TPM-SIM: A Framework for Performance Evaluation of Trusted Platform Modules

Jared Schmitz
jschmit2@cs.binghamton.edu

Jason Loew
jloew@cs.binghamton.edu

Jesse Elwell
jelwell1@cs.binghamton.edu

Dmitry Ponomarev
dima@cs.binghamton.edu

Nael Abu-Ghazaleh [*]
nael@cs.binghamton.edu

Department of Computer Science
State University of New York at Binghamton

ABSTRACT

This paper presents a simulation toolset for estimating the impact of Trusted Platform Modules (TPMs) on the performance of applications that use TPM services, especially in multi-core environments. The proposed toolset, consisting of an integrated CPU/TPM simulator and a set of microbenchmarks that exercise the major TPM services, can be used to analyze and optimize the performance of TPM-based systems and the TPM itself. In this paper, we consider two such optimizations: (1) exploiting multiple TPMs; and (2) reordering requests within the software stack to minimize queueing delays. Our studies indicate that both techniques result in significant performance improvement, especially as the number of concurrent applications using the TPM increases.

1. INTRODUCTION

Security has become a first order design consideration for modern computer and communications systems. For secure system operation, it is essential to defend against dynamic attacks that exploit software vulnerabilities during program execution, but also necessary to ensure secure system bootstrapping by verifying the integrity of any code before it actually runs on the system. To support secure bootstrapping and also to allow remote parties to verify that only authorized code runs on their systems, the Trusted Platform Module (TPM) - a small security co-processor - was recently introduced [14].

TPM chips have already been deployed in over 300 million computers [14]. The introduction of the TPM is arguably the most significant change in hardware-supported security in commodity systems since the development of segmentation and privilege rings [11]. Some commercial applications that explicitly use the TPM have already been developed, including Microsoft's BitLocker Drive Encryption software [3]. A number of research efforts [10, 7] have introduced new software security architectures that extensively rely on TPM support.

In order to foster wide-spread adoption, a primary consideration of TPM design was low implementation cost. Performance issues were an afterthought, as the expected primary usage of the TPM was to measure the BIOS and the operating system kernel during system boot. However, two recent developments require re-examination of TPM performance characteristics and, potentially, even its architecture and placement within the system. First, with the proliferation of multicore and many-core processors, it is possible that several applications running on different cores will request TPM operations simultaneously, causing significant execution delays as these operations have to be serialized. Second, some recent solutions have been proposed that advocate a more active use of the TPM in the course of dynamic program execution [10, 7]. These techniques either experience substantial slowdowns [10] or are outright impossible to implement with current TPMs [7] due to the long latencies of TPM operations.

In this paper, we attempt to answer the following three questions related to TPM performance. We view these questions as seminal steps towards determining how the next generation of TPM systems should be designed.

- What is the impact of currently deployed TPMs on the performance of applications that use them, either in isolation or when executed concurrently with other applications also requiring TPM support?
- For multiple applications requiring TPM services, what is the impact of TPM request scheduling policies on the overall throughput? Should such scheduling be an important design consideration or is the current First Come First Served (FCFS) discipline sufficient?
- What is the performance impact of increasing the number of TPMs in the system?

To quantitatively address the first question, we develop a set of simple C programs that exercise the major TPM services and measures their performance impact. To the best of our knowledge, no such benchmark suites are currently available in the public domain. We evaluate these benchmarks on a quad-core machine equipped with a TPM and isolate TPM delays from the delays incurred on the system busses and within the software stack.

[*] This work was done while Nael Abu-Ghazaleh was on a leave of absence at Carnegie Mellon University in Qatar.

DAC 2011. June 5-10, 2011. San Diego, California, USA.
Copyright 2011 ACM ACM 978-1-4503-0636-2/11/06 ...$10.00.

To address the second and third questions, we integrate a TPM software emulator [2] with a cycle-accurate simulator of a modern microprocessor [5]. We use empirically obtained TPM delays to model TPM latency in the simulator. The tool (called *TPM-SIM*) allows us to accurately explore design alternatives in a realistic environment. We use it to study the impact of using multiple TPMs and to evaluate the impact of request scheduling policies on performance. In general, the combination of the new benchmark set and an integrated CPU-TPM simulator provides a valuable TPM performance analysis tool that can be used both to assess the performance of applications using the TPM as well as to guide the design of future TPMs. Our studies indicate that for four programs simultaneously using TPM services, simple request reordering to minimize queueing delays results in a 35% performance improvement on average, while adding two more TPMs (for a total of three TPMs in the system) improves the performance by a factor of 2.8X.

2. BACKGROUND

This section provides a short summary of TPM, sufficient to understand the operation of the benchmarks that we developed in this study. For more detailed description, we refer the readers to [4, 6].

2.1 TPM and the Root of Trust

The TPM (Trusted Platform Module) was designed to provide a local root of trust for each computing platform, providing both static and dynamic root of trust. Static Root of Trust for Measurement (SRTM) measures code modules at system's boot time before the code is allowed to execute. If a measured code fails (its hash does not match the expected value), the boot process can be blocked by a trust aware bootloader such as TrustedGRUB[16].

Dynamic Root of Trust for Measurement (DRTM) is a mechanism available in implementations such as IBM's Integrity Measurement Architecture in the Linux kernel. It allows measuring the integrity of a code module at any time during system operation. In either case, the code integrity measurements amount to computing a cryptographic hash over the code before it is executed. For SRTM, the measurement is performed on the TPM chip itself, while for DTPM the cryptographic hashes are computed in software and are then extended into a Platform Configuration Register (PCR) in the TPM chip. This mechanism makes it possible to perform TPM-based remote attestation to a third party interested in the platform state.

In PC-based systems, the TPM is located on the Low Pin Count (LPC) bus. Because the TPM has a relatively simple byte-stream interface to the LPC bus, a software stack is specified by the TCG to expose a more flexible set of operations to user-space code. The Trusted Software Stack (TSS), informally known as **T**rou**S**er**S**, provides richer functionality for application developers and abstracts away vendor-specific interfaces. Its most important abstraction is the serialization of requests to the TPM. The TPM can only process one command at a time and will return a busy signal when prompted to perform another. The Trusted Core Services daemon (TCSD) is responsible for this serialization.

2.2 TPM Performance Limitations

In general, existing TPM implementations favor cost over performance. Because of its location on the LPC bus on x86-based architectures, the TPM has a low bandwidth (2.56 MB/s average throughput) [8]. Although the LPC is capable of direct memory access, the TCG specification forbids the TPM from using it at all. There are two main sources of delay within the TPM logic: RSA key generation and encryption calculation. The RSA algorithm [12] requires prime numbers, which are fed into the RSA engine from the TPM's random number generator. Since there is no guarantee that the numbers generated will be prime, a statistical test must be done to check for primality. This is an expensive process and is also non-deterministic if the numbers are truly random. In addition, when the sealing operation (encrypting to TPM state) is performed on data, the RSA encryption must be done internally within the TPM [10], as the current state of the PCRs are recorded in the resulting ciphertext and malicious programs cannot be allowed to spoof PCR values.

Another problem impacting performance relates to the implementation of the Trusted Software Stack through which applications interact with the TPM. Because the TPM only reads and writes byte-streams, all of its operations must be atomic. To enforce atomicity, the TCSD will serialize all requests to the TPM. In the current implementation, these requests are handled in a First Come First Served (FCFS) fashion, which can severely delay applications with short requests and substantially degrade system throughput. Consequently, alternative request scheduling schemes can be more efficient, and we evaluate them in this paper.

To address these performance bottlenecks without changing the internal architecture and functionality of TPM chips themselves, the following options are possible:

- Alternative placement of TPMs closer to the main CPU (such as within the memory controller).
- The use of multiple TPM chips to allow concurrent processing of multiple requests
- Better TPM request scheduling algorithms for multi-programmed workloads.

In the rest of the paper, we present evaluation of these proposals using the combination of a real TPM-based multicore platform and a novel simulation framework (TPM-SIM).

3. TPM BENCHMARK SUITE

Because there were no publicly available TPM benchmarks, the first goal of this study is to develop a benchmark suite that exercises the set of major TPM services. Our goal is not to create large-scale applications; rather, we seek a set of simple mini-benchmarks that intermix the TPM calls with some regular processing. Having a separate benchmark for each of the major services allows us to evaluate them in isolation or in combination (on multiple cores).

3.1 TPM Benchmarks

The benchmarks were written using the Trusted Service Provider interface (TSP). There is one instance of the TSP for each application that wishes to use TPM services, and each TSP establishes a software context with the TCSD. The TSP interface is a simple C API, giving the developer as much flexibility as possible, without relying on knowledge about the internals of the software stack. Every benchmark is written to exercise only one TPM functional unit such as the Random Number Generator (RNG). The benchmarks are described in more detail below.

Key Generation Benchmark (KEY): The first benchmark (called KEY) exercises the secret RSA key generation capability of the TPM. The measured function is *Tspi_Key_CreateKey()*. KEY times how long it takes the TPM to create a user-defined key wrapped with the storage root key. Simplified pseudo-code for the KEY benchmark is shown in Figure 1.

```
...
for(i = 0; i < key_count; i++) {
        ...
        // ONLY call down to TPM is here
        clock_gettime(CLOCK_REALTIME, &start);
        result = Tspi_Key_CreateKey(hSigning_Key, hSRK, 0)
              ;
        clock_gettime(CLOCK_REALTIME, &end);
        ...
}
...
```

Figure 1: C code for Key Benchmark

Key Loading Benchmark (LOAD): This benchmark (called LOAD) loads a single key or a hierarchy of keys into the TPM and then evicts it. The measured function calls are *Tspi_Key_LoadKey()* and *Tspi_Key_UnloadKey()*. It times how long it takes to load and unload a daisy chain of keys to and from the TPM. The calls are measured identically to the method used above.

Remote Attestation Benchmark (PCR): The purpose of this benchmark is to exercise the TPM's remote attestation capability. It measures the time of the function calls *Tspi_TPM_PcrExtend()* and *Tspi_TPM_Quote2()*. Essentially, this benchmark quantifies how long it takes the TPM to extend PCRs and to fulfill a quote request conforming to v1.2 of the TCG specification: these are the basic operations supporting remote attestation. The benchmark allows the user to specify the length of the data to be extended into the PCRs, how many times to extend each PCR, the number of Quote operations to perform, and which PCRs should be involved in these operations. Simplified pseudo-code is presented in Figure 2.

```
Perform required setup of TPM and variables

Query to find out how many PCRs are available

Create Random data to Extend PCRs with

For each PCR that we wish to extend:
        If user specifies data length:
                Hash in software and extend PCR
        Else:
                Extend PCR

Generate AIK for quoting PCRs
For each PCR to be quoted:
        Quote PCR
```

Figure 2: Pseudocode for PCR Benchmark

Timing the Random Number Generator (RNG): This benchmark times the random number generator on the TPM. The measured function calls are *Tspi_TPM_StirRandom()* and *Tspi_TPM_GetRandom()*. Thus it measures the latency of adding to the entropy pool of the onboard RNG and to generate random numbers of variable length.

Data Sealing Benchmark (SEAL): The final benchmark measures the perfomance of the RSA encryption engine, which supports data sealing. It measures the following function calls: *Tspi_Data_Seal()* and *Tspi_Data_Unseal()*. Specifically, this benchmark quantifies the latency of sealing and unsealing a single block of data according to the PCR state.

The TPM is limited in the maximum data size that in can seal in one function call by the modulus of the key used for encryption. It is therefore the application's responsibility to break longer data into blocks, call *Tspi_Data_Seal()* repeatedly and then recombine the encrypted data. However, this behavior introduces cache misses, along with disk and memory I/O into the timings. Instead, we chose to mandate a small input size and call *Tspi_Data_Seal()* on the same input buffer numerous times, reporting the average.

The benchmark code is available at `http://www.cs.binghamton.edu/~tpmsim/download.html`.

4. EXPERIMENTAL EVALUATION

In this section, we present results obtained by evaluating the set of benchmarks described in the previous section on a modern multicore system. The experiments are performed on a a quad-core Intel Core i7 processor that supports Intel's Trusted Execution Technology (TXT). Figure 3 shows the high-level architecture of the experimental platform. The parameters of the TPM chip deployed in the machine used for our experiments are described in Table 1.

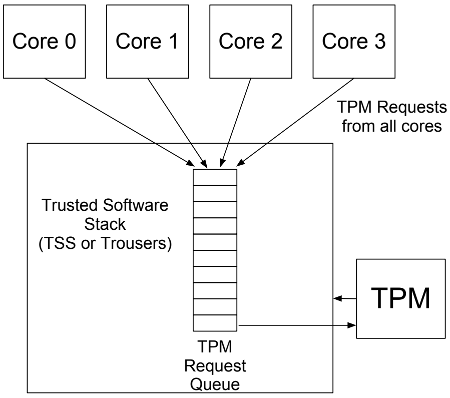
Core 0
Core 1
Core 2
Core 3
TPM Requests from all cores
Trusted Software Stack (TSS or Trousers)
TPM
TPM Request Queue

Figure 3: System Overview

Parameter	**Value**
TCG Compliance	V1.1B / V1.2
Low Pin Count Interface	33Mhz, V1.1
Architecture	1088-bit Modular Arithmetic Processor Hardware-based SHA-1 accelerator True RNGs (FIPS 140-2 / AIS-32 compliant)

Table 1: ST19NP18-TPM Specifications[13]

All benchmarks allow for a user-specified number of iterations and the average and total times for all operations are reported. For our results, we ran 1000 iterations with each data set (e.g. different key lengths or data sizes for sealing) for all tests to ensure statistical accuracy. Only those function calls that require TPM processing are timed. We use the real-time clock to measure TPM response times. TPMs that were manufactured to conform to specifications older than 1.2 [15] are not required to support interrupts, so timing becomes difficult (i.e. threads sleeping versus polling).

Using real time gives the beneficial option of running multiple instances of the benchmark to get the accurate measurements for concurrently running benchmarks.

We first present the latencies for the individual TPM calls made by the benchmarks. In total, 18 different calls were profiled as summarized in Table 2; please note that the call ids in this figure will be used to refer to these calls in the performance analysis figures. Figure 4 presents the latencies incurred by each such call. As seen from the results, by far the longest latency is seen in the key generation benchmark for generating 2048-bit keys. The Load benchmark also exposes relatively high latencies. Conversely, requests for quoting the PCR registers and requests to the random number generator are very fast.

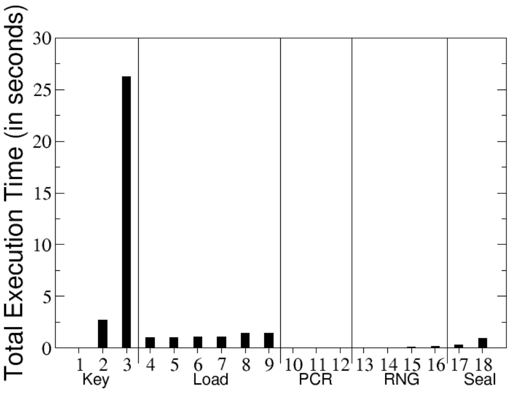

Figure 4: Average Latency of Each Call ID

Call ID for TPM Benchmarks		
Title	Call ID	Description
Key	1	Create a 512-bit RSA Key
	2	Create a 1024-bit RSA Key
	3	Create a 2048-bit RSA Key
Load	4	Loads a single 512-bit Key and evicts it
	5	Loads a single 1024-bit Key and evicts it
	6	Loads a single 2048-bit Key and evicts it
	7	Loads a hierarchy of keys and evicts them
	8	Loads a hierarchy of keys w/o evicting
	9	Loads a single key w/o eviction
PCR	10	Quotes a PCR
	11	Extends a PCR with the default block size
	12	Extends a PCR with random data and size
RNG	13	Generates a 512-bit random number
	14	Generates a 1024-bit random number
	15	Generates a 2048-bit random number
	16	Generates a 4096-bit random number
Seal	17	Seals a file
	18	Unseals a file

Table 2: TPM Call Summary

Table 3 shows the percentage of time that each call spends in the software stack; the rest of the time is the sum of bus traversal and processing delays within the TPM itself. As seen from the table, software delays represent less than 1% of the overall TPM call delay for all of the cases. The experiments also show that the bus delays (including the delays incurred in traversing the LPC bus) are negligibly small, making the TPM processing delays by far the primary culprit in the overall cost of the TPM calls. For example, the fastest TPM call that we experimented with has a hardware delay of 100 milliseconds (200 bytes of data are sent to the TPM in this call). If we assume that the LPC bus has a width of 4 bits and is clocked at 33 MHz, then 400 bus cycles are required to transmit the data with a cycle time of about 30 nsec. Even in this case, the overall bus delay is only 12 microseconds, which is four orders of magnitude smaller than the delays within the TPM itself. For other calls, the difference is even larger. For this reason, we do not distinguish bus delays into a separate component, but rather consider them as part of the TPM delay.

TPM Benchmarks: Software Time		
Benchmark	Call ID	Time Spent in Software
Key	1	0.44%
	2	0.09%
	3	0.01%
Load	4	0.13%
	5	0.13%
	6	0.13%
	7	0.10%
	8	0.09%
	9	0.13%
PCR	10	0.80%
	11	0.30%
	12	0.67%
RNG	13	0.21%
	14	0.13%
	15	0.06%
	16	0.04%
Seal	17	0.83%
	18	0.40%

Table 3: Percentage of Time Spent in Software Stack

5. ALTERNATIVE TPM DESIGNS

The high delays associated with TPM primitives have significant impact on performance for applications that use the TPM. This is especially true in multi-core environments, where many such applications may execute concurrently. To study this impact on system performance, and to enable the exploration of system optimizations, we designed a new cycle-accurate simulation tool (TPM-SIM) that integrates TPM models into existing traditional CPU and memory system simulators. In this section, we give an overview of TPM-SIM design, then use it to study two optimizations to TPM performance.

5.1 TPM-SIM Architecture

TPM-SIM integrates a cycle-accurate simulator of a modern microprocessor with a software emulator of the TPM. Our processor simulator is M-Sim [9], which was built on top of Simplescalar simulator [5]. M-Sim provides cycle-accurate models of out-of-order processors, including support for modeling multithreaded and multicore designs. TPM modeling is provided through a publicly-available software TPM emulator [1] that can be compiled to either simulate a TPM or route calls to a hardware TPM.

The two simulators are interfaced as follows. When M-Sim encounters a TPM call, it pauses the executing thread and invokes the TPM simulator with the corresponding call. The TPM simulator returns the number of cycles required to service this call and the return values (if necessary) that are provided by the TPM. The delays for each call are determined by our experiments on the hardware TPM, as presented in Table 2. When a TPM call is encountered, the application that performed this call is stalled for the number of cycles corresponding to the delay. All other applications continue to execute. Figure 5 graphically demonstrates the interactions and interfaces between the TPM-SIM components.

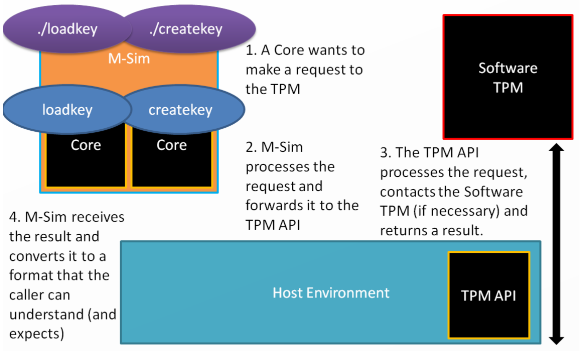
./loadkey
./createkey
M-Sim
loadkey
createkey
Core
Core
1. A Core wants to make a request to the TPM
Software TPM
2. M-Sim processes the request and forwards it to the TPM API
3. The TPM API processes the request, contacts the Software TPM (if necessary) and returns a result.
4. M-Sim receives the result and converts it to a format that the caller can understand (and expects)
Host Environment
TPM API

Figure 5: Architecture of the TPM-SIM Simulator

5.2 TPM System Optimizations

We use TPM-SIM to study two performance optimizations that do not require redesigning of the TPM chip itself. The first study explores re-ordering of pending TPM requests to avoid having short requests block behind expensive ones. Specifically, we compare Shortest Job First (SJF) scheduling to the default First Come First Served (FCFS) implementation in the TSS scheduler. Note that a large number of scheduling schemes (taken from standard operating system scheduling) can be applied here; our goal is to simply illustrate that the choice of scheduling discipline is an important issue for TPM in multi-threaded setting. The best scheduling policy will obviously depend on the workloads being executed by the system in question. Figure 6 shows the performance of workloads composed of multiple applications developed in this study.

Table 4 shows the benchmark combinations used. Since TPM operations are atomic, and the next request cannot be generated until the previous request from the same application is serviced, there is no difference between the various scheduling policies when only two applications are running (because at most one request is in the queue at any time). Therefore, we omit 2-threaded experiments from this study. For the rest of the simulated workloads, SJF provides significant improvement in performance (35% on average across all simulated mixes with a maximum improvement of 81% for one of the 4-threaded mixes). We note that the performance gains increase with the number of concurrent applications sharing the TPM. In this graph, performance is calculated as the total number of cycles spent executing these benchmarks (including both calls to the TPM and the rest of the code) on all cores.

As a final note, we mention that this study only considered non-preemptive service within the TPM. However, there does exist a function in the software stack to issue a cancellation call to the TPM and it is possible to use Tddli_Cancel() (from the lowest layer of the software stack) as a very primitive "preemption" if it is augmented in such a way that it returns the canceled request back into the request queue. In the current TPM implementation, jobs are not restartable, so exploration of this modification is left for future studies.

Table of Benchmark Mixes		
Number of Threads	Mix Number	Benchmarks Mixed
2 Threads	1	Key - Load
	2	Key - PCR
	3	Key - RNG
	4	Key - Seal
	5	Load - PCR
	6	Load - RNG
	7	Load - Seal
3 Threads	8	Key - Load - PCR
	9	Key - Load - RNG
	10	Key - Load - Seal
	11	Key - PCR - RNG
	12	Key - PCR - Seal
	13	Load - PCR - RNG
	14	Load - RNG - Seal
	15	PCR - RNG - Seal
4 Threads	16	Key - Load - PCR - RNG
	17	Key - Load - PCR - Seal
	18	Key - Load - RNG - Seal
	19	Key - PCR - RNG - Seal
	20	Load - PCR - RNG - Seal

Table 4: Concurrent Mixes of Applications

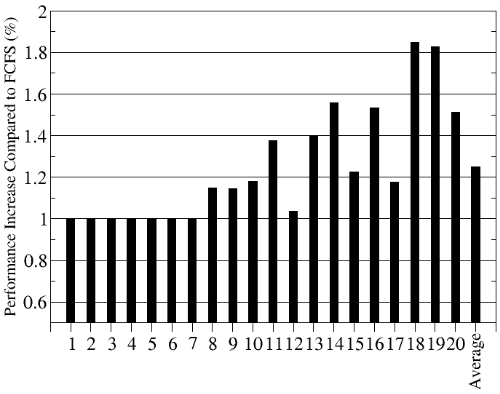

Figure 6: SJF vs. FCFS Scheduling by Mix Number

The next optimization we evaluated is the use of multiple TPMs located on the same LPC bus. This way, the servicing of multiple requests can proceed concurrently. Figure 7 shows the relative performance of the system with multiple TPMs (we considered 2, 3, and 4 TPMs) with respect to the performance of a single-TPM machine. Naturally, each application can only use one TPM in these experiments (to maintain consistency), but requests from multiple applications can be executed on different TPMs concurrently.

On average across all 4-threaded mixes[1], the system with 2 TPMs results in a 1.9X speedup. If 3 TPMs are used, a 2.7X speedup is observed, while using 4 TPMs results in a 3.3X performance improvement. In practice the magnitude of the

[1] For readability and due to space constraints, we only show the graphical results for the scenarios with four concurrently running applications.

performance gains will be determined by the frequency of TPM calls and the proportion of time that applications wait for the TPM services. In our experiments, the execution time was dominated by the TPM calls.

For two concurrent benchmarks, the speedup when using two TPMs is 81% on average. For three-threaded workloads, adding a second TPM improves performance by 72% and adding the third TPM achieves a 1.5X performance improvement on average.We note that placing the TPM closer to the processor (i.e. within the memory controller) does not result in performance improvements unless the TPM chip architecture changes, because the bus latency is negligible compared to the delay within the TPM itself.

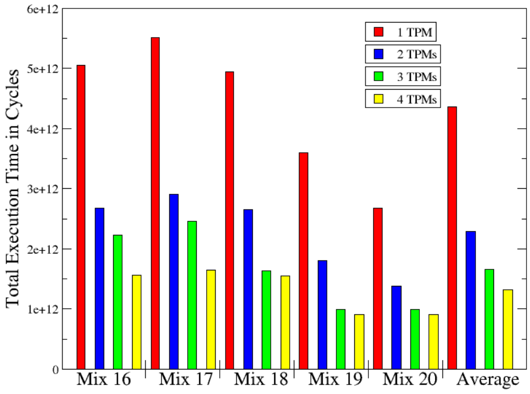

Figure 7: Performance of Multiple TPMs

Other studies that explore alternative TPM designs and organizations are enabled with TPM-SIM. The tool (both the developed benchmarks and the integrated simulation environment) is publicly available at
`http://www.cs.binghamton.edu/~tpmsim`.

6. CONCLUDING REMARKS

The contribution of this paper is to develop a toolset and a collection of benchmarks to evaluate TPM performance under a range of realistic use cases and design options. Another contribution is the initial investigation of two performance optimizations: reordering of TPM requests within the software stack and the use of multiple TPMs. We demonstrate that both techniques can lead to significant performance improvements, especially as the number of concurrent applications using the TPM increases. Our studies indicate that for four programs simultaneously using TPM services, simple request reordering to minimize queueing delays results in an average of 35% performance improvement, while adding two more TPMs improves the performance by a factor of 2.7X. The developed toolset will provide designers the environment for evaluating future generations of TPM designs. The tool will be made publicly available, encouraging widespread reevaluation of TPM performance.

7. ACKNOWLEDGEMENTS

This material is based on research sponsored by Air Force Research Laboratory under agreement number FA8750-09-1-0137 and by National Science Foundation under grant number CNS-0958501. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies and endorsements, either expressed or implied, of Air Force Research Laboratory, National Science Foundation, or the U.S. Government. Jared Schmitz and Jesse Elwell were partially supported by the National Science Foundation REU program under grant CNS-1005153.

8. REFERENCES

[1] E. D. Berger and B. G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *In Proc. of PLDI'06*. ACM, June 2006.

[2] S. Berger, R. Caceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vtpm: Virtualizing the trusted platform module. In *Usenix Security Symposium*, July 2006.

[3] Bitlocker drive encryption - windows 7 features, 2010. Available online at: `http://windows.microsoft.com/en-US/windows7/products/features/bitlocker`.

[4] D. Challener, K. Yoder, R. Catherman, D. Safford, and L. Van Doorn. *A Practical Guide to Trusted Computing*. IBM Press, 2008.

[5] D.Burger and T.Austin. The simplescalar toolset: Version 2.0, June 1997.

[6] D. Grawrock. *Dynamics of a Trusted Platform: A Building Block Approach*. Intel Press, 2009.

[7] A. K. Kanuparthi, M. Zahran, and R. Karri. Feasibility study of dynamic trusted platform module. In *Proc. IEEE ICCD*, October 2010.

[8] Low pin count interface specification, August 2002. Available online at: `http://www.intel.com/design/chipsets/industry/lpc.htm`.

[9] M-sim: The multi-threaded simulator: Version 3.0, September 2010. Available online at: `http://www.cs.binghamton.edu/$\sim$msim`.

[10] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proc. of the ACM EuroSys in Computer Systems (EuroSys)*, Apr. 2008.

[11] B. Parno, J. M. McCune, and A. Perrig. Bootstrapping trust in commodity computers. In *31st IEEE Symposium on Security and Privacy*, May 2010.

[12] R. L. Rivest, A. Shamir, and L. M. Adleman. US patent 4,405,829: Crytographic communication system and method, 1983.

[13] St19np18-tpm specification, 2006. Available online at: `http://www.st.com/stonline/products/literature/bd/12803/st19np18-tpm.htm`.

[14] Replacing vulnerable software with secure hardware, 2008. Available online at: `http://www.trustedcomputinggroup.org/resources/replacing_vulnerable_software_with_secure_hardware`.

[15] Pc client specific tpm interface specification (tis), July 2005. Version 1.2 available online at: `http://www.trustedcomputinggroup.org/resources/pc_client_work_group_pc_client_specific_tpm_interface_specification_tis_version_12`.

[16] Trustedgrub, August 2010. Available online at: `http://sourceforge.net/projects/trustedgrub/`.