To appear in the *2010 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*.

Timing-Accurate TPM Simulation for What-If Explorations in Trusted Computing

Vandana Gunupudi
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052–7329
Email: vgunupudi@gmail.com

Stephen R. Tate
Department of Computer Science
University of North Carolina at Greensboro
Greensboro, NC 27402
Email: srtate@uncg.edu

***Abstract*—The addition of security-oriented hardware devices such as Trusted Platform Modules (TPMs) to computing systems, as promoted by the Trusted Computing Group, leads to many interesting possibilities. Many interesting research questions are of the form "What if the TPM could do x?", but since the functionality of these chips is necessarily fixed it is difficult to explore these questions experimentally and evaluate the performance of proposed solutions. In this paper, we develop a timing model for TPMs which we can instantiate based on experiments with existing TPMs. We perform experiments that validate this model, building "performance profiles" for TPMs from different manufacturers. Our validation process shows that our timing model is very accurate (errors less than 20ms and typically less than 1%) for all but one of the TPMs tested. While the accuracy for the remaining TPM is less than ideal, approaching 25% error, we were able to hand-tune this model so that errors in estimated time were reduced to less than 16%.**

In this paper we also describe our work incorporating these performance profiles into an existing software-based TPM simulator. This modified simulator then allows us to add new functionality with performance that accurately reflects the time such functionality would require if implemented in actual devices. As an example application of this, we describe results of our experiments with implementing random oracles on TPMs, using enhanced functionality not available on existing TPMs.

I. Introduction

"Trusted Computing" technologies, in the sense defined by the Trusted Computing Group [1], revolve around capabilities enabled by the addition of a hardware device called a Trusted Platform Module (TPM) to a computing system. TPMs are small embedded computing devices which provide a specific, standard set of operations to the host platform. These devices are typically built using small (8-bit) processors with embedded firmware to provide the operations, and since these devices serve as a hardware root of trust for the computing platform, it is vital that the operations work as defined and not be modifiable or "hackable" by a user (although the specification provides a mechanism for the manufacturer to perform an authenticated firmware update).

Trusted Platforms (systems incorporating a TPM) potentially have many interesting applications, and several researchers have considered what could be possible if a modified set of operations were provided by the TPM. Examples include using TPMs to provide count-limited objects [2], to instantiate random oracles [3], to support verifiable encryption and fair exchange [4], or to support privacy-preserving operations [5]. Unfortunately, the previously-described reasons why end-user updating of TPM firmware would destroy the basic trust in this hardware mean that directly experimenting with these applications of modified TPMs is impossible by people outside of the companies that manufacture the TPMs.

In this paper, we describe a timing model of TPM operations and a way of testing existing TPMs to build "performance profiles" that allow us to predict the performance of TPMs on new functionality that could be implemented. The basic idea of our model is to break TPM commands into component operations, focusing on cryptographic operations which dominate the time taken by TPM commands, and to derive the time for these component operations through carefully designed experiments. While this has been done in an ad hoc manner for several previous "what-if" projects on TPM functionality [3], [6], we approach the modeling from a generic setting that is widely applicable to different applications. Next, we describe experiments we have performed with TPMs from four different manufacturers (Atmel, Winbond, Infineon, and ST Microelectronics) to derive performance profiles. We then validate our model and performance profiles by estimating and then measuring the times required for additional operations that were not used in making the profiles. On all but one of the TPMs, our validation shows that the models and corresponding profiles accurately predict the time required by the TPM for these new operations, but on one TPM (from ST Microelectronics) our model underestimates the time significantly on some operations. We are unable to explain the differences for this TPM, as some measured times are significantly longer than the sum of the component times would suggest; nevertheless, we managed to hand-tune a model for this TPM that is within 16% of actual times for all tested operations.

Building on our work establishing and validating a TPM timing model, we next describe our work in integrating this timing model into a TPM simulator [7] that will run at a time consistent with the model of real TPMs, both on existing operations and on new operations that we might create in "what-if" TPM research. Since the processors used to make

This material is based upon work supported by the National Science Foundation under Grant No. 0915735.

TPMs are significantly weaker than modern desktop CPUs, the simulator performs cryptographic operations much faster than a TPM, so our solution works by inserting appropriate delays into basic functions used by the simulator. As a result, new TPM functionality added to the simulator calling on these building-block functions operates at a speed that accurately reflects the speed at which a real TPM would run. Finally, we show that this simulator can be used to evaluate the practical performance of certain TPM modifications that enable the instantiation of random oracles by TPMs, using algorithms from previous work [3]. The simulator extensions and timing profiles described in this paper are publicly available [8].

II. Related Work

In this section, we present related work on proposed TPM extensions that add functionality for useful applications. One such effort is the work by Sarmenta *et al.* [2] on creating a potentially unlimited number of virtual monotonic counters using the hardware monotonic counter in the TPM. These virtual counters can then be used to enable count-limited objects, or *clobs*, which can be used to create useful applications like delegated n-time-use signature/encryption keys, n-time-use certificates, and more. To efficiently manage these virtual counters, modest changes to the TPM are required.

Sarmenta *et al.* define a virtual monotonic counter as a mechanism that stores a value and provides `Read` and `Increment` commands to operate on the value. Their design ensures that the virtual counter produces verifiable output and that replay attacks can be prevented. Using these virtual counters, Sarmenta *et al.* have proposed count-limited objects, or *clobs*, as an interesting and important primitive. These are proposed objects that utilize the ability of a TPM to encrypt data or keys into "blobs" such that they can only be decrypted when the TPM is in a specified state, which in current TPMs is limited to conditions based on the PCRs. In Sarmenta's construction, these encrypted blobs are then linked to a virtual monotonic counter which is used to track/limit the usage of the blob. Since each clob requires a dedicated virtual counter, they proposed an efficient scheme (requiring modifications to the TPM) that uses hash trees to allow the TPM to track a large number of virtual monotonic counters.

Sarmenta *et al.* recently extended their work and showed how to create offline count-limited certificates (or *clics*) [9] using existing TPM functionality. These digital certificates depend on irreversible counters and have guarantees that any attempt to subvert the protocol can be detected even if all parties are offline. These certificates can have diverse applications including using n-time-use certificates and migratable certificates in e-commerce and offline trading scenarios. These clics are implemented using existing TPM features like monotonic counters, special signing keys created on the TPM and exclusive and logged transport sessions. They tested their protocols using TPM/J [10], a Java-based interface to the TPM, on 2 laptop machines with TPMs. They timed the various operations on the TPM and extrapolated the results to get timing results for their protocol.

Using Sarmenta's work on *clobs*, Gunupudi and Tate showed how to utilize count-limited objects to remove interaction requirements in oblivious transfer [6]. Oblivious transfer (OT) is a fundamental primitive used in many cryptographic protocols, including general secure function evaluation (SFE) protocols. However, interaction is an unavoidable component of any OT protocol on standard systems (i.e., those without additional hardware security components such as TPMs). This work introduced a new primitive, called "Generalized Non-interactive Oblivious Transfer" (GNIOT), which is an efficient method of performing a collection of general (k-out-of-n) independent oblivious transfers with a single request, using a single *clob*, in a non-interactive manner. The GNIOT primitive was then applied to mobile agent computation, where strong security is often enforced by interactive oblivious transfer in various agent protocols. Removing the interaction from these agent protocols removes a significant bottleneck to their efficiency and practicality.

Additional work by Gunupudi and Tate [3] shows how to create a random oracle using a slightly modified TPM. The random oracle model is an idealized theoretical model that has been successfully used for designing many practical cryptographic algorithms and protocols such as the Probabilistic signature scheme (PSS) [11] and Optimal asymmetric encryption padding (OAEP) [12]. In this model, all parties, including the adversary, have oracle access to a random function. An instantiation of the algorithm in the standard model (or "real world") is obtained by replacing the random oracle with a suitable function like a "good" cryptographic hash function. One of the potential candidates for a real-world instantiation of a random oracle is a pseudorandom function (PRF). A PRF produces output that is computationally indistinguishable from random to a polynomial-time attacker. For a PRF to be used as a secure random oracle, the seed used to key the PRF must be kept secret, something that is impossible to do without augmented hardware support. This prior work showed how a Certified Migratable Key (CMK) can be securely transmitted to various parties (using the "protected storage" functionality of TPMs) and then used as the seed to the HMAC function in the TPM to create a "TPM oracle".

III. A TPM Timing Model

The construction of our timing model starts with identifying the component operations that are used when a TPM command is executed. As with many cryptographic systems, the most time-consuming operations are those that use public key cryptography: In TPMs this includes public key encryption and decryption, as well as digital signatures. Other non-trivial components of the time required for a TPM command include the overhead for communicating the command to the TPM and dispatching the command within the TPM, HMAC-based authentication for commands, random number generation, and computing a cryptographic hash. There is a significant amount of basic field and parameter validation, but as these are typically just quick comparisons, we assume the time required for basic data manipulation like this is insignificant.

After some initial tests, we identified 7 significant variables for our timing model, which we define below.

T_{ohc}: Constant overhead for invoking a TPM command.
T_{ohir}: The "overhead rate" for input data, measured in cost per byte.
T_{ohor}: The "overhead rate" for output data, measured in cost per byte.
T_{dec}: The time to perform an RSA decryption operation.
T_{enc}: The time to perform an RSA encryption operation.
T_{sign}: The time to perform an RSA signature operation.
T_{hashr}: The hash rate, or the time to calculate a SHA1 hash, measured in cost per byte.

Looking through the TPM command definitions, given in Part 3 of the TPM Specification [13], we identified several commands with time dominated by each of the public-key cryptography operations. Using our basic variables, we can break down the time required for these TPM commands into components as follows (coefficients in these formulas are derived from the specification, which gives the sizes of the input and output parameters for each operation — for those commands that have variable input or output size, the sizes below correspond to the sizes we used in our experiments, described below):

$$T(\texttt{TPM\_Seal}) = T_{ohc} + 98\,T_{ohir} + 319\,T_{ohor} + T_{enc} + 2240\,T_{hashr}$$
$$T(\texttt{TPM\_UnBind}) = T_{ohc} + 319\,T_{ohir} + 66\,T_{ohor} + T_{dec} + 2112\,T_{hashr}$$
$$T(\texttt{TPM\_Sign}) = T_{ohc} + 83\,T_{ohir} + 311\,T_{ohor} + T_{sign} + 1024\,T_{hashr}$$

These commands work by first validating parameters passed to the command, verifying use of internally stored data, and verifying authentication for keys, before performing the main operation. In order to extract the individual component times, we designed a series of tests in which various error conditions were intentionally invoked, effectively aborting the commands at various stages of execution. For example, with the `TPM_Sign` command we invoked the command correctly, but also invoked the command with a bind key (instead of a signing key), with a bad authentication for the key, and with an empty data area to sign.

Timing Technique: To interface with the TPM device driver, which is now a part of the standard Linux kernel, we used TPM/J which is an object-oriented Java interface to the TPM, developed at MIT by Sarmenta *et al.* [10]. TPM/J provides direct interaction with the TPM, which is more appropriate for our experiments than the abstracted interface of the TCG's standard Trusted Software Stack (TSS). While not all TPM commands are implemented in the current TPM/J distribution, the modular design made it easy for us to add support for new commands (e.g., the commands dealing with Certifiable Migratable Keys). As we complete our experiments, we plan on sending our code additions back to the TPM/J project so that support for all TPM commands is available for everyone using TPM/J.

In order to time the commands in TPM/J, we modified the TPM/J library to insert timing measurements directly before the raw command was written to the TPM device driver and immediately after the result was read. While we put these commands as close to the actual device write and read as possible, we were not sure how the overhead of Java libraries would affect our timings and so performed the following test: We ran the tests for the `TPM_Sign` command using the Linux `strace` command with the `-tt` option to save the time at each system call, and compared the times obtained in this way to the times obtained by our instrumentation of the Java TPM/J library. Since `strace` ties directly into the system calls, after any user-level libraries (such as the Java JVM implementation), these "raw" times are as close as we can get to actual device times without modifying the kernel driver. The results of these tests gave us great confidence that the times reported by our instrumentation of TPM/J are extremely accurate: for one test the time derived from the system call trace was 726.55 milliseconds, whereas the time reported from our TPM/J instrumentation was 726.75 milliseconds, showing an overhead of less than 0.03% (3 hundredths of one percent).

For final timings reported here, we ran each test 100 times and calculated the average and standard deviation of the times for each test. With the exception of key generation commands (which use randomized algorithms whose time can vary greatly) the standard deviation for each test was small, indicating that consistent and reliable times were obtained. It is interesting to note that the standard deviation of times was significantly smaller on the ST Microelectronics TPM, and the lack of variation in times suggests that the designers of this TPM may have included measures to reduce vulnerability to timing attacks.

Interestingly, while the specification defines TPM commands using a sequence of operations, our timings reveal that not all TPM manufacturers perform the tests in the same order that they are given in the specification. By sorting the times when invoked by various error conditions, we could determine which order the tests were performed inside the TPM, and taking differences we could derive the times for individual tests and operations.

Timing Results: In our lab we have systems that include TPMs by all major TPM manufacturers: Atmel, ST Microelectronics, Infineon, Broadcom, and Winbond. Our goal was to build performance profiles for all five TPMs, but problems with the Broadcom TPM made it impossible to run the timing tests on that TPM at the time of this writing. We hope to add results for the Broadcom TPM to our results for the other four TPMs at a future date. For comparison, we also ran the same timing tests with a TPM simulator running on an Intel Core 2 T7200 clocked at 2GHz.

While we had good results extracting the larger component times (those from public key operations), it proved challenging to derive the smaller values, particularly those that are rate variables. To solve these problems, rather than trying to calculate the smaller values directly we used least squares approximation to give good values for the remaining variables

TABLE I
TIMING MODEL PARAMETERS — TIMES ARE IN MILLISECONDS

TPM	T_{ohc}	T_{ohir}	T_{ohor}	T_{dec}	T_{enc}	T_{sign}	T_{hashr}
Atm	18.7	0.001	0.001	766	55	760.58	0.009
Win	9.8	3e-05	3e-05	493	103	492	0.001
Inf	19.2	0.003	0.001	304	161	309	.0001
STM	.001	0.150	0.083	744	192	690	0.003
Sim	–	–	–	14.8	0.65	14.7	–

(all except T_{enc}, T_{dec}, and T_{sign}). The results are given in Table I, where some values are truncated to control table width — full precision is in the online performance profiles [8].

IV. Model Validation

In designing our model, we assumed that TPM command times could be broken down into component times which we could measure, and that these times would allow us to predict the times for additional TPM commands (both implemented and proposed). In order to verify this hypothesis and give us confidence in results derived from this model, we identified additional TPM commands that were not tested as part of the model derivation described above, estimated the time that our model predicted the commands would take, and then ran tests to test the accuracy of our model on these previously untested commands. Five such commands, broken down into component times, are shown below.

$$T(\texttt{TPM\_Quote}) = T_{ohc} + 84\,T_{ohir} + 800\,T_{ohor} + T_{sign} + 2048\,T_{hashr}$$
$$T(\texttt{TPM\_Unseal}) = T_{ohc} + 372\,T_{ohir} + 107\,T_{ohor} + T_{dec} + 2816\,T_{hashr}$$
$$T(\texttt{TPM\_AuthMigrationKey}) = T_{ohc} + 341\,T_{ohir} + 357\,T_{ohor} + 1664\,T_{hashr}$$
$$T(\texttt{TPM\_CreateMigrationBlob}) = T_{ohc} + 672\,T_{ohir} + 554\,T_{ohor} + T_{dec} + T_{enc} + 5696\,T_{hashr}$$
$$T(\texttt{TPM\_ChangeAuth}) = T_{ohc} + 388\,T_{ohir} + 352\,T_{ohor} + T_{dec} + T_{enc} + 4160\,T_{hashr}$$

Next, we evaluated the formulas from our mathematical model of TPM times for the original three TPM commands as well as these five new commands, and compared the calculated results with those from timing experiments, and the results are given in Table II.

From these validation tests, it can be seen that our model is very accurate for the Atmel, Winbond, and Infineon TPMs, with only a few times being off by more than 10 milliseconds. Even the highest error (20.1 milliseconds) is less than a 5% error, and most errors are under 1%.

Unfortunately the results aren't as good for modeling the ST Microelectronics TPM, with several errors in the hundreds of milliseconds. We are not entirely sure what is causing this problem. We initially speculated that the low variation we saw in times from this TPM indicated that results were delayed in order to have a fixed time for commands and thereby protect against timing attacks. However, even this would not account for the long time required by the ST Microelectronics TPM, which takes far longer for some commands than component times would suggest. When we created the model from all commands (rather than our limited set of three used initially), the model is slightly more accurate, with a maximum error of around 16%. While accurate enough to get ballpark estimates for protocols using this TPM, it will not give precise estimates.

V. Integration Into a TPM Simulator

In this section, we describe how we incorporate our timing model into an existing TPM simulator, originally designed by Mario Strasser and now an open-source project [7].

A. TPM Simulator

The TPM simulator consists of a kernel module and a user-space daemon that implements the TPM simulator. The kernel module, called `tpmd_dev`, simulates the hardware TPM by providing a character device `/dev/tpm`. This interface is the same as for a physical TPM, so application software written for a real TPM will run under the simulator without modification, and vice-versa. When the simulator is loaded, all commands destined for this device are written to the TPM simulator daemon, `tpmd`, which implements the actual simulator. This consists of the daemon application, the TPM simulator engine and the cryptographic module. The simulator daemon, `tpmd`, listens for requests on a Unix socket, which has the default name `/var/tpm/tpmd_socket:0`. Received commands are processed by the simulator engine and responses are returned to the socket and thus back through the kernel driver to the application that sent the command.

The basic operation of the simulator is controlled by 3 main functions. The `tpm_emulator_init()` function initializes the simulator. To handle various commands, the `tpm_handle_command()` is used. Finally, a call to the `tpm_emulator_shutdown()` shuts the simulator down. At startup, all TPM-related data is set to default values. The simulator can be started in various modes, as specified in the TCG specifications. All persistent data is written to a specific file, and if the simulator is started in the "save" mode, all relevant data is restored from that file. If the simulator shuts down normally, all persistent data is written to the same file. Otherwise, in case of abnormal shutdown or if the simulator is started in the mode "clear", all internal data of the simulator are destroyed.

B. TPM Commands and Execution

TPM commands and responses are byte arrays which consist of three main parts: a request or response header, some command-specific input/output parameters, and an optional authorization trailer. Presence or absence of the authorization trailer depends on the command (response) type, defined by the tag field. There are three different types of commands that vary in how the command is authorized.

The execution of a TPM command is initiated by calling the function `tpm_handle_command()` with the marshaled TPM command (converted into a byte array) as input parameter. The command is then processed in three steps:

TABLE II
TIMES FROM OUR DERIVED TIMING MODEL, COMPARED TO MEASURED TIMES (ALL TIMES ARE IN MILLISECONDS)

	Atmel			Winbond			Infineon			ST Microelectronics		
TPM Command	Est	Actual	Error	Est	Actual	Error	Est	Actual	Error	Est	Actual	Error
Seal	94.0	95.7	-1.7	114.6	114.1	0.5	180.8	180.8	0.0	239.9	239.9	0.0
Unbind	803.9	798.0	5.9	505.3	505.0	0.3	324.5	324.3	0.2	803.4	803.4	0.0
Sign	788.8	791.8	-3.0	502.9	503.7	-0.8	328.7	328.9	-0.2	731.6	731.6	0.0
Quote	798.4	798.4	0.0	503.9	504.8	-0.8	329.3	341.5	-12.2	775.2	804.8	-29.6
Unseal	810.2	820.1	-9.9	506.0	509.5	-3.6	324.7	335.5	-10.8	816.8	852.2	-35.5
AuthMigKey	34.2	35.7	-1.5	11.5	11.8	-0.3	20.6	24.0	-3.4	85.5	125.6	-40.0
CreateMigBlob	891.5	889.3	2.2	611.5	624.8	-13.4	486.9	470.5	16.5	1099.4	1464.5	-365.1
ChangeAuth	877.4	881.8	-4.5	609.9	618.1	-8.2	485.9	465.7	20.1	1035.7	1350.6	-314.9

1) The command is first decomposed into its three main components: the request header, the command parameters, and the (optional) authorization trailer. If it is a legal command, the input parameter digest is computed and the parameters are further unmarshaled according to the specific TPM command.
2) The command is executed and the command response parameters are setup. For commands that require authorization, the authorization trailer is verified in order to guarantee command authorization and integrity of the input parameters.
3) Finally, the response authorization is computed and combined with the response parameters and the response header. The response is then marshaled into its byte representation and returned to the caller.

C. Our Extensions

In this section, we present details of the extensions to the simulator and TPM/J. In addition to the modifications to support timing simulation, in order to support a wider range of tests and support the application described in Section VI we also added functionality to support migration of regular keys as well as CMKs.

1) Extensions to the TPM Simulator and TPM/J: Our extensions to the TPM simulator allow us to load a performance profile (a set of variables for our timing model) and delay completion of the simulator commands so that the actual time taken by an application using this modified simulator reflects the estimate from the model. Since these timing tests are probably something that is only needed sometimes (when benchmarking an application, for example), we added a startup option (`-t`) to the simulator to turn on our delays, and added a second option (`-p`) to set an arbitrary performance profile (by default the profile `/etc/tpm/timing/default.pro` is used). So to start the simulator with timing delays enabled and the Infineon performance profile, we can use the command:

```
tpmd -t -p /etc/tpm/timing/infineon.pro
```

The performance profiles are simply text configuration files that give values for the variables in our model. The following is an example of a profile for the Infineon values shown earlier (lines starting with # are comments):

```
# Infineon profile

overconst: 19.225
overinrate: 0.0025
overoutrate: 0.001

decrypt: 304.16
encrypt: 160.76
sign: 308.88

hashrate: 0.0001
```

In order to use the timing profiles, we added delays in two main places: The command dispatch function (to handle the three overhead times) and the cryptographic routines. This way, any new functionality that uses the cryptographic library will automatically get the appropriate delays from our model. We designed a timing library so that adding the delays in the code was particularly easy, and so augmenting the model with new variables should also be simple. For example, to add the appropriate encryption delay, we just had to insert the following single line into the `tpm_rsa_encrypt` function:

```
TPMTiming_addDelay1(TPMTIMING_ENCRYPT);
```

Our first attempt at inserting delays was to do the actual delay at this point; however, in running tests we discovered that the simulator makes many calls to the hash function with very small data sizes (even single bytes), and so attempting to add many such tiny delays (e.g., 3 μ-seconds for a single byte hash) worked very poorly. Our current code simply accumulates a measure of the delay required, and then performs the accumulated delay once, immediately before the result is returned to the application. By doing this we can also subtract the actual time taken by the simulator on the complete command, to produce a much more accurate delay.

2) Testing Results: We next ran all of our original tests again, through TPM/J with the TPM/J timing instrumentation, but this time with the simulator rather than the real TPMs. The tests include both the first set of tests that we used to derive our model, plus the tests on additional commands that we used in model validation. The goal of these tests is to see how well our simulator reflects the times of the real hardware,

TABLE III
TIMES FROM THE INSTRUMENTED SIMULATOR, COMPARED TO MEASURED TIMES (ALL TIMES ARE IN MILLISECONDS)

	Atmel			Winbond			Infineon			ST Microelectronics		
TPM Command	Est	Actual	Error	Est	Actual	Error	Est	Actual	Error	Est	Actual	Error
Seal	94.1	95.7	-1.6	114.7	114.1	0.6	180.8	180.8	0.1	240.0	239.9	0.1
Unbind	803.4	798.0	5.5	505.0	505.0	0.0	324.1	324.3	-0.2	803.1	803.4	-0.4
Sign	788.3	791.8	-3.5	502.4	503.7	-1.3	328.2	328.9	-0.7	731.1	731.6	-0.5
Quote	797.9	798.4	-0.5	503.4	504.8	-1.4	328.8	341.5	-12.7	774.7	804.8	-30.1
Unseal	809.8	820.1	-10.3	505.6	509.5	-3.9	324.3	335.5	-11.1	816.3	852.2	-35.9
AuthMigKey	33.6	35.7	-2.1	10.8	11.8	-1.0	19.9	24.0	-4.0	85.0	125.6	-40.6
CreateMigBlob	891.1	889.3	1.8	611.2	624.8	-13.7	486.5	470.5	16.0	1099.3	1464.5	-365.2
ChangeAuth	877.4	881.8	-4.5	609.8	618.1	-8.3	485.4	465.7	19.7	1035.7	1350.6	-314.9

and the results are given in Table III.

The results from these tests are extremely close to the calculated results in our model validation tests, showing that our additions to the TPM simulator are indeed reflecting the mathematical model that we defined and validated earlier. As before, notice that the errors are all quite small for the Atmel, Broadcom, and Infineon TPMs, but the same problems as before arise with the ST Microelectronics TPM.

VI. EXAMPLE USE: RANDOM ORACLES

In this section, we apply our performance estimation techniques — both the mathematical model and the augmented simulator — to evaluate a previously proposed technique for instantiating random oracles using TPM functionality [3]. In this problem, n hosts need to participate in an n-party cryptographic protocol that was designed in the random oracle model. Below, we give a brief overview of this work and then present results that demonstrate the usefulness of our timing model and simulator modifications.

A. Overview of TPM oracle construction

The random oracle model is a technique for designing and reasoning about cryptographic protocols in which it is assumed that all parties have access to a random function. While at first it seems like a pseudo-random function (a "PRF") can substitute for any random function when the attacker is a probabilistic polynomial time algorithm, the subtle flaw in this reasoning is that for a PRF to be computationally indistinguishable from a real random function the key must be secret — clearly this is impossible for parties participating in the protocol in the traditional model of computing. However, TPMs give us the capability to use cryptographic functionality with keys that are kept secret, even from the owner of the system. The basic building blocks for creating a random oracle using a TPM are: (1) the ability to share secrets between TPMs in a way that is controlled so that all parties have assurance that these secrets only exist in usable form within protected TPMs, and (2) the ability of a TPM to compute HMAC. However, we need very slight modifications to standard TPMs which makes it impossible to directly test these techniques on real TPMs, motivating our use of the timing models described in this paper. In particular, we need the following modifications:

- *Support for direct migration of CMKs.* Currently, a CMK must be under the control of a migration authority (MA), represented by a signing key that approves TPM destinations. Unfortunately, this MA can arbitrarily authorize additional destinations without the knowledge of protocol participants. As we would like to avoid requiring a trusted party, we suggest a change in which only public storage keys (and no signing keys) are on the approved "msaList" when the key is created, and direct migration to those destinations are implicitly allowed without further authorization. This is in fact a *more* restricted version of migration than the already-supported CMK migration and so does not affect the security of currently supported CMK uses.
- *Support for symmetric HMAC CMKs.* Currently only RSA keys are supported as CMKs, but a simple random HMAC key type integrated into the TPM would allow sharing HMAC keys between TPMs.
- *Command for executing HMAC with a CMK.* HMAC is already supported in TPMs for authenticating commands, so this just adds an external interface with the HMAC CMKs just described.

These modifications are all trivial and do not affect the security of existing TPM functionality. We cannot extend real TPMs ourselves to experiment with these ideas, but we can estimate performance mathematically using our model and experimentally explore these ideas through our timing-accurate simulator.

Below, we summarize the TPM operations required for each phase of our TPM oracle construction. For details of these operations, refer to the TPM specifications [14].

Part 1: Creation of a Certified Migratable Key (CMK) on a TPM (designated as the source)

(a) `TPM_CMK_ApproveMA` is used to authorize a list of destination keys — the source should check key certificates to verify that all keys are TPM-bound storage keys.
(b) `TPM_CMK_CreateKey` command creates an HMAC key in which migration is restricted to the list of destination keys provided in step (a).
(c) The newly created CMK must now be certified using a special signing key called an AIK with the

TPM_CertifyKey2 command. For this step, both the CMK and the AIK must first be loaded using the TPM_LoadKey2 command.

(d) For each of the $n-1$ destination hosts, the source must first authorize the destination using the TPM_AuthorizeMigrationKey command, and then create a "migration blob" for transmission using the TPM_CMK_CreateBlob command. The original CMK public key, the certificate from step (c), and the appropriate migration blob is sent to each destination.

Step 2: Installation of the CMK at each destination

(a) The destination host should verify the certificate on the original CMK and each destination, to see that the CMK is restricted to TPM-bound parent keys. This does not require any TPM operations.

(b) TPM_CMK_ConvertMigration is used to decrypt and re-encrypt the migrated key under its local parent key.

(c) The resulting key is loaded using TPM_LoadKey2 so that it is ready for use.

Step 3: "TPM Oracle" use.

The "TPM Oracle" requires a new command, which we call TPM_HMAC, to compute an HMAC using the loaded CMK.

B. Timing Estimations for TPM Operations

Before describing the use of our timing model and simulator to estimate performance, we describe the ad hoc timing estimation we used in our initial work on this protocol. It was, in fact, this work that led directly to the current paper as we abstracted the techniques into a more generally useful model.

There are a total of 7 different TPM operations required for the TPM oracle setup, on source and destination hosts, as described in the previous section. Of these, only two can be tested directly, as our proposed TPM operation modifications involve introducing a new key type that is needed but unsupported in 5 of the operations. Fortunately, for 4 out of 5 of these operations, the only real difference is the size of the key, since the key is not used but rather is simply wrapped and unwrapped as a blob. Therefore, for all except the TPM_CMK_CreateKey operation we simply measured times for operations using RSA keys, and used those times as estimates for our new operations. For the TPM_CMK_CreateKey command, we used the TPM_Seal command as a time estimate, since it does many of the same operations as TPM_CMK_CreateKey when we create an HMAC key. While this technique gave decent ballpark estimates of times, the estimates had a component of "guessing" involved, and lacked the more rigorous model validation that we have performed for the general timing models in this paper.

Given the work described in this paper, we can more accurately and reliably estimate the times for the operations required in the TPM Oracle construction. Counting sizes and operations of the necessary TPM commands, we derive the following formulas:

$$
\begin{aligned}
&T(\texttt{TPM\_CMK\_ApproveMA}) = \\
&\quad T_{ohc} + 75\,T_{ohir} + 71\,T_{ohor} + 1024\,T_{hashr} \\
&T(\texttt{TPM\_CMK\_CreateKey}) = \\
&\quad T_{ohc} + 154\,T_{ohir} + 362\,T_{ohor} + T_{enc} + 3008\,T_{hashr} \\
&T(\texttt{TPM\_LoadKey2}) = \\
&\quad T_{ohc} + 370\,T_{ohir} + 55\,T_{ohor} + T_{dec} + 2240\,T_{hashr} \\
&T(\texttt{TPM\_CertifyKey2}) = \\
&\quad T_{ohc} + 148\,T_{ohir} + 436\,T_{ohor} + T_{sign} + 1920\,T_{hashr} \\
&T(\texttt{TPM\_CMK\_CreateMigrationBlob}) = \\
&\quad T_{ohc} + 683\,T_{ohir} + 513\,T_{ohor} + T_{dec} + T_{enc} + 5440\,T_{hashr} \\
&T(\texttt{TPM\_AuthMigrationKey}) = \\
&\quad T_{ohc} + 341\,T_{ohir} + 357\,T_{ohor} + 1664\,T_{hashr} \\
&T(\texttt{TPM\_CMK\_ConvertBlob}) = \\
&\quad T_{ohc} + 680\,T_{ohir} + 311\,T_{ohor} + T_{dec} + T_{enc} + 5440\,T_{hashr}
\end{aligned}
$$

Any of the timing profiles can be used in these formulas to estimate the time that would be required for these operations if they were implemented in a TPM with the new HMAC key type. We also implemented the new key type and operations in the TPM simulator. It is important to note that because of the way we embedded timing into the cryptographic and general command processing places, our implementation did not require giving any consideration at all to timing issues — that was a "free" benefit of implementing the operations. Implementing in the simulator gives several benefits over the mathematical models, including the ability to measure overhead of management and other non-TPM operations, and the ability to test and debug protocols.

Next, we report the results of this testing. To make things concrete, we selected the Infineon profile, which we use for the remainder of this section. The times for TPM operations were calculated using the timing model, and were measured on implemented operations in the simulator. Results are given below, where times are in milliseconds.

Reference	Operation	Model	Simulator
T_1	CMK_ApproveMA	20	20
T_2	CMK_CreateKey	181	181
T_3	LoadKey2	325	326
T_4	CertifyKey2	329	332
T_5	CMK_CreateBlob	487	489
T_6	AuthMigKey	21	24
T_7	CMK_ConvMig	487	489

Notice that the times are in very close agreement, validating our simulator modifications as an accurate reflection of the underlying mathematical model. Next, we use these times to estimate the time of the full TPM Oracle operations described in the previous section.

1) Source Operations: Extracting the commands from Step 1 in the previous section, and noticing that steps 1a-1c are performed once, while step 1d is performed for each of the $n-1$ destination hosts, we derive the following formula for total time of the TPM Oracle operations at the source:

$$T_1 + T_2 + 2T_3 + T_4 + (n-1)(T_5 + T_6)\,.$$

Using calculated values from our model above, we get a total time for TPM operations at the source of

$$1180 + 508(n-1) = 672 + 508n \text{ milliseconds.}$$

In particular, for a two-party protocol ($n = 2$), we have an estimated time of 1,688ms, or 1.688 seconds. As mentioned above, the simulator allows us to estimate other components of this time, such as the non-TPM operations, or overhead. Measuring a complete Java application to perform source operations, we measured a total time of 2.1 seconds. Looking closer, we found that a significant number of operations were performed to set up authorization sessions, authorizing use of various keys and migration, and those TPM operations in the simulator took a total of around 0.2 seconds. The remaining 0.2 seconds is overhead for non-TPM operations and Java virtual machine startup and management time. The lesson to be learned from this is that these "overhead" operations can add up — taking almost 20% of the total time. Being able to measure this overhead directly made the additional work of implementing operations in the simulator worthwhile.

2) Destination Operations: We similarly model the operations at the destination host, given in Step 2 of the previous section, to arrive at a total TPM time of $T_7 + T_3$, which is 812 seconds in our Infineon timing model. The actual measured time for the operations was 974 milliseconds, with 40 milliseconds coming from overhead of TPM authentication sessions, and the remaining 122 milliseconds being general application overhead.

3) TPM-Oracle Queries: There is no current user interface to the HMAC engine within a TPM, but there is a direct way to use the SHA1 engine. Since the HMAC computation consists entirely of multiple calls to the SHA1 function, this is a good way to estimate the performance of HMAC in the TPM. We tested SHA1 on inputs of size 16k, 32k, 48k, and 64k, and found a very consistent increase of 1.15 seconds per 16k increment.

Here's how we derive the estimates. For the SHA-1 estimation: the SHA-1 block size is 512 bits, or 64 bytes. Therefore there are $16k/64 = 256$ blocks in each 16k increment. Therefore, we obtain $\frac{1.15}{256} = 4.5$ milliseconds as the estimate for one application of the SHA1 function. The ultimate goal is to estimate the performance of HMAC, not just the SHA1 function. In HMAC, the input key extension adds 1 block to the hashing, and the output hash adds an extra 2 blocks.

Based on these benchmarks, we see that if the TPM-oracle is answering queries from a domain that requires b_i input blocks to hash, and we are generating an output that needs b_o output blocks, then the time would be at most $4.5(b_i + 3)b_o$ ms. To make this concrete, note that for 1024-bit inputs ($b_i = 3$), with 160-bit outputs ($b_o = 1$), each random oracle query would take no more than 27 milliseconds.

4) Summary: In summary, our model allows us to estimate the TPM-oracle using current hardware with an Infineon TPM. For a two-party protocol the setup operations, combining source and destination operations but excluding communication cost, is roughly 3.1 seconds, and realistically sized TPM-oracle queries take less than 30 milliseconds. The ability to estimate these times with a high level of confidence, when we cannot directly implement and test them, is a significant contribution of the modeling work in this paper.

VII. Conclusions

In this paper, we described a model and software tool that we developed that will allow researchers to experimentally explore the efficiency of algorithms and protocols using Trusted Platform Models (TPMs) when new functionality is added over and above the functionality defined by the Trusted Computing Group. We defined a clean mathematical model of the time required for a TPM to complete a command, validated this model through experiments, and described our instrumentation of a software TPM simulator that forces it to answer queries in an amount of time that is consistent with our model. Our experiments show that our model is indeed quite accurate in estimating the time required by TPMs manufactured by Atmel, Winbond, and Infineon, although there were some significant errors in our modeling of the ST Microelectronics TPM. Finally, we described a use of this instrumented simulator in order to estimate the practical performance of previously-developed protocols for implementing a secure random oracle using TPMs. We believe this modeling and tool will be very useful to other researchers exploring "what-if" questions in trusted computing, and the code that we have developed for these experiments is publicly available [8].

References

[1] "Trusted Computing Group," Website. http://www.trustedcomputinggroup.org.

[2] L. F. G. Sarmenta, M. van Dijk, C. W. O'Donnell, J. Rhodes, and S. Devadas, "Virtual monotonic counters and count-limited objects using a TPM without a trusted OS," in *Proceedings of the First ACM Workshop on Scalable Trusted Computing (STC 06)*, 2006, pp. 27–42.

[3] V. Gunupudi and S. R. Tate, "Random oracle instantiation in distributed protocols using trusted platform modules," in *Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops (AINAW'07)*, 2007, pp. 463–469.

[4] S. R. Tate and R. Vishwanathan, "Improving cut-and-choose in verifiable encryption and fair exchange using trusted computing technology," in *Proceedings of the 23rd Annual IFIP WG 11.3 Working Conference on Data and Application Security*, 2009, pp. 252–267.

[5] A. Iliev and S. Smith, "Private information storage with logarithmic-space secure hardware," in *Proceedings of the Workshop on Information Security Management, Education, and Privacy*, 2004, pp. 201–216. [Online]. Available: http://www.cs.dartmouth.edu/ sws/papers/is04.pdf

[6] V. Gunupudi and S. R. Tate, "Generalized non-interactive oblivious transfer using count-limited objects with applications to secure mobile agents," in *Proceedings of the 12th International Conference on Financial Cryptography and Data Security (FC 2008)*, 2008, pp. 98–112.

[7] M. Strasser, "A software-based TPM emulator for Linux," Master's thesis, Eidgenossische Technische Hochschule (ETH), Zurich, 2005, project web page, https://tpm-emulator.berlios.de/documentation.html.

[8] V. Gunupudi and S. R. Tate, "TPM simulator extensions — project page." [Online]. Available: http://span.uncg.edu/tpmtiming

[9] L. F. G. Sarmenta, M. van Dijk, J. Rhodes, and S. Devadas, "Offline count-limited certificates," in *Proceedings of the ACM Symposium on Applied Computing (SAC 08)*, 2008, pp. 2145–2152.

[10] L. Sarmenta, J. Rhodes, and T. Muller, "TPMJ: Java-based API for the trusted platform module," http://projects.csail.mit.edu/tc/tpmj/, 2007.

[11] M. Bellare and P. Rogaway, "The exact security of digital signatures - How to sign with RSA and Rabin," in *Proceedings of EUROCRYPT*, 1996, pp. 399–416.

[12] V. Shoup, "OAEP reconsidered." in *Proceedings of CRYPTO*, 2001, pp. 239–259.

[13] Trusted Computing Group, "TPM Main – Part 3 – Commands, specification version 1.2, revision 103," 2007, available at `http://www.trustedcomputinggroup.org`.

[14] ——, "TPM Main Specification Version 1.2, Revision 103, Parts 1-3," 2007, available at `http://www.trustedcomputinggroup.org`.