The Diversity of TPMs and its Effects on Development

A Case Study of Integrating the TPM into OpenSolaris

Anna Shubina*
Dartmouth College
Hanover, New Hampshire
shubina@dartmouth.edu

Sergey Bratus
Dartmouth College
Hanover, New Hampshire
bratus@dartmouth.edu

Wyllys Ingersoll
Oracle, Inc.
wyllys.ingersoll@oracle.com

Sean W. Smith
Dartmouth College
Hanover, New Hampshire
sean.w.smith@dartmouth.edu

ABSTRACT

Broad adoption of secure programming primitives such as the TPM can be hurt by programmer confusion regarding the nature and representation of failures when using a primitive. Conversely, a clear understanding of the primitive's failure modes is essential for both debugging and reducing the attack surface in the mechanisms built on it. In particular, differences in error processing and reporting logic significantly detract from such understanding.

We present a case study of diversity in TPM behaviors and its effects on a TSS implementation, which emerged from the Sun/Dartmouth TCG/OpenSolaris project, one of the goals of which was instrumenting TPM support on Solaris. At the start of the project, both parties believed the instrumentation to be well-defined and, although time-consuming, relatively straightforward. In the course of the project we had to reexamine our assumptions concerning the state of the hardware and the software involved and the view of the system as presented to someone unfamiliar with its internals. We describe some failure modes we encountered and suggest directions for remediation.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security

*This effort was supported in part by Sun/Oracle and by the U.S. Department of Homeland Security under Grant Award Number 2006-CS-001-000001. The views and conclusions contained in this document are those of the authors, not the funders, and do not represent their official policies.

STC'10, October 4, 2010, Chicago, Illinois, USA.

1. INTRODUCTION

Since 2006, many computers have been sold with the Trusted Platform Module (TPM) chip built in. Among the manufacturers of these chips are Atmel, Broadcom, Infineon, Intel, ST Microelectronics, and Winbond. Many Acer, Dell, Fujitsu, Gateway, Lenovo, Toshiba, and Hewlett-Packard computers now come with the TPM chip. Yet despite the availability of the TPMs, the adoption of the secure programming primitives they offer by both production developers and R&D efforts appears to be comparatively slow compared to adoption rates typical of the IT industry.

1.1 Motivation and our case study

As researchers who see promising applications of trusted computing to many mission-critical problem areas of practical computing, we are compelled to examine potential causes for this slowness.

Neither the problem nor the technical approach appear to be at fault. Indeed, the problem of trust in computer systems (including but not limited to remote trust) is still one of the most fundamental technological problems facing the industry, and still generally lacks comprehensive technical solutions, so lack of interest is hardly the case. Similarly, the TCG's chain of trust based on a hardware root of trust at boot appears as a necessary condition of system trust in the light of attacks moving to lower and earlier layers of the OS (e.g., rootkits' evolution from system call interfaces to the driver method implementations, to bootkits). Notably, the TCG technology as a basis for secure systems attracts independent security researchers highly skilled in attack techniques (e.g., [6]).

Earlier, we wrote about the *"policy gap"* ([2, 3]) between the TCG architecture security model and the process runtime properties that developers associate with the successful enforcement of application-level policies. We argued for providing developers with programming primitives based on the TCG architecture that would allow them to express *application process and data runtime properties* that they consider critical for their security policy goals. However, this gap should stimulate systems developers' attempts to bridge it rather than discourage them.

Yet, we see only a few independent projects that attempt to build on the TPM – e.g., a Sourceforge search as of the time of this writing produces a mere 11 relevant projects,

of which 3 emulate TPM hardware rather than interact with it. We should therefore consider the hypothesis that *some combination of circumstances is frustrating developers and gets in the way of their projects' starting up or maturing.*Anecdotally, we have been made aware of student projects that were abandoned because of various technical roadblocks.

We consider this hypothesis worth discussing for yet another reason. True scalability of a technology is hard to achieve without a critical mass of committed developers. Many recent examples suggest that the breadth of the developer community and its ability to attract a critical mass of programmers – its "developer capital" – significantly helps its adoption. As Perens [5] suggests, this ability is strongly associated with the perception of "enabling" developers and lowering the technological barriers to starting with the technology. Moreover, Couch [4] describes initial developer experience with a programming environment as crucial to adoption with the new generations of programmers.

In this paper we present our experiences derived from integrating and testing support for two vendors' TPMs and *TrouSerS*, an open Trusted Software Stack (TSS) into OpenSolaris on x86. In the course of this effort, we encountered a number of issues potentially frustrating for developers and possibly discouraging for newcomers to the field.

1.2 What this paper is not about

We would like to stress that this paper is not meant to single out any particular TPM vendor for either criticism or commendation. We definitely do not intend to attach blame or "name names." Rather, it seeks to formulate and substantiate engineering challenges for all vendors and the TCG community at large.

Moreover, this paper is not meant as a criticism of either TrouSerS, its TSS API test suite, OpenSolaris, or other tools and technologies mentioned therein. Indeed, we have not encountered any tools that would have served us better.

Furthermore, the TrouSerS project has seen an increase in activity in the past 6 months and has incorporated many of our reported bug fixes and enhancements into its main codebase. There have been several releases of TrouSerS in the past year and the early problems we encountered with resource exhaustion have been addressed. The TSS Test Suite has also seen some activity in recent months; several problems that were causing our tests to fail have been identified and corrected.

We proceed from the understanding that, as a technology spreads to more applications and platforms, and adds features and dependencies – in other words, as its uses scale up and diversify – issues connected with economics of programming effort and programming environment diversity inevitably emerge. They motivate creation of new tools or even subsystems, such as – during the evolution of UNIX – its startup system, tracing and debugging subsystems, TCP/IP stacks, etc.

2. WHAT WE EXPECTED

One of the tasks of the joint Dartmouth College and Sun Microsystems TCG/OpenSolaris project was instrumenting the TPM support on OpenSolaris. At the start of the project, both parties believed this to be well-defined and, although time-consuming, relatively straightforward.

Testing the TPM for compliance. Testing TPMs for compliance is a hard engineering challenge, as shown, e.g., by the important work of Sadeghi et al. [7]. A comprehensive TPM compliance testing effort [1] (primarily of TPM 1.1b implementations, and of some TPM 1.2 chips) was then performed by Sirrix AG in collaboration with researchers from the Horst Görtz Institute for IT Security. Five TPMs were tested, but only two of them were found to be compliant. Testing of the other TPMs revealed numerous bugs. Unfortunately, their test suite was not available to us.

We used TPM 1.2 chips from Intel and Atmel on Dell and Lenovo platforms, with drivers developed by Sun Microsystems engineers and a researcher from our PKI/Trust Lab (see 5.2.1). We also used the *TrouSerS* TSS.[1]

We decided to test our Trusted Software Stack by using the open source *TSS API test suite* for *TrouSerS*.[2] We anticipated that the test suite would not be complete, due to new functionality that needed to be tested and due to functionality that is intrinsically hard to test (such as, for example, key migration).

However, we expected that for the straightforwardly testable and previously tested TPM functionality, the suite would provide us a *"push-button"* ability to tell us that our TPM driver and TSS were functioning correctly, and would thus enable us to test the more sophisticated features and scenarios.

Intuitively, we expected a situation similar to testing a "layered" design ultimately grounded on a physical device, such as an OSI-flavor network stack, which could be tested layer-by-layer, starting with the basic ability of the physical layer device to reliably perform transmit (`tx_*`) and receive (`rx_*`) operations on arbitrary payloads, shared by all platforms. We also expected the ability to conclusively test the lower layers across all implementations with cookie-cutter tests.

In the course of the project we had to reexamine our assumptions concerning the state of the hardware and the software involved in the task and the view of the system as presented to someone unfamiliar with its internals.

3. WHAT WE FOUND

Much to our surprise, after building the test suite on OpenSolaris, we realized that no such "push-button" capability to test the lower layers of the trusted software stack actually existed. We could not conclusively tell from the tests whether TPM driver and TSS were functioning correctly w.r.t. their "essential" functionality, let alone advanced features.

We found that this behavior was partially due to the *different sets of commands implemented and enabled on different TPM chips.* This diversity made it hard to develop a single high-level procedure for testing *either* the "basic" *driver* functionality *or* the correctness of *TSS*, because interacting with the driver to test it required complex wrapping of data that was the raison d'être for much of the TSS, and so using the TSS wrapper code is the natural choice; however, this code is not designed to isolate faults in specific TPM features, and makes assumptions about certain TPM features acting correctly.

[1] `http://sourceforge.net/projects/trousers`.
[2] `http://sourceforge.net/projects/trousers/files/TSS+API+test+suite/`.

Furthermore, test failures associated with unimplemented or disabled features were reported differently by different TPM drivers, and caused different observable behaviors in the TSS, thus making it hard to associate a specific TPM error condition with a particular TSS behavior.

These differences turned out to significantly complicate debugging of higher level TSS and test suite code, which in turn increased the effort required to fix any given bug in the higher layers of the software stack.

Testing layered designs. We note that designing tests for layered architectures with complex low level interfaces – such as the TPM – is a non-trivial research problem. Indeed, interaction with these interfaces requires encapsulation of complexity, which is most naturally abstracted to the higher layers of the software stack; however, "test" and "production" kinds of encapsulation code are likely to be different – which in turn poses the requirement that these two code branches be kept synchronized and must induce equivalent state transitions within the TPM. The latter property becomes harder to validate as the complexity of the input data grows.

A representative experience. It is hard to claim that any particular set of uncovered bugs gives an unbiased idea of the "typical" debugging or security challenges for a platform. Even in methodologies like fuzz-testing, wherein bugs are found by random searches of the spaces of all possible inputs and system states and code coverage is established in terms of the part of the target's instructions or basic blocks traversed during execution, these metrics are only accepted due to the lack of better ones.

However, since our experience involved a *test suite* of TPM features and related TSS functions, we suggest that we have covered many of the common cases that a beginning developer or a start-up project would encounter.

In the following section we classify the kinds of errors we encountered. and discuss their implications.

4. TPM FAILURE MODES AND PATTERNS

When running TSS API test suite on Atmel TPMs and on Intel TPMs, we witnessed the following types of failures.

4.1 Expected test failures

Certain kinds of test failures were expected by the creators of the test suite and are possible (but not guaranteed) on different TPM chips, due to variety in the sets of commands available on those chips.

- Some tests fail with the TPM_E_BAD_ORDINAL error code, which means that the index passed to the TPM does not correspond to a valid TPM command. This kind of failure may happen, for example, if a command was deprecated and removed from a TPM chip.
- Some tests fail with the TPM_E_DISABLED_CMD error code, which means that the command is disabled inside the chip. The command may be disabled permanently or conditionally.
- Some tests fail with the TSS_E_NOTIMPL error code. This error code means that the command is not implemented in TSS.

We note that checking for each of these error conditions in the upper layers of the TSS, and distinguishing between their intended semantics – whenever the semantic distinctions are indeed intended – creates additional complexity for the developer.

4.2 Cascading test failures

The test suite is structured so that some tests (in particular, NVRAM tests) expect the system to be in a certain state after the previous tests. If one test fails, so will the consequent ones.

This kind of failure represents the hardest challenge from the point of view of predictable TSS behavior – and, therefore, developer debugging activities. To correctly identify these error conditions, *the code must track the state of the TPM components*, which essentially means maintaining a (correct) model of the TPM within the code. This requirement is clearly excessive, and yet necessary for correct error interpretation.

We believe that a clear understanding of the secure programming primitive's failure modes is essential for both efficient debugging and for reducing the potential attack surface in the mechanisms built on it.

In particular, differences in the logic that propagates error conditions to higher layers of the API and the resulting difference in error reporting can be very confusing to programmers and reduce their overall productivity, making development more expensive and less secure.

Whereas such differences may be unavoidable or even necessary in a developing standard, they should be offset by *a clear conceptual model of data and control flow* that developers would use while debugging. We discuss this issue further in Section 7.

4.3 TPM resource exhaustion

Most operations with the TPM require *authentication sessions* established between the TPM and the software components serving it, an abstraction that is implemented via parts of the hardware's stored internal state and the software's corresponding state. The TPM's data storage limits them to a small integer, with a minimal compliant number prescribed by the TCG.

As described in Section 6, on some TPMs TrouSerS did not always release authentication sessions. This resulted in the test suite DOS-ing these TPMs by taking up all the available authentication sessions.

4.4 TPM issues and TSS bugs

We witnessed test failures that appeared to be due to problems with respective TPMs or due to TSS bugs. These problems and fixes to them were discussed on the `trousers-tech` mailing list.[3] The full history of fixes to TSS bugs can be viewed in the TrouSerS git repository.

The test suite also contained directory `init/` of tests that could not be automatically run, due to their effect on the TPM - such as, for example, TPM deactivation.

5. CONFUSION ON "FIRST DATE"

In this section we describe unexpected behavior variations that we encountered even before writing any of our own code and dealing with the issues above, a cautionary tale early

[3]See `http://sourceforge.net/mailarchive/forum.php?forum_name=trousers-tech`

adopters of the TPM-enabled platforms intending to offer *customer support* across a diverse TPM deployed base.

Whereas issues described below may not present any significant obstacle to seasoned developers, addressing them as a matter of product/SDK support would likely divert resources from other tasks. Even though monetary costs of such diversion are hard to estimate, we suggest that any difference in development platform behaviors does have a support cost that cannot be entirely eliminated without losing some of the customer base.

We also point the reader to Couch's insightful analysis [4] of modern development practices by younger generations of programmers. While we find the picture it paints is disturbing from the point of view of resulting software trustworthiness, we credit the author with astute "anthropological" observations, no matter how uncomfortable.

The Trusted Computing Group (TCG)'s guide [8] summarizes the enabling and use of TPM in four steps:

1. Turn on the TPM from the BIOS.
2. Load available TPM utility software.
3. Enable the TPM and take ownership.
4. Use the TPM to generate keys.

One would expect that, if all required software is available, and if all configurations are correct, the first three steps would either work straightforwardly, without requiring the developer to spend time and effort to figure them out, or failures in these steps would be easy to interpret. Also, one would expect that TPM functions, such as key generation, would always work according to the Trusted Computing Group's specification. In [7], Sadeghi et al. test these assumptions for earlier TPM versions.

In practice, we found that, given TCG's specifications of the TPM and of user mode interfaces to the TPM, and given diversity of chips, systems, software, and configurations, success in these steps cannot be taken for granted. Moreover, when a step fails, it is not always clear to a non-specialist *why* it failed, and what should be done to remedy the situation.

We describe these three steps and their failure modes in 5.1, 5.2, and 5.3.

5.1 Enabling the TPM chip in the BIOS

For the TPM chip to become functional, it needs to be activated in BIOS.

A new computer with a TPM chip is likely to come with that TPM chip disabled. In order to enable it, the user needs to boot the computer, go to the BIOS menu, find the TPM settings, change them, and save the settings. The BIOS allows not only to activate the TPM chip, but also allows to deactivate it, and to clear the TPM. Clearing the TPM resets all the information stored in it, removing the owner authorization secret, the Storage Root Key (SRK), the other keys, and all the handles.

In the BIOSes of computers, TPM chip settings are frequently listed in the "Security Settings" menu, without any mention of the word TPM. Typically, no version/vendor information about the chip is listed, making it necessary to refer to external data sheets and to assume that the chip is the same as in the standard configuration.

Worth noting is also that on Lenovo laptops the option to clear the TPM is not displayed in the BIOS until the computer is powered down and booted back up. Merely rebooting the computer, without a total power down, does not allow one to see the option to clear the TPM. This has been a source of a lot of confusion of users of TPM on Lenovo machines.

5.2 Loading TPM utility software

As described in the TCG architecture specification overview [9], the communication with the TPM happens through the TPM device driver and through the user mode components of the TCG stack. In Sun/Dartmouth implementation the latter is implemented as the daemon `tcsd`.

5.2.1 Loading TPM device driver

The TPM device driver enables the actual TPM device to communicate with the TSS device driver library. An early prototype of the TPM 1.2 device driver for Solaris was developed by Kwang-Hyun Baek of Dartmouth College, who based it on the Atmel TPM unit. It then had to be significantly rewritten by Sun engineers to accomodate other TPMs. In particular, the initialization functionality of the driver had to be entirely rewritten to make the driver compatible with the various TPMs that Sun developers added support for.

Currently, the TPM driver is packaged into an OpenSolaris package, `SUNWtpm`, available from pkg.opensolaris.org. When the package is installed, it adds to `/etc/devlink.tab` an entry corresponding to the TPM driver and attempts to add the driver by running `add_drv`. The computer needs to be rebooted in order to attach the driver.

If the driver attached successfully, it can be seen by running `modinfo | grep TPM` and checking for the presence of TPM. Running the OpenSolaris GUI utility "Device Driver Utility" (listed under "System Tools" of OpenSolaris 2008-2009 versions) will show the driver for TPM devices even if it failed to attach. The driver may fail to attach if it is incompatible with the TPM (if it failed initial checks), or if the TPM is disabled in the BIOS. If the driver failed to attach, the first thing to check would be whether the TPM is activated in BIOS.

Sun's current release of the TPM driver is intended to support any 1.2 TPM. Sun did extensive testing on Atmel and Infineon TPM devices; Dartmouth extensively tested Atmel and Intel TPMs.

Notably, one of the more generic problems that encountered in the course of this testing was that the register sets that the TPM devices presented to the kernel were not always arranged the same way on different TPMs. In some cases there were multiple registers and the device driver had to query each one to identify the correct one to attach with instead of assuming that the first one was correct.

Instructive examples of bugs, reported and fixed in the later versions, included not checking the registers correctly upon completion of a command, and one of the TPMs unexpectedly returning much more data than others in the version info record, with the result that the data got rejected by the driver.

5.2.2 Trusted Software Stack

As specified by Trusted Computing Group, the TSS interacts with the TPM driver, simplifying TPM access for a

programmer, and also provides APIs for some extra functionality (such as the ability to store keys on disk).

Sun uses the open-source TCG Software Stack TrouSerS (`http://trousers.sourceforge.net`), originally implemented for Linux, to provide TSS functionality; the third author ported TrouSerS to Solaris, fixing a number of Solaris-specific and non-Solaris-specific problems.

The Solaris build of TrouSerS is packed into the packages `SUNWtss` and `SUNWtss-root`. The packages install and start up the TSS Core Service daemon `tcsd`, which allows the user applications to communicate with the TPM.

The daemon `tcsd` needs to run in order for any TSS-based applications to be able to talk to the TPM. The user can check whether it is running by `pgrep tcsd`. If for some reason `tcsd` is not running, it can be enabled with `svcadm enable tcsd`.

In the course of this effort Dartmouth has extensively tested `tcsd`. Bugs in `tcsd` had previously caused it to die due to conditions the user could not diagnose.

5.3 Taking Ownership

For taking ownership of the TPM, Sun provides the utility `tpmadm`, part of the package `SUNWcsu`. Besides taking ownership, `tpmadm` allows to run a few other commands.

```
usage: tpmadm command args ...
where 'command' is one of the following:
status  init             clear [owner | lock]
auth    keyinfo [uuid]   deletekey uuid
```

Running `tpmadm status` allows to examine the state of the TPM, as shown in Figure 1.

```
TPM Version: 1.2 (ATML Rev: 13.9, SpecLevel: 2, ErrataRev: 1)
Contexts: 16/16 available   Sessions: 2/3 available
Auth Sessions: 2/3 available  Loaded Keys: 21/21 available

PCR 0: 7A A0 EC 26 E8 59 1D E9 55 A3 D8 4B BB 03 B8 6F D8 07 8E 6B
PCR 1: 5B 93 BB A0 A6 64 A7 10 52 59 4A 70 95 B2 07 75 77 03 45 0B
PCR 2: 5B 93 BB A0 A6 64 A7 10 52 59 4A 70 95 B2 07 75 77 03 45 0B
PCR 3: 5B 93 BB A0 A6 64 A7 10 52 59 4A 70 95 B2 07 75 77 03 45 0B
PCR 4: AF 98 77 B8 72 82 94 7D BE 09 25 10 2E 60 F9 60 80 1E E6 7C
PCR 5: 7A E0 C1 A8 BD C0 8E 18 D9 9C 31 89 45 4B A9 C3 9C E3 2A 85
PCR 6: 5B 93 BB A0 A6 64 A7 10 52 59 4A 70 95 B2 07 75 77 03 45 0B
PCR 7: 5B 93 BB A0 A6 64 A7 10 52 59 4A 70 95 B2 07 75 77 03 45 0B
PCR 8: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
.....
.....
PCR 16: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
PCR 17: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR 18: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR 19: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR 20: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR 21: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR 22: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR 23: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 1: Output of `tpmadm status`

For `tpmadm` to be able to interact with the TPM (in particular, to take ownership), the following conditions must be satisfied:

- The TPM chip must be activated.
- The TPM driver must be attached.
- The TCS daemon `tcsd` must be running.

If `tpmadm` is able to take ownership, every component necessary for working with the TPM is in place.

6. TPM RESOURCES

Every TPM can hold only a limited number of sessions, contexts, authentication sessions, and keys. However, each TPM vendor implements different limits. As shown above, `tpmadm status` lists the available TPM resources: Contexts, Sessions, Auth Sessions, and Loaded Keys. The TPM user can (for example, through faulty code) cause the TPM to run out of a certain kind of available resources, thus preventing any operation that requires creation of new objects of that type.

In particular, we discovered that some TPMs failed to release authentication sessions when certain `GetCapability` functions fail and quickly run out of sessions. At the time, the chosen fix was for TrouSerS to reset the TPM after such a failure, clearing authentication sessions; it was possible that other solutions existed at the time, but were hard to find. Later, these resource exhaustion problems were reportedly corrected.

If the TPM runs out of a certain kind of a resource, it may become unusable until objects of this kind are released. Releasing the objects may involve power-cycling the system; in some cases, this may be possible to address through software.

Clearing the TPM removes all stored information and makes it possible to use the TPM again. However, this is probably not a measure a user should be forced to revert to.

7. A DIRECTION FOR REMEDIATION?

We see the path to remediating these difficulties in a series of measures to increase the transparency of the TPM device driver, the TSS layer, and the related TSS test suites, that would facilitate observation of each stage on the path of data from and to the TPM, through the driver, and throughout the levels of the TSS.

In this section,we briefly survey the historical precedent for similar improvements in other systems, and then provide specific recommendations for the TCG architecture.

7.1 Historical Precedent

Historically, architectural improvements in complex multi-layered systems has been effected through designing and implementing a system of debugging hooks associated with the paths of data units through the layers. These hooks were "dataflow-centric", in the sense that they not only allowed examination of the lower layers and events occurring in them, but also reported all objects – including all session-related data structures – associated with a particular data flow.

We find a motivating example in the evolution of another layered architecture, TCP/IP network stacks. Despite the obvious differences between network and TSS programming, we note that layering is essential to both for controlling complexity and maintaining future extensibility and that both stacks build on physical devices with limited interfaces and complex behaviors (e.g., the 802.11 link layer requires handling over a dozen different types of frames, with many kinds of information elements). Moreover, normal data flows in each require creating and maintaining additional "association", "connection", or "session" data structures in various levels of the OS kernel.

Initially, network stacks were known to contain multiple bugs and, as a result, vulnerabilities; their defence was relegated to external "firewall" systems, which quickly developed

into implementations of stateful models of the target stacks' state.

The change was brought about by kernel hook systems such as *Netfilter* that instrumented the *path of a communication unit* (packet) through the kernel, and made it possible to *extract and examine* these units at different stages of their processing, as well as to debug the system by injecting specially prepared data structures.[4] The result was a marked improvement in the trustworthiness of network stacks.

7.2 Directions for TPM/TSS

The lessons of our case study can be generalized as follows:

TPM/TSS testing needs to be designed for each specific TPM condition as much as possible, in particular: disabled, enabled but un-owned, enabled and owned.

Testing needs to be broken up so that each layer can be tested before moving up to the higher layers:

- kernel driver layer – passing commands and reading responses from the TPM registers,
- TDDL layer – communicating directly with the kernel driver through the defined kernel interfaces - e.g. whether the TCSD daemon can read and write to the device as expected,
- TSPI layer – can the client applications send and receive commands to the TCSD as expected? Can the applications issue commands and get the expected response all the way through the stack?

Testing of this nature would be *much* easier if there were a known working **emulator** that we could know for sure was returning the proper responses. However, software emulators are just as prone to have implementation bugs as hardware or any other complex layered software solution so it is often difficult to identify where failures lie - in the device, the application software, the test suite, or in the assumptions made by the testers.

8. CONCLUSION

Our experience derived from implementing a TPM/TSS test suite for OpenSolaris leads us to conclude that diversity of TPM feature sets and driver and TSS behaviors, and, in particular, variations in handling of TPM and driver error conditions throughout the stack may present an obstacle to the TCG technology adoption. The mismatch between our expected and actual effort on this project confirms this.

The direct result of our case study was that Sun/Dartmouth TPM support on OpenSolaris is at this time fully functional on all version 1.2 TPM chips, as far as we know. We found the open source TrouSerS test suite to be a useful tool for finding mistakes in TSS implementations and TPM problems, despite some incompleteness and deficiencies. The Sun TSS test suite will soon be available as a package from the testing repositories on OpenSolaris.

However, we also concluded that the failure modes of the system remain confusing to the users, despite many fixes and clarifications, from the very first steps such as loading the driver and taking ownership of the TPM. In some cases, we could not always tell if the bugs were in the test suite, the TSS implementation, or the TPMs themselves. Thus, whereas a number of bugs uncovered with the help of the test suite have been fixed, potential for developer confusion remains.

We contend that this issue is an important one for the TCG community to address, and we see the direction for remediating it in further layer-by-layer, dataflow-centric instrumentation of the Trusted Software Stack, which will enable developers to follow the path of their data through the TSS, and to develop an understanding of its internals and failure modes, and, ultimately, encouraging wider adoption of the TCG architecture. We abstracted and described several design priciples that we believe should be followed to achieve this goal.

[4]See, e.g., `http://netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO.html`.

9. REFERENCES

[1] Sirrix AG. TPM compliance test results. `http://www.sirrix.com/content/pages/test_results_en.htm`, 2008.

[2] S. Bratus, M. E. Locasto, A. Ramaswamy, and S. W. Smith. New Directions for Hardware-assisted Trusted Computing Policies. In *Future of Trust in Computing*, 2009.

[3] Segey Bratus, Michael Locasto, and Brian Schulte. SegSlice: Towards a New Class of Secure Programming Primitives for Trustworthy Platforms. In *Proceedings of the TRUST 2010 Conference*, June 2010. Berlin, Germany.

[4] Alva Couch. Programming with Technological Ritual and Alchemy. *;login:*, June 2010.

[5] Bruce Perens. The Emerging Economic Paradigm of Open Source. `http://perens.com/Articles/Economic.html`.

[6] Bruce Potter. High Time for Trusted Computing. *IEEE Security and Privacy*, 7(6):54–56, 2009.

[7] Ahmad-Reza Sadeghi, Marcel Selhorst, Christian Stüble, Christian Wachsmann, and Marcel Winandy. TCG Inside? - A Note on TPM Specification Compliance. In *Proceedings of the First ACM Workshop on Scalable Trusted Computing (STC'06)*, 2006.

[8] Trusted Computing Group. How to Use the TPM: A Guide to Hardware-Based Endpoint Security. `http://www.trustedcomputinggroup.org/resources/how_to_use_the_tpm_a_guide_to_hardwarebased_endpoint_security`.

[9] Trusted Computing Group. TCG Specification Architecture Overview. `http://www.trustedcomputinggroup.org/.../TCG_1_4_Architecture_Overview.pdf`, August 2007.