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

Performance Evaluation of TPM-based Digital Wallets

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

Roopa Vishwanathan
Dept of Computer Science and Engineering
University of North Texas
Denton, TX 76203
Email: rv0029@unt.edu

Abstract—**The Trusted Platform Module (TPM) manufactured and promoted by the Trusted Computing Group (TCG) [1] is an embedded chip found on laptops and desktops and is fast becoming a standard component on most computing systems. Previous work in trusted computing-based research has suggested that we can facilitate e-commerce transactions such as digital wallets, e-cash and offline payments using TPMs. In this paper, we experimentally evaluate the performance of digital wallets that are implemented using TPM-based functionality. In particular, we study how different approaches of implementing TPM-based monotonic counters that are used in the digital wallet application can perform with respect to time and cost, experimentally evaluate and compare how each of these approaches perform under conditions like different workloads and distributions, and how they perform in non-standard environments where one would need a simulator to estimate performance.**

I. Introduction

In recent years, with the increasing availability of Trusted Platform Modules (TPMs) on laptops and desktops, researchers have devised many interesting applications that use TPMs, including ones that use a standard TPM as well as applications that use enhanced TPMs. A TPM is a hardware chip embedded on the motherboard of a computing device that provides a set of commands, operations or capabilities to the host platform on which it is resident. The main components of a standard TPM are a crypto engine, a block of non-volatile storage, a set of 160-bit PCR registers that act as volatile memory, an I/O port and an 8-bit processor. The crypto engine provides a set of standard algorithms that include RSA for public-key encryption, SHA1 for hashing, HMAC for signed Message Authentication Codes (MAC) and a random number generator. Besides this, the TPM has a host of other capabilities such as Direct Anonymous Attestation (DAA), migration of keys and data, secure, logged and exclusive transport sessions, monotonic counters, delegation of authority from the owner, processes, and more.

With commercial transactions between users and online merchants growing at an ever-increasing rate, a digital wallet is a useful application which helps a single user buy goods from several online merchants without active participation of an online credit/debit card issuer. In a digital wallet application, a user may have a *credit issuer* who sets a credit limit for the user (e.g., a credit card company issuing a credit card), and stores a copy of the current credit balance on the user's machine. The user might then use the card to buy goods from online merchants. Whenever a user makes a purchase from a merchant, the merchant would want some assurance that the user is not overspending his credit limit *without* having to communicate with the credit-card issuer who may not always be online. An easy way to do this is for the credit-card issuer to sign the user's balance, and the online merchant can verify it easily without the credit-card issuer's help. This would prevent a user from arbitrarily changing his credit balance, but wouldn't prevent a user from replaying old signed balances (balances with a valid signature by the credit issuer). If the user's device is equipped with a TPM chip, Sarmenta et al. [2] presented a way to deal with this problem by implementing a large number of *virtual monotonic counters* using the user's computing device. Whenever a merchant wants to update a user's credit balance stored on its untrusted device equipped with a TPM, the merchant first increments the monotonic counter on the user's device and signs a token which includes the new monotonic counter value and the new value of the user's balance. When the user deals with the next merchant, the new merchant checks whether the value of the monotonic counter on the user's device matches that which was signed in the token. This prevents a malicious user from replaying old credit balances and overspending their credit limit. Sarmenta et al. [2] suggested and briefly outlined ways in which the virtual counter mechanism can be used to support various e-commerce applications; in this paper, we show how to realize the *digital wallet* application with virtual counters and experimentally evaluate the performance of different schemes to generate virtual counters for the digital wallet application.

A. Our Contributions

Our contributions in this paper can be summarized as follows:

1) We demonstrate a methodology for estimating performance of TPM-based protocols in non-standard environments, including use of a simulator to estimate performance. To this end, we present a protocol to

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

realize digital wallets using TPM-based virtual monotonic counters [2] and do an experimental performance evaluation of the same.
2) The performance analysis presented in this paper should be of use to designers of other applications which use TPM-based counters. Since we also evaluate performance using a simulator and estimate what the performance would be if extra commands were added to a real TPM, one can weigh the benefits gained against the cost of adding functionality to the TPM specification for any particular application.

II. Related work

The idea of using trusted hardware for secure e-commerce transactions is an old one and had been first introduced by Chaum and Pedersen [3] for e-cash schemes. Some other applications of using trusted hardware, specifically TPMs are in securing peer-to-peer networks [4], certificate management, and private information retrieval [5]. Sarmenta et al. [2] used a TPM to support a large number of virtual monotonic counters which can then be used in various applications like digital wallets, offline payments and virtual trusted storage. In a subsequent paper [6], they discuss how to prevent forking attacks and replay attacks by an untrusted server in the virtual trusted storage application. Extending the concept of trusted computing beyond TPM chips, a new trusted computing infrastructure called Flicker [7] has been proposed which can be used to execute security-sensitive code in an untrusted environment (untrusted OS, BIOS, memory). Flicker leverages the capabilities of trusted components built into commodity processors such as AMD's SVM (secure virtual monitor) and Intel's TXT (trusted execution technology) along with that of a TPM chip to provide a secure environment for the execution of security-sensitive code.

On the theoretical side, the idea of using trusted hardware has been studied by Katz [8], Damgard et al. [9] who have presented ways to generate hardware tokens which can then be used in various applications. Gunupudi and Tate have proposed ways to use trusted hardware, and TPMs in particular to improve the efficiency of *Oblivious Transfer* [10] which is an operation used in various cryptographic protocols. More recently Goldwasser et al. [11] have put forth the idea of *one-time programs* which are programs that can only be run once and lend themsleves naturally to e-tokens or e-cash schemes.

To put things in context, in this paper, we focus on examining some of the more practical aspects of designing TPM-based e-commerce applications such as a digital wallet. To this end, we study and estimate the performance of TPM-based monotonic counters on real TPMs and a TPM simulator. Virtual monotonic counters are useful building blocks which can be used by online e-commerce transactions, and previous work in this area has not provided any detailed experimental results on how costly it is to implement counter operations in terms of time. Since time and cost are both important considerations for any application designer, this work could help application designers weigh the benefits of using TPMs against the cost of adding functionality to the TPM.

III. Digital Wallet Application and Implementation of Virtual Counters

Since we use virtual monotonic counters as building blocks in designing TPM-based digital wallets, we first define the notion of a virtual monotonic counter below based on [2]:

Virtual Monotonic Counter: A virtual monotonic counter is a mechanism implemented in hardware and/or software that stores a value and provides two commands to access that value: READ and INCREMENT. The read command returns the current value of the counter, and the increment command increments the value and returns the new value of the counter. The properties that a virtual monotonic counter is expected to have are that the counter value should be *non-volatile*, should be *irreversible* and the read and increment commands should both be *atomic*. In addition, we require that the counter be *tamper-evident*, so if an adversary causes the counter to behave incorrectly, we should at least be able to detect the anomaly, if not prevent the adversary from tampering with the counter. Additionally, the counter should return an *unforgeable* execution certificate on the execution of a READ or INCREMENT operation.

A. Application

Although virtual monotonic counters can be used in a variety of applications, in this paper we focus on digital wallets. In a digital wallet application, we have a credit issuer (e.g., a credit card company) which stores a copy of a buyer's credit balance on the buyer's computing device equipped with a TPM. Everytime the buyer makes a purchase from a merchant and pays from its available credit balance, the merchant should not have to check the credit balance with the credit issuer, and needless to say, the merchant would not trust the buyer to report the correct value of its credit balance. Whenever the buyer purchases some goods from a merchant, the merchant would update the credit balance on the buyer's device by subtracting the price of the goods purchased. The purpose of using a TPM here is that the merchant does not have to go through a centralized credit issuer every time the merchant wants to check if the credit balance stored on the buyer's device is correct or not. The way to use a TPM, as given in [2], is to have a large number of virtual monotonic counters on the buyer's device, each counter tied to a buyer's credit balance. Assuming the counters (and hence the TPM) are trusted by the credit issuer and the merchant, whenever the buyer buys goods from a merchant, the merchant decreases the credit balance stored on the buyer's device, increments the virtual counter value associated with it, and signs a token with the new counter value and the credit balance. Whenever the buyer buys goods from another merchant, the new merchant simply has to check if the credit balance and the counter value match those on the signed token or not. Table I depicts an outline of the multi-round protocol involving the merchant and

TABLE I
ILLUSTRATION OF DIGITAL WALLET MECHANISM

Round 1: Buyer initiates a transaction, buys goods, and is ready to pay. Sends the following tuple to the Merchant: $S = Sign(ctrID \| \| ctrVal \| \| currBalance)$
Round 2:Merchant gets $ctrVal$ from S and sends a request to the Buyer to read the current counter value
Round 3: Buyer passes on request to its TPM, which returns the current counter value currCtrVal, and issues a signed execution certificate, $execCert = Sign_{AIK}(currCtrVal)$. Buyer sends execCert to Merchant
Round 4:Merchant gets currCtrVal from execCert, If $(currCtrVal == ctrVal)$, computes $newCtrVal = currCtrVal + 1$, decrements balance, passes request to Buyer to increment counter, Creates $S_1 = Sign_{Merchant}(ctrID \| \| newCtrVal \| \| newBalance)$, returns S_1 to Buyer
Round 5:Buyer stores S_1 for future use, passes request to its TPM to increment its counter with $newCtrVal$

buyer in the digital wallet application. Next, we proceed to explain how virtual counters are implemented.

B. Implementation of virtual counters

Let n_c be the total number of virtual counters available that are managed by the buyer's TPM, and consider a request sequence $r_1, r_2, r_3, \ldots$, where each request is a pair $r_i = (c_i, op_i)$ consisting of a counter name c_i and an operation op_i such as INCREMENT or READ. When considering a request, one important measure is the number of requests since the last change to the same counter, so we define p_t to be the request number of the most recent such time:

$$p_t = \max\{i \,|\, c_i = c_t \text{ and } op_i \in \{\text{CREATE}, \text{INCREMENT}\} \quad i < t; \text{ or } i = 0\}.$$

We use p_t to define the set U_t, the set of all requests since p_t that update counters other than c_t:

$$U_t = \{i \,|\, p_t < i < t \text{ and } op_i \in \{\text{CREATE}, \text{INCREMENT}\}\}.$$

The schemes that we evaluate in this paper use a combination of operations on the TPM and on the host CPU (a much more powerful processor). To characterize times required by various critical operations, we define the following notation:

T_{hh}	=	Time for the host CPU to hash one block
T_{th}	=	Time for TPM to hash one block
T_{ts}	=	Time for TPM to create a signature
T_{hv}	=	Time for host CPU to verify a signature
T_{ti}	=	Time for TPM to increment a physical counter
T_{tr}	=	Time for TPM to read a physical counter
T_{tt}	=	Time to build a logged transport session in the TPM
S_s	=	Size of a signature
S_h	=	Size of a hashed value
S_c	=	Size of a counter

We use the schemes outlined by Sarmenta et al. [2] for realizing the virtual counter mechanism using TPMs: the *log-based* scheme and the *hash-tree based* scheme. Below we describe both the schemes and the request sequences of counters.

1) Log-based scheme: In the *log-based* scheme, the TPM's physical counter is used as a global clock and the value of a virtual counter at a particular instant is defined as the value of the physical counter when the virtual counter was last changed. Note that the ever-increasing global counter insures that all virtual counters are monotonically increasing, but virtual counters do not necessarily increase by 1. All requests are processed by the buyer's TPM in an exclusive and logged transport session. For an INCREMENT request, the buyer increments the TPM's physical counter and returns a certificate called the *increment certificate* that consists of a structure containing the virtual counter's value, the counter ID, the antireplay nonce given by the merchant, and the hash of the transport session log which consists of all the commands during the session. The TPM signs this structure using its Attestation Identity Key or AIK, so the merchant can verify the certificate by checking the counter ID and nonce, the contents of the transport log, and verifies the signature as coming from a valid, certified AIK.

For a READ request, the buyer returns the virtual counter's current value along with a *read certificate* which the merchant can verify. The read certificate consists of the current value of the physical counter, the virtual counter's last increment certificate and all the increment certificates issued to all the counters since the time of that particular counter's last increment. All these certificates bunched together make up the *log* from where the scheme gets its name. Reporting that the value of a particular virtual counter is X is equivalent to saying that the last request to change this virtual counter was when the physical counter had value X, which can be verified from this log. Note that this operation always requires a read of the TPM's physical counter (for getting its current value) which takes place inside an exclusive and logged transport session.

This process is illustrated below, for a simple situation with 4 virtual counters and 10 requests (showing INCREMENT requests only, since READ requests do not change the state of the system).

Physical Counter Value:	1	2	3	4	5
Request:	INC VC4	INC VC2	INC VC4	INC VC1	INC VC3

Physical Counter Value:	6	7	8	9	10
Request:	INC VC2	INC VC2	INC VC4	INC VC1	INC VC4

At the end of this request sequence, the four virtual counters have the values shown in Table II

In order to prove that the value of VC1 is 9, the host only needs to prove that the current value of the physical counter is 10, that VC1 was incremented at step 9 (using the saved increment certificate), and that VC1 was *not* changed at step

TABLE II
RESULT OF REQUEST SEQUENCE

Counter	Value
VC1	9
VC2	7
VC3	5
VC4	10

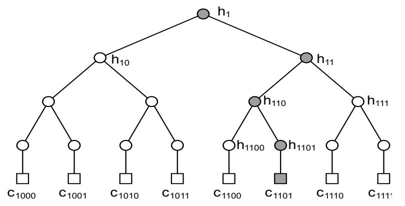

Fig. 1. Hash-tree Based Scheme Illustration — based on Figure 2 in Sarmenta *et al.* [2]

10 (by providing the increment certificate for VC4 at step 10). As another example, to prove that the value of VC3 is 5, the host must supply the 6 increment certificates for steps 5–10.

We estimate the costs of the operations as follows:

Reset/Create/Increment:	$T_{th} + T_{ts} + T_{ti} + T_{tt}$
Read:	$\lvert U_t \rvert \cdot T_{hv} + T_{tr} + T_{tt} + T_{ts}$

The size of the counter attestation is $|U_t|(S_h + S_s)$.

2) Hash-tree based scheme: The *hash-tree based* scheme is based on a popular data structure used in the cryptographic literature: a Merkle hash tree which is used for verifying the integrity of a set of data objects. A Merkle hash tree is a binary tree with data objects as its leaves and the values of nodes at each subsequent level are the concatenation of the cryptographic hashes (e.g. SHA-1) of their children. Thus, the root of the tree is a cryptographic hash that depends on the entire tree. This data structure is illustrated in Figure 1 in which, for example, the node labeled h_{110} would hold hash value $h_{110} = SHA1(h_{1100}||h_{1101})$.

To implement virtual counters using this mechanism, the counters are stored in the leaves of the hash tree, and the root of the hash tree is stored in and maintained by the TPM. Note that the leaves and the intermediate nodes of the tree can be stored in the host's untrusted memory. To prove to a remote party the current value of a counter such as c_{1101}, the information in the leaf (the counter value) is sent to the remote party, along with the hash values from siblings of all nodes on the path from c_{1101} to the root, which allows the remote party to compute all the hash values in the path to the root, culminating in h_1. The remote party then challenges the TPM to attest the value h_1 using a random nonce, and the TPM signs the nonce and the current value of h_1 with an Attestation Identity Key (AIK).

To update a counter, only those hash values on the path from the counter to the root need to be verified and modified — for example, to increment counter c_{1101} only the shaded nodes in Figure 1 are involved. Therefore, if the system is maintaining n_c virtual counters, only $\log_2 n_c$ hash values need to be checked and modified with any update operation. For additional details about the hash-tree based scheme, please refer to Sarmenta *et al.* [2].

We estimate the costs of the operations as follows:

Reset/Create/Increment:	$(2\log_2 n_c)\ T_{th} + T_{ts}$
Read:	$(\log_2 n_c)\ T_{th} + T_{ts}$

The size of the counter attestation is $S_c + (\log_2 n_c)\ S_h + S_s$.

3) Comparison: The log-based scheme has the advantage that it can be implemented on existing TPMs since it does not require any additional commands beyond the TPM v.1.2 specification. But the TPM operations used in the log-based scheme can slow down its performance considerably: the READ and INCREMENT virtual counter operations in the log-based scheme always require a read and increment respectively of the TPM's physical counter which can take quite a bit of time on present-day TPMs - $0.028 - 0.041$ sec for read and $0.44 - 0.70$ sec for the increment operation, and the construction of the exclusive and logged transport session takes $0.06 - 0.13$ sec. Apart from this, each transport session log needs to be signed and a signature operation can take around 0.8 sec on present-day TPMs. This bears out in our experiments where the hash-tree scheme outperforms the log-based scheme in both the read and increment counter operations. Another major drawback of the log-based scheme is that the length of the log can be unbounded in time — the length of the log returned after performing a read operation for a particular virtual counter really depends on how far apart two consecutive increment requests for that virtual counter were. So if a virtual counter i does not get incremented for a long time while other virtual counters get incremented many times, the log for i would have to include the increment certificates of all other virtual counters. In other words, the latency of the read counter operation can be unbounded in practice. This problem manifests itself in applications where the counter requests aren't sequential and follow uneven probability distributions like the *zipf* distribution where 20% of the counters get incremented 80% of the time. In such situations, the time taken for creating the log and the size of the log can be unbounded in time. Lastly, we cannot implement arithmetic counters with the log-based scheme. We can only implement counters that are monotonically increasing, but cannot predict what the next value of a particular counter is going to be. In the hash-tree based scheme, we have bounded communication and computation costs as each update operation requires $O(\log n)$ hashes to update the root hash in a tree of n leaves (each leaf is a virtual counter). Also, we can implement arithmetic counters and *clobs* (count-limited objects) using the hash-tree based scheme. The main disadvantage of the hash-tree based scheme is that it cannot be implemented using current TPMs and requires an extension to the TPM v.1.2 specification. But this addition might be something worth considering given the efficiency and low latency of the hash-tree based scheme as compared to the log-based scheme.

IV. Experimental Setup and Request Sequences

For our experiments, we consider a setup where we have a machine with a TPM that acts as a buyer's device equipped with 1024 virtual monotonic counters. These counters are used by various applications and parties, including the merchants who issue counter requests to read or increment the virtual counters stored on the buyer's machine. We generate two kinds of counter request sequences: round-robin requests in which counters are read and incremented sequentially and randomized requests where counter requests are generated randomly with a few counters getting read/incremented more than others. The sequences in which we issue counter requests are as follows:

1) Round-robin requests: We have 1024 virtual counters with requests for reading/incrementing each counter and measure the time taken for the increment/read requests in a sequential manner.
2) Randomized requests: We have 1024 virtual counters with incoming requests for reading/incrementing them based on requests generated from the *zipf* distribution [12]. The *zipf* distribution is a power law distribution that is used to characterize real-world phenomena like occurences of words in texts, Internet browsing habits and user discussion forums, to name a few. A zipf distribution models the scenario where a few occurences of a particular event are extremely common, while a large number of occurences of events are very rare. For example, in natural languages, some words are used much more frequently than others (e.g., the word "the"), while a majority of words never get used. In web browsing habits of users, a few websites get visited multiple times every day (e-mail providers, search engines), while a vast majority of websites are visited rarely by the user, if at all. In e-commerce applications, a user might regularly purchase goods from a few trusted merchants or dedicated online stores that sell popular consumer products, while a vast majority of merchants that the user might not have heard of or doesn't trust are never visited. We simulated this scenario by having a few virtual counters incremented/read a lot of times while a majority of the virtual counters would rarely get read/incremented. The distribution function for zipf's law [12] is given as follows where ρ is a positive parameter, $\zeta(z)$ is the Riemann zeta function and $H_{(n,r)}$ is a generalized harmonic number:

$$D(x) = \frac{H_{x,\rho+1}}{\zeta(\rho+1)}$$

V. Experimental results

We used TPM/J developed by Sarmenta et al. [13], a Java-based front-end interface, for communicating with both the real TPM and the TPM simulator which interfaces with the real TPMs' and the TPM simulator's device driver. We have used v.0.5 of the TPM simulator by Mario Strasser [14] in our hash-tree experiments and added the required command, TPM_ExecuteHashTree for simulating the hash-tree to it. For experiments on the log-based scheme on a real TPM, we have used TPMs from four major manufacturers: Atmel, Infineon, STMicro and Winbond. We have run the log-based scheme with both a real TPM and the TPM simulator — one of the reasons for doing so is to give a fair idea of how accurately the simulator can model a real TPM. Specifically, the hash-tree based scheme cannot be run on a real TPM and needs to be run entirely on the simulator, the results of the log-based scheme run on a simulator gives an indication of how accurate the hash-tree results are. In general, the benefit of doing this is: by estimating what the performance would be if new commands were added to a TPM, one can weigh the benefits against the cost of adding functionality to the TPM specification.

A. Round-robin Counter Requests

The time taken for the increment and read virtual counter operations with round-robin requests for the log-based scheme are shown in Table III, Table IV and Table V, Table VI respectively. The tables show the time taken for increment and read virtual counter operations on real TPMs as well as on a TPM simulator run with the performance profile of the corresponding TPM manufacturer. These performance profiles were part of work due to Gunupudi and Tate [15] in which they had benchmarked profiles of four real TPMs: Atmel, STMicro, Infineon and Winbond on the TPM simulator. The performance profiles are simply time delays introduced in the simulator (which runs on the host CPU) in order for it to better mimic the speed of a real TPM, since real TPMs' processing speed is far slower than that of a standard desktop CPU. There are a few operations that contribute to the total time taken in the increment virtual counter and/or read virtual counter requests in the log-based scheme that aren't shown in the tables, viz. the signature operation, the time taken for generating a random anti-replay nonce, the time taken for constructing an exclusive and logged transport session and the time taken for incrementing and reading a TPM's physical counter. This is because the unit time taken for each of these operations is the same regardless of the number of counters being read or incremented. All times given in the tables are in seconds, represent the total time taken for the read/increment operations, and the time difference between the real TPM and the simulator ("Error") is taken as a relative value; in cases where the relative error is $\leq 0.005\%$, we give it as 0.

The hash-tree based scheme cannot be implemented on a real TPM and needs to be run on a TPM simulator. For measuring the hash-tree read/increment counter request times, we ran the simulator with the performance profiles of four real TPMs: Atmel, Infineon, STMicro and Winbond. Table VII and Table VIII show the time taken for the Increment and Read virtual counter operations in the hash-tree based scheme respectively. As in the log-based scheme, there are a few operations (signatures, hashing, etc.) that contribute towards the total time in the hash-tree based scheme that aren't shown in Table VII and Table VIII since the unit time taken for

TABLE III
TIME TAKEN FOR VIRTUAL COUNTER INCREMENT OPERATION IN LOG-BASED SCHEME - ATMEL, INFINEON TPMS

Ctrs	Atmel			Infineon		
	Sim	Real	Error	Sim	Real	Error
2	1.80	1.87	3.74%	0.89	0.98	9.18%
4	3.69	3.75	1.6%	2.09	2.19	4.56%
8	7.44	7.53	1.19%	4.00	4.11	2.43%
16	14.94	15.06	0.72%	11.41	11.51	2.43%
32	26.50	26.60	0.37%	20.10	20.21	0.49%
64	53.17	53.30	0.00%	48.90	49.00	0.20%
128	106.17	106.30	0.00%	91.71	91.82	0.11%
256	211.93	212.51	0.27%	198.75	198.88	0.06%
512	423.09	424.09	0.23%	385.98	386.11	0.03%
1024	846.78	847.79	0.11%	800.02	800.16	0.01%

TABLE IV
TIME TAKEN FOR VIRTUAL COUNTER INCREMENT OPERATION IN LOG-BASED SCHEME - STMICRO, WINBOND TPMS

Ctrs	STMicro			Winbond		
	Sim	Real	Error	Sim	Real	Error
2	1.60	1.63	1.84%	1.05	1.11	5.40%
4	3.06	3.25	5.84%	2.09	2.16	3.24%
8	6.92	6.49	8.78%	4.34	4.43	2.03%
16	12.98	13.02	7.98%	8.58	8.67	1.03%
32	23.66	25.96	8.75%	17.42	17.52	0.57%
64	49.81	51.95	4.11%	34.39	34.40	0.29%
128	106.01	104.03	2.85%	69.24	69.38	0.20%
256	210.06	208.25	0.86%	136.50	136.76	0.19%
512	415.11	410.11	1.21%	273.58	273.81	0.08%
1024	844.85	839.83	0.59%	383.661	384.00	0.08%

TABLE V
TIME TAKEN FOR VIRTUAL COUNTER READ OPERATION IN LOG-BASED SCHEME - ATMEL, INFINEON TPMS

Ctrs	Atmel			Infineon		
	Sim	Real	Error	Sim	Real	Error
2	1.82	1.82	0%	1.44	1.44	0%
4	3.60	3.60	0%	2.64	2.64	0%
8	7.14	7.14	0%	5.17	5.17	0%
16	14.26	14.27	0%	10.33	10.33	0%
32	28.59	28.59	0%	20.65	20.66	0.04%
64	57.05	57.06	0%	41.30	41.30	0%
128	113.98	113.99	0%	82.59	82.57	0.02%
256	287.76	227.77	0%	165.22	165.23	0%
512	455.41	455.42	0%	330.51	330.52	0%
1024	910.07	910.08	0%	661.56	661.57	0%

TABLE VI
TIME TAKEN FOR VIRTUAL COUNTER READ OPERATION IN LOG-BASED SCHEME - STMICRO, WINBOND TPMS

Ctrs	STMicro			Winbond		
	Sim	Real	Error	Sim	Real	Error
2	1.88	1.89	0.5%	1.18	1.18	0%
4	3.69	3.78	2.11%	2.26	2.26	0%
8	7.50	7.54	0.5%	4.48	4.48	0%
16	14.89	15.08	1.25%	8.91	8.91	0%
32	29.88	30.16	0.92%	18.16	18.16	0%
64	60.10	60.25	0.24%	36.23	36.23	0%
128	118.51	120.57	0.01%	72.30	72.30	0%
256	236.81	240.95	0.01%	144.52	144.52	0%
512	474.66	481.89	0.01%	282.63	282.63	0%
1024	955.04	963.48	0%	564.78	564.78	0%

these operations remain the same regardless of the number of counters being read or incremented. All time units are seconds and represent the total time taken for the entire increment operation.

B. Performance Analysis of Round-robin Requests

From the tables, it is clear that the hash-tree based scheme outperforms the log-based scheme by a significant margin in both, the read and increment virtual counter operations. This is mainly because the log-based scheme involves TPM operations that are time-consuming such as reading/incrementing a physical counter and constructing an exclusive, logged transport session (the operations common to both the schemes are the TPM generating a hash, and a TPM signature operation). The time taken for an increment physical counter operation is 0.500-0.703 sec on an Atmel TPM, 0.449 sec on an STMicro TPM, 0.509 sec on an Infineon TPM and 0.6 sec on a Winbond TPM. The time taken for constructing a transport session is 0.06 sec on an Atmel TPM, 0.087 sec on a STMicro TPM, 0.137 sec on an Infineon TPM and 0.03 sec on a Winbond TPM. Time taken for a signature on an Atmel TPM is around 0.65-0.9 sec, on an Infineon and STMicro TPM is 0.8 sec, and on a Winbond TPM is 0.6 sec. If we consider the formulae we had introduced in Section III-B1, the cost of a read operation is $|U_t| \cdot T_{hv} + T_{tr} + T_{tt}$; in the round-robin request sequence, the size of $|U_t|$ is 1 (assuming this is the first round of increments), since we have just 1 increment certificate — the current time certificate in the log. So, the total cost of the read operation is $T_{hv} + T_{tr} + T_{tt}$. This is validated by our experiments where for example, the cost of a read operation for 2 counters on an Atmel TPM is $2*(0.060+0.041+0.8) = 1.801$ sec, or the time taken for 1024 counters is $1024*(0.060+0.041+0.8) = 922.624$ sec. We can see the same trend for the increment counter operations where the difference between our model and the actual experimental results is about 0-15 sec. We have not considered the cost for preliminary operations like loading a key into the TPM, initializing the driver and going through the layers of Java libraries in our formula. We estimate that the few seconds difference between our model and the actual experimental results are due to these issues. The hash-tree based scheme has a much lower margin of error between our estimate from the model and the actual experimental results, compared to the log-based scheme. This is partly due to having a larger set of TPM commands to execute in the log-based scheme — signature, hash, transport, physical counter read/increment, as opposed to just two TPM commands in the hash-tree based scheme: signature and hash. Also, the hash-tree experiments were run on the simulator which simulates each operation with exactly the same time every time we run it. In the real TPM, this isn't necessarily the case, e.g., an Atmel TPM can take anything between 0.6 and 0.9 sec to create a signature, and may not give the exact same time each time if we run it for 1000 times. We estimate that if TPM_EXECUTEHASHTREE were an actual TPM command (which can be run on real

TABLE VII
TIME TAKEN FOR VIRTUAL COUNTER INCREMENT OPERATION IN HASH-TREE BASED SCHEME

No.of ctrs.	Atmel	Infineon	STMicro	Winbond
2	0.91	0.51	0.81	0.61
4	0.92	0.52	0.83	0.62
8	0.94	0.53	0.83	0.63
16	0.94	0.55	0.84	0.64
32	0.95	0.55	0.84	0.64
64	0.95	0.57	0.86	0.66
128	0.97	0.57	0.86	0.67
256	0.98	0.58	0.89	0.69
512	0.99	0.59	0.89	0.69
1024	1.00	0.6	0.9	0.7

TABLE VIII
TIME TAKEN FOR VIRTUAL COUNTER READ OPERATION IN HASH-TREE BASED SCHEME

No.of ctrs.	Atmel	Infineon	STMicro	Winbond
2	0.905	0.505	0.80	0.605
4	0.91	0.51	0.81	0.61
8	0.91	0.51	0.81	0.61
16	0.92	0.52	0.82	0.62
32	0.92	0.52	0.82	0.63
64	0.93	0.53	0.83	0.63
128	0.93	0.53	0.83	0.64
256	0.94	0.54	0.84	0.64
512	0.94	0.54	0.84	0.65
1024	0.95	0.55	0.85	0.65

TPMs), we would have a margin of $0 - 0.01$ sec between our model and the actual experimental results. Figure 2 illustrates and compares the performance of the log-based and hash-tree based schemes on the Atmel and Infineon TPMs (the Y-axis is on a log scale). It can be seen that the hash-tree based scheme is much more scalable than the log-based scheme. For example, incrementing 1024 counters on an Atmel TPM just takes 0.95 sec with the hash-tree based scheme as compared to ≈ 847 sec in the log-based scheme. Although an expensive signature operation is unavoidable in the hash-tree scheme (signature takes ≈ 0.8 sec), the scalability of the hash-tree based scheme is still very high. We could go up to 2048 or 4096 virtual counters on an Atmel TPM with just around 0.955 sec and 0.96 sec for each respectively.

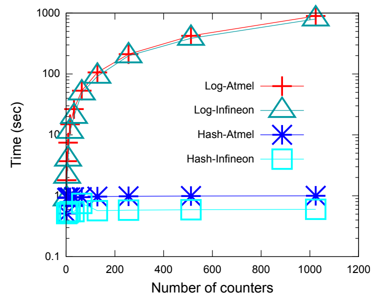

Fig. 2. Comparison of log-based and hash-tree based schemes for round-robin requests

C. Randomized Counter Requests

When we tried the read and increment virtual counter requests with the zipf distribution, we realized that the distribution of requests has no effect on the increment virtual counter operation in the log-based scheme or the hash-tree based scheme. In the log-based scheme, the increment certificate returned by the server consists of the virtual counterID, an anti-replay nonce, the transport log and its signature. This increment certificate is of constant size and each counter's increment certificate does not depend on any other counter's increment certificate. So whether counters are incremented sequentially or randomly makes no difference. In the hash-tree based scheme too, since the execution certificate returned by the TPM is completely independent of other counters' increment/read certificates, it doesn't matter whether the increment request was generated deterministically or randomly.

On the other hand, we found that the order of counter requests does matter in the read virtual counter request in the log-based scheme and is one of the main considerations for anyone implementing the log-based scheme in applications. In the read counter operation in the log-based scheme, the server returns a read certificate which consists of the current counter value, the counter's latest increment certificate and all the increment certificates generated from the time of that particular counter's last increment. If a large number of counters have been incremented since the time of that particular counter's last increment, the read certificate for such a counter would be incredibly long, and the time taken for constructing such a read certificate would increase proportional to the time that has elapsed since the counter's last increment. Also, the size of the log, U_t, increases as the time between counter requests increases. We ran the zipf distribution experiments using the Atmel TPM as a baseline measurement, i.e., for the log-based scheme, we used a real Atmel TPM, and for the hash-tree based scheme, we ran the simulator with the Atmel TPM performance profile. The maximum, minimum, average, median and standard deviation of the time taken in seconds are given in Table IX.

TABLE IX
STATISTICS FOR READ COUNTER OPERATION IN LOG-BASED SCHEME AND HASH-TREE BASED SCHEME WITH ZIPF DISTRIBUTION

Scheme	Min	Max	Avg	Median	Std.Dev.
Log-based	1.82	2146	453.07	14.047	737.674
Hash-tree	0.90	0.98	0.931	0.92	0.029

D. Performance Analysis of Randomized Requests

The experiments with the zipf distribution clearly bring out the differences between the performance of the log-based scheme and the hash-tree based scheme. Both the schemes were tested with 1024 counters with varying increment gap between two successive counter increments, the gap being determined by the zipf sampling function. In our context, we define the "increment gap" of a particular virtual counter to be the number of virtual counters incremented since the counter in question's last increment. The zipf sampling function was used for picking random counter ID's and we manually sorted them in ascending numerical order of increment gaps. The largest gap generated by the zipf sampling function was 700 (i.e., for a counter with counter ID x, after the zipf function generated x, it generated 700 other counter IDs before it generated counter ID x again). Here it might seem anomalous that although we simulated the zipf distribution tests with 1024 counters, the maximum gap generated by the distribution was 700. We note that this might seem like a lot of counters never get used, but this is the whole point of the zipf distribution - a few counters get used a lot of times, and some counters rarely get used, if at all.

Figure 3 compares the performance of the log-based and hash-tree based schemes for the zipf distribution on the Atmel TPM where the Y-axis is on a log scale. From the statistics of the log-based scheme, we can see that as we go from a low increment gap of 0-2 counters to a high increment gap of 700 counters, the time taken increases sharply. At the same time, the time taken in the hash-tree based scheme is 0.90 for an increment gap of 0 counters and 0.98 for increment gap of 700 counters with a low standard deviation of 0.029. This is because in the hash-tree based scheme, each virtual counter read/increment operation is completely independent of other virtual counters' read or increment operations. In the log-based scheme, each virtual counter's read certificate directly depends on all the virtual counters that have been incremented since that particular counter's last increment. As a result, in the hash-tree based scheme, we do not need to protect against a worst-case scenario like in the log-based scheme where if the increment gap between two successive increments of a counter reaches a tipping point or an upper bound, the system would get saturated take forever to generate the read certificate for that counter, besides the length of the read certificate (the log) would be unbounded. This can affect the performance of the system greatly in high-workload applications where we have thousands of counters being updated every second in an uneven manner. Clearly the log-based scheme, despite the advantage of being implementable on present-day TPMs, is suitable only for applications where there are a small number of counters, and all the counters get incremented according to a somewhat uniform distribution.

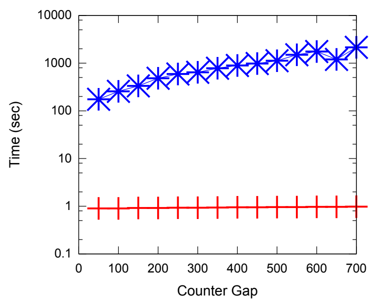

Fig. 3. Comparison of log-based and hash-tree based schemes for zipf distribution requests

VI. Conclusion

We have demonstrated a methodology to estimate the performance of TPM-based protocols in realizing secure digital wallets. To realize our methodology, we used the virtual monotonic counter schemes presented by Sarmenta et al. [2] and analyzed performance with two virtual counter generation schemes: the log-based scheme and the tree-based scheme. Our experiments show that the log-based scheme is inefficient and does not scale well to large numbers of counters, and the performance degrades if the counters requests are processed randomly as in real-world e-commerce scenarios; however the log-based scheme has the advantage that it can be implemented on current TPM chips. On the other hand, the hash-tree based scheme has a very low performance overhead, even for large numbers of counters, but requires an extension to the current TPM specification. With the increasing prevalence of e-commerce transactions, we believe that applications such as secure digital wallets and e-cash have great commercial value. As part of future work, it would be interesting to explore if, instead of proposing minor extensions to the TPM chip to facilitate such applications, we can implement such applications using a trusted computing infrastructure such as Flicker [7].

References

[1] T. C. Group, "Trusted Platform Module Specifications – Parts 1–3," Available at https://www.trustedcomputinggroup.org/specs/TPM/.

[2] L. Sarmenta, M. van Dijk, C. O'Donnell, J. Rhodes, and S. Devadas, "Virtual monotonic counters and count-limited objects using a TPM without a trusted OS," in *ACM CCS-STC '06*, 2006, pp. 27–42.

[3] D. Chaum and T. P. Pedersen, "Wallet databases with observers (extended abstract)," in *Advances in Cryptology - CRYPTO 92*. Springer-Verlag, 1992, pp. 89–105.

[4] S. Balfe, A. Lakhani, and K. Paterson, *Securing peer-to-peer networks using trusted computing, chapter 10, Trusted Computing*. C. Mitchell, 2005.

[5] A. Illiev and S. Smith, "Private information storage with logarithmic-space secure hardware," in *Workshop on Information Security, Management, Education and Privacy*, 2004, pp. 210–216.

[6] M. van Dijk, J. Rhodes, L. Sarmenta, and S. Devadas, "Offline untrusted storage with immediate detection of forking and replay attacks," in *ACM CCS-STC '07*, 2007, pp. 41–48.

[7] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for TCB minimization," in *EuroSys*, 2008, pp. 315–328.

[8] J. Katz, "Universally composable multi-party computation using tamper-proof hardware," in *EUROCRYPT '07*, 2007, pp. 115–128.

[9] I. Damgard, J. B. Nielsen, and D. Wichs, "Isolated proofs of knowledge and isolated zero knowledge," in *EUROCRYPT '08*, 2008, pp. 509–526.

[10] V. Gunupudi and S. R. Tate, "Generalized non-interactive oblivious transfer using count-limited objects with applications to secure mobile gents," in *Financial Cryptography*, 2008, pp. 98–112.

[11] S. Goldwasser, Y. Kalai, and G. Rothblum, "One-time programs," in *CRYPTO '08*, 2008, pp. 39–56.

[12] A. Gelbukh and G. Sidorov, "Zipf and heaps laws' coefficients depend on language," in *Conference on Intelligent Text Processing and Computational Linguistics*, 2001, pp. 332–335.

[13] T. Muller, L. Sarmenta, and J. Rhodes, "TPM/J - Java based API for the Trusted Platform Module TPM," Available at http://projects.csail.mit.edu/tc/tpmj/.

[14] M. Strasser, H. Stamer, and J. Molina, "Software-based TPM simulator," http://tpm-emulator.berlios.de.

[15] V. Gunupudi and S. R. Tate, "Timing-accurate TPM simulation for what-if explorations in trusted computing," To appear in SPECTS '10.