform Module* (TPM). The current widespread implementation of the TPM is a small tamper-resistant chip capable of securely storing cryptographic keys and other security critical information and providing cryptographic functions. Using the TPM functionality one can attest the initial configuration of the underlying platform and seal and bind data to a specific platform configuration. Further, digital signatures can be used for a public key infrastructure. The TPM acts as root of trust and is the basis for all other specifications of the TCG. Vendors already deploy computer systems, e.g., laptop computers, that are equipped with a TPM chip. Currently used TPMs are, e.g., from Atmel [4, 5, 6, 3], Broadcom [11, 8, 9, 10], Infineon [15], National Semiconductor (now Winbond) [19], and ST-Microelectronics [21, 22, 23].

The TPM has currently two specifications, 1.1b [24] and 1.2 [25]. Since these documents have a certain degree of complexity, one may expect that not all TPM chips operate exactly as specified. Actually, during some of our development projects [1, 2] we encountered problems with certain chip models. In practice, different vendors may implement the TPM differently. They may exploit the flexibility the specification itself provides or they may deviate from it by inappropriate design and implementation[2]. The latter might lead to security weaknesses. Many vendors currently equip their platforms with TPMs claiming to be TCG compliant. However, there is no feasible way for users of these systems to verify this fact. Moreover, many applications may use the functionality provided by the TPM in the near future and rely on the claimed functionality and their correctness.

In this context, it is crucial to have means for testing the compliance as well as analyzing the security of different TPMs. A TPM compliance test is not a

*Accepted for publication at the First ACM Workshop on Scalable Trusted Computing (STC'06)

[1] http://www.trustedcomputinggroup.org

[2] One may argue that vendors can deviate from the TPM specification because the TCG does not enforce its brand name strongly enough. However, the TCG specification itself has a certain degree of flexibility which may be exploited. This is also the case with many other standards. Pushing a brand name or logo "TCG inside" may not in general be sufficient given the complexity of the TPM.

security test, however, it can be useful as an input to a security analysis since it considers the behavior of the TPM. Furthermore, such tests should be developed by an organization independent from TPM vendors in order to provide an unbiased test allowing the comparison of various TPM implementations. Proposing a compliance test suite is not only the responsibility of the standardization body, in this case the TCG. However, it is not trivial to define test cases that lead to meaningful results and keep the test space feasible. Moreover, designing a feasible state machine for testing another state machine (in our case a TPM) have been subject of research in the past see, e.g., [12, 16, 18]. The TPM specification by TCG is complex and there is no thorough analysis publicly available that determines the minimal functionalities a TPM has to provide based on which appropriate security architectures can be build. Designing an efficient compliance test provides insights towards achieving this goal in practice.

TPM compliance test as we consider is not only a test specification on the TPM command level. It includes also methods for users of TCG-enabled platforms to ensure that the main functionalities are available that are important for users' trust (e.g., the availability of TPM identifiers and a valid manufacturer's certificate). Evaluation and certification according to Common Criteria is a common industry practice. This typically includes certain security checks depending on a protection profile[3]. But there are differences to a compliance test. First, a compliance test is not a security test as mentioned before. Second, the protection profile considers those *security* aspects that are defined in it by, e.g., the TPM vendor under a certain evaluation level. Third, it is not obvious that a future TCG proposed test would cover all the aspects we are considering and those which will be important in the future. Application developers and platform vendors may have individual requirements that diverge from a given test suite by the TCG. If a TCG test was obvious then it should have existed before implementing TPMs, and most of the built-in TPMs were compliant, and as our test shows, this is not the case. Moreover, there may be TPM models which conform to a certain TCG specification version but which also include some extra functions of a later version. An independent compliance test would enable developers and users to test whether a TPM model conforms to their specific requirements, which may include the extra functionality. Last but not least, TPM vendors may also use such a compliance test during early prototype testing before the product is going to be evaluated.

In this paper, we introduce the prototype test suite we have developed for TPM compliance tests. Based on our test results, we point out the non-compliance and bugs of many TPM implementations currently available at the market. We present our testing strategy, sample test results, and analysis for different TPM implementations of different manufacturers. Moreover, we discuss how one can construct attacks by exploiting non-compliance and deficiencies of protection mechanisms. We stress that, although our test suite does not cover the whole specification, we believe that it is representative enough to show the importance of a compliance test. Currently, we are developing a comprehensive test suite for the TCG specification.

[3]There does already exist a protection profile for TPMs: TCPATPMPP Version 1.9.7, see `http://www.commoncriteriaportal.org/public/files/ppfiles/`

2 Related Work

In general, compliance testing is the formal approach for validating systems. A commonly used industry practice is a formal verification of the design or testing the system in a simulation environment. The former often models the system's specification as finite state machine [12] and provides a tool that verifies the compliance of the implementation (see [16] and the references in [13]). Testing by simulation uses a fixed set of rules that are checked against violation. The authors of [17] give an example of how to combine the formal verification approach with simulation.

The main problem regarding compliance testing, based on a finite state machine (FSM), is to find the FSM model when not explicitly given by the test subject's specification. In [18], the authors present a systematic method for deriving the FSM model from the specification. The model is then used in a monitor-based approach for compliance test of an On-Chip Bus device. They run the target-of-test in a simulation environment, whereas a bus master controls the bus, i.e., delivers the input to the target, and a bus monitor checks the output of the target, i.e., verifies the bus specification compliance. However, this approach is rather low-level and tailored to bus signals and states. Thus, we believe it cannot be directly applied to a more high-level perspective of a TPM.

Hardware devices can also be treated as a black box providing certain functionality through interfaces defined by a specification. Compliance testing in this way is comprised of (i) generating input data, (ii) sending the input data to the device, and (iii) analyzing the received output data. If input and output match the specification, then the device is said to be compliant. Test suites are used to perform the compliance test. They usually consist of various test scripts running in a defined test environment. For instance, [20] gives an overview of such a compliance testing regarding IPv6 devices, and [7] describes the principles of black box testing in general. Although the latter is related to software testing, they can be partially applied to hardware as well.

3 TPM Preliminaries

In this section we introduce the main TPM properties, which are used throughout this paper. Basically, every TPM provides *volatile* and *non-volatile memory*, which store information about the TPM's state and its capabilities. Moreover, every TPM has different types of *registers*, namely a number of Platform Configuration Registers (PCRs) providing 160-bit storage for *integrity measurements*, and *Data Integrity Registers* (DIR)[4]. Every TPM is equipped with a *cryptographic co-processor* needed for asymmetric encryption (RSA), and hashing (SHA-1, HMAC) and a *hardware random number generator* in order to deliver secure random numbers, e.g., for key generation. For example, the command `TPM_CreateWrapKey` generates an asymmetric key pair (RSA). The input parameters required for the key generation are the key type (e.g., storage, signing, binding), the key size, additional attributes (e.g., migratable, non-migratable) and authorization data (e.g., key password).

Before using a TPM it is necessary to assign an owner to it. In this case the TPM stores the authorization information for the new owner and creates an asymmetric key pair called *Storage Root Key* (SRK), which is used to securely store keys (generated by the TPM) outside the TPM.

In order to authenticate with the TPM, it has different authorization protocols, which are needed to create a shared session secret between the user and the TPM. In order to select the corresponding secret, the TPM returns a session handle upon a successful session request. The most common authorization protocols are the *Object-Independent Authorization Protocol* (OIAP) and the *Object-Specific Authorization Protocol* (OSAP). The OIAP session was designed for efficiency, because it is possible to authorize different entities within one session (object independent). OIAP sessions can live indefinitely until either the TPM or the user terminates the session. In contrast, the OSAP is establishing an authenticated session for a single entity only (e.g., the TPM owner). The OSAP session furthermore enables the secure exchange of new authorization data.

4 Compliance Test Strategy

Our goal is to provide a practical framework for TPM compliance tests in real-world scenarios. Instead of testing the design specification of a TPM model, e.g., by using formal methods, we want to test concrete instances of TPMs in a practical way. This is reasonable since on the one hand, design specifications are not public in most cases, and on the other hand tests should be verifiable by platform vendors and application developers as well as ordinary users. Vendors and application developers may be interested in a detailed TPM compliance statement regarding their product development. In contrast, end-users may only be interested whether they can use a certain TPM as they would like to, e.g., checking if the counter has not expired, a certificate is available, etc. Application developers and users of a TPM-enabled platform should be able to verify its TCG compliance without the need for special, probably expensive, test equipment. Therefore, we assume that the tests will be performed on an ordinary off-the-shelf computer system. We also assume that the tests use a reduced testbed to determine the compliance. Based on these assumptions, our approach uses a real-world execution environment to state the compliance of a concrete TPM by comparing its observed behavior with the expected one.

In order to actually determine the compliance to the TCG specification, we need to find reasonable test cases (*what* shall be tested) and test procedures (*how* shall be tested).

What shall be tested? We test the individual commands of a TPM by providing a certain input and determine the compliance of the output. Each TPM command has an input data structure of fixed length, i.e., 32 bits. In theory, we would have to test all of the 2^{32} different input values for each command. However, this is not practical. To keep the testbed feasible, we categorize the input data structure into parameter fields as defined by the specification. For each TPM command we model its execution as a "state transition" into a return code. Instead of testing all of the 2^{32} bit parameter values, we test a representative of each parameter category, which significantly reduces the testbed. Moreover, we categorize the TPM commands regarding their main functionality and the different states a TPM can take according to the specification.

How shall be tested? We test each TPM command by running a series of test types. The tests are performed on an ordinary PC system. We operate on two levels: At the *application level*, we perform high-level tests by testing the TPM functionality from the view (and interfaces) of an application. At the *protocol level*, we perform low-level tests by directly operating on the command data structures used to communicate with the TPM. We operate on these two levels in order to cover most operational conditions. The high-level tests use the TPM functionality as real applications would do. The low-level tests give us the opportunity to test specific conditions which probably would arise very rarely in applications deployed in real life environments.

At each level we run the following test types. We run *functional tests* to check the intended behavior, i.e., if the outputs for certain given inputs are con-

[4] TCG 1.2 has specified a dedicated non-volatile index to replace the previous DIR. Furthermore, non-volatile memory is directly supported in TCG 1.2.

form to the functional specification. *Integrity tests* are used to check if the TPM performs resistantly in case of failures and when operated in a not specified manner, e.g., by providing incorrect input parameters. Lastly, we use *stress tests* to determine the stability of the TPM under abnormal or extreme condition beyond the common operational capacity. An example is to call a certain command many times one after another. To measure the security quality, we use the results of functional, integrity, and stress tests and analyze them according to their security implications. Moreover, we make time measurements in all test types and use them to state the performance quality. However, security and performance qualities only belong partly to the TPM specification compliance, i.e., when explicitly stated in the specification.

4.1 Defining Test Cases

A detailed analysis of the TPM specification results in a dependency graph that defines the input/output and calling dependencies of all TPM commands. Figure 1 shows a subset of this graph we use for our prototype test suite from an application-level point of view. The dependency graph can then be transformed into an activity graph defining the required order of calling the individual TPM commands. Figure 2 shows the TPM commands (in the required order) where each path from start to end represents one application level node of Figure 1. This activity graph is related to the tests we have performed and it represents a part of the complete test suite needed for the TCG specification.

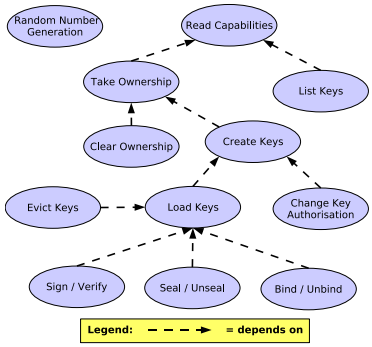


Figure 1: Dependency graph of functional tests.

Having the graphs we can derive a general abstract finite state machine of the TPM. A state will contain several combination of commands. For determining the specification compliance it will only be necessary to test the TPM in all states and all state transitions instead of testing all combination of commands and parameter values. This will reduce the test space and keep it feasible. In the following, we describe a way to define the test cases of each command.

Every TPM command has a specified set of input parameters. There are different classes of input: valid, invalid, illegal and unsupported input parameters. We examined these classes for all command parameters and determined the corresponding return codes. In order to test if a TPM acts as specified, we choose a value for each type of input and examine whether it answers with a valid return code.

Valid input parameters should result in the TPM answering with a valid response and the `TPM_SUCCESS` return code indicating that no error has occurred during the processing of the command. *Illegal* parameters have been modified such that the expected data structure of the parameter does not match or contains values that have not been specified by the TCG. *Invalid* parameters contain wrong values (e.g., *paramSize* parameter is too short/long). *Unsupported* parameters contain valid values but are not supported in the context of the command (e.g., *keySize* field set to a value < 2048). For the last three classes of input errors, a TPM must respond with a return code different from `TPM_SUCCESS`.

The TPM specification [25] specifies a number of return codes for only a few error conditions that might occur during the processing of a command and its parameters. However, for the majority of possible erroneous inputs, the specification only states that the TPM should return a "legal return code" from a list given in [25, part 2, ch. 16]. Indeed, this list gives a short description for every return code but does not always allow the distinct assignment to a specific error condition. Thus different TPMs might return different return codes for the same input error but are both compliant to the TCG Specification. Therefore we determine the set of "legal" return codes for all classes of input errors for every parameter and combinations of them for each command.

In order to determine the set of valid return codes for a specific command, we first identified the states of the TPM that support the execution of the respective command. If the TPM is disabled and thus not allowed to execute the desired command it must return `TPM_DISABLED`. For the case that a TPM is deactivated or unowned but the execution of a command requires an active or owned TPM, the specification does not define a return code directly. Thus we had to determine all applicable return codes for this scenario.

After determining TPM states that allow the command to be executed, we test valid input parameters which should result in a successful execution of the command. This means that the TPM should return the `TPM_SUCCESS` return code. Note that still one has to examine that the TPM has executed the command properly. Then we test the three remaining variants of illegal, invalid and unsupported input parameters. The order of the processing of a command's parameters is often not exactly specified, so it is not always possible to assume which parameter is checked first

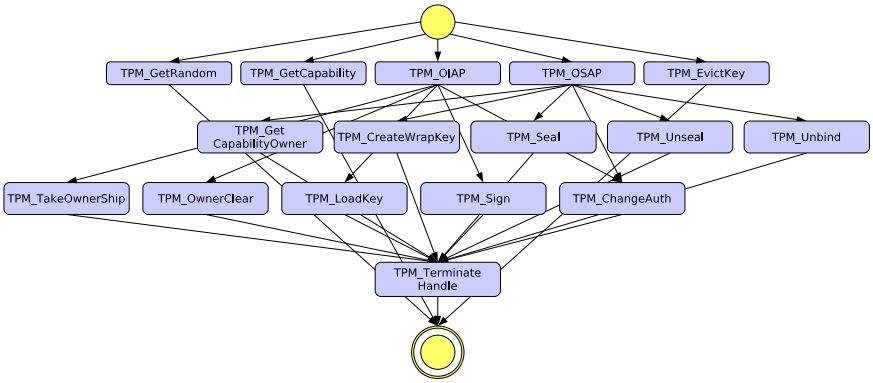


Figure 2: Activity graph of TPM commands on protocol level.

and therefore which return code can be expected. Thus, it should be sufficient to test if the TPM answers with a return code from the union of the sets of legal return codes. Note that if the result is not as expected, this does not strictly mean that the TPM is not compliant, but that the TPM handles the incoming data in a different order.

At this point, one may interpret the TCG specification such that the TPM vendor is free to define what error code is reasonable in each case. However, in our opinion it is important to take the return codes into account. First, if all error codes of a TPM were vendor-specific, it would have been much easier to define only one generic error code instead of different ones in the specification. Second, and more importantly, security-sensitive applications may rely on the functionality of the TPM and they may react differently to different error codes. Thus, if a TPM returns a different return code as expected, the application may behave in an unforeseen manner, possibly resulting in a security risk.

Finally, we give a short hint how to construct a complete finite state machine of the TPM. Starting from an initial state, one can execute a TPM command. Depending on the output, the TPM will be in a state ready for calling another command (e.g., after a key is loaded, the key can then be used for a different command) or it will result in an error code, which can be interpreted as a state of its own. The categorization of command parameters can be used to derive the corresponding state transitions into "error states", while the dependency and activity graphs can be used to derive the state transitions from one command to another. These are the first steps towards a generic state machine which may be easily adopted to future specification versions.

5 Test Case Example

We demonstrate through an example our procedure of determining valid TPM return codes using the command `TPM_CreateWrapKey`.

The TCG defines eight states for a TPM which depend upon whether the TPM has an owner installed or not, is enabled or not, and is active or not. Figure 3 gives an overview about these states. A disabled or inactive TPM only provides a very limited set of commands (see [25, part 2, ch. 17]). The only difference between disabled and inactive is that a disabled TPM cannot execute the `TPM_TakeOwnership` command, which installs a new TPM owner.

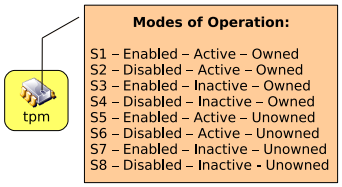
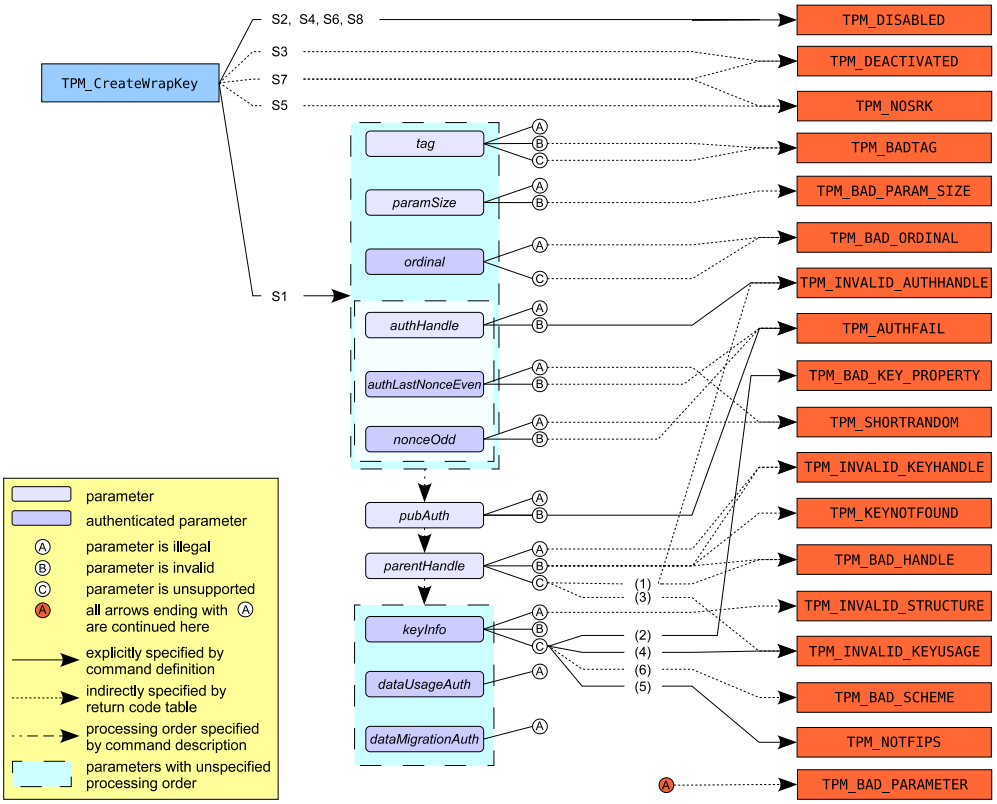


Figure 3: TPM modes of operation.

To find out the TPM states in which `TPM_CreateWrapKey` can be executed we used the table provided in [25, part 2, ch. 17]. The command is not executable if the TPM is deactivated, disabled or unowned. Considering the modes of operation for a TPM (Figure 3, [25, part 1, ch. 9.4]) the command can only be executed successfully if the TPM is in state S1. The TPM must return `TPM_DISABLED` when it is disabled and receives this command. This is the case for the states S2, S4, S6 and S8. In state S3 the TPM is inactive. There is no directly specified return code for this case. According to its description, `TPM_DEACTIVATED` is considered to be the only return code that is valid for this situation. Using the same procedure leads to the result that the TPM should answer with `TPM_NOSRK` when it is unowned. State S7 means that the TPM is inactive and unowned. Since the specification does not mention whether inactivity or ownership is checked first, the valid return codes for state S7 are the union of sets of return codes for both cases.



(1) to (6): Depending on the value of a data element within a parameter the resulting return code can vary. E.g., (1) occurs if the session type for *parentHandle* is not OSAP, while (3) occurs if *parentHandle.keyUsage* is unequal to `TPM_KEY_STORAGE`.

Figure 4: Valid return codes for all possible errors concerning the `TPM_CreateWrapKey` command.

Every TPM command consists at least of the *tag*, *paramSize* and *ordinal* parameter. According to [25, part 2, ch. 16], the TPM should return, e.g., `TPM_BADTAG` if "the *tag* value is invalid" and `TPM_BAD_PARAM_SIZE` if "the *paramSize* argument to the command has the incorrect value". Other return codes are also reasonable, e.g., `TPM_BAD_PARAMETER` since "one or more parameter is bad". We determined all valid return codes as given in Figure 4.

To integrate the new key into the existing key hierarchy, the authorization data for the parent key that should protect it has to be provided. `TPM_CreateWrapKey` needs authorization which is provided through the OSAP protocol. This has two consequences: First, the command `TPM_OSAP`, which creates an OSAP session, and `TPM_TerminateHandle`, which terminates a session, have to function correctly and have been performed before. Second, the TPM specification demands that the corresponding authorization session has to be verified before `TPM_CreateWrapKey` is allowed to be executed. This verification includes the validation of the parameter *authHandle*, which should uniquely identify the authorization session, and the calculation of an HMAC that includes the SHA-1 hash of certain other parameters and uses the authorization data for the parent key as secret. The result of this HMAC computation is sent as *pubAuth* parameter to the TPM. Modifications of these parameters without adaption of *pubAuth* must result in the `TPM_AUTHFAIL` return code. This is therefore subsumed by the case of an invalid *pubAuth* parameter. In the case of an invalid *authHandle* parameter, the TPM must return `TPM_INVALID_AUTHHANDLE`.

However, if authorization has been successful, all authorized parameters may still contain illegal, invalid or unsupported values. Unsupported values are usually discussed in the command's description and can be adopted. The return codes corresponding to illegal or invalid parameters have not been specified directly in most cases and must be determined for each parameter from the table in [25, part 2, ch. 16].

The graph shown in Figure 4 is the result of exercising this procedure to all parameters of the `TPM_CreateWrapKey` command. It is not necessary to test unsupported parameters for *authHandle* and *pubAuth* since all values are possible and allowed. The same holds for nonces which are random bytes used to provide security against replay attacks. The parameters *dataUsageAuth* and *dataMigrationAuth*

contain encrypted authentication data for the use and migration of the key to be created. Modifications of these parameters will not be noticed by the TPM but by the entity that started the command after it received the response message from the TPM. Therefore we do not have to consider invalid and unsupported values for these parameters. A wrong value for the *ordinal* parameter should result in the TPM executing the command associated with the submitted ordinal. This means that we would have to examine the behavior of the command the modified ordinal points to. An unsupported *ordinal* might be a vendor specific command.

6 Testsuite Implementation

Our test suite implementation is a restricted instantiation of the test strategy defined in Section 4. All tests performed within the suite have been selected by analyzing the functionality offered by the selected software needed for the implementation (see Section 6.1). Missing tests that are needed for a whole TPM compliance test suite are work in progress. The first implementation shall be seen as a prototype to give first results about the need for compliance tests.

A test subject does always consist of only one TPM command (target of test). After identifying and defining the test subject, we derived the corresponding test cases as described in section 4.1. The test cases define the input data as well as the expected output we use for the implementation of the corresponding functional, integrity, and stress tests.

For each test subject we implemented a separate module. Depending on the test subject, the test case and the mode of operation, we describe the expected test results and take into account that the results may differ depending on the state the TPM is currently in. After performing a test, it should be assured, that the TPM is in the same state as it was before. In particular, this means: we unload all loaded keys, free all used resources, terminate all key handles, and switch back eventually changed authorization data to the original one. In case the effects of a test cannot be undone (e.g., extension of a PCR), we have to make sure to not interfere with subsequent tests (e.g., do not clear ownership if a subset of keys exist). This will allow for comparability.

6.1 TPM Library libtpm

Our test suite prototype reuses as much as possible of existing open source TPM software. Currently, only some open source TPM applications exist, each following different approaches. The *libtpm*[5] was the first open source implementation of the most common features of the TCG specification 1.1b. Since *libtpm* works directly with the TPM on the protocol level, we have direct access to the input and output buffer and therefore have a single point of interaction, since all communication with the TPM is done in one function.[6]

Figure 5 shows the overall framework architecture and the interaction between our test suite implementation and the *libtpm*. In order to manage the test subjects and their dependencies, we have developed a test suite management application. The management software selects the test subjects and the test cases. Afterwards, the *libtpm* is notified about the selected test subject and test cases. We use the *tpm utilities* provided by the *libtpm* to perform the desired tests. Those utilities themselves use the *libtpm* to build the command structure as defined by the specification and to communicate with the TPM by using a transmit function. Instead of allowing the libtpm to directly communicate with the TPM chip, our test framework modifies the buffer as specified by the test case, sends it to the TPM, receives the response from the chip and afterwards analyzes the results. The obtained information will then be collected and returned to the test suite management application.

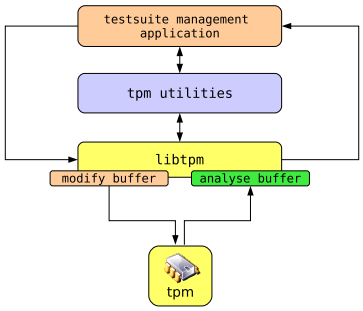


Figure 5: Test suite architecture

Although the *libtpm* does not fully implement the whole functionality provided by the TCG specification 1.1b, it is nevertheless useful as a prototype implementation. In the following we describe which tests can be done using *libtpm*.

6.2 Application-Level Tests

The first part of tests will cover the functionality of the TPM. Due to the dependencies between different TPM commands, a functionality test on the application level will cover more than one TPM command. Figure 1 shows the application level functional tests we perform and their dependencies. Note that this only gives an abstract view of the dependency graph considering the main relations. However, in some cases the dependencies may be more subtle and hence

[5] We use *libtpm* version 3.0.3 under GNU Linux.

[6] Meanwhile, IBM has developed the open source TSS implementation *"TrouSerS"*, which we will use in the future.

the graph may include more details (e.g., commands can change states and capabilities of a TPM during runtime.)

6.2.1 Functional Tests

The required functional tests at the application level are briefly described below.

TPM Capabilities: Read out all possible TPM capabilities. This includes gathering information about the current state (see Figure 3) of the TPM. Additionally, one can check whether the TPM has an *Endorsement Key* installed and - depending on the TPM vendor - whether a valid TPM credential (e.g., a vendor certificate) exists.

Hardware random number generator: Request the maximum number of available random bytes and measure the speed of processing the request.

Take Ownership: Take Ownership of the TPM in order to enable further TPM commands.

Create keys: Create all possible subkeys.

Load keys: Load all created subkeys. This will test if the TPM is able to decrypt the given key with its private part of the SRK. Afterwards, the TPM will return a key handle which is needed to perform key operations with this key (depending on the key properties).

List keys: Check if the TPM has loaded any keys and returns the depending key handles.

Change Key Authorization: Change the authorization data for a given key. This key can either be the Storage Root Key (SRK) or any subkey created within the TPM.

Key operations: Use keys loaded into the TPM to perform key operations, e.g., sealing, unsealing, or signing.

Evict keys: After successful usage of keys, evict[7] loaded keys out of the TPM in order to free TPM resources.

Clear Ownership: Clear the ownership of the TPM.

6.2.2 Integrity Tests

Currently, we have not implemented integrity tests at the application level. This is due to the fact that the application-level software, i.e., *libtpm*, already intercepts and handles incorrect input parameters at this level. Thus, no TPM command would really be tested for correct behavior this way. Instead, we perform integrity tests at the protocol level.

[7]By evicting a key, the key will be deleted from TPM internal memory. All associated key handles are invalidated and the key slot resource is available again. The key itself is still available outside the TPM.

6.2.3 Stress Tests

Key related stress test: This test consists of all actions that are related to keys, e.g., as the following:

- Create all possible subkeys regarding the combination of key type, key size, and key attributes.
- Load all possible subkeys.
- Load more keys into TPM than key slots available.
- Evict all keys.
- Evict a key with an invalid key handle.
- Load two keys, evict the first one, and load a third key and compare the received key handle to the first freed to check whether the TPM assigns new key handles or if it uses recurrent and thus predictable handles.
- Test the quality of the key handles regarding the security aspects defined in Section 8.4.
- Try all possible combinations of keys and key operations, e.g., use a storage key to perform binding.
- Perform long-term key creation and loading. In detail, try to create n keys of any type and try to load and evict these keys alternately.

Access/authorization related stress test: This test concerns the TPM behavior when stressing the authorization/access mechanisms.

By performing a dictionary attack[8] on the TPM, the behavior of the TPM is analyzed. If changes in behavior are observed because of countermeasures we will perform a valid authentication (e.g., creating or loading a key) and afterwards continue the dictionary attack.

Hardware random number generator: By collecting a huge number of random numbers, we can perform additional tests on the received random numbers in order to qualify the randomness and security of the TPM's random number generator. Moreover, we can measure the delay between processing two random number requests (delays may play a role in profiling TPM chip models, see section 8.3).

6.3 Protocol-Level Tests

Tests on the protocol level mainly focus on performing integrity and stress tests on specified test subjects.

[8]This is a technique for defeating authentication by trying to determine a passphrase. In contrast to a brute force attack, only possibilities are considered which are most likely to succeed, typically derived from a list of words in a dictionary.

6.3.1 Functional Tests

Currently, we do not have implemented functional test at the protocol level. They are implicitly done through functional tests of the application level, since tests on the higher level can only be successful if the underlying commands have been executed correctly and if the TPM behaves in the correct manner. Moreover, most commands on the protocol level require certain input parameters (e.g., key or session handles) which depend on the output of prior commands. Therefore, either test vectors have to be defined or the tests have to be done implicitly on a higher level.

6.3.2 Integrity Tests

Integrity tests at the protocol level are performed on a TPM command by manipulating the command parameter structure. Manipulating either one of those parameters has to lead to an error.[9]. In our test suite implementation we cover integrity tests on the protocol level for all commands that are executable via the *libtpm* utilities. We therefore call the selected test subject as often as needed to be able to modify every field entry once. Note that manipulating the authorization field might result in a locked TPM state.

6.3.3 Stress Tests

Session related stress tests: The TPM specification defines several protocols on how to authenticate with the TPM and how to create shared session secrets. For each session, which is agreed upon via the session protocols, the TPM has to have internal memory to store the session handle, the session shared secret, and all other properties assigned to this session. Due to its limited amount of internal memory, the TPM has a limited number of concurrent sessions. We perform the following stress tests:

- Open as many concurrent OIAP / OSAP sessions possible with the TPM.
- Close all sessions.
- Close a non-existing session.
- Open two sessions, close the first one, and open a third session. Compare the received session handle to the first freed and check whether the TPM assigns new handles or uses recurrent and thus predictable handles.
- Open an OSAP session with all possible entity types.
- Open alternating OIAP and OSAP sessions to see if the session handles come from the same session pool.
- Analyze the quality of the session handles.

[9]For example, modifying the `TPM_AUTHDATA`-field should lead to *Error 1 - Authentication failed*

7 Compliance Test Results

Based on the test strategy as described in Section 4, we used our test suite implementation to test a sample of TPM chip models from different vendors. The analysis of the test results reveals that, as expected, the different TPM models differ in their implementation. Moreover, several TPMs show non-compliant behavior with respect to the TCG specification. Table 1 gives a brief overview of the tested TPM models and their results. In the following, we describe the test results for each test subject separately. For a more detailed list of the conducted tests and updates about potentially future tests see our website on [14].

TPM Vendor	Chip	Firmware	Compliant
Infineon	1.1b	1.1.1.6	no
Atmel	1.1b	1.1.0.6	no
Winbond	1.1b	1.1.4.22	yes
ST Microelectr.	1.2	1.2.2.6	no
Infineon	1.2	1.2.1.0	yes

Table 1: Tested TPM models. (Compliance statement regarding the tests we performed.)

Atmel AT97SC3201 The Atmel AT97SC3201 is claimed to be compliant with [24]. As specified, it has 16 PCRs and two DIRs. It provides 10 key slots and does not precalculate keys.[10] It can handle two concurrent OIAP and OSAP authorization sessions as specified in [24]. The AT97SC3201 supports all cryptographic algorithms required by the TPM 1.1b specification (RSA, SHA1, HMAC). Our prototype test suite reveals the TPM not to be compliant with [24] since it has the following bugs: The TPM changes the *authData* parameter inside the SRK-blob after calling `TPM_TakeOwnership` for taking ownership.[11] Another bug our prototype test suite revealed is that this TPM returns `TPM_SUCCESS` instead of `TPM_AUTHFAIL` when modifying the *authHandle* parameter, which uniquely identifies the authorization session to be used with that command. This pertains `TPM_LoadKey`, `TPM_LoadKey2`, `TPM_Sign`, `TPM_Unbind`, `TPM_Seal` and `TPM_Unseal`.

This TPM uses randomized key handles but authorization session handles that always start with `0x00000000`, which is increased for each new handle. Handles are immediately re-assigned to new sessions when they have been freed.

Infineon SLD 9630 (1.1b) The SLD 9630 is claimed to be compliant with [24]. It provides 16 PCRs, 16 DIRs and four key slots. It is able to precalculate keys and thus responds faster to

[10]The precalculation of keys is no requirement of the TPM specification and only mentioned for comparison with other TPMs.

[11]For details refer to trousers-bugfix in `trousers/src/tspi/spi_tpm.c`.

commands that create a new key. This TPM is able to manage 20 concurrent OSAP and OIAP authorization sessions. According to our test results, the SLD 9630 is not compliant with [24] since it does not differ between the TPM entity types `TPM_ET_KEYHANDLE` and `TPM_ET_SRK`. Therefore, an authentication error occurs for keys of the type `0x0004`. This error holds for `TPM_CreateWrapKey`, `TPM_Seal`, `TPM_UnBind`, `TPM_ChangeAuth`. It is fixable by a TPM firmware update or a patch to the *libtpm* library. The SLD 9630 partly increases the key and session handles in minor or major steps and immediately re-assigns freed session handles.

Winbond (1.1b) The Winbond TPM, which has been developed and distributed by National Semiconductor, claims to be compliant with [24]. It provides 16 PCRs, two DIRs and 9 key slots. It can precalculate keys and is able to manage eight concurrent OSAP and OIAP sessions. This TPM is the only 1.1b TPM that is, according to our prototype test suite, compliant with [24]. Despite the 1.1b specification does not require countermeasures against dictionary attacks, this TPM provides a mechanism that refuses to execute any further TPM command after 10 invalid authorization attempts. After some seconds and a valid authorization, the TPM allows only one command in a time frame of 5 seconds. After 10-30 minutes, the TPM works normally again. This TPM chooses key handles by starting at `0x00000001`, which is steadily increased. After a TPM restart, the key handle still has its last value, which is set back to 0 after some hours of powered down state. Session handles always start at `0x00000000` and are increased for each new session. Freed session handles are immediately re-assigned.

Infineon SLB 9635 (1.2) The SLB 9635 is claimed to be compliant with [25]. It provides 24 PCRs, one DIR and 10 key slots. It is able to precalculate keys and can manage 20 concurrent authorization sessions. According to our prototype test suite it is compliant with the TPM 1.2 specification. Key and session handles are chosen according to [25, part 2]. In case of a dictionary attack, the TPM denies the execution of commands after 10 invalid authorization attempts. If further authorizations fail it will increase the time frame in which a valid command can be executed.

STM ST 19 WP 18 (1.2) The STM TPM is claimed to be compliant with [25]. It provides 24 PCRs, one DIR and 9 key slots. Is is not able to precalculate keys and can manage 12 concurrent OIAP and 6 OSAP sessions. According to our prototype test suite, this TPM is not compliant with [25].

According to [25, part 2], the three LSBs of a handle must contain collision resistance values that must be generated randomly, which is both not considered by this TPM. Key handles are returned after `TPM_LoadKey` and `TPM_LoadKey2`. Session handles are returned after `TPM_OSAP` and `TPM_OIAP`.

As specified in [25], a TPM must support an increment rate of once every 5 seconds for monotonic counters, which is violated by this TPM. This fact can be verified with `TPM_GetCapability` for capability area `CAP_PROP_MIN_COUNTER`. Moreover, when using the command `TPM_ReadCounter`, which returns the current value of the referenced counter, the TPM must return `TPM_BAD_COUNTER` if an invalid *countId* is requested. This parameter uniquely identifies an active monotonic counter managed by the TPM. However, the TPM returns `TPM_BAD_PARAMETER`. Additionally, this TPM returns `TPM_SUCCESS` instead of an appropriate error code when invalidating the send-buffer by modifying the *paramSize* parameter of the following TPM commands: `TPM_OIAP`, `TPM_GetCapability`, `TPM_PCRRead`, and `TPM_EvictKey`.

In case of a dictionary attack, the TPM starts increasing its answer time. We used the `TPM_CreateWrapKey` command, which we called with a false SRK authentication. The first 15 responses are received immediately, then every two commands the return time is increased by one second. After about 40 faulty authentication attempts, the TPM response time is at about 10 seconds. Then we called `TPM_LoadKey2` using valid authentication data. The dictionary attack countermeasure resets itself after one successful authentication.

This TPM generates session handles by starting with zero and incrementing this value. If a session handle is freed, it will be immediately re-assigned to the next session. The same holds for key handles, which start at `0x00000100`.

Analyzing the results of our prototype test suite, we found out that – upon the tests we performed – three out of five TPMs are not TCG compliant. For example, some TPM implementations return `TPM_SUCCESS` on manipulated commands, although invalid parameters should lead to an error. Moreover, some of them have security related bugs, of which we will describe the impact in the following section.

8 Security Implications

A compliance deviation from the specification and inappropriate implementation of TPM chips may have crucial security impact. We were able to exploit this and point out the severity in case of one widespread TPM, namely ST Microelectronics TPM 1.2 [21, 22, 23]. In the following, we discuss the security aspects of the corresponding test results.

8.1 Endorsement Key

The *Endorsement Key (EK)* is a 2048-bit RSA key pair representing the platform identity. It is cryptographically unique, bound to a certain TPM, and only available for two operations: establishing the TPM Owner and establishing *Attestation Identity Keys (AIK)*. Due to this, both the public and the private key part of the EK have privacy and security concerns. Usually, the EK will be generated by the TPM vendor before the platform is shipped to end users or resellers. This is necessary because the entity generating the EK (in this case, the TPM vendor) does also create a credential attesting the validity of the TPM and the EK. The credential (e.g., a certificate) contains the public part of the Endorsement Key and asserts that the holder of the private key is a TPM conforming to TCG specifications.

If the TPM is shipped without an EK, it seems to be no problem at the first view, since the EK can be created within the TPM by calling the according TPM command `TPM_CreateEndorsementKeyPair`. However, the corresponding certificate will be missing, since this can only be generated by the vendor and not by the TPM owner. Note that a missing EK is not a violation of the TCG specification, but in our opinion the requirements of users of TPM-enabled systems should also be considered. Thus, the TPM cannot be used in trust models and application scenarios where a certificate signed by the vendor or a trusted entity is required. This also pertains TPMs shipped with an EK but without a corresponding vendor certificate.

8.2 Dictionary Attack

The TCG specification version 1.1 does not suggest any requirements for dictionary attack mitigation, but version 1.2 adds the requirement that the TPM vendor provides some measures against dictionary attacks, where the internal mechanism is vendor specific. The protection mechanism against dictionary attacks – as suggested in the TCG specification 1.2 – is to delay the response of the TPM to any request for a time out period, if a threshold of failed authorization exceeds. This time out period doubles each time the threshold exceeds again. The owner of a TPM is able to resolve (reset) this locked state by performing a `TPM_ResetLockValue` command. [24, ch. 8.1].

We have tested the implementation of such protection mechanisms by performing a false authentication at the TPM and analyzing the resulting behavior. In detail, we repeatedly send invalid authorized commands to the TPM until the TPM starts increasing its response time. When we recognize any countermeasure against dictionary attacks, we perform a different command with a valid authentication and subsequently continue the dictionary attack in order to check whether the behavior of the TPM has changed.

By using this method, we were able to identify that the protection mechanism of one TPM implementation resets its locked state immediately after one successful authorization even without executing the `TPM_ResetLockValue` command. This allows us to continue the dictionary attack, which – from a security point of view – annihilates the protection mechanism against dictionary attacks.

8.3 Profiling

Every response includes a return code, which can be used to analyze how the TPM internally processes the incoming data. Sending a correct command with valid parameters has to lead to the return code `TPM_SUCCESS`. If one or more of the parameters are modified, one can figure out whether the TPM has recognized the manipulation and – depending on the return code – which parameter is checked first by the TPM. Even the time needed to process the request might be interesting, but we did not explore this in our test suite.

Comparing the results of different TPM models and comparing the results of different test cases, one may be able to obtain a detailed profile of the TPM behavior. We figured out that some TPM implementations have bugs inside their parameter handling. This means they do not check certain parameters inside the command buffer and therefore return `TPM_SUCCESS` instead of an error when the buffer has been manipulated. Moreover, some TPMs return error codes that are completely unrelated to the actual command. This may be legal in terms of the specification, but it may also be useful as an entry point for an attack.

8.4 Handles

The TPM 1.2 specification requires certain quality standards for the randomness of handles. Those include the key handles returned during a `TPM_LoadKey` as well as any session handle created with `TPM_OIAP` or `TPM_OSAP`. There may be security problems relying in the missing freshness of the received handle. An application may still have the reference to a handle while the handle refers to a different key or session the application assumes. Depending on the application, this may probably result in unanticipated behavior, which may lead to a violation of the user's security policy. For instance, if data is sealed with a different key as intended, a subject being capable of unsealing may unintentionally gain access to the data. An attacker can easily figure out (due to the profiling property) how the TPM generates and associates session and key handles and can therefore perform an attack using this information if he knows the application behavior and, e.g., that a freed key

handle will be immediately assigned to the next key loaded into the TPM.

Since the TCG specifications do not consider performance issues, an application executing a TPM command has no estimation about the processing time. Therefore, the attack can be extended to a man-in-the-middle attack, where, e.g., a malicious device driver may intercept the communication with the TPM, analyze the command, and try to exchange the key to be used. If the attacker is aware of the corresponding parent key password (e.g., due to a dictionary attack), it is possible to evict a key K_1, create an own key K_2, and load that key into the TPM. There are two different ways of getting the application to use the second key. First, if the TPM suffers from the bug of reusing key handles, the application automatically refers to K_2. The attacker can send the unmodified command buffer to the TPM. Second, if the TPM does not have this bug, the attacker can modify the buffer before sending it so that it contains a handle referring to K_2. Since key handles are not part of the HMAC computation of the authorization data of a command, this modification will not be detected at first. However, in both cases the TPM verifies the authorization data of the transmitted command, which includes the authorization data of the referenced key. Since the HMAC of the transmitted command was computed by including the authorization data of K_1, the verification, and thus the attack, only succeeds if K_2 has the same authorization data as K_1.

9 Conclusion and Future Work

Many vendors currently equip their platforms with a TPM claiming to be TCG compliant. However, there is no feasible way for users of these systems to verify this fact. To face this problem we have developed a prototype test suite for TPM compliance tests according to the TCG specification. We have presented our testing strategy describing what shall be tested and how, and we gave an overview of our current test suite implementation. First results of our test sample show the non-compliance and bugs of several TPM implementations. Furthermore, we have pointed out security problems that can arise as consequence of non-compliance. Although our tests cover only a part of the TCG specification we believe that the results are reasonably representative. We are currently working on a complete test suite specification for TPM implementations. Since our test case model is based on a graph description language, future work may also include the development of a compiler that can directly transform the test case model into test procedure scripts.

References

[1] TrustedGRUB. http://www.prosec.rub.de/trusted_grub.html.

[2] Linux Device Driver for Infineon TPMs. http://www.prosec.rub.de/tpm/index.html, 2006.

[3] Atmel. AT97SC3201 — The Atmel Trusted Platform Module. http://www.atmel.com/dyn/resources/prod_documents/doc5010.pdf, August 2004.

[4] Atmel. AT97SC3203 Advanced Information Summary. http://www.atmel.com/dyn/resources/prod_documents/5116s.pdf, July 2005.

[5] Atmel. AT97SC3203S for SMBus Protocol Summary. http://www.atmel.com/dyn/resources/prod_documents/5132s.pdf, August 2005.

[6] Atmel. Trusted Platform Module AT97SC3201 Summary. http://www.atmel.com/dyn/resources/prod_documents/2015s.pdf, June 2005.

[7] B. Beizer. *Black Box Testing*. John Wiley & Sons, 1995.

[8] Broadcom. Broadcom Revolutionizes LAN Communications by Introducing the World's First PCI Express Gigabit Ethernet Controllers for Server, Desktop and Mobile PCs. http://www.broadcom.com/press/release.php?id=461159, October 2003.

[9] Broadcom. BCM5752 Product Brief. http://www.broadcom.com/collateral/pb/5752-PB00-R.pdf, 2005.

[10] Broadcom. BCM5752M Product Brief. http://www.broadcom.com/collateral/pb/5752M-PB00-R.pdf, 2005.

[11] Broadcom. Broadcom Controllers Integrate TPM 1.2 enabling OEMs to Offer Hardware-Based Security as a Standard Feature on all PCs. http://www.broadcom.com/press/release.php?id=700509, 2005.

[12] T. S. Chow. Test design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978.

[13] K. El-Fakih, N. Yevtushenko, and G. v. Bochmann. FSM-based incremental conformance testing methods. *IEEE Transactions on Software Engineering*, 30(7):425–436, 2004.

[14] Horst Görtz Institute for IT Security, Ruhr-University Bochum, Applied Data Security Group. Technical Report. http://www.prosec.rub.de/tpmcompliance.html, May 2006.

[15] Infineon Technologies AG. Product Brief — TPM 1.2 Hardware. http://www.infineon.com/tpm, May 2005.

[16] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1123, 1996.

[17] L. Li, S. A. Szygenda, and M. A. Thornton. Combining simulation and formal verification for integrated circuit design validation. In *Proceedings of the 9th World Multi-Conference on Systemics, Cybernetics and Informatics (WMSCI)*, pages 92–97, 2005.

[18] H.-M. Lin, C.-C. Yen, C.-H. Shih, and J.-Y. Jou. On Compliance-Test of On-Chip Bus for SOC. In *Proceedings of the 2004 Asia and South Pacific Design Automation Conference (ASP-DAC'04)*. IEEE Press, 2004.

[19] National Semiconductor. Product Brief: PC8374T SafeKeeper Desktop TrustedI/O. http://www.winbond-usa.com/products/winbond_products/pdfs/APC/PC8374T.pdf, August 2004.

[20] J. Ruiz, A. Vallejo, and J. Abella. IPv6 conformance and interoperability testing. In *Proceedings of the 10th IEEE Symposium on Computers and Communications (ISCC 2005)*. IEEE Press, 2005.

[21] Data Brief: ST19WP18-TPM-A Trusted Platform Module. http://www.st.com/stonline/products/literature/bd/10926.pdf, 2004.

[22] Data Brief: ST19WP18-TPM-B Trusted Platform Module. http://www.st.com/stonline/products/literature/bd/10927.pdf, 2004.

[23] Data Brief: ST19WP18-TPM-C Trusted Platform Module. http://www.st.com/stonline/products/literature/bd/10928.pdf, 2004.

[24] Trusted Computing Group (TCG). TCPA Main Specification, Version 1.1b. https://www.trustedcomputinggroup.org/specs/TPM/TCPA_Main_TCG_Architecture_v1_1b.pdf, February 2002.

[25] Trusted Computing Group (TCG). TPM Main Specification, Version 1.2 Revision 94. https://www.trustedcomputinggroup.org/specs/TPM/, March 2006.