Offline Count-Limited Certificates[*]

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

ABSTRACT

In this paper, we present the idea of ***offline count-limited certificates*** (or ***clics*** for short), and show how these can be implemented using minimal trusted hardware functionality already widely available today. Offline count-limited certificates are digital certificates that: (1) specify usage conditions that depend on irreversible counters, and (2) are used in a protocol that guarantees that any attempt to use them in violation of these usage conditions will be detected *even if* the user of the certificate and the verifying party have no contact at all with the outside world at the time of the transaction. Such certificates enable many interesting applications not possible with traditional (unlimited use) certificates, including count-limited delegation and access, offline commerce and trading using cash-like *migratable* certificates, and others. We show how all these applications can be made possible by using only a simple ***trusted timestamping device (TTD)***, which can in turn be implemented using existing trusted hardware devices such as smartcards, and the Trusted Platform Module (TPM) chips embedded in PCs available today. Significantly, our solutions do *not* require trust in any other components in the host machines aside from the TTD itself; they remain tamper-evident as long as the TTD is not compromised, even if the entire host system, including the BIOS, CPU, OS and memory, is compromised. This not only provides better security by minimizing the required trusted computing base, but also makes implementation possible on present-day machines without requiring a particular kind of OS. We demonstrate all these ideas by implementing a prototype application that runs under both Linux and Windows, and presenting experimental performance results.

Categories and Subject Descriptors:
D.4.6 [Operating Systems]: Security and Protection
C.3 [Special-Purpose and Application-based Systems]: Smartcards

General Terms: Security

Keywords: Trusted Platform Module (TPM), smartcards, offline payments, authentication

---

1. INTRODUCTION

Offline count-limited certificates (or ***clics*** for short) are digital certificates that: (1) specify usage conditions that depend on irreversible counters, and (2) are used in a protocol that guarantees that any attempt to use them in violation of these usage conditions will be detected *even if* the user of the certificate and the verifying party have no contact at all with the outside world at the time of the transaction. Such certificates enable many interesting applications not possible with traditional (unlimited use) certificates, including:

Offline count-limited delegation, access, and authorization. Using n-***time-use certificates***, we can allow a user, Alice, to *delegate* to another user, Bob, the authority to execute transactions on her behalf, *but only at most n times*. This would be useful, for example, if Bob is Alice's stock broker, and Alice wants to enable Bob to trade on her behalf but only to a limited extent.

Count-limited delegation in turn can be used to implement count-limited access, in a way analogous to one-time or n-time-use tickets, passes, or authorization letters in real life. For example, in real life, if Alice wants to give a doctor one-time access to her medical records at a hospital, she may write and sign an authorization letter on a piece of paper. Once the doctor presents the letter to the hospital, the hospital takes the piece of paper away (or stamps it with a note saying it has been used). This way, the doctor can only access the records once, but cannot return to get access again in the future (unless he gets another letter from Alice).

Count-limited migratable certificates and offline trading and payments. Counter-based usage conditions for certificates can be used not only to specify n-time-use limits, but also to allow for limiting the *migration* of a certificate. For example, a ***one-copy migratable certificate*** is a certificate that can be transferred from one party to another under the condition that at most only one party can use the certificate at any given time (i.e., after the transfer, the previous holder of the certificate cannot use it anymore even if he keeps a copy and tries to replay this copy offline). Applying this mechanism to delegation, access, and other applications described above can lead to many interesting applications. For example, Alice can create a delegation certificate for Bob, and allow him to pass on this certificate to Charlie (without needing to contact her), but under the condition that once Bob does that, he cannot use the certificate anymore (unless Charlie gives it back to him).

Note that in real-life, cash is actually a form of migratable one-copy certificate, where the certificate is signed by a bank or the government. This means that if we can implement migratable one-copy-use certificates digitally, we can use these certificates as virtual banknotes which would enable people to securely trade and transact *offline* without having to contact the original issuer or any another trusted third party during the transaction. Furthermore, un-

like most existing digital cash schemes today, a scheme using migratable count-limited certificates would allow for "multi-hop" or "circulatable" digital cash, which users can securely pass on to each other without needing to contact the bank or the original issuer of the certificates. (We note, however, that unlike other digital cash schemes, schemes based on count-limited certificates may not necessarily have the same anonymity properties as real cash.)

1.1 The Problem

A major obstacle to implementing all these applications, however, is the problem of **replay attacks**. That is, in an *offline* context and without some form of trusted component, it is impossible to prevent the holder of a count-limited certificate from reverting his machine's state and reusing the certificate beyond its limits with different disconnected parties.

Consider, for example, the following scenario:

1. Alice gives Bob a one-time-use authentication certificate that allows Bob to authenticate himself to anyone, but only at most one time.

2. After receiving the certificate from Alice, *but before he uses it*, Bob copies his machine's entire state (e.g., by copying the hard disk of the machine).

3. Bob uses the certificate with Dave, and succeeds (legally).

4. Bob restores the machine to its old state from step 2.

5. Bob *re*-uses the certificate with Ed, and succeeds (illegally).

This attack works because the states of Bob's machine after steps 2 and 4 are indistinguishable, and because Dave and Ed are disconnected from each other and the outside world; since Ed has no other point-of-reference except for Bob's machine itself, there is now way for Ed to know that Bob has already used the certificate.

In general, as long as Bob is able to restore the *entire* state of his machine to *exactly* what it was before the certificate was used, it is impossible to prevent him from illegally *re*-using the same certificate with different offline parties. This is the reason why to date, it has not generally been possible to implement secure offline count-limited certificates using *software-only* solutions on commodity PCs — i.e., in today's PCs, typically the only place to store the machine's non-volatile state is on the hard disk or some other storage device whose contents are easily copyable and reversible by the owner of the machine (Bob).

Thus, one solution to this problem would be to employ some form of *irreversible state change* in the usage mechanism for these certificates. That is, what we need is some form of *trusted memory* on Bob's machine (other than the untrusted and reversible hard disk) that is somehow changed irreversibly during the usage protocol, such that it would be infeasible for Bob to revert his machine to a previous state. By including the contents of such a trusted memory as part of Bob's state, the replay attack above can then be avoided because the state after step 4 would *not* be the same anymore as the state after step 2 — even if the adversary is able to backup and restore the machine's hard disk.

1.2 Overview

In this paper, we present a solution to the replay attack problem that uses minimal trusted hardware — namely, a **trusted timestamping device (TTD)**. This minimal scheme is significant because it allows implementation using trusted hardware components available today, including in particular, the current version of the Trusted

Platform Module (TPM 1.2) [20], an inexpensive secure coprocessor that is currently becoming a standard component in new commodity PCs and laptops today, and the MTM [19], a version of the TPM for mobile devices. This is in contrast with the original schemes introduced in [16], which enable the implementation of offline count-limited and migratable delegation, authentication, and authorization, but require new hardware features not yet present in TPM 1.2 or the MTM. (These techniques are discussed further in Sect. 5.)

Our paper is arranged as follows: We begin in Sect. 2 by presenting our model of a trusted timestamping device, and giving an overview of our log-based scheme for using such a device to implement offline count-limited certificates that are secure against replay attacks. In Sect. 3, we describe in detail the different protocols for creating, using, and migrating count-limited certificates. We then present a proof-of-concept implementation of these protocols using TPM 1.2 chips in Sect. 4, and some experimental results. Finally, we discuss related work in Sect. 5, and conclude in Sect. 6.

2. SOLUTION OVERVIEW

2.1 Trusted Timestamping Device (TTD)

Abstractly, a **trusted timestamping device (TTD)** is a device with the following key properties:

- It has an **arithmetic monotonic counter**, which is a variable t whose value can be made to go up (by 1) using "increment" operations, but which cannot be made to revert to an older value — even by the owner of the timestamping device.

- It has a **unique private signing key** (SK), which can be used in special timestamping operations to produce unforgeable signatures that can only be produced using the device itself (and which cannot be used to sign arbitrary data). This private signing key has a corresponding **unique public verification key** (PK), which is certified by a trusted certificate authority (using a traditional certificate), and which can be used by any third party to verify the signatures produced with the signing key.

- It supports the following **timestamping operations**:

  ReadSign(rec), which outputs
        ($X = $ ("Read", t, rec), Sign(X)), and

  IncSign(rec), which atomically increments t and outputs
        ($X = $ ("Inc", t_{new}, rec), Sign(X)),

  where rec is a record containing arbitrary data, and where Sign(X) indicates an unforgeable and verifiable signature over X produced by the device using its unique signing key.

- It is **secure** — i.e., there must not be any commands or attacks that would allow an adversary (even one that owns and can give arbitrary commands to the device) to successfully rewind the value of t, or produce valid ReadSign and IncSign outputs without actually invoking the ReadSign and IncSign commands themselves.

Note that although we call our device a *time*stamping device, our form of the device does not actually use real time values, as usually done in secure timestamping (e.g., [9]).

2.2 Protocols Overview

Assuming a trusted timestamping device (TTD) on all users' machines, we can implement count-limited certificates by using a **log-based** scheme — i.e., a scheme that uses logs of the timestamps generated by the TTDs. The idea is roughly as follows (a more detailed and accurate version will be described in Sect. 3):

1. **Creation.** First, the issuer, Alice, gets and verifies Bob's current counter value by asking Bob to do a ReadSign operation on a random nonce. Then, she creates a certificate C (signed with her key), which includes Bob's TTD's PK_B and Bob's counter value. This links the certificate to Bob's TTD, as well as to a particular point in time on Bob's TTD.

2. **Usage.** In order for Bob to use his certificate with another party, Dave, Bob needs to do two things: First, he uses his machine's TTD to perform an IncSign operation on a *usage record* containing (among other things): a random nonce from Dave (to prove freshness to Dave), the desired usage type or *opcode* (e.g., "spend"), and the identity of the certificate which he wants to use. Then, he sends Dave the original certificate C, together with a **proof-of-right-to-use (PRU)**, which consists of a log of all timestamps produced by IncSign operations on Bob's TTD, covering all values of the counter from the creation time signed by Alice up to, and including the timestamp he has just created.

3. **Verification.** The PRU proves to Dave that Bob actually has the right to use the certificate, and that this right has not expired or been depleted. Dave can verify the PRU by going through the log of timestamped usage records to check that Bob is not using the certificate in a way that would violate the conditions signed by Alice in the original count-limited certificate. For example, if the original count-limited certificate is a one-time-use certificate, Dave would check that *none* of the usage records in the log refer to that particular certificate *except* for the very last one which Bob has just timestamped in the usage step above.

Note that this scheme is **tamper-evident**, and satisfies the security properties required for offline count-limited certificates as long as the TTD itself is secure. Suppose, for example, that after using the certificate once with Dave, Bob tries to use it again with Ed. Nothing prevents Bob from *trying* to use the certificate again by calling IncSign again. However, even if he does so, Bob would not be able to produce a *valid* PRU after the first time he uses the certificate. This is because the log he needs to show Ed would still need to start from the creation time of the certificate, and would thus necessarily include the usage record created when Bob used the certificate with Dave. Thus, *even if Ed is offline* and has no contact with Dave, Alice, or anyone else, Ed would still know that Bob has used the certificate before, and would not accept Bob's PRU.

The basic schemes for usage and verification can also be slightly modified to implement **migration**. First, if Bob wants to migrate his certificate to Dave, he begins by getting and verifying Dave's current counter value (by asking Dave to do a ReadSign on a random nonce). Then, using this counter value, he creates a usage record which indicates "migrate" (instead of "spend") as the operation type, and includes Dave's TTD's PK, and Dave's current counter value in it. Given this usage record, Bob then proceeds to use the IncSign operation and give Dave the PRU, just as in the standard usage step. Dave can now verify this PRU (which in this case can be called a "proof-of-migration"), by checking that Bob has indeed used the irreversible IncSign operation, *and* that
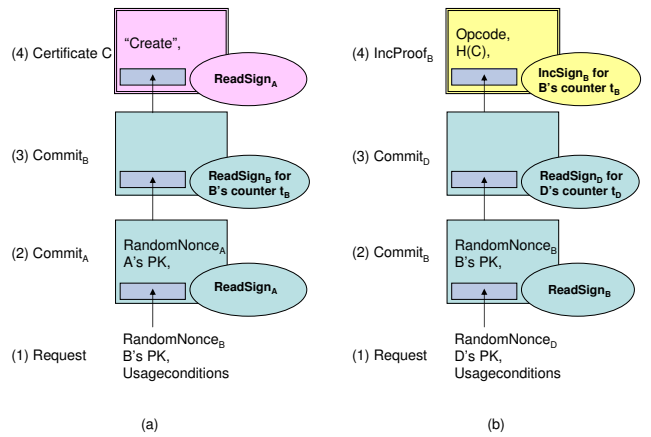

Figure 1: Data structures used in (a) the creation protocol and (b) the usage protocol. Arrows represent containment (i.e., the lower data structures are contained in the upper ones). Ovals represent operations on the data in the rectangles, which produce a tuple containing the data and a signature.

Bob's log does not contain any other *previous* migration operations for the same certificate (which would mean that he does not have the right to migrate the certificate anymore). If Bob's PRU is valid, then the certificate is now considered to be migrated to Dave, and Dave can now himself use or migrate this certificate with/to another party, Ed. Now the usage protocol between Dave and Ed is the same as before, except that in this case, the PRU Dave would present to Ed would not just include the certificate C, and a log of timestamps from his own TTD, but will also include the log of timestamps from Bob's TTD (i.e., the proof-of-migration which Bob had previously presented to Dave). In this way, Ed can then check the whole chain of timestamps (all the way back to the original certificate from Alice) to verify that the usage conditions of the original certificate have not been violated.

3. SOLUTION DETAILS

Section 2 describes the general intuition behind our ideas. In this section, we present the actual protocols in more detail.

Note that a given assumption in all these protocols is that the identity certificates that certify the public verification keys of the different parties' TTDs (i.e., the traditional non-count-limited identity certificates signed by the trusted certificate authority) are available to the parties beforehand. In practice, such certificates would be presented by transacting parties to each other during their first encounter. Each party would then verify the authenticity of the other's certificate and then save the PK in the certificate in a list of "verified" PK's, so that if they meet again in the future, "verifying" the PK's would just involve checking whether it is on the "verified" list.

3.1 Creation

Figure 2 summarizes the creation protocol in which Alice creates and issues a certificate C to Bob. The relationships between the different data structures used in the protocol are depicted in Fig. 1(a). Note that the final output of this protocol is the certificate C, which is given by Alice to Bob. As shown, it contains the following: the issuer's identity (A's PK from step 2), the holder's identity (B's PK from step 1), the usage conditions (requested by Bob in step 1) Bob's current counter value (signed by the TTD in step 3, together with a random nonce from Alice from step 2, which proves

freshness), and finally, Alice's signature over the entire structure (produced in step 4). As noted in Sect. 2, this signature serves to unforgeably link the certificate to Bob's particular TTD, and a particular point in time on that TTD.

3.2 Basic Usage and Verification

Figure 3 depicts and explains the *general* form of the usage protocol, which is used for both spending and migration (as well as any other type of usage). The relationships between the different data structures used in the protocol are depicted in Fig. 1(b). Note that the usage protocol is actually very similar to the create protocol. The difference is only in the final step of the protocol where a valid PRU for C is contructed and transmitted.

Figure 4 depicts an example of spending, wherein Alice creates a count-limited spending certificate C which she issues to Bob, and Bob then spends it once with Dave, and then a second time with Ed. The vertical column on the left depicts the steps of the creation protocol, ending with the creation of the certificate C. The middle box in this column shows Bob's current counter value, t_B, which marks the creation time of the certificate, and is included in the certificate signed by Alice. The row of boxes marked with $t_B + 1$ to $t_B + m$ represent *incproofs* (i.e., outputs of the `IncSign` operation) produced by Bob's TTD at different times after the creation of C. As shown, most of these incproofs refer to *other* certificates, while the incproofs at times $t_B + n$ and $t_B + m$ correspond to the spending of C with Dave and Ed respectively (depicted by the vertical columns at those times).

Note that at step 4 of the spending protocol, as depicted in Fig. 3, Bob is required to present the PRU for C, which consists of the log of incproofs since the creation of C. In Fig. 4, the PRU that Bob shows to Dave consists of the incproofs from time $t_B + 1$ to $t_B + n$, while the one that Bob shows to Ed consists of the incproofs from $t_B + 1$ to $t_B + m$ (including the one at $t_B + n$). Given this PRU, and the original certificate C, the verifier (Dave in the first case, and Ed in the second), can then check all the entries to verify that the usage conditions are not violated. In the case of Ed, for example, Ed would go through the following steps to verify the certificate C and the PRU presented by Bob:

1. Ed verifies the signatures of certificate C and distills the counter value t_B and B's PK. Ed then verifies B's PK.

2. Ed uses B's PK to verify the incproof signatures of *all* the incproofs in the PRU. This is necessary even for incproofs that are not labeled $H(C)$. (Otherwise, Bob can replace a real incproof for $H(C)$ with a fake incproof with a different label, and thus hide a previous usage of C.)

3. Ed distills the counter values from each incproof, and verifies that valid incproofs for all counter values from the time after C's creation time, t_B, to the current time, are presented.

4. Ed extracts a *sublist* of incproofs that correspond to the usage of C. These are exactly those incproofs that are labeled with $H(C)$. In this example, only the n-th and m-th incproofs are extracted.

5. Ed considers the `Opcode` in each of the extracted incproofs and determines the remaining rights of usage of C and verifies whether the usages of C's in the log were valid. For example, note that Ed would see in the log that the `Opcodes` in the two extracted incproofs are equal to "spend". If C's usage conditions describe C as a 3-time-spending certificate, then Ed concludes that C's usages were valid, and accepts Bob's proof. However, if C is only a one-time-use certificate,

then Ed would see that the last spending (i.e., the current one with Ed) is *invalid* because there has already been another spending in the past (i.e., with Dave at time $t_B + n$). Thus, Ed would reject Bob's proof and not give him the goods that he is trying to purchase.

3.3 Migration

Figure 5 depicts an example in which Alice issues a certficate C to Bob, Bob migrates C to Dave, and then Dave spends C with Ed.

The vertical column in the middle depicts the migration step from Bob to Dave. This step is just like the usage protocol except that the opcode used in Bob's incproof is "migrate" instead of "spend". The PRU at this stage consists of the incproofs from time $t_B + 1$ to $t_B + n$. Dave verifies this PRU as in the spending protocol, checking that no violations have occured. This time, however, he also needs to make sure that the final incproof (at time $t_B + n$) includes his (Dave's) identity and current counter value (which he gave to Bob as part of his commitment in step 3 of the usage protocol). If Dave is satisfied with Bob's proof, Dave saves the entire log for future use. At this point, the certificate has now been migrated to Dave, and Dave now has the right to use it. (Bob does not have the right to use it anymore, and will not be able to use it in the future because if he ever tries to spend or migrate the certificate again, the verifier at that time would see the "migrate" incproof at time $t_B + n$ and know that Bob no longer has the right to use the certificate.)

The vertical column on the right represents Dave spending C with Ed. In this case, the usage protocol proceeds as before, except that now, the PRU consists not only of the log of incproofs generated by Dave's TPM (i.e., the incproofs from time $t_D + 1$ to $t_D + k$), but *also* the log of incproofs generated by Bob's TPM (from $t_B + 1$ to $t_B + n$). The verifier (Ed) would then verify the PRU as in the spending protocol, with the additional step of also verifying the migration step from Bob to Dave. That is, in addition to verifying the signature of the incproof from Bob at time $t_B + n$, Ed also distills Dave's counter value t_D from the incproof (it is included as part of `Commit`$_D$), and then verifies that the log on Dave's side starts from $t_D + 1$ and is complete up to the current time $t_D + k$.

3.4 Complex Usage Conditions and General-Purpose Count-Limited Certificates.

Note that a count-limited certificate can have separate and arbitrarily complex conditions for spending and migration. Different conditions would lead to different kinds of certificates.

For example, the most basic and common kind of migratable certificate would probably be a ***one-copy migratable certificate***, which can be migrated from one host to another, but which can only be used by at most one host at any one time. Such a certificate can be implemented by a condition which requires that in order for a user Dave to be able to use the certificate, he must be able to present a valid log wherein the *most recent* incproof that contains a "migrate" opcode in the log is one which designates the user himself as the destination. Note that this maintains the one-copy property because only at most one user (the last one in the chain) would be able to satisfy this property. For all the previous holders, the most recent incproof with a "migrate" opcode would be the one migrating the certificate *away* from the host to another host.

Given, a one-copy migratable certificate, however, many variations are possible depending on the exact usage conditions. One could have an "infinitely circulatable certificate", which can be migrated any number of times, or one could also limit the total number of migrations allowed, creating a "hop-limit" to the certificate. One could also vary the spending condition orthogonally to the migra-

Alice: Bob:

                              (1) Request   Request = (RandomNonce$_B$,
Verify B's PK,   ←——————————————————   B's PK, Usageconditions)
Commit$_A$ = ReadSign$_A$((2) Commit$_A$
RandomNonce$_A$, A's PK, request) ——————————————→   Verify A's PK,
                                                                Verify Commit$_A$,
Verify Commit$_B$,   (3) Commit$_B$ Commit$_B$ = ReadSign$_B$(Commit$_A$)
Opcode = "Create",   ←——————————————————
C = ReadSign$_A$(Opcode, Commit$_B$),   (4) Certificate C
                          ——————————————————→   Save and verify C
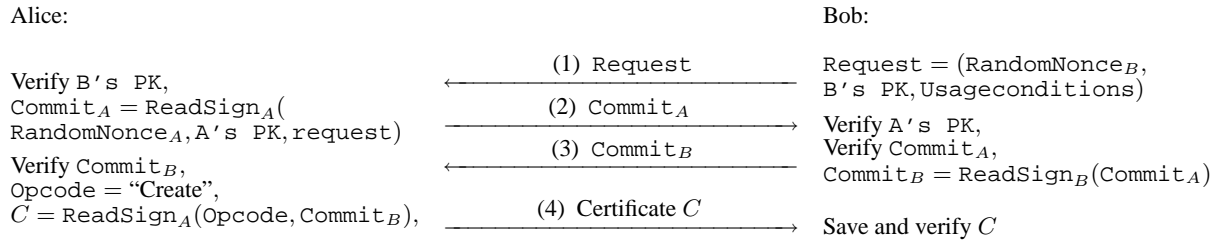
Figure 2: Creation protocol for certificate C. Steps (1-3) authenticate Alice and Bob to one another as well as commit both Alice and Bob to the create protocol. Note that the output of step (3) includes the current counter value of Bob's timestamping device. In step (4), certificate C is created and issued.
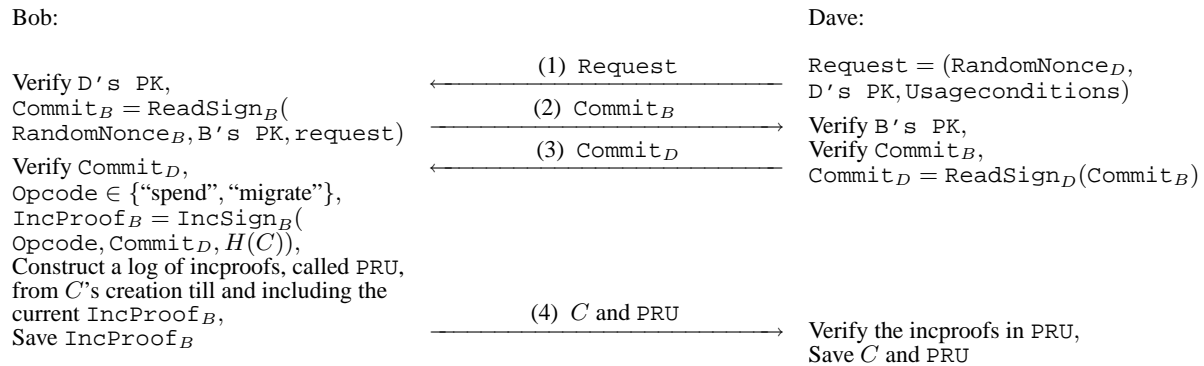
Bob: Dave:

                              (1) Request   Request = (RandomNonce$_D$,
Verify D's PK,   ←——————————————————   D's PK, Usageconditions)
Commit$_B$ = ReadSign$_B$((2) Commit$_B$
RandomNonce$_B$, B's PK, request) ——————————————→   Verify B's PK,
                                                                Verify Commit$_B$,
Verify Commit$_D$,   (3) Commit$_D$ Commit$_D$ = ReadSign$_D$(Commit$_B$)
Opcode ∈ {"spend", "migrate"}, ←——————————————————
IncProof$_B$ = IncSign$_B$(
Opcode, Commit$_D$, $H(C)$),
Construct a log of incproofs, called PRU,
from C's creation till and including the
current IncProof$_B$,   (4) C and PRU
Save IncProof$_B$ ——————————————————→   Verify the incproofs in PRU,
                                                                Save C and PRU

Figure 3: General usage protocol for C (used for both spending and migration). Steps (1-3) authenticate Bob and Dave to one another as well as commit both Bob and Dave to the usage protocol. Note that the output of step (3) includes the current counter value of Dave's timestamping device. In step (4), the output of the IncSign operation, called the *incproof*, forms part of the PRU which Bob presents to Dave, and serves to prove to Dave that Bob has irreversibly spent or migrated the certificate C. The PRU also contains a log of past incproofs to prove that Bob has not already over-spent (or over-migrated) the certificate in the past.
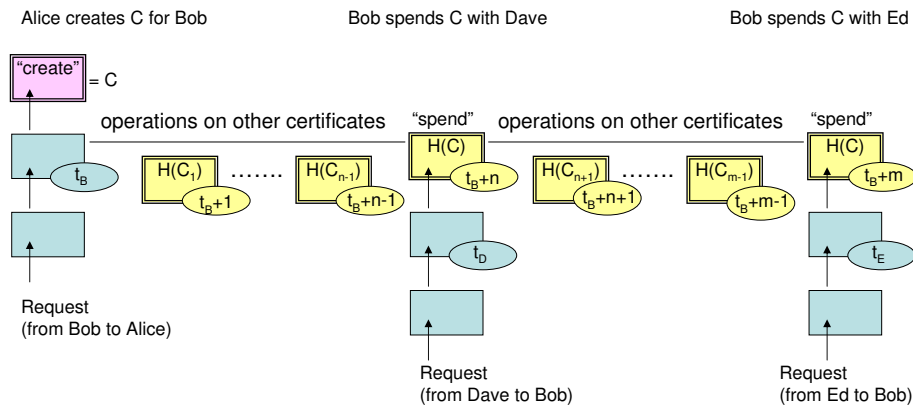
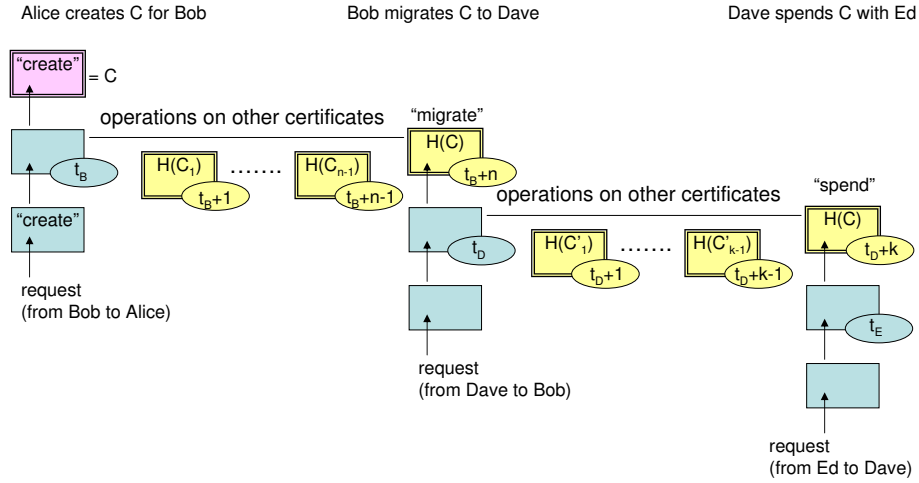

Figure 4: The proof of right to use C during spending.

Alice creates C for Bob   Bob migrates C to Dave   Dave spends C with Ed

"create" = C

operations on other certificates   "migrate"

H(C) t_B+n

H(C_1) t_B+1 ......   H(C_{n-1}) t_B+n-1

t_B

"create"

request
(from Bob to Alice)

operations on other certificates   "spend"

H(C) t_D+k

H(C'_1) t_D+1 ......   H(C'_{k-1}) t_D+k-1

t_D

request
(from Dave to Bob)

t_E

request
(from Ed to Dave)

Figure 5: The proof of right to use C during migration.

tion condition. Thus, one might have a one-copy migratable *unlimited use* certificate, or a one-copy migratable n-time-use certificate, which can be migrated many times but can only be spent at most n times total across all holders in the chain. All this and many other conditions are possible, and can be enforced by simply having the verifier check the sequence of opcodes in the log.

Generalizing this idea to the extreme, we note that the usage and verification protocols can actually allow for an arbitrary set of different opcodes (aside from spending and migration), and for arbitrarily complex usage conditions. This leads to the idea of *general-purpose count-limited certificates*, which can be used to implement many potential applications, such as secure mobile agents, and complex authorization and delegation systems.

3.5 Limitations and Possible Solutions

One obvious drawback of our scheme as described so far is the need for the user of a certificate C to keep and present a log of *all* the increment operations done using the TTD's monotonic counter since the creation of the certificate — even those increments which do not refer to the particular certificate C itself. As will be shown in Sect. 4, this issue limits *performance*, since the communication time required during a transaction will be proportional to the length of this log. At the same time, it also limits *privacy* since the log exposes to the recipient the details of *all* the transactions that the certificate's user has executed since the certificate's creation. For example, in Fig. 4, Ed (the recipient) can see the PK's of all the parties that Bob (the user) has transacted with since time t_B.

The issue of privacy can partly be addressed by using the hash $H(\text{Commit}_D)$ instead of Commit_D in the argument to IncSign in step 4 of Fig. 3. In this way, Ed can verify the incproofs given only Opcode, $H(\text{Commit}_D)$, and $H(C)$, for each incproof in the log. These allow Ed to see whether the incproof refers to the desired certificate, and to see and check the opcodes that have already been executed using that certificate, but does not reveal any of the PKs in any of the intermediate transactions since time t_B, including Dave's, in the case of Fig. 4. (Note that Commit_D includes random nonces unknown to Ed, so Ed cannot use dictionary attacks to derive D's PK from $H(\text{Commit}_D)$.)

This solution, however, still has its limitations. First, in the case above, even if Bob is able to hide the identity of the parties he has transacted with, Ed would still be able to see at least the number of transactions he has performed, as well as *which* certificates

were used in these transactions (as identified by their respective $H(C)$'s). Second, note that when a certificate is used by its holder (Bob) the verifier needs to be given the public key of the holder's TTD in order be able to verify the validity of the incproofs. Also, in the case of migratable certificates, the verifier would similarly need to know the public keys (and thus the identities) of all the senders and receivers of the certificate in the whole migration chain from the issuer all the way to the last user of the certificate.

Occasional log validation by a Trusted Third Party (TTP). One possible solution that can help address both the performance and privacy problems is to allow users to *occasionally* connect to a trusted third party (TTP), *not* during transactions with other users, but at a separate time – i.e., at the users' leisure, and/or when the user has a higher-bandwidth connection to the Internet (e.g., at home at night). At such a time, a user's device can contact an (online) TTP and transmit to the TTP the device's log of incproofs since the last time it has contacted the TTP. From this log, the TTP can check the usage status of each of the user's certificates, and produce *validation certificates* that attest to the usage status of the user's certificates at that time. Thus, in a future transaction with another user (*not* requiring contact with the TTP), the user need only present a PRU containing the validation certificate plus any *new* incproofs, and does not anymore need to present any old incproofs that have been seen and validated by the TTP. This approach not only improves performance (since it decreases the size of the PRUs, and thus decreases communication time), but also improves privacy (since the user's activity before validation certificate is no longer visible to other users). At the same time, it still retains the benefits of offline usability of the certificates since contact with the TTP is not required during user-to-user transactions.

4. IMPLEMENTATION AND RESULTS

We have implemented our protocols and tested them on machines equipped with a TPM 1.2 chip. We implemented the abstract secure timestamping device described in Sect. 2 by: (1) using the TPM's *built-in monotonic counter* as the arithmetic monotonic counter, (2) using an *attestation identity key (AIK)* as the unique private signing key, and (3) implementing the ReadSign(rec) and IncSign(rec) operations by using the TPM's TPM_Read-_Counter and TPM_Increment_Counter command (respectively) inside an *exclusive and logged transport session*, using the AIK as the signing key, and the hash of rec as the input nonce.
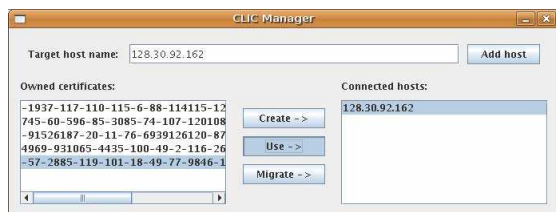
Figure 6: The graphical user interface of our simple prototype application.
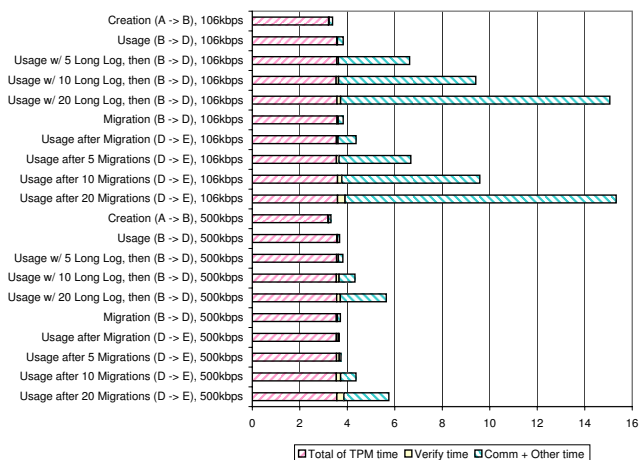

Figure 7: Performance Measurements.

We implemented our prototype application using TPM/J [15], a cross-platform Java-based API for the TPM which provides support for using transport sessions and monotonic counters on TPM 1.2 chips. Figure 6 shows the graphical user-interface of this prototype application. To use this application, a user would start the program on his host, and then type in or select the address or hostname of another host (also with a TPM and running the same program) with which he wants to transact. He can either create a new certificate for that host, or use an existing one from the list on the left. He can also migrate a certificate to another host (if the certificate's usage conditions allow it).

Figure 7 shows some experimental performance results using two laptops with TPM 1.2 chips connected by a bandwidth-limited LAN connection. This setup simulates a scenario where two users' machines are connected to each other using a local connection (e.g., Bluetooth, infrared, etc.), but are disconnected from the Internet. As shown, for each transaction, about 3-4 s in total were spent performing TPM operations (the `ReadSign` and `IncSign` operations, including the monotonic counter operation and the signed transport session takes about 0.9 to 1.4 s each on the TPM chips that we used), and the rest was spent mostly on communication time, increasing proportionally to PRU's length.

Although the performance of our prototype may not be good enough for applications such as allowing access to subway turnstiles by swiping one's device via a contactless interface, we note that it can be acceptable for many other less time-critical applications such as count-limited delegation or access, or even trading between two people with TPM-enabled devices. Note, for example, that when purchasing an item by credit card in a store or through a web site, it is still not unusual for the credit card verification process, which is online, to take 10-15 s or more. We also note that our current implementation of the timestamp operations produces some redundancy in the timestamp data structures, which can be removed at the cost of increasing complexity of the marshalling, unmarshalling, and verifying code. We expect that doing so would result in 30%-50% less communication time.

5. RELATED WORK

Traditional solutions to solving the problems of replay attacks and "double-spending" in distributed computing applications have followed one, or a combination, of three general approaches.

The first approach is to require contact with an *online* trusted third party (TTP), which keeps track of and limits the usage of the object. Online transactions done with credit cards, for example, use this approach, as do certain online multi-player games, and online music and media services. Bauer et al. have presented an interesting idea they call "*consumable credentials*" [1], which has some similarities to our idea of count-limited certificates. Their implementations, however, require the use of online trusted verifiers (known as "ratifiers") to ensure security. The problem with online solutions like this is that they are not usable in offline situations, and thus have more limited scalability and availability.

The second approach is to use mathematical techniques that ensure that if a user uses an object beyond its allowed limit, such use will be detected *eventually* and the user can be identified and punished. Electronic cash schemes (e.g., [7]) use this approach, as do other more recent applications involving certain "one-time" or "n-time" authentication, authorization, or delegation (e.g., [5, 10, 12, 18, 6]). This approach has the advantage of requiring neither an online TTP nor any trusted components during user-to-user transactions. Its disadvantage, however, is that it does not *immediately* detect illegal use, and is thus not effective in cases where it is possible for the adversary to escape from being punished, or where one must catch the malicious activity on-the-spot.

The third approach is to use some form of *trusted component* that enables *irreversibility* as described in Sect. 1. Unlike the two previous approaches, this approach allows offline transactions *and* can catch malicious behavior immediately during the transaction itself. For this reason, this approach is the most widely used in offline applications today. For example, many electronic offline payment systems today rely on special smartcards [14] for storing and keeping track of digital tokens. Similarly, media player devices and software that implement a Digital Rights Management (DRM) scheme that limits the number of times media files can be used or shared typically make use of "secure" modules in the form of special hardware and/or proprietary obfuscated software.

Our solution follows the third approach, but reduces the trusted component to a very simple hardware device (a TTD), and does not rely on the security of any other hardware or software components. This means, for example, that even if a malicious user has the ability to hack the BIOS and OS of his system, tap and alter memory, alter the protocol software, and even alter the execution of the CPU, he still cannot successfully perform replay attacks without being caught by the parties he is transacting with. Thus, our techniques provide better security (since hardware is harder to break than software, and also since simpler hardware is easier to secure than more complex hardware), and also better interoperability and personal freedom (since our techniques can be implemented on any OS on any machine with any *standard* trusted device capable of implementing a TTD, such as a TPM 1.2 chip, MTM, or smartcard, and do not require new trusted operating systems such as Microsoft's NGSCB [13] or new or proprietary secure hardware features).

As noted in Sect. 1, the work here is related to, but not the same as the work described in [16]. In that paper, we proposed techniques for achieving count-limited delegation, authentication, and authorization using *count-limited objects*, specifically *count-limited*

keys, which are encrypted blobs containing *someone else's* private key, the ID of a *virtual monotonic counter* dedicated to that key, and the count-limit condition for the blob. Although these techniques are potentially more efficient (i.e., no logs are required for verification), they require proposed features that are not yet available in existing TPM chips today (and will not be available unless they are included by the TCG in future TPM specifications). In contrast, the techniques here are implementable *today* using existing TPM 1.2 chips, and should also be implementable using the Mobile Trusted Module (MTM) recently defined by the TCG [19] (assuming an MTM that implements monotonic counters and signed transport sessions, which are optional in the MTM).

Moreover, even if the features proposed in [16] become available in future TPMs, our techniques in this paper can still be useful for certain applications. Specifically, we note that count-limited *certificates* are different from count-limited *keys* in that they do not involve allowing the TPM to use *someone else's* private key, even internally. This may provide better security in cases where the compromisability of a TPM chip is a concern. Also, given the ability to create unlimited virtual monotonic counters through the mechanisms proposed in [16], we can actually use the same log-based protocols described here but with a separate *dedicated* monotonic counter for each certificate. This would significantly improve efficiency since the log of incproofs in the PRUs would then only include incproofs pertaining to the certificate itself, and not to any other certificate. (Note that the log-based protocols we introduce in this paper are similar to, but different, from the log-based scheme described in [16] for implementing non-deterministic counters using TPM 1.2 chips. Here, we extend the technique and define new protocols, not proposed in [16], that implement count-limited certificates in general, and support the use of arbitrary opcodes and usage conditions.)

Finally, we note that our techniques also bears similarities with other well-known security techniques. The idea of verifying valid usage by checking a log of secure timestamps has similarities to work in *digital notarization* [9] and *secure audit logs* on untrusted machines [17]. Also, it is already common practice to limit the usage of traditional digital certificates through things such as policies (e.g., [4, 3, 8, 2]), expiration dates, and certificate revocation lists (e.g., [11]. Our work can be considered as another way of implementing limited-use certificates in an *offline* manner, using simple and widely available trusted hardware devices instead of an online trusted third party.

6. CONCLUSION

In this paper, we have presented the idea of ***offline count-limited certificates***, and have shown how these can be implemented using commodity TPM-enabled PCs available *today*, without requiring any changes to these machines or requiring new operating systems. Our implemented prototype applications show promising results, and encourage us to pursue future research in building actual real-world applications using these count-limited certificates.

7. REFERENCES

[1] 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.

[2] M. Y. Becker and P. Sewell. Cassandra: Flexible trust management, applied to electronic health records. In *17th IEEE Computer Security Foundations Workshop (CSFW'04)*, pages 139–154, 2004.

[3] M. Blaze, J. Feigenbaum, and A. D. Keromytis. KeyNote: Trust management for public-key infrastructures (position paper). In *LNCS 1550*, pages 59–63, 1999.

[4] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *IEEE Symposium on Security and Privacy*, pages 164–173, 1996.

[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] J. Camenisch, S. Hohenberger, M. Kohlweiss, A. Lysyanskaya, and M. Meyerovich. How to win the Clonewars: Efficient Periodic n-times anonymous authentication. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 201–210, New York, NY, USA, 2006. ACM Press.

[7] D. Chaum. Blind Signatures for Untraceable Payments. In *Advances in Cryptology - Crypto '82 Proceedings*, pages 199–203. Plenum Press, 1982.

[8] D. Clarke, J.-E. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Science*, 9(4):285–322, 2001.

[9] S. Haber and W. S. Stornetta. How to Time-Stamp a Digital Document. In *CRYPTO '90: Proceedings of the 10th Annual International Cryptology Conference on Advances in Cryptology*, pages 437–455, 1991.

[10] 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.

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

[12] 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.

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

[14] W. Rankl and W. Effing. *Smart Card Handbook (Third Edition)*. Wiley, 2003.

[15] L. F. G. Sarmenta and contributors. TPM/J: Java-based API for the Trusted Platform Module (TPM). http://projects.csail.mit.edu/tc/tpmj/, Dec. 2006.

[16] L. F. G. Sarmenta, M. van Dijk, C. W. O'Donnell, J. Rhodes, and S. Devadas. Virtual Monotonic Counters and Count-Limited Objects using a TPM without a Trusted OS. In *Proceedings of the 1st ACM CCS Workshop on Scalable Trusted Computing (STC'06)*, Nov. 2006.

[17] B. Schneier and J. Kelsey. Secure Audit Logs to Support Computer Forensics. *ACM Transactions on Information and System Security (TISSEC)*, 2(2):159–176, 1998.

[18] 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.

[19] Trusted Computing Group. Mobile Phone Specifications. https://www.trustedcomputinggroup.org/specs/mobilephone/.

[20] Trusted Computing Group. TCG TPM Specification version 1.2. https://www.trustedcomputinggroup.org/specs/TPM/.