Virtual Monotonic Counters and Count-Limited Objects using a TPM without a Trusted OS*

Luis F. G. Sarmenta, Marten van Dijk,
Charles W. O'Donnell, Jonathan Rhodes, and Srinivas Devadas
Computer Science and Artificial Intelligence Laboratory (CSAIL)
Massachusetts Institute of Technology
Cambridge, MA 02139
{lfgs,marten,cwo,jrhodes,devadas}@mit.edu

ABSTRACT

A trusted monotonic counter is a valuable primitive that enables a wide variety of highly scalable offline and decentralized applications that would otherwise be prone to replay attacks, including offline payment, e-wallets, virtual trusted storage, and digital rights management (DRM). In this paper, we show how one can implement a very large number of *virtual* monotonic counters on an untrusted machine with a Trusted Platform Module (TPM) or similar device, without relying on a trusted OS. We first present a *log-based scheme* that can be implemented with the current version of the TPM (1.2) and used in certain applications. We then show how the addition of a few simple features to the TPM makes it possible to implement a *hash-tree-based scheme* that not only offers improved performance and scalability compared to the log-based scheme, but also makes it possible to implement *count-limited objects* (or "*clobs*" for short) – i.e., encrypted keys, data, and other objects that can only be used when an associated virtual monotonic counter is within a certain range. Such count-limited objects include n*-time use keys*, n*-out-of-*m *data blobs*, n*-copy migratable objects*, and other variants, which have many potential uses in digital rights management (DRM), digital cash, itinerant computing, and other application areas.

Categories and Subject Descriptors:
D.4.6 [Operating Systems]: Security and Protection
C.3 [Special-Purpose and Application-based Systems]: Microprocessor/microcomputer applications and Smartcards
E.3 [Data Encryption]: Public key cryptosystems

General Terms: Security, Design

Keywords: trusted storage, key delegation, stored-value, e-wallet memory integrity checking, certified execution

*An extended version of this paper will be available as an MIT CSAIL Technical Report. This work was done as part of the MIT-Quanta T-Party project, funded by Quanta Corporation.

STC'06, November 3, 2006, Alexandria, Virginia, USA.

1. INTRODUCTION

The increasing availability of the Trusted Platform Module (TPM) [30] as a standard component in today's PCs and mobile computers creates many exciting new possibilities in the realm of secure scalable and distributed computing. In the past, applications requiring security have generally assumed that users' machines are untrusted and have thus required *online* interaction with *centralized* trusted servers. Today, as more ordinary users' machines start including a TPM, however, it now becomes possible to avoid having to contact a central server by placing trust in the TPM chip on users' machines instead. This means that one can now create a variety of *decentralized* and *offline* secure applications which have much higher levels of scalability than previously possible with centralized schemes.

A few applications taking advantage of the TPM in this way have already been proposed, including applications such as distributed certificate authorities [8], peer-to-peer computing with enhanced security [1], controlling mobile access to broadcasted media [10], and others [20, 23]. In this paper, we propose using the TPM not just to implement one particular application, but to implement a *fundamental primitive* that in turn enables a wide variety of useful applications. Namely, we show how a TPM can be used to implement a potentially unlimited number of trusted *virtual monotonic counters* on an untrusted machine without a trusted OS.

A *trusted monotonic counter* – i.e., a tamper-resistant counter embedded in a device whose value, once incremented, cannot be reverted back to a previous value – is a very valuable primitive because it enables one to implement a wide variety of highly scalable applications that would otherwise be vulnerable to *replay attacks*. These include several applications of great interest and commercial value today, such as secure offline payments, e-wallets, virtual trusted storage, digital rights management (DRM), and digital cash.

In addition, the ability to dedicate and use an *unlimited* (or at least very large) number of monotonic counters on a single device is even more valuable. For one, it enables a user's personal device, such as a PC or mobile device, to be used in an arbitrary number of independent applications at the same time, even if each of these applications requires its own dedicated monotonic counter. Similarly, it also enables a single server to provide dedicated monotonic counters to an arbitrary number of clients. Finally, it makes possible new classes of applications and mechanisms that cannot be implemented with only one or a small number of monotonic counters. A particular example of these is the idea of *count-limited objects* which we present in Sects. 2 and 6, and which have many potential applications in secure and pervasive computing.

In the latest version of the TPM specification, version 1.2 [30], the Trusted Computing Group (TCG) has introduced built-in sup-

port for monotonic counters into the TPM [29]. However, because the low-cost TPM chip can only afford to have a small amount of internal non-volatile memory, this new functionality is necessarily limited. Specifically, a TPM 1.2 chip is only required to be able to store four independent monotonic counter values at a time, and only one of these counters is usable during a particular boot cycle (i.e., once one of the counters is incremented, the other counters cannot be incremented until the host machine is rebooted). The intent of the TCG in designing the TPM this way is not for the built-in monotonic counters to be used by user-level applications directly, but rather for the single usable counter to be used by a trusted OS component to implement an arbitrary number of "*virtual monotonic counters*", which in turn can then be used by user-level applications [29]. In this way, a TPM 1.2 chip can theoretically be used to provide an arbitrary number of dedicated virtual monotonic counters to different applications using only a single monotonic counter.

The problem with this approach, however, is that although it is theoretically sufficient from an abstract point-of-view, its use in practical applications is limited by the complex security mechanisms needed to implement a trusted OS. For example, one scheme for implementing virtual monotonic counters outlined by Microsoft as part of their proposed Next Generation Secure Computing Base (NGSCB) system [24] seems simple and straightforward by itself, but if we look at the security requirements for NGSCB, we find that it needs not only a PC with a TPM, but also (at least) the following hardware-based security features as well [9]: (1) a trusted BIOS that acts as the Core Root-of-Trust for Measurement (CRTM), (2) a built-in security mechanism in the main CPU, such as Intel's LaGrande Technology [16], that can be used to implement an *isolation kernel*, and (3) a memory controller or chipset that facilitates protection from DMA attacks. Furthermore, it also requires users to switch to an entirely new OS built according to the NGSCB architecture, and requires OS vendors to perform extensive security testing on their OS components every time they make a change.

Given the importance of virtual monotonic counters as an enabling primitive in many useful applications, we argue that it is worth the effort to ensure that it is possible to implement such counters using a TPM alone *without* relying on a trusted OS or trusted software. In this paper, we discuss how this goal can be achieved, and present concrete solutions, recommendations, and applications.

We begin in Sect. 2 by identifying the many potential applications of being able to keep track of a large number of virtual monotonic counters on a single host. In Sect. 3, we present an abstract model for how virtual monotonic counters can be used in applications and how they can be implemented. In Sect. 4, we present a *log-based* scheme that can be implemented with TPM 1.2. We note, however, that this scheme has two drawbacks. First, it can only implement a *non-deterministic* form of monotonic counters which are useful in stored-value and trusted storage applications, but it cannot be used to implement the stronger *arithmetic* form of monotonic counters which is useful in a broader range of applications. Moreover, it has a potentially unbounded worst-case read and increment latency. In Sect. 5, we solve these problems by presenting a new scheme based on the idea of Merkle hash trees [19] that can easily be implemented with the addition of relatively simple new functionality to the TPM. Unlike the log-based scheme, this *hash tree-based scheme* has a small bounded worst-case read and increment latency of $O(logN)$, where N is the number of virtual counters. Moreover, it can be used to implement *arithmetic* monotonic counters which the log-based scheme cannot implement. This in turn enables us to implement a new idea we call *count-limited objects*, which we discuss in Sect. 6. In Sect. 7, we present some experimental peformance measurements using present-day TPM 1.2 chips. We cite related work throughout this paper as appropriate and discuss other related works in Sect. 8 as well. Finally, we summarize our contributions in Sect. 9. An extended version of this paper containing more details will be available as an MIT technical report [25].

2. APPLICATIONS

A monotonic counter is a highly valuable primitive because it can be used to detect (and thus prevent) *replay attacks* in offline and decentralized secure applications. In this section, we present examples of such applications.

Offline payments, stored-value, and e-wallets. We first consider the problem of *offline payment* or *stored-value* systems. The goal in such systems is to allow a credit issuer to store a credit balance on a user's device, and then, when the user makes a purchase, allow merchants to securely verify and reduce this balance *without needing to communicate with the credit issuer or a third party*. The security challenge in these systems is obvious: since the value of the balance is stored in the untrusted user's own device, how does one prevent the user from *double-spending* his credits by simply changing or rewinding the value as he pleases? Preventing *arbitrary* changes to the stored balance is easy. A credit issuer can simply use a private or secret key, known only to itself and to trusted hardware in the merchant's devices, to produce a digital signature or message authentication code (MAC) for the account balance stored on the user's device. This way, only the credit issuer and the trusted merchant devices can write valid balances on the user's device.[1] The harder problem is that of preventing the user from *rewinding* or *replaying* his account balance. That is, even if the balances are signed with an unforgeable signature, the user can still easily copy *old* signed balances and reuse these copies with different merchants. A merchant who has no contact with a centralized server, and who has not seen the same or a newer signed balance from that user before, can tell that the balance is *authentic* because it is signed with the credit issuer's private key. However, he has no way of telling if the balance is *fresh* – that is, that it is not an old copy.

Without some sort of trusted memory or trusted component on the user's device, preventing such *replay attacks* in an offline system would be impossible. If the user's device, however, has a trusted monotonic counter (trusted by the credit issuer and the merchants), then a relatively simple solution is possible. Whenever the credit issuer or a merchant's device increases or reduces the stored value in the user's device, it first increments the monotonic counter on the user's device, and then signs a token including the new monotonic counter value and the new balance value. Then, when the user presents the token to another merchant at a later time, that merchant's device checks not only that the signature on the token is valid but also that the counter value in the token matches the current value of the monotonic counter on the user's device. This prevents a malicious user from replaying an old credit balance since the counter value signed with an old balance will not match the latest counter value in the trusted monotonic counter.

Note that in order for this scheme to work, a *dedicated* counter is needed *for each balance that needs to be protected*. Thus, if we want to be able to store different independent credit balances (e.g., from different credit issuers) in a single user's device, then that device must be able to keep track of multiple independent monotonic counters. This is why a mechanism for implementing a large number of virtual monotonic counters on a single device would be

[1]We assume for now that some other mechanism allows us to protect against malicious merchants tampering with their devices.

useful. With such a mechanism, a user's device can effectively become an *e-wallet* – that is, a digital equivalent of a real wallet that can store cash, credit cards, and generally different forms of currency and credits from different credit issuers.

Virtual trusted storage. A related use of virtual monotonic counters is in implementing *virtual trusted storage*. The idea here is to create a potentially unlimited amount of private, tamper-evident, and replay-evident virtual storage using untrusted storage and a *small* and *constant-sized* trusted component such as a TPM.

Consider, for example, a user who wants to store his data on a third-party server on the Internet and wants to be able to retrieve it at a later time from any one of several client devices that he (or his friends) own. If the other client devices can be offline at different times or do not have any secure means of communicating directly with each other (except through storing and retrieving data on the untrusted server itself), then the user's data can be vulnerable to a replay attack by a malicious third-party server. That is, a second client device retrieving data from the server would have no way of knowing if the data on the server is in fact the latest version.

Note that this problem is actually a more generalized form of the problem in the stored-balance offline payment system described earlier, except that the directions are reversed. That is, here, the user is storing his data on a third-party machine, instead of the other way around (i.e., a third-party such as the credit issuer or merchant storing data on the user's device). Thus, a monotonic counter can also be used to protect the data from replay attacks by the untrusted server in the same way as described earlier. In this case, the "balance" being protected is the user's trusted data, and the server takes the role of the e-wallet host while the user's devices take the role of the credit issuer and merchant's devices.

In this application, the ability to have a very large number of dedicated monotonic counters becomes useful for the server, since it would allow the server to handle an arbitrary number of independent users, each of whom may in turn want to securely store an arbitrary number of indepedent pieces of data. This ability in turn can enable us to implement many different applications in mobile and distributed computing, including file storage, synchronization, and sharing applications.

Count-limited objects (*aka* clobs). A very useful feature of *existing* TPM chips (to be described in more detail in Sect. 6) is the ability to perform operations using *encrypted blobs* containing keys or data that have been encrypted such that only a particular TPM can decrypt and use them. At present, there is no limit to the number of times a host can make a TPM use an encrypted key or data blob once it has a copy of that blob. However, if we can enable a TPM to keep track of a large number of virtual monotonic counters, then we can link a blob with a particular virtual monotonic counter so that the TPM can use this counter to track and limit the usage of these blobs. Such blobs would then become what we call ***count-limited objects***, or "***clobs***" for short.[2]

Count-limited objects can take many different interesting and useful forms, including:

- n-***time-use clobs***. Here, each clob has its own dedicated counter, which is incremented every time the clob is used. Useful forms of these include n-***time-use decryption keys*** (e.g., Alice gives Bob a key that lets Bob decrypt anything encrypted by Alice's public key, but only at most n times), and n-***time-use signing keys*** (e.g., Alice gives Bob a key that lets Bob sign anything with Alice's signature, but only at most n times).

- ***Shared-counter interval-limited clobs***. These are clobs that are tied to the *same* virtual counter. One form of such clobs are ***time-limited clobs***, wherein the shared counter is one whose value is tied to real time so that the valid interval for the clob corresponds to the real-time interval during which it is allowed to be used. Another form are n-***out-of***-m ***clobs***, including n-***out-of***-m ***encrypted data blobs***, which are blobs that share the same counter and all have a usage interval of 1 to n, such that one can only use *at most* n out of m encrypted blobs (regardless of m).[3] Still another form are or ***sequenced clobs*** or ***ordered clobs***, which have different usage intervals set in such way as to ensure that certain clobs cannot be used before others.

- n-***copy migratable objects***. Here, a virtual counter is used to limit the number of times a clob can be *migrated* (i.e., re-encrypted) from a particular TPM to another TPM such that copies of the clob can be *circulated* indefinitely (i.e., Alice can migrate a clob to Bob and Bob can migrate the clob back to Alice without needing a trusted third party), but only *at most* n copies of a clob are usable at any point in time (where n is the count-limit range of the original clob).

- ***Count-limited TPM operations***. Extending the existing idea of *wrapped commands* in TPM 1.2, we can have a clob that contains a wrapped command together with a count-limit condition. This allows us to apply the various types of count-limit conditions (e.g., n-time-use, n-out-of-m, n-time migratable, sequenced, etc.), to any operation that a TPM is capable of executing.

The different form of count-limited objects, have many exciting new applications, which we discuss next.

Digital rights management (DRM). The idea of limiting the use of data and programs is central to DRM. Thus, clobs naturally have many direct applications to DRM. For example, n-time-use decryption keys and n-out-of-m encrypted data blobs can be used to allow a copyright owner to create and store many encrypted media files on a user's device, while limiting the number of media files that the user can decrypt and use. Time-limited clobs would allow for media files that can only be used within a certain real time interval. Most interestingly, n-copy migratable clobs can make it possible to create protected media files that users can freely *lend* or *circulate* to other users much like people do with physical books and CDs.

Digital cash. Clobs also have potential applications as a way to implement or supplement *digital cash* schemes which require the ability to perform *offline* and *anonymous* transactions. Consider, for example, an e-wallet mechanism where instead of storing a user's total credit amount as an account balance, we store a collection of n-time-use signing keys. When a user with this kind of e-wallet purchases goods from a merchant, a merchant receives payment from the user by asking the user to sign a random nonce with the credit issuer's key using one of these signing keys. If we consider each signature produced using the keyblob as having a certain value (where different keys can represent different denominations), then the count limit on a user's keyblob represents the total stored value of that keyblob, and this value is reduced accordingly

[2] This idea is *not* related to the character large object (CLOB) data type in some databases.

[3] Note that in this case, the encrypted data blobs can be encrypted with different keys which do not necessarily have to be count-limited as long as they are protected by the TPM.

every time the keyblob is used. This scheme is more secure than the stored-balance scheme described earlier because it does not require merchant's devices to know the credit issuer's private key. Moreover, another advantage of this scheme is that it preserves the user's *anonymity*. This is because the signed nonces that the user gives to the merchant are signed with the *credit issuer's* key, *not* the user's. Thus, at the end of the transaction, the merchant has proof that the transaction was valid, and can go to the credit issuer to exchange the signed nonce for real money, but neither the merchant nor the credit issuer has any information on who the user was.

Using n-copy migratable clobs, an even more interesting form of digital cash is possible. If a credit issuer, for example, creates a "digital coin" as a one-time migratable clob, then such a coin can be migrated from user to user an arbitrary number of times *without requiring contact with the credit issuer or a trusted third party*. This more closely corresponds to how real cash is used in the real world. If we assume that all users have TPMs and that all these TPMs are trusted and working properly, then transactions are both secure (i.e., at most one valid copy of a coin exists at any time), and anonymous (i.e., the identity of the previous holder of a coin need not be exposed in a transaction).

Finally, if we assume that TPMs *can* get compromised, then more complex schemes would be necessary, but are possible if we have count-limited TPM operations involving special types of signing or decryption. One possibility, for example, is to use Brands' scheme [3], wherein a trusted hardware device called an *observer* is used to produce a signature needed for a successful transaction. To prevent a user from double-spending his digital coins, the observer is trusted to only produce this signature at most once (i.e., for each coin, the observer stores a random number that is needed to produce the signature for that coin, and then erases it after using it once). However, even if the observer is compromised, the cryptographic property of the e-cash scheme itself is not compromised, and double-spending can still be detected (and the offender identified) eventually by the credit issuer. In our case, we can implement a digital coin as a one-time-use clob representing this special signature operation. This allows us to implement Brands' idea of an observer, but with the advantages that we can handle an aribtrary number of coins at the same time, and that we can do it using a *non-dedicated* secure coprocessor (i.e., the TPM) that is not limited to e-cash, but can also be used for other applications.

Itinerant computing. Clobs would also be useful in *itinerant computing* applications. Here, a user's code runs *not* on the user's own machine, but on *other people's* machines, using resources on those machines as necessary, and then moving on to other machines (also belonging to other people) to continue the computation.

A traditional example of such itinerant computing applications are applications involving *mobile agents* that move from one host to another, executing code on behalf of its owner [7, 13, 17]. In such applications, clobs such as count-limited keys and commands can improve security by allowing a user's mobile agent to use the user's private keys as it executes on a host, while preventing the host from using these keys after the agent leaves, even if the host makes a copy of the mobile agent's code and wrapped keys. (In a way, this is a hardware-based alternative or supplement to Hohl's idea of time-limited blackbox security [13].)

Clobs also enable new forms of itinerant computing where the *the user himself* is itinerant. Suppose, for example, that a user is traveling and visiting different places where he needs to be able to run certain programs that require use of his private key, but suppose that he prefers not to bring his own computer with him as he travels (e.g., perhaps because airlines have banned passengers from carrying-on electronic devices). In this case, if his host institutions have machines with a TPM, then the user can create clobs on his hosts' machines before his trip. (To prevent a clob from being used before the user arrives, a clob can include an encrypted authorization password like TPM wrapped keys do.) When the user gets to a host institution, he can use the clobs he has previously sent over to do his required computations. Then, when he is done, he makes sure to increment the clobs' counters beyond their usable range. (Or, if he has created migratable clobs, he can also migrate his clobs to his next host institution.) This way, even though the host can keep a copy of the clobs or even steal the authorization passwords from the user as he types using their keyboard, the host cannot use the clobs outside of the count-limit.

Count-limited objects and virtual monotonic counters. Many other applications of count-limited objects are possible. We emphasize, though, that the crucial feature that makes count-limited objects possible is the ability to keep track of a large number of virtual monotonic counters. This is because we need a different virtual counter value for each independent clob (or group of shared-counter clobs). Having only one or a few monotonic counters, like the existing TPM currently has, is not good enough since it does not allow us to freely create counters when needed for a new clob.

3. VIRTUAL MONOTONIC COUNTERS

Having given an appreciation of the many different applications of virtual monotonic counters, we present in this section a model and framework for how such virtual monotonic counters can be implemented and used in an actual system.

Basic definition. We model a ***monotonic counter*** as a mechanism (implemented in hardware or software or both), which stores a value and provides two commands to access this value: the Read command, which returns the current value, and the Increment command, which increments the current value according to a specified *increment method* and returns the new value of the counter. This mechanism must have the following properties: First, the value must be ***non-volatile***. That is, it must not change or be lost unless explicitly incremented. Second, it must be ***irreversible***. That is, once the value has been changed (by invoking Increment), there must be no command or series of commands that can make the counter assume any previous value that it has had in the past. And third, the monotonic counter must behave as if the Read and Increment commands were ***atomic***. That is, if several commands are submitted to the counter at the same time, then the output of the counter must be as if the commands were executed one at a time in some sequential order.

In real-world applications, a monotonic counter would not be used alone, but as part of a system containing other hardware and software components. Thus, in addition to having the properties above, it must also be *trusted* and remain *secure* even if it or the components around it are exposed to both software-based and physical attacks by an adversary. This means that a monotonic counter must also satisfy the following security properties:

1. The counter should ideally be ***tamper-resistant***, but must *at least* be ***tamper-evident***. That is, it must be infeasible for an adversary to directly or indirectly cause the counter to behave incorrectly without at least being detected. In particular, an adversary must not be able to set the value of the counter arbitrarily, cause it to revert to a past value, cause it to generate false execution certificates (as defined below), or cause it to fail in any other way without being detected – even if the

adversary owns and has physical access to the hardware and software used to implement and use the counter.

2. In response to a command, the counter must produce a ***verifiable*** output message that certifies the output and the execution of that command. That is, if a user invokes a command $cmd(t)$ on the counter at some real time t and then subsequently receives a corresponding output response message $Out(t)$ from the counter, there must be a ***verification algorithm*** that the user can follow to check $Out(t)$ and convince himself that the counter has in fact executed $cmd(t)$, and that $Out(t)$ is indeed $cmd(t)$'s correct output. We call this verifiable output an ***execution certificate***.

3. Valid execution certificates must be ***unforgeable***. It must be infeasible for an adversary using any method (including using another counter, using a fake monotonic counter, or acting as a man-in-the-middle) to produce an acceptable execution certificate certifying an operation not actually executed by the counter.

Attestation identity keys. In a concrete implementation, the last two conditions above can be satisfied if we assume that the counter has at least one unique and protected public-private keypair that it can use for signing. In keeping with TPM terminology, we call this the counter's *attestation identity key* (AIK). The private key of the AIK is kept in secure non-volatile memory, and it must be impossible for an adversary to know this private key. The public key is certified by a trusted certificate authority (CA), and presented to users of the counter when needed to enable them to verify the counter's signatures.

Given such an AIK, the counter can be used as follows: First, the user of the counter generates a random nonce $nonce$ and then sends it to the counter together with the Read or Increment command request. The counter returns an output message, which we call the *execution certificate* for the command, that includes the output of the command (i.e., the current or new value of the counter), the nonce, and a signature using the AIK over the output and the nonce together. The user can then verify this execution certificate by checking that the signature is valid according to that counter's public key (this protects against an adversary using another counter or a fake counter), and checking that the nonce included in the output message is the same as the nonce that the user gave (this protects against replay attacks by a man-in-the-middle adversary giving a copy of an older execution certificate).

Virtual vs. physical monotonic counters. As noted earlier, in order to implement the applications we would need to be able to keep track of a large number of monotonic counters. Although non-volatile RAM (NVRAM) for general storage is rapidly growing cheaper today, *securing* large quantities of non-volatile RAM is still not easy to do. Thus, secure *and* low-cost hardware components such as the TPM are currently limited to having only small amounts of NVRAM. This problem motivates the idea of ***virtual monotonic counters***, as opposed to the ***physical monotonic counters*** currently implemented in TPM 1.2. Here, the idea is to use a small *bounded-sized* tamper-resistant hardware component together with ordinary *untrusted* memory and storage (which we assume to be effectively unbounded in size) to simulate a potentially unlimited number of independent "virtual" monotonic counters.

Of course, since virtual monotonic counters need to use untrusted memory, it is impossible to make virtual monotonic counters truly tamper-resistant like physical ones. As we will show, however, it *is* possible to implement virtual monotonic counters that are *tamper-evident*. With such virtual counters, attempts to change the value of a counter might not be preventable, but would always be detected by the client. Thus, the worst things that an adversary can do are denial-of-service attacks, such as destroying a counter or dropping command requests. These attacks are still worth noting, but are much less dangerous than arbitrary tampering, since in many applications, it is in the adversary's interest *not* to destroy or slow down a counter. In an offline payment system for example, the adversary (i.e., the user) has no incentive to disable or slow down his monotonic counter because he cannot use his credits without it. (This situation is analogous to the real world where one always has the ability to destroy or throw away cash in one's wallet, but one does not gain anything by doing so, so one does not do it.)

System model. Figure 1 depicts our model of how virtual monotonic counters are used and implemented. Here, we have two interacting systems: the ***host*** and the ***client***. The host contains the virtual monotonic counters and some application-specific functions and data, while the client runs an application program that needs to make use of the data, functions, and counters on the host.

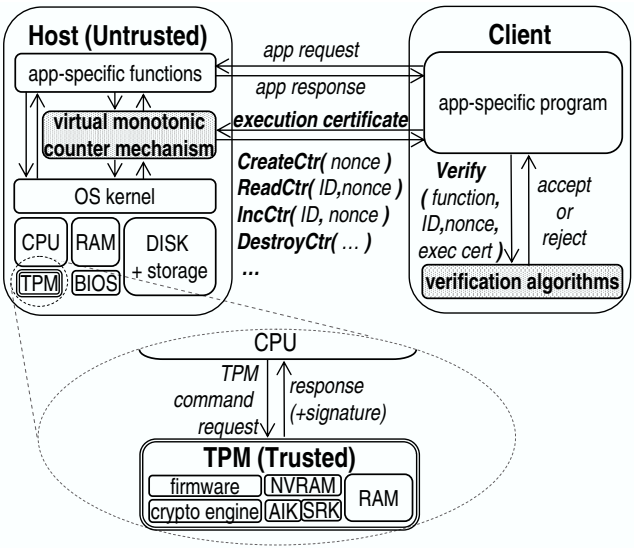

Figure 1: System Model for an application using virtual monotonic counters. The TPM is a passive secure coprocessor, and is the only trusted component in the host. An implementation scheme for virtual monotonic counters needs to define the shaded software components, given the TPM's functionality.

The virtual monotonic counter mechanism (shown in Fig. 1 as the shaded box in the host) is a software component that simulates a potentially unlimited number of virtual monotonic counters using the host's untrusted memory and storage and the TPM (as will be described below). This mechanism is meant to be used by different clients to create and use monotonic counters as needed in their respective applications. The virtual monotonic counter mechanism must support the following functions:

- CreateNewCounter(Nonce): Creates a new virtual monotonic counter and returns a ***create certificate***, which contains the new counter's unique CounterID and the given nonce.
- ReadCounter(CounterID, Nonce): Returns a ***read certificate*** containing the current value of the virtual counter specified by the given CounterID, together with the CounterID itself, and the given nonce.
- IncrementCounter(CounterID, Nonce): Increments the

specified virtual counter, and returns an ***increment certificate*** containing the *new* value of the virtual counter together with the CounterID and the given nonce.

- DestroyCounter(CounterID, Nonce): Destroys the specified virtual counter (so that the same CounterID cannot be used again), and returns a ***destroy certificate*** containing the CounterID and the given nonce.

Note that output certificates of these functions are not necessarily single signed messages. In general, these certificates can be complex data structures (possibly containing multiple signatures) that are designed to be *verifiable* by the client through the use of a corresponding set of verification algorithms, which the client runs on his own machine (which he trusts). The verification algorithm takes the execution certificate and checks that it is valid for the same function, counter ID, and nonce that the client originally gave.

Security model. In our model, we assume that the virtual monotonic counter mechanism can be invoked remotely by an arbitrary number of independent client devices, each of which may create an arbitrary number of independent virtual monotonic counters. We define the *owner* of a virtual counter to be the owner of the client device that requested its creation. This owner may own several client devices, each of which may be used to access the virtual counter. We assume that independent owners do not trust each other, and generally do not share virtual counters. However, we assume that client devices of the same owner *generally* trust each other. Specifically, we allow different client devices of the same owner to run applications that depend on secret or private keys known only to the other devices of that owner. However, we assume that the client devices of an owner may be offline at different times or may have no other way of communicating with each other except indirectly through the counters they share. Thus, such client devices are *not* allowed to depend on the ability to share state information with each other except indirectly through the virtual monotonic counters themselves.

Given this model, our main security goal is to implement a virtual monotonic counter mechanism that is *at least tamper-evident from the owner's point-of-view*. That is, at worst, all the client devices of the owner of a virtual monotonic counter must always be able to detect any failure or erroneous behavior of the counter caused by an attack by the host or another owner. Ideally, however, we would also want to be able to detect tampering by compromised client devices of the same owner, whenever possible.

As shown in Fig. 1, we assume that *all* hardware and software components on the host, except for a Trusted Platform Module (TPM), are untrusted – i.e., possibly under the control of an adversary working against the client. This includes not only memory, disk, and all application software, but even the CPU, the BIOS, and the OS kernel. In particular, note that the software implementing the virtual monotonic counter mechanism itself is considered as open to being attacked and controlled by the adversary.

In this paper, we assume a TCG-type TPM chip. Abstractly, however, our techniques should work with any trusted coprocessor with similar functionality. The TPM is assumed to contain the following: (1) a cryptographic engine, (2) internal firmware for implementing a set of TPM commands that the host can invoke, (3) a small amount of trusted internal memory (both volatile and non-volatile) that is not visible outside the TPM, and (4) a small number of protected keys usable only within the TPM, including at least an *attestation identity key* (AIK) for signing information generated by the TPM, and a *storage root key* (SRK) for encrypting and decrypting data. The AIK can be used to sign outputs of a TPM, and can thus provide certification that requested operations have been executed in the secure environment of the TPM. This is the crucial feature that would allow clients to verify the virtual monotonic counter mechanism's outputs. The SRK is a keypair whose private key is generated internally and never leaves the TPM. Its public key can be used by an external user or application to encrypt keys and other data that are meant to be decryptable and usable only inside the TPM. This key makes encrypted (wrapped) keys and data blobs possible, as described in Sect. 6. (For a good description of how all these TPM features work and are used, see [23].)

It is important to note here that the TPM is a *special-purpose* coprocessor. That is, it does not run arbitrary application software, but can only be used to execute a limited set of pre-defined commands as defined by the TPM specifications (see [30]). Furthermore, it is also a *passive* processor. That is, it cannot read or write directly into memory or other devices, and cannot do anything unless the CPU requests it. It also cannot prevent a CPU from submitting a request to it. It can return an error message in response to a CPU request, but only according to the pre-specified definition of the requested TPM command. The challenge, therefore, is how to be able to use the TPM in the host to implement a tamper-evident virtual monotonic counter mechanism without relying on any other trusted hardware or software on the host. This is what we will show in the following two sections.

4. LOG-BASED SCHEME

Since the TPM was not explicitly designed to support virtual monotonic counters without needing a trusted OS, it is impossible (to our knowledge) to use a TPM 1.2 chip to directly implement unlimited *arithmetic* virtual monotonic counters, where the counter value is incremented by 1. We *can*, however, implement a weaker form of virtual monotonic counter which can be used directly in virtual trusted storage and stored-value applications.

Implementation. The idea here is to use one of the TPM's physical monotonic counters as a "*global clock*" where the current "time" t is defined as the value of the monotonic counter at a particular moment in real time. Given this global clock, we then define the value of a particular virtual counter as the value of the global clock at the last time that the virtual counter's Increment command was invoked. Note that this results in a ***non-deterministic monotonic counter***, i.e., a counter that is irreversible, but whose future values are not predictable. Although such a virtual counter does not have all the advantages of an arithmetic counter, it can still be used in virtual trusted storage and stored-value applications as described in Sect. 2. This is because these applications only need to be able to tell if the value of a monotonic counter has changed from its previous value or not. It does not matter what the new value is, as long as it is different from any other value in the past.

We can implement the IncrementCounter function of the virtual monotonic counter mechanisms by using the TPM's built-in TPM_IncrementCounter command (which increments the TPM's physical monotonic counter) inside an *exclusive and logged transport session*, using the AIK as the signing key and the hash of the counter ID and the client's nonce (i.e., $H(counterID||nonce)$) as the anti-replay nonce for the final TPM_ReleaseTransportSigned operation. This produces a signature over a data structure that includes the anti-replay nonce and a hash of the *transport session log*, which consists of the inputs, commands, and outputs encountered during the entire transport session. This signature can then be used together with the counter ID, the client's nonce, and the transport session log, to construct the *increment certificate* which the client can verify. Note that by making this transport ses-

sion *exclusive*, we ensure that the TPM will not allow other exclusive transport sessions to successfully execute at the same time. This ensures the *atomicity* of the increment operation.

The verification algorithm for such an increment certificate is as follows: First, the client checks that $counterID$ and $nonce$ are the same as what it gave to the host. If they are, the client then computes $H(counterID\|nonce)$ and uses this hash together with the transport log, the signed output, and the certified public key of the TPM's AIK to verify the certificate. Finally, if the certificate verifies as valid, the client gets the *virtual* counter's value as the *physical* counter's value, which is included in the log of inputs and outputs given by the host as part of the certificate.

The more challenging problem in this scheme is that of implementing ReadCounter. We begin by having the host keep an array of the *latest* increment certificates for each virtual counter in its memory and disk storage, and return the appropriate one upon a client's request (since by definition, the global clock value at the time of the latest increment is the value of the counter). This is not enough, however, since a malicious or compromised host can easily reverse a particular counter by replacing its latest certificate with an older certificate for the same counter. Thus, an extra mechanism is needed to protect against this replay attack.

Our solution is as follows: On a ReadCounter request from a client, the host first reads the global clock by issuing a TPM's built-in TPM_ReadCounter command in an exclusive logged transport session. This produces a *current time certificate*, analogous to the increment certificate produced by using the TPM_IncrementCounter command. Then, the host gets the latest increment certificate for the client's desired counter from the array described above. Finally, it gets *all* the increment certificates it has generated (regardless of counter ID) from the time of the client's latest certificate to the current time. The *read certificate* for the ReadCounter command is then composed as a list, or *log*, of all these certificates, plus the current time certificate.

The verification algorithm for such a read certificate is as follows: First, the client checks the current time certificate. Then, starting from the increment certificate for its desired CounterID, it goes through the log making sure that: (1) there is a valid increment certificate for each global time value until the current time, and (2) *none* of the increment certificates are for the desired CounterID, except for the first one. If this verification algorithm succeeds, then the client is convinced that the first increment certificate indeed corresponds to the latest increment operation on that virtual counter. The value of the counter is then read as the value of the global counter included in that certificate.

Security. This scheme is provably secure if we assume that the TPM is trusted and cannot be compromised. One security issue, however, is that of a *fake* increment. That is, the host can pretend that it received an increment command from the client, even when it did not. The host cannot reverse the virtual counter in this way, but can make the counter go forward without the owner of the virtual counter wanting it.

In many applications, this is not a major concern because it would be to the host's disdvantage if it increments the counter without the client requesting it. For example, in the stored-value offline payment application described in Sect. 2, if the adversary (the user) performs a fake increment, he still cannot replay old stored values of an account balance, and would in fact lose his ability to use his latest available balance at all, since its signed counter value will not match the new counter value.

Nevertheless, if protection against fake increments is desired, then there are at least two solutions. One solution is to require client devices that request increment operations to send a *confirmation certificate* after verifying the increment certificate it receives. The confirmation certificate includes a signature of the incremented counter value generated using the client's secret key, so that it would be impossible for the host to generate fake confirmations. Then, when a read request for a counter is made at a later time, the host includes the confirmation certificate of the counter's latest increment as part of the read certificate. This allows a client to verify that the latest update was not a fake one. If a client receives a valid increment certificate but does not receive a valid confirmation, then it can suspect the host of executing a fake increment. Another solution is to use a nonce specially constructed and signed by the client as detailed in [25]. The advantage of this scheme is that it is more robust against network failures since there is no danger of a confirmation being lost between the client and the host.

Another possible problem worth noting is what happens if power to the host fails some time after the TPM_IncrementCounter but before the host is able to save the increment certificate to disk. If this happens, then the host will not have a valid execution certificate for the increment operation, and will have a gap in the log. This problem *cannot* be used for a replay attack because clients will still be able to at least detect the gap during the read counter operation. However, it does make all counters before the power failure untrustable (because client devices would not have proof that these counters were *not* incremented during the time slot of the gap). This problem cannot easily be avoided because of the limitations of existing TPMs, and is one disadvantage of the log-based scheme compared to our proposed hash tree-based scheme in Sect. 5. Note, however, that recovery of a counter's value is still possible if *all* the client devices of the counter's owner are able to communicate together and agree on the last valid value of the counter. Then, they can perform a special increment operation after the gap, and sign a special confirmation together indicating the correct value of the counter after the gap.

Finally, note that the functions CreateCounter and DestroyCounter can be implemented like IncrementCounter with a special confirmation or special nonce to indicate a creation or destruction event for the desired counter ID. However, since the TPM does not check the nonce given to the TPMIncrementCounter operation, there is nothing actually stopping a client device, in collusion with the host, from incrementing a virtual counter which has not been created or which has already been destroyed (thus generating a new increment certificate). Thus, the usefulness of the create and destroy functions are limited when using the log-based scheme (unless we can assume that the client devices are trusted and never misbehave).

Performance. The log-based scheme is relevant because it is implementable using *existing* TPM 1.2 chips, and it is usable in virtual trusted storage and stored value applications. This means that we can implement such applications using existing hardware today. Performance-wise, however, the log-based scheme has a significant drawback: if a virtual counter C is not incremented while other counters are incremented many times, then the read certificate for C would need to include the log of *all* increments of *all* counters (not just C) since the last increment of C. The length of this log is *unbounded* in time and can easily grow very large.

In some applications – either where there are only a few counters (e.g., a small e-wallet), or all counters are incremented frequently, this may be acceptable since the log would not get very long. It is also possible to do *adaptive time-multiplexing* as described in [25]. This reduces the size of the log required for verification when reading. However, it still results in potentially unbounded and long

waiting times for increments. Another disadvantage of this scheme is that it cannot currently be used to implement count-limited objects because these require arithmetic counters and require modifications to the TPM that allow it to prevent signing and decryption operations based on the value of a virtual monotonic counter.

5. HASH TREE-BASED SCHEME

If we can add new commands to the TPM, then a better solution is possible which not only has a bounded (and small) computation, communication, and latency cost for virtual counter operations, but which also enables us to implement ***arithmetic virtual monotonic counters*** and the idea of ***count-limited objects*** described in Sect. 2. In this section, we present a basic version of this solution consisting of a new TPM command, TPM_ExecuteHashTree, and some minor changes to existing TPM commands. We discuss the implementation of count-limited objects in Sect. 6.

Merkle Hash Trees. Our solution is based on the idea of a *Merkle hash tree*, a well-known technique for efficiently checking the integrity of a large number of data objects [19]. In a Merkle hash tree (such as the one shown in the middle of Fig. 2), a leaf node is created for each data object, and contains a collision-free *hash* of the object's contents. Then, a binary tree is formed, where the value of an internal node is the hash of the concatenation of its left and right children. The root of this tree, called the *root hash* is then itself a collision-free hash for the entire set of data objects, and is guaranteed to change if *any* of the data objects change.

The advantage of using a Merkle tree over other ways of producing a collision-free hash over a large data set is that once the tree has been initialized, it only takes $O(logN)$ steps to update the root hash whenever there is a change in one of the N data objects. Specifically, whenever a piece of data is changed, we go up the hash tree along the path from the changed leaf node to the root. At each step, the new value of a node is hashed with its sibling in order to produce the new value of its parent, and this process is repeated until the root hash itself is updated. Verifying the integrity and freshness of a data object also only takes $O(logN)$ steps. Here, we take the *current* version of the data object in question, and compute a root hash in the same way as above. The computed root hash can then be compared with a saved value of the latest root hash known to the verifier to determine if the given version of the data object is in fact the latest version of that object.

In the context of secure and trusted computing, Merkle trees have been proposed as an efficient way of protecting the integrity and freshness of a large (practically unbounded) amount of data stored in untrusted memory using a much smaller *bounded-sized* trusted component. The idea here is to require all *legitimate* read and update requests for the data objects to go through a *trusted component* which maintains a hash tree and uses it to verify the integrity of the data before proceeding. It can be shown that as long as the root hash is kept in persistent trusted memory, then it is possible to achieve tamper-evident operation, even if hash tree nodes themselves are stored in untrusted memory. This is because the use of collision-free hash functions means that even if the adversary can illegitimately change the data objects or any of the nodes in the tree, it would be computationally infeasible for him to produce a combination of these corresponding to a *different* set of leaf node values but hashing to the *same* root hash node.

In previous work, different forms of such a trusted component have been proposed and used. (The reader is referred to the papers cited here for alternative explanations of how Merkle hash trees work.) Applications involving *authenticated dictionaries* [21] and trusted databases [18] have been proposed that use a trusted computer running trusted software to authenticate data stored in storage that is accessible to other untrusted computers. The AEGIS project [27] proposes a *secure CPU* which ensures privacy by encrypting any data it stores in main memory, and decrypting it internally. To protect itself against replay attacks on its externally stored data, the AEGIS processor uses a Merkle tree with the root hash stored in trusted memory inside the secure CPU. In other recent work, hash trees have also been proposed as way of protecting the data integrity and freshness in a system with a TPM and the Nexus trusted OS [31, 26]. In this case, as in the case of Microsoft's scheme for virtual monotonic counters cited in Sect. 1, the trusted component is a trusted OS loaded through a secure boot process, and requires not only a TPM, but also a trusted BIOS, and certain security features in the main CPU and hardware of the system.

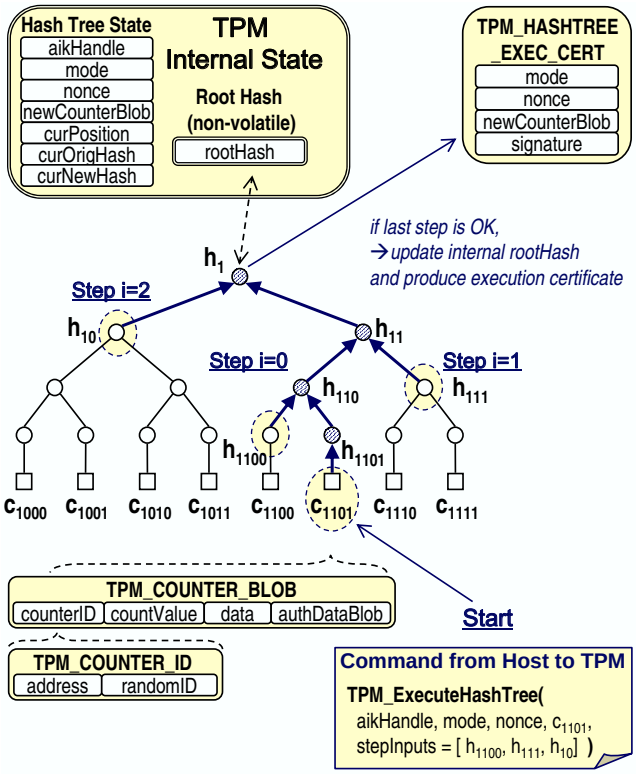

Figure 2: Hash tree-based scheme data structures and example. The counter blobs (squares) and hash nodes (circles) are all stored in the host's memory. To read or update counter c_{1101}, the host sends the TPM the command shown. Dashed circles show the inputs given to the TPM. The shaded internal tree nodes are computed internally by the TPM given these inputs. Arrows show the flow of computation inside the TPM.

Our new scheme borrows the idea of using Merkle trees from these previous works, but takes it further by allowing the trusted component to be a simple and *passive* coprocessor like the TPM, instead of a more complex and *active* one, such as a main CPU like AEGIS, or a trusted OS like NGSCB or Nexus.

TPM Implementation. Figure 2 presents the basic version of our scheme, which uses a new TPM command, TPM_ExecuteHashTree, shown in Fig. 3. In this scheme, the data objects being protected by the Merkle tree are a set of *counter blobs*, each representing an independent virtual monotonic counter. Aside from containing the actual value of the counter ($countValue$), each counter

Command: TPM_ExecuteHashTree
Inputs: int *aikHandle*, byte *mode*
TPM_COUNTER_BLOB *counterBlob*
TPM_NONCE *nonce*
TPM_DIGEST *stepInputs*[]
(optional) byte[] *command*
Outputs: If successful, returns TPM_HASHTREE_EXEC_CERT
(or output of *command*)
Else returns error code
Actions:

1. Check authorizations for the AIK, for *counterBlob*, and for *command* and ABORT on failure (i.e., return error code and clear *hts*)
2. Check *mode* and ABORT if illegal
3. Check *counterBlob.counterID.address* and ABORT if illegal

4. *HASHTREE_START* routine:
   Initialize the Hash Tree State
   a. Create a new TPM_COUNTER_BLOB, *newCounterBlob*
      i. Copy *all* fields of *counterBlob* to *newCounterBlob*
      ii. if *mode* is INCREMENT then
         (1) *newCounterBlob.countValue* = *counterBlob.countValue* + 1
         (2) *newCounterBlob.data* = *nonce*
      iii. else if *mode* is CREATE then
         (1) *newCounterBlob.counterID.randomID* = new random number
         (2) *newCounterBlob.countValue* = 0
         (3) *newCounterBlob.data* = *nonce*
         (4) *counterBlob* = null *// old blob should have been null*
   b. Setup TPM's internal Hash Tree State for leaf node
      i. Let *hts* be the TPM's internal Hash Tree State
      ii. Set *hts.aikHandle* = *aikHandle*
      iii. Set *hts.mode* = *mode*
      iv. Set *hts.nonce* = *nonce*
      v. Set *hts.newCounterBlob* = *newCounterBlob*
      vi. Set *hts.curPosition* = *newCounterBlob.counterID.address*
      vii. Compute *hts.curOrigHash* = Hash(*counterBlob*)
      viii. Compute *hts.curNewHash* = Hash (*newCounterBlob*)
      ix. if *mode* is equal to RESET then
         hts.curNewHash = *KnownNullHashes[height of position]*
      x. *hts.command* = *command*

Notes:
1. *mode* can be READ, INCREMENT, CREATE, or RESET. EXECUTE is an option bit which can be OR'd into *mode* (usually with INCREMENT or READ).
2. EXECUTE can be used with or without *command*. If used without *command*, *hts* is remembered so it can be checked by the *immediately* following command given to the TPM

5. *HASHTREE_STEP* loop:
   FOR each *i* = 0 TO *stepInputs.length* DO
   a. *siblingHash* = *stepInputs*[*i*]
   b. *isRight* = *hts.curPosition* & 1 *// (i.e., get lowest bit)*
   c. *// Set hts "current" state to refer to parent*
      if (*isRight* is 0) then
         hts.curOrigHash = Hash(*hts.curOrigHash* || *siblingHash*)
         hts.curNewHash = Hash(*hts.curNewHash* || *siblingHash*)
      else
         hts.curOrigHash = Hash(*siblingHash* || *hts.curOrigHash*)
         hts.curNewHash = Hash(*siblingHash* || *hts.curNewHash*)
   d. *hts.curPosition* = *hts.curPosition* >> 1 (right shift)
6. Check if computed original root hash is same as trusted root hash
   a. If (*hts.curPosition* is not 1)
      then ABORT *// not enough stepInputs presented*
   b. If ((*hts.curOrigHash* is NOT EQUAL to *TPM.rootHash*)
      AND (mode is NOT EQUAL to RESET))
      then ABORT *// original values fed in were not correct*
7. Execute command according to *mode*
   a. If (*hts.mode* is INCREMENT)
      OR (*hts.mode* is CREATE)
      OR (*hts.mode* is RESET)
      then *TPM.rootHash* = *hts.curNewHash*
   b. If (*hts.mode* does NOT have EXECUTE bit set)
      OR (*hts.command* is null) then
      i. Create new TPM_HASHTREE_EXEC_CERT *execCert*
      ii. *execCert.mode* = *hts.mode*
      iii. *execCert.nonce* = *hts.nonce*
      iv. *execCert.newCounterBlob* = *hts.newCounterBlob*
      v. *execCert.signature* = Sign(*hts.mode* || *hts.nonce* || *hts.newCounterBlob*) using AIK specified by *hts.aikHandle*
      vi. if (*hts.mode* has EXECUTE bit set)
      then remember *hts* for *immediately* following command
      else erase *hts*
      vii. Return *execCert*
   c. else *// i.e., hts.mode* has EXECUTE and *hts.command* is not null
      i. Get count-limit condition pertaining to *hts.command*
      ii. Compare *mode* and *counterID* in count-limit condition with those in *hts*, and ABORT on failure
      iii. If *hts.newCounterBlob.countValue* is within the valid range in count-limit condition then execute *hts.command* and return result, else ABORT

3. For READ and INCREMENT, input counterBlob should have the current counter value. For CREATE, input counterBlob contains address and encrypted authDataBlob from owner/creator. For RESET, input counterBlob should have address of node or subtree to be reset, and encrypted authDataBlob with TPM owner authorization.

Figure 3: The TPM_ExecuteHashTree command pseudocode.

blob also contains a *counter ID*, an arbitrary *data* field, and an encrypted data blob for authentication information, *authDataBlob*. The counter ID is composed of an *address* field, and a *randomID* field. The address contains the position of the counter blob in the tree, expressed as a "1" followed by the binary representation of the path from the root to the counter blob, while the *randomID* field contains a random number generated by the TPM at the creation of the virtual monotonic counter. The use of the random ID field here allows the address of a virtual counter that has been destroyed to be reused without compromising any clients who depend on the old counter at the same address. The arbitrary *data* field is not strictly necessary for basic functionality, but is used to make certain applications possible. In our current implementation, we simply use this field to store the nonce given by the client. Finally, the encrypted *authDataBlob* field is analogous to the authorization data fields in key blobs in the TPM. It specifies a secret that a caller to the TPM would be required to demonstrate knowledge of, through the TPM's OSAP or OIAP authorization protocols, before the TPM would allow any operation involving this counter blob to proceed. A client can use this authorization mechanism to prevent the host from performing fake increments. (Note that the TPM's OSAP and OIAP protocols work without exposing the authorization secret in the clear between the TPM and the caller. Thus, it is possible for the host to act as a man-in-the-middle between the client and the TPM without learning the secret.) Confirmations or specially-constructed nonces, as discussed in Sect. 4, can also be used instead of or in addition to this authorization mechanism.

In the beginning, before any virtual counters are created, all the counter blobs are assumed to have a special null value (i.e., all-zeros), and both the TPM and the host assume a hash tree computed from such null values. Since such a tree is symmetric, the hashes corresponding to internal nodes at the same depth are equal to one another. Thus, we can pre-compute all of the nodes of the tree by pre-computing a set of $log_2 N$ distinct *null hashes*, one for each level, given a maximum number of virtual counters N. The value of the highest-level hash is used as the initial value of the root hash.

The pre-computed values of all the null hashes are also kept by both the TPM and the host for reference. The host can use these constants when it needs to produce a hash for an unused or reset subtree. The TPM can store these constants in internal ROM (or hardwired circuitry) and use them when resetting subtrees in the tree (at the request of the host) as is done in line 4b.ix of Fig. 3.

Starting from this null state, the host then responds to each legitimate create, read, increment, and destroy request from a client by invoking the TPM_ExecuteHashTree command, shown in pseudocode in Fig. 3.[4] This command takes in an AIK handle, a $mode$ parameter to specify the desired operation, a nonce, and the *current* counter blob corresponding to the desired virtual counter (or an empty counter blob with only the $address$ field and encrypted authorization blob, when creating a new counter or resetting a counter or subtree). It also takes a list, $stepInputs$, corresponding to the hashes of the siblings of the leaf's ancestors along the path to the root. (An example is shown in Fig. 2.) These are provided by the host from the host's copy in untrusted memory. Given these input parameters, the TPM computes the root hash corresponding to the current counter blob. For create, increment, and reset operations, the TPM also generates an *updated* counter blob ($newCounterBlob$) and computes the corresponding root hash for it. If the root hash computed using the *current* (original) counter blob matches the TPM's internal copy of the root hash, then the TPM replaces the internal copy with the *new* computed root hash, and generates an execution certificate signed by the specified AIK. This execution certificate can then be passed by the host to the client, which can then verify it by checking the counter ID, nonce, and mode in it, and verifying the signature from the AIK.

Whenever an update is made to any of the counter blobs, the host also updates the corresponding hash tree nodes in its own untrusted memory. Note that the TPM only needs to produce the final execution certificate, and does not need to output the intermediate values in the hash tree. This is because the host can easily compute these values by itself given the new counter blob. Also note that the host need not store counter blobs or hash tree nodes in subtrees with no virtual counters, since the hash values of these are pre-computed as discussed above. Thus, even if a host may logically have a tree containing billions of virtual counters, it only needs memory proportional to the number of active virtual counters. And, significantly, the TPM only needs a small *constant* amount of memory, namely, non-volatile memory for the root hash, and a constant amount of volatile memory for the hash tree state used during the execution of the algorithm.

We assume that TPM_ExecuteHashTree, like other TPM commands, is an atomic operation. That is, we assume that the TPM will not allow other TPM commands to be invoked while TPM_ExecuteHashTree is still executing. This satisfies the atomicity requirement for our virtual monotonic counter functions since such functions are implemented here directly as a single call to TPM_ExecuteHashTree.

Also note that if there is a power loss during an increment operation before the host is able to get the execution certificate from the TPM or return it to the client, then the host can simply return an error code to the client. The client can then issue a ReadCounter request to check whether the counter has actually been incremented or not. (The client, not the host, needs to do this because authorization may be needed.) In this case, the host performs the ReadCounter operation using a *new* counter blob derived from the old blob by incrementing the count value, and feeds this to TPM_ExecuteHashTree. If the operation succeeds, then the host and client know that the counter has been incremented, otherwise, the client simply reissues the IncrementCounter command. (Note that this assumes that a power loss while TPM_ExecuteHashTree is executing results in either the root hash being untouched or being updated to its new correct value, but not an indeterminate value. This is actually not guaranteed by the current TPM 1.2 specifications for NVRAM in general, but is possible to guarantee with very high probability given extra internal hardware in the TPM.)

Variants. Variants of this instruction are possible. One variant is to split TPM_ExecuteHashTree into two commands: a *start* command, which the TPM calls at the beginning with the AIK handle, mode, nonce, and original counter blob, and a *step* command, which takes a single step input (sibling hash) and is called for each successive step up the tree. The start command would essentially correspond to lines 1 to 4 of Fig. 3 and the step command would correspond to one iteration of the loop in line 5, and then lines 6 and 7 when the position reaches the root. (Atomicity can be preserved by treating the start-step sequence like an exclusive transport session.) This has the advantage that it only requires the TPM to hold a very small of amount data in secure volatile memory at a time, and can be useful if the input buffer or the internal volatile memory are small. Another variant would be to modify the hash tree data structure such that counter blobs are contained in internal hash tree nodes as well, and not just the leaves. Combined with the start-step variant described above, this variation makes it possible to have *dynamically growing hash trees* that enable us to support a *truly unbounded* number of virtual counters.

We note, however, that even if we assume a tree of depth of 32, supporting 2^{32} virtual monotonic counters, the $stepInput$ array (which forms the bulk of the input data) only amounts to 32 hash values of 20 bytes each, or 640 bytes total. This is still considerably smaller than the 4K byte input buffer that present-day TPMs already have. Thus, we expect that the TPM_ExecuteHashTree command as we have defined it will be a practical solution for actual TPMs.

Other variants are also possible, such as allowing for multiple independent hash trees, and allowing for multiple increments of different counters to be done in one step (saving the time it takes to produce the final execution certificate signature). A general optimization technique worth noting that can be applied orthogonally to any variant scheme would be that of *caching* certain internal nodes. This would improve performance by allowing a TPM to stop checking the hashes as soon as it reaches a cached node in its internal secure volatile memory. This technique was originally proposed in [11] in the context of memory integrity checking schemes, and was used in AEGIS [27].

6. COUNT-LIMITED OBJECTS

We can implement the idea of ***count-limited objects*** or ***clobs*** presented in Sect. 2 by combining our proposed new features for virtual monotonic counter features with existing features in the TPM for supporting encrypted keys and data blobs. In this section, we show how this can be done.

Background: wrapped keys and encrypted data blobs. One of the useful features of *existing* TPM chips today is the ability to sign or decrypt data using a ***wrapped key*** – i.e., a public-private keypair where the private key has been encrypted (by the TPM itself or by an external party) using a key protected by the TPM (e.g., the SRK

[4]We leave the name of this command general since it is possible to define other ways for using this command by simply defining new modes. This allows this command to potentially support other useful mechanisms as well in the future (e.g., non-monotonic virtual trusted storage, etc.)

mentioned in Sect. 3), such that it can only be decrypted and used internally by a particular TPM. There are several forms of such keys and many applications.

A ***non-migratable key***, for example, is a wrapped key where the private key is generated internally by a TPM and encrypted using that TPM's unique key, so that it can only be used by that particular TPM. If one encrypts data using the public part of such a key, then the encrypted data can only be decrypted by the particular TPM with the private key. This allows one to tie data to a particular machine, such that, for example, if a data thief somehow copies the entire hard disk of a PC, the thief cannot decrypt the data without stealing the actual PC itself. This feature also has potential use in DRM since it can allow a media distributor, for example, to send protected media to a consumer such that the data can be decrypted only on the consumer's particular TPM-enabled device.

An ***externally wrapped key*** is another useful kind of wrapped key. Here, an external party, Alice, takes a public-private keypair that she owns (i.e., where the private key is known only to her) and creates a wrapped key for another party, Bob, by using the public key of the SRK of Bob's TPM. If Alice then gives the wrapped key to Bob, Bob can now use this key to sign data with Alice's signature (if it is a signing key), or to decrypt data encrypted with Alice's public key (if it is a decryption key). However, Bob can only do this on the machine with his particular TPM. If he tries to use the wrapped key on another machine, it will not work because the other machine would not be able to decrypt the private part of the wrapped key. Thus, for Alice, this is a kind of *key delegation* mechanism that gives the assurance that her delegated key (i.e., the wrapped key) can be used only on a specific machine (i.e., Bob's).

In addition to wrapped keys, the TPM also supports ***encrypted data blobs***. There are two forms of these. *Bound* data blobs are blobs that have been encrypted using the public key of a wrapped key protected by a TPM. Such blobs can be created by anyone (without using a TPM), but can only be decrypted on a particular TPM using a particular wrapped key. *Sealed* data blobs are created using the TPM itself, and can only be *unsealed* by the same TPM *and* only if the values in the platform configuration registers (PCRs) of the TPM match the values specified in the $sealInfo$ field of the data blob. Such blobs can be used to hold data that can only be decrypted while a certain trusted program (represented by the particular PCR values) is running.

Wrapped keys and sealed data blobs can also include an encrypted usage authorization secret. This adds an extra layer of security which ensures that a key or a blob can be used only if the caller knows its authorization secret. (As noted earlier, the TPM's OSAP and OIAP authorization protocols work without exposing the authorization secret in the clear, so it is possible for the host to act as a man-in-the-middle between the client and the TPM without learning the secret.) In addition, wrapped keys, like sealed data, can also be tied to PCR configurations such that they can only be used while running certain trusted software.

A wrapped key can also be a ***migratable key***. Such a wrapped key includes a migration authorization secret encrypted in the blob together with the private key and the usage authorization secret. A migratable key wrapped for a source TPM A can be migrated to a destination TPM B by invoking a migration command on TPM A with the public key of TPM B and the migration authorization secret of the key. (Note that the TPM does not certify that the other TPM's public key is authentic, but relies on the assumption that the trusted party who knows the migration authorization secret trusts the public key of the other TPM.)

Implementing count-limited objects. Currently, there is no limit to the number of times that a host can use a wrapped key or encrypted data blob as long as it has the correct TPM, and authorization secret. If the new mechanisms for virtual monotonic counters that we propose in Sect. 5 are included in a future version of the TPM, however, then these can be used to provide *count-limited* versions of the TPM's existing abilities to handle wrapped keys and encrypted data. This leads to the various forms of clobs described in Sect. 2.

To do this, we first modify the existing TPM data structures for wrapped keys and encrypted data blobs to include an optional *count-limit condition* field, containing the counter ID of a virtual monotonic counter plus, the mode and the range of counter values that are required for valid use of the key or data. At present, both the TPM_KEY and TPM_STOREDDATA structures for wrapped key blobs and sealed data blobs, respectively, already have a variable-length field for specifying a required PCR configuration, if desired. We propose to have the TPM allow a count-limit condition structure to be used in this field instead of, or in addition to, the PCR information. (Note that the count-limited condition, like the PCR information, is stored in unencrypted form to allow the host to know the condition. However, as done with wrapped keys and sealed data blobs in TPM 1.2, a hash of the unencrypted parts of the blob – including the condition – is included in the encrypted part of the blob. This prevents the host from altering the count-limit condition.) Correspondingly, the TPM_LOADKEY command must be changed to include the count-limit condition information as part of the information loaded and kept in the TPM's memory so that it can be checked whenever the key is used. (Note that TPM_LOADKEY need not do any checking itself, though.)

Second, we modify the TPM_ExecuteHashTree command proposed earlier to allow for an EXECUTE option bit in the $mode$ input parameter. If this bit is set, then the TPM will remember the final hash tree state (hts) of a successful TPM_ExecuteHashTree execution such that it can be checked by the TPM command invoked *immediately after* it (and then erased afterwards). In typical use, we expect the EXECUTE bit to be used with the INCREMENT mode so that using a clob requires incrementing a counter. However, it may also be used with the READ mode to allow us to create clobs that do not require the counter to be incremented each time they are used. This allows for clobs that the host can use an unlimited number of times until *someone else* (e.g., the owner of the counter or another clob) increments the counter.

Finally, we modify the TPM_Sign, TPM_Unbind, and TPM_Unseal commands to add a simple check when using keys or data blobs that have a count-limit condition field. Specifically, these commands must first check the count-limit condition field (if any) in the corresponding loaded key information or data blob and make sure that the $counterID$ and $mode$ in the TPM's hash tree state match the values in the count-limit condition, and that $countValue$ is within the valid range. (In the case of TPM_Unseal, we also modify the command such that if there is a count-limit condition, then it does not require the sealed data blob to have a PCR configuration or $tpmProof$ field.)

To allow for virtual counters that can only be incremented by using a clob (and not by calling TPM_ExecuteHashTree by itself), we can also allow the desired TPM command (e.g., TPM_Sign, TPM_Unbind, or TPM_Unseal), together with all its input parameters, to be included as an optional variable-length input parameter of TPM_ExecuteHashTree in a similar way to how wrapped commands are included in the TPM 1.2's TPM_ExecuteTransport command). This is useful, for example, in implementing *sequenced clobs* as described in Sect. 2, which require that the shared counter cannot be incremented except by executing the clobs themselves.

Using count-limited objects. Given these modifications, using a count-limited object, or clob, is easy. If Alice, the *issuer* or *delegator*, wants to give a count-limited object to Bob, the *recipient* or *delegatee*, then they take the following steps:

1. First, Alice checks that Bob's host machine has a genuine and secure TPM. Exactly how this is done is not the focus of this paper, but well-known schemes for doing this include Direct Anonymous Attestation [4], a scheme supported in TPM 1.2 that allows verification while preserving Bob's anonymity.

2. Then, if Alice wants to create an n-time-use clob, or the first clob among a set of shared-counter interval-limited clobs, she gets a *new* virtual monotonic counter ID from Bob by invoking his CreateCounter function remotely. Alternatively, she could also use an *existing* counter ID of Bob's, if she wants to create a shared-counter clobs using that counter.

3. Alice then constructs the count-limit condition field with the counter ID, count range, and mode (i.e., READ or INCREMENT) that she desires. A mode of INCREMENT means that the counter must be incremented before each use of the clob. A mode of READ means that the counter can be used an unlimited number of times until someone else increments the counter.

4. Given the appropriate counter ID, Alice then uses the public key of Bob's TPM's SRK (or another storage keypair whose private key is known by Bob's TPM but not revealed to Bob) to construct a wrapped key blob or sealed data blob for Bob. The resulting encrypted blob is usable only on Bob's TPM, and only according to the count-limit condition included in it by Alice.

5. On Bob's side, Bob can use a count-limited key or data blob exactly as he does an ordinary TPM wrapped key or data blob, except that he has to first invoke TPM_ExecuteHashTree *immediately before* calling his desired operation (e.g., TPM_Sign, TPM_Unbind, or TPM_Unseal). This reads or increments the appropriate counter, and sets up the hash tree state so that the desired operation called after it can check it before proceeding. Alternatively, he can also feed the desired TPM command as an additional input into TPM_ExecuteHashTree itself.

Count-limited migratable objects. One of the more intriguing variants of clobs are ***n-time migratable*** or ***n-copy migratable objects***, described briefly in Sect. 2. To support such clobs, we create new commands that work similarly to the TPM's existing set of commands for supporting migratable keys, except that they take into account the count-limit condition field. These new migration commands must enforce the condition described in Sect. 2. Specifically, if a clob's count-limit range is 1 to n and its corresponding virtual counter on the source TPM A currently has the value c (where $c \leq n$), then TPM A can create a new clob for the destination TPM B with a count-limit range of 1 to k provided that the virtual counter of TPM A's clob is first incremented by k and the new counter value does not exceed n. Given this rule, a clob can be *circulated* indefinitely (i.e., TPM B can migrate the clob back to A, thus creating a new clob with a separate counter from the original one), but the total usable ranges of the count limits of the original clob and clobs migrated from it (as well as clobs migrated from those) cannot exceed n at any one time, where n is the count limit of the original clob.

To *use* migratable clobs, we simply use the modified TPM commands described above (e.g., TPM_Sign, TPM_Unbind, or TPM_Unseal) immediately after a TPM_ExecuteHashTree command as before. The mode in the count-limit condition of a migratable clob determines how the clob can be used. If the mode is INCREMENT, then the total number of times that a clob can be *used* is limited to n regardless of which machine uses them. If the mode is READ, then a host holding a clob can use it an unlimited number of times, as long as it has not been migrated from that host more than its count limit. (That is, if the host migrates the clob more than n times where n is the count limit in the host's copy of the clob, then the counter exceeds the count limit and the host's TPM starts disallowing use of that clob.) This allows for a clob that can be circulated indefinitely and used an unlimited number of times on multiple hosts, but only in at most n hosts at any one time. This variant is notable because it allows for the media "lending" example mentioned in Sect. 2, among other applications.

(Note that here, we assume that a *new* virtual counter must be created at the destination TPM and the counter ID of this new counter must be included by the source TPM in the reencrypted blob. However, if the destination TPM *does* use an old counter whose value is not zero, then there is no security problem because at worst, it can only *reduce* the count limit on the blob, and not increase it.)

One important question for n-time migratable blobs is that of how the source TPM can know that the destination public key is that of a valid and trustworthy TPM. In the current version of the TPM (1.2), the migration commands assume that either the owner of the TPM or the process invoking the commands (which could be a remote process on a trusted machine) is trusted to verify the destination public key and to only authorize migration if the destination public key is that of a valid TPM. The TPM itself does not check the trustworthiness of the destination public key given to it. However, in our model, neither the owner of the TPM nor any processes in the host are trusted. Thus, the TPM needs to be able to verify the destination public key *by itself* so that the secret data in the blob is guaranteed to only be reencrypted for another trusted TPM, and never exposed to any untrusted parties.

One possible solution to this problem is to include a *verification key* inside the clob. This verification key should be the public key of a certificate authority trusted by the issuer of the clob. (Like the count-limit condition, the verification key can be unencrypted but is included in a hash that is in the encrypted part of the blob to prevent the host from altering it.) Then, when the clob is to be migrated, the receiving host presents a valid certificate chain, rooted at the trusted certificate authority, to certify the destination key that it is giving. (An example would be a certificate chain including a DAA signature [4] on the receiving TPM's AIK, which in turn certifies the destination key as a non-migratable storage key on that TPM.) Given the verification key in the original blob and this certificate chain, the source TPM can then verify the destination public key and reencrypt the blob only if the destination key is valid. (Note that the same verification key is included in the reencrypted blob.)

Count-limited TPM operations. Existing TPM 1.2 chips already support the idea of ***wrapped commands*** as part of *transport sessions*. If we extend this idea by creating a clob containing a wrapped command and a count-limit condition, then we can apply various types of count-limit conditions (e.g., n-time-use, n-out-of-m, n-time migratable, sequenced, etc.) to any operation that the TPM is capable of executing. Furthermore, if we create *sequenced* clobs (as described in Sect. 2), with such wrapped commands, then we can create a count-limited *sequence* of TPM operations. This would

be analogous to a transport session, with the advantages that: (1) it would be count-limited, and (2) it can be executed by the untrusted host without needing online contact with the remote party issuing the operations.[5] Note, however, that in these cases, the sequences are no longer atomic operations, unlike the individual wrapped commands, so care must be taken in designing them. Alternatively, we can also allow a single clob to contain a small number of wrapped commands in sequence (as would fit in the TPM's internal memory), so that atomicity can be ensured by the TPM as it executes the operations internally.

Variant: Using physical monotonic counters. Note that count-limited objects can also be implemented if the TPM had a larger – but not necessarily unlimited – number of *physical* monotonic counters. Suppose, for example, that we have a trusted (tamper-resistant) table of N finite-sized "slots" in NVRAM, each indexed by an *address* i. We can use this table to store up to N TPM_COUNTER_BLOB structures, each representing a virtual monotonic counter. Using this trusted table, the read and increment operations can be implemented by simply having the TPM read or increment the appropriate blob directly (i.e., no hash tree computation is required). As in our hash-tree based scheme, we use a *random ID* field together with the slot address of a blob to give the blob's virtual counter ID. This allows us to safely reuse the NVRAM space of a counter which has been destroyed. Given such an implementation, all the variations of clobs we have described can be implemented just as before, except that no hash tree computations are needed anymore to verify and update the counters.

Such an implementation would have the benefits of better performance and reliability (since there is no risk of the host losing the counter blobs and hash tree nodes). The main disadvantage here, of course, is that the number of monotonic counters that the host can keep track of *at a time* would be limited, and thus the number of clobs that a host can hold would be limited too. In some applications, however, this may be acceptable. For example, in digital cash applications, this would simply mean that the host can only hold at most N digital coins at a time, and would need to use a coin before it can get a new one. This is not different from the real world, where a real wallet can only hold a limited number of real coins. The only requirement, then, is for the number of secure NVRAM slots N to be large enough for the needs of the user. Thus, if it becomes possible in the future to implement sufficiently large tamper-resistant NVRAMs, then this variant may be a practical way to implement virtual monotonic counters and clobs.

Note, however, that even if it does become possible to make tamper-resistant NVRAMs large enough for users' needs, using our hash-tree scheme still has its benefits. For one, it would still be much easier for the TPM manufacturer to guarantee the physical security of a single NVRAM register for storing the root hash than that of a large number N of NVRAM slots. Thus, a TPM using our hash-tree scheme can arguably be made cheaper for the same level of security (or alternatively, more secure for the same price) than one depending on many secure NVRAM slots.

Variant: unique clob counter IDs. Another interesting implementation variant is one where the counter ID of a clob is derived from a function (such as a collision-free hash) that generates a *unique* ID based on certain parts of the clob's contents. This allows us to skip the step of having to create a new virtual counter on the host before creating the clob. In this case, the issuer of the clob can simply create the clob and give it to the host. Then, just before using the clob for the first time, the host issues a special command to the TPM, which then computes the unique counter ID from the blob and gives it to the host.[6] Given this unique ID, the host then performs a CreateCounter operation using the given address.[7]

It is important to note that this scheme requires a counter ID address space large enough (e.g., 160-bits) so that the probability of collisions is negligibly small. Otherwise, such collisions can allow someone or something else other than the clob itself to increment the clob's counter (whether maliciously or unintentionally). Thus, this is a case where using our hash-tree-based scheme for implementing virtual monotonic counters offers a significant advantage over using physical monotonic counters as described earlier. Since our hash-tree-based scheme requires only $O(logN)$ steps for each counter operation, implementing even a very large virtual counter address space would still take a reasonable amount of time, and can still be useful in many non-time-critical applications. For example, as noted in Sect. 7, if we assume the speed of present-day TPM chips, then handling 160-bit counter ID addresses would only take around 3 s – which is an acceptable delay if, for example, the clob in question is used for decrypting a media file being migrated from one secure media player to another. In contrast, it is not obvious how one can implement, or even simulate, a collision-free 160-bit virtual counter ID address space using physical monotonic counters, even if it were cheap to implement thousands or millions of these physical counters. (An interesting possibility, however, is proposed in [25].)

7. PERFORMANCE ISSUES

Experimental TPM Performance Results. To get a feel for the practical performance that we can expect to get from our schemes, we measured the execution times of various TPM instructions on an HP DC7600 with a Broadcom TPM 1.2 chip. We used IBM's `tpmdd` device driver [14] as the low level device driver providing the TDDL-level interface, and used JTPM, a Java API that we have developed ourselves to allow us to access TPM 1.2-specific functionality such as monotonic counters and transport sessions, which are not supported by other freely available TPM software stacks today. Note that the TPM is slow enough compared to the main CPU that any slowdown due to the use of Java (vs. C) was verified by us to be negligible.

Roughly, we found that on average, TPM_PCRExtend (which computes the hash of two 160-bit values concatenated together) takes about 12 ms, generating a signature takes about 0.9 s, and a call to TPM_IncrementCounter wrapped in a logged transport session takes about 1.4 s (about 0.4 s to increment the counter, and about 1 s to generate the signature of the transport log). However, on the Broadcom chip, the latter can only be done once every 2.1 s. (To prevent burnout of the monotonic counter's NVRAM, the TPM 1.2 specifications allow TPM implementations to throttle the monotonic counter to be incremented only once every 5 s.)

We have implemented the log-based scheme described in Sect. 4, and have verified that, as predicted, we can indeed execute an IncrementCounter operation approximately once every 2.1 seconds (with the operation itself taking around 1.4 s but requiring a wait

[5] In TPM 1.2 transport sessions, wrapped commands are encrypted with a random session key, and thus requires online contact with the remote party. In the case of a clob with a wrapped command, this is unnecessary since the wrapped command can be encrypted using the public key of the TPM's SRK, just as in other clobs.

[6] The host cannot compute this ID by itself because the function for computing the ID may use secrets encrypted in the blob itself so that the ID can only be computed internally by the TPM.

[7] The TPM in turn cannot "create" the counter by itself because it needs the step inputs from the host in order to update its root hash.

before it is used again). We cannot implement the hash-tree based scheme on a real TPM chip since TPM 1.2 does not support our proposed TPM_ExecuteHashTree command. However, from our measurements above, we can preliminarily estimate that the hash-tree based scheme would take about 1.7 s per operation assuming a 32-level hash tree allowing a maximum of 2^{32} virtual counters (i.e., 0.9 s signing time, plus 32 hash operations at 12 ms each, and around 0.4 s to write to the NVRAM), and would take only about 3.2 s per operation (i.e., same computation as before with 160 hash operations instead) even if we used a 160-bit counter ID address space allowing a maximum of 2^{160} independent virtual counters. Moreover, in a real implementation, the actual time would probably be less because the 12 ms cost per hash operation that we use in these estimate is actually the cost of invoking a separate TPM command. This cost likely includes a significant amount of communication overhead which will not exist in our proposed implementation where all the inputs can be given in one command.

8. RELATED WORK

The idea of implementing "virtual monotonic counters" using a single physical monotonic counter and untrusted storage was previously presented in the context of TPM 1.2 by the TCG [29] and NGSCB by Microsoft [24]. Their schemes, however, rely on a trusted OS, which in turn relies on a trusted BIOS and special security support in the CPU and other hardware (as noted in Sect. 1). To our knowledge, we are the first to present a scheme for implementing a potentially unbounded number of virtual monotonic counters trusting only in a small passive coprocessor like the TPM with only a very small amount of secure non-volatile storage, *without* trusting in the OS or even the CPU.

The idea of data, operations, or programs that can only be used a certain number of times is an old idea that forms the core of several computer security application areas such as DRM, digital cash, and others. For example, one may consider limited-used trial software or media that expire after n uses or n days as a form of count-limited object. Similarly, digital coins in existing digital cash schemes are another form of count-limited object since they are not supposed to be used more than once.

In [2], Bauer et al. present a logic model that can be used to analyze, develop, and prove systems that use what they call "*consumable credentials*" – i.e., credentials that provide authorization only a limited number of times, such as coins, tickets, and similar tokens in both the real and digital worlds. However, as far as we understand, although they discuss how consumable credentials (which are essentially count-limited objects as well) can be *used*, they do not answer the question of how these can be *implemented*.

To date, there have been two main approaches to enforcing the usage limitations of consumable credentials and count-limited objects. One approach is to trust that the hardware and software of the executing platform will prevent a count-limited object from being used outside of its count-limit. This approach is used in existing DRM schemes for limiting the use of software and media files. The problem with this approach, however, is that many implementations of this approach today – including most DRM applications running on PCs – rely on general-purpose non-secure hardware, and implement security through obscurity in the trusted operating system or trusted software. This makes it possible for motivated hackers to eventually be able to break security by disassembling the software.

An alternative approach is not to trust the hardware or software at all, but to design the application such that if a user uses a count-limited object beyond its limit, such use will be detected eventually and the user identified and punished. This approach is used in application areas such as digital cash. In Chaum's e-cash scheme [6], (as well as Brands' scheme [3], described earlier in Sect. 2), for example, the idea of *blind signatures* allows users to engage in legal transactions offline and anonymously, but ensures that if a user double-spends an e-cash coin then his identity will eventually be exposed to the issuing bank, and the bank can prosecute him. More recent examples of "one-time" or "k-time" operations of this type and their applications include [5, 17, 22, 28], among others. The approach used by these schemes has the advantage of being secure even if the hardware or software used by the user is compromised. The disadvantage, however, is that it does not actually prevent malicious activity from happening in offline transactions but only detects and punishes it later. Thus, it is not effective if it is possible for the adversary to hide and escape from being punished, or if there is a need to actually prevent the malicious activity from happening at the time of the offline transaction itself.

Our approach is a variation of the first approach above, with the difference that we do not rely on the security of the host CPU and OS, but only on that of a much smaller, simpler, and passive co-processor such as the TPM. This makes our solutions more secure and harder to break than other existing DRM solutions that rely on a trusted CPU and OS.

In recent work, Goldwasser et al. have coined the terms *one-time programs* and *n-time programs* to refer to *general* programs that can only be run a limited number of times [12]. They have also shown the first implementation (to our knowledge) of such programs using very simple trusted hardware, and have proposed the application of such programs to digital cash and DRM. Their scheme uses a very different technique from ours, and assumes even simpler trusted hardware than ours does. Our schemes and ideas about count-limited keys, data, and TPM operations, and the applications of such count-limited objects to digital cash, DRM, and other application areas were developed independently of Goldwasser et al.'s work, and do not use any of their techniques. Inspired by their ideas, however, we are developing the idea of *count-limited general-purpose programs* in ongoing work.

9. CONCLUSION

In this paper, we make two major contributions: First, we present a *hash tree-based scheme* that makes it possible to implement a very large number of virtual monotonic counters using only a small constant amount of trusted space and a single simple new instruction for the TPM. Unlike previous schemes, our scheme guarantees tamper-evident operation even if everything other than the TPM on the host platform implementing the virtual monotonic counters is completely untrusted, including the software, the OS, BIOS, and even the CPU. This provides a significant improvement in security over existing schemes by making it impossible for hackers to break the security of our scheme without physically breaking into the TPM chip. Second, we show how we can use these virtual monotonic counters with the existing idea of wrapped keys, data, and commands already implemented by the TPM to implement the new idea of *count-limited objects* or *clobs*, which have many useful applications.

The changes to the TPM that can make all these things possible are simple, elegant, and efficient, and are easily implementable given the internal features already present in the TPM. Thus, we hope that the changes we have proposed in this paper will be considered for inclusion in future TPM specifications.

Meanwhile, we have also presented a log-based scheme which can be implemented using the current TPM 1.2 chip without any new instructions. Although the log-based scheme cannot be used to implement clobs, it can be used for virtual trusted storage and stored-value applications.

Finally, we note that our techniques are not limited to systems using TCG's TPM chip, but can also be applied to other secure coprocessor systems as well. For example, our tree-based scheme and our mechanisms for clobs may be a useful feature to include in smart cards or even in security technologies meant to be embedded in CPUs, such as IBM's SecureBlue [15]. Even though these systems are already designed to be fully secure themselves, the benefit of our techniques would be that they provide a way to support a large number of monotonic counters and count-limited objects using only a small amount of trusted space. This potentially makes it possible not only to build smaller and cheaper smart cards and other secure components, but also to improve the security of such components, since a small trusted computing base is much easier to guarantee security for than a bigger one.

10. REFERENCES

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

[2] L. Bauer, K. D. Bowers, F. Pfenning, and M. K. Reiter. Consumable credentials in logic-based access control. Technical Report CMU-CYLAB-06-002, CyLab, Carnegie Mellon University, Feb. 2006.

[3] S. Brands. Untraceable off-line cash in wallet with observers (extended abstract). In *CRYPTO '93*, volume 773 of *Lecture Notes in Computer Science*, Aug. 1993.

[4] E. Brickell, J. Camenisch, and L. Chen. Direct Anonymous Attestation. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, 2004.

[5] L. Bussard and R. Molva. One-time capabilities for authorizations without trust. In *Proceedings of the second IEEE conference on Pervasive Computing and Communications (PerCom'04)*, pages 351–355, March 2004.

[6] D. Chaum. Blind signatures for untraceable payments. In *Advances in Cryptology - Crypto '82 Proceedings*, pages 199–203. Plenum Press, 1982.

[7] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, and G. Tsudik. Itinerant agents for mobile computing. *IEEE Personal Communications*, 2(5):34–49, Oct. 1985.

[8] A. Dent and G. Price. Certificate management using distributed trusted third parties. In C. Mitchell, editor, *Trusted Computing*, chapter 9. IEE, 2005.

[9] E. Gallery. An overview of trusted computing technology. In C. Mitchell, editor, *Trusted Computing*, chapter 3. IEE, 2005.

[10] E. Gallery and A. Tomlinson. Secure delivery of conditional access applications to mobile receivers. In C. Mitchell, editor, *Trusted Computing*, chapter 7. IEE, 2005.

[11] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and Merkle Trees for Efficient Memory Integrity Verification. In *Proceedings of Ninth International Symposium on High Performance Computer Architecture*, New-York, February 2003. IEEE.

[12] S. Goldwasser, G. Rothblum, and Y. Kalai. One-time programs. Personal communication, June 2006.

[13] F. Hohl. Time limited blackbox security: Protecting mobile agents from malicious hosts. *Lecture Notes in Computer Science*, 1419, 1998.

[14] IBM. Linux TPM Device Driver. http://tpmdd.sourceforge.net/.

[15] IBM. SecureBlue. http://domino.watson.ibm.com/comm/pr.nsf/pages/news.20060410_security.html, 2006.

[16] Intel. LaGrande Technology. http://www.intel.com/technology/security/, 2003.

[17] H. Kim, J. Baek, B. Lee, and K. Kim. Secret computation with secrets for mobile agent using one-time proxy signature. In *Proceedings of the 2001 Symposium on Cryptography and Information Security*, 2001.

[18] U. Maheshwari, R. Vingralek, and W. Shapiro. How to Build a Trusted Database System on Untrusted Storage. In *Proceedings of OSDI 2000*, 2000.

[19] R. Merkle. A certified digital signature. In *manuscript*, 1979.

[20] C. Mitchell, editor. *Trusted Computing*. The Institution of Electrical Engineers, 2005.

[21] M. Naor and K. Nissim. Certificate revocation and certificate update. In *Proceedings 7th USENIX Security Symposium (San Antonio, Texas)*, 1998.

[22] L. Nguyen and R. Safavi-Naini. Dynamic k-times anonymous authentication. In *Applied Cryptography and Network Security (ACNS 2005)*, volume 3531 of *Lecture Notes in Computer Science*, pages 318–333, 2005.

[23] S. Pearson, editor. *Trusted Computing Platforms: TCPA Technology in Context*. Prentice-Hall, 2005.

[24] M. Peinado, P. England, and Y. Chen. An overview of NGSCB. In C. Mitchell, editor, *Trusted Computing*, chapter 4. IEE, 2005.

[25] 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 (Extended Version). MIT CSAIL Technical Report (to be published), Sept. 2006. http://publications.csail.mit.edu/.

[26] A. Shieh, D. Williams, E. G. Sirer, and F. B. Schneider. Nexus: a new operating system for trustworthy computing. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–9, New York, NY, USA, 2005. ACM Press.

[27] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *Proceedings of the* 17^{th} *Int'l Conference on Supercomputing (MIT-CSAIL-CSG-Memo-474 is an updated version)*, New-York, June 2003. ACM.

[28] I. Teranishi, J. Furukawa, and K. Sako. k-times anonymous authentication (extended abstract). In *ASIACRYPT 2004*, volume 3329 of *Lecture Notes in Computer Science*, pages 308–322, 2004.

[29] Trusted Computing Group. TPM v1.2 specification changes. https://www.trustedcomputinggroup.org/groups/tpm/TPM_1_2_Changes_final.pdf, 2003.

[30] Trusted Computing Group. TCG TPM Specification version 1.2, Revisions 62-94 (Design Principles, Structures of the TPM, and Commands). https://www.trustedcomputinggroup.org/specs/TPM/, 2003-2006.

[31] D. Williams and E. G. Sirer. Optimal parameter selection for efficient memory integrity verification using merkle hash trees. In *Proceedings of IEEE Symposium on Network Computing and Applications (NCA '04)*, 2004.