Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2006-064 September 11, 2006

# Virtual Monotonic Counters and Count-Limited Objects using a TPM without a Trusted OS (Extended Version)

Luis F. G. Sarmenta, Marten van Dijk, Charles W. O'Donnell, Jonathan Rhodes, and Srinivas Devadas

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

# Virtual Monotonic Counters and Count-Limited Objects using a TPM without a Trusted OS (Extended Version)*

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, digital voting, itinerant computing, and other application areas.

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

## 1. INTRODUCTION

The increasing availability of the Trusted Platform Module (TPM) [32] 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 [11], peer-to-peer computing with enhanced security [2], controlling mobile access to broadcasted media [13], and others [23, 26]. 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 [32], the Trusted Computing Group (TCG) has introduced built-in support for monotonic counters into the TPM [31]. 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*

*A shorter version of this paper will appear in the 1st ACM CCS Workshop on Scalable Trusted Computing (STC'06), ©ACM, 2006. This work was done as part of the MIT-Quanta T-Party project, funded by Quanta Corporation.

*counters*", which in turn can then be used by user-level applications [31]. 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 [27] 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 [12]: (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 [19], 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 [22] 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.

## 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 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

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

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 forms 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 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

[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.

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 [4], 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.

**Digital voting.** The properties that make clobs useful for digital cash may also make them useful for certain forms of *digital voting*. As in digital cash, two major goals in digital voting (among others) are anonymity (i.e., the voter's identity should not be traceable from his ballot) and the prevention of double use (i.e., in this case, double voting instead of double spending). Correspondingly, just as in digital cash, a one-time-use signing key can also be used to address these problems. Specifically, in this case, the election organizers can issue a one-time-use signing clob to each voter. This clob contains the *issuer's* key and can be used by the voter to sign his votes at election time. In this case, double-voting is prevented because the clob lets the voter sign only one set of votes, and then becomes unusable after that. At the same time, the voter's anonymity is preserved because the signature produced over the vote is made using the *issuer's* key, and not the voter's (just as in the digital cash case).[4] An additional advantage here is that if we ensure that the clob is *non*-migratable from its host TPM, then we ensure that a voter can only cast his vote using his personal registered device (with his personal TPM). This makes it harder for a user to give his voting power to another person, since doing so would require *physically* giving the other person the user's personal device at least at the time of voting. This is not only inconvenient for both the user and the other person, but if the TPM involved is also being used by the user for other personal applications (i.e., as an e-wallet, storage for his personal private keys, etc.), then the user also has a strong incentive *not* to give the other person access to his personal TPM.

In addition to anonymity and double-voting, we note that there are several other issues in digital voting that need to be addressed. (For discussions on some of these, see [6, 1].) We do not claim that our scheme above addresses all these problems, but we are currently working on possible solutions to these problems using other functionality that count-limited objects can provide beyond one-time signatures. Meanwhile, although the scheme above may not be good enough for political elections at this time, it may already be useful for less critical applications such as marketing or social computing applications that involve voting and secure counting.

[4]For better security in this application (as well as in digital cash), the clob should actually contain one of *many* keys of the issuer. This way, even in the unlikely event that a TPM would be comprosed and one of the issuer's keys is revealed, damage can be limited. At the same, though, there should not be *too many* keys relative to the total number of voters, because then it would be possible for the clob issuer to intentionally give particular keys to particular voters and thus be able to track their votes later.

**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 [9, 16, 20]. 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 [16].)

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).

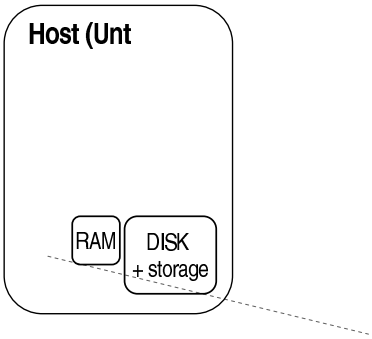
**Virtual vs. physical monotonic counters.** As noted earlier, in order to implement the applications we would need to be able to keep track of a large number of monotonic counters. Although non-volatile RAM (NVRAM) for general storage is rapidly growing cheaper today, *securing* large quantities of 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 make virtual monotonic counters truly tamper-resistant like physical ones. That is, an adversary with access to the untrusted memory can easily erase or alter the data in the memory, causing any counters stored in the affected memory areas to either act incorrectly or fail completely. 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 (preventing other users from using it in the future) or dropping command requests (making the counter look to an external user trying to use it as if it is slow or not working at all). These attacks are still worth noting, but are much less dangerous than arbitrary tampering, since in many applications, it is to 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.

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.


Host (Unt
RAM
DISK
+ storage