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

Luis F. G. Sarmenta (lfgs@mit.edu), Marten van Dijk (marten@mit.edu), Charles W. O’Donnell, Jonathan Rhodes, Srini Devadas

MIT Computer Science and Artificial Intelligence Laboratory

November 3, 2006 (these slides edited on November 4, 2006)
1st ACM Workshop on Scalable Trusted Computing

* This work was funded by Quanta Corporation as part of the MIT-Quanta T-Party project.
Our Paper

• **Monotonic Counter**: A counter whose value cannot be reversed to an old value
  - even if an adversary has complete control of the host machine containing the counter mechanism

• **Enables several offline (and thus highly scalable) applications:**
  - **Replay-evident Trusted Storage using Untrusted Servers**
    * where clients can be offline relative to each other
    * monotonic counters can be used for time-stamping
  - **Count-Limited Objects (“clobs”) and operations (“clops”):**
    * Objects/operations which can only be used once
    * e.g., one-time or n-time use signing/encryption keys, etc.
    * Potential: DRM, offline payment (e-cash), e-voting, etc.

• **Our paper**: *Virtual monotonic counters using TPM without a Trusted OS*

• **Two solutions**
  - Log-based scheme (works with TPM 1.2, but has drawbacks)
  - Hash-tree based scheme (small new proposed TPM functionality)
    * More efficient, and allows count-limited objects and operations
Count-Limited Objects and Operations

- Objects or commands which an untrusted host can successfully use/execute only a limited number of times
  - even if host can keep and replay old objects and data
- Examples and Applications
  - n-time-use delegated signing/encryption keys
    * Alice gives Bob a token which lets Bob to sign/encrypt using Alice’s key n times
    * Useful for n-time offline authorization, authentication, encryption
    * Potential: e-tickets, e-cash, etc.
  - n-time-use decryption keys
    * Bob can decrypt using Alice’s key n times
    * Potential: DRM, Personal DRM
  - shared-counter limited-use objects/operations
    * Several different objects share the same counter
    * n-out-of-a-group operations
    * Interval-limited (including time-limited) operations
    * sequenced and generating clobs/clops
  - n-copy migratable / circulatable objects
    * Users can transfer an object to another user
    * BUT at most n users can use the object at a time
    * Potential: circulatable DRM tokens, e-cash, etc.
  - count-limited (or counter-linked) operations
    * Operations / functions / algorithms in general whose behavior and output depend on values of certain monotonic counters
    * Potential: secure delegated time-stamping, mobile agents, outsourced execution, etc.
How Can We Implement Count-Limited Objects?

- Three general approaches
  - Online Trusted Third Party
    * Used in software/media licensing, online payments, etc.
    * Not always possible. Not scalable. Not topic of this paper.
  - Cryptography
    * Detect and trace double-spending (> n-times use)
    * Works for certain applications (e.g., e-cash, n-time anonymous authentication, etc.)
    * But, cannot prevent double-spending at time of offline transaction
  - Using Trusted Component
    * Require trusted component to produce results
      • can be hardware, software or combination
    * Trusted component securely counts usage of object
    * Actually prevents double-spending at time of offline transaction
    * But, assumes trusted component is not compromised

- We follow the third approach, but using only a TPM
  - Minimize trusted computing base
Count-Limited Objects using Monotonic Counters

• **Note:** We need to keep trusted *independent* state for each object

• *such as … a dedicated monotonic counter per object*
  – Irreversible, non-volatile register
  – Needs to be implemented using secure internal non-volatile memory

• **Problem:**
  – It is hard to have a lot of secure NVRAM in a small secure chip
    * small space inside trusted chip
    * wear-out problem
  – So, existing secure chips only support a few monotonic counters

• **Example: Built-in (aka Physical) Monotonic Counters in TPM 1.2**
  – TPM 1.2 chips can create and keep track of at least 4 independent monotonic counters
  – BUT … can only increment 1 per boot cycle (!)
  – Allowed to throttle increments to once every 5 seconds, good for 7 years
Virtual Monotonic Counters with Trusted OS

• If we have a trusted OS or trusted software, then keeping a large number of monotonic counters is straightforward

• Example: TCG/Microsoft scheme for “virtual monotonic counters”
  – Trusted OS keeps track of an arbitrary number of virtual counters
  – To increment a virtual counter:
    * OS increments global physical counter
    * OS “seals” the new virtual counters’ collective state together with counter’s value as timestamp (can only be decrypted by TPM when Trusted OS is running)
    * OS stores sealed data on untrusted disk
    * OS can detect replay attacks by comparing time-stamp with current value of global counter

• Trusted OS can also enforce Count-Limited Objects/Operations
  – Trusted OS checks the virtual counters before executing clob/clops

• Current DRM-enabled devices do something similar (but not using TPM)
  – either using trusted firmware, or obfuscated software as trusted component
Problems with depending on Trusted OS

- **Problem: Trusted OS is a BIG requirement**
  - requires TPM
  - requires trusted BIOS (CRTM)
  - requires trusted CPU (with special features)
  - requires other hardware support
  - requires new OS, which must be fully tested

- **Can we implement trusted virtual monotonic counters using just a TPM, but *without* a trusted OS?**

- **Note:** Most *real-world* TPM apps that *ordinary* people actually use *today* do *not* use trusted boot
  - E.g., mostly use ability of TPM to protect and use encrypted keyblobs

- **VMCs and Clobs are fundamental primitives that *should* also be supported without requiring Trusted OS**
  - can even help in implementing Trusted OS’s
Our Solutions

• **Using TPM 1.2 : Log-Based Scheme**
  – Use one built-in monotonic counter
  – Use log of increment operations as a freshness proof
  – Good enough for implementing trusted storage on untrusted servers
  – Advantage: works with existing hardware
  – But has drawbacks

• **Better: Hash-tree based scheme**
  – Use Merkle Hash Tree
  – Simple Proposed additional TPM functionality
    * 1 new TPM command
    * 1 word (160-bits) of secure NVRAM space for root hash
  – Advantages
    * More efficient
    * Enables *count-limited* objects and operations
      • (with simple additional changes to other operations)
Log-Based Scheme (Using TPM 1.2)

- **Idea:** Use one built-in monotonic counter as a global counter.
- **On increment of virtual counter A**
  - TPM does an “increment-and-sign” of global counter
    * with nonce = H(virtual counter ID A | client’s random nonce)
- **On read of virtual counter A, client gets**
  - current global counter value
  - Latest inc certificate for virtual counter A
  - Log of inc certificates between A and current time
  - Client checks that no other increments on A were done in between
- **Drawbacks**
  - Non-deterministic
    * Value of individual virtual counter goes up by unpredictable amounts
  - Proof of freshness grows linearly in time
    * If a certain counter is not used while others are used a lot, then proof for that counter can become very long
  - Cannot do arbitrary count-limited operations since TPM does not limit execution
- **Useful for now**
  - Non-deterministic counter is OK for time-stamping and trusted storage
  - n-time use certificates are possible, though complex and unwieldy
Hash-Tree based scheme

- Each Leaf contains an individual virtual counter’s state
  - Virtual Counter ID
  - Current Counter Value
  - Other meta-data

- Leaves and nodes are stored by untrusted OS in untrusted storage
  - Hashes for empty subtrees are well-known, so need not be stored
    * Allows for sparse trees

- Root hash is kept by TPM in trusted internal NVRAM

- All reads, updates, and secure use of virtual counters must invoke TPM as final step
Proposed New Command: TPM_ExecuteHashTree

**Command:** TPM_ExecuteHashTree

**Inputs:**
- int aiHandle, byte mode
- TPM_COUNTER_BLOB counterBlob
- TPM_NONCE nonce
- TPM_DIGEST stepInputs[]

**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.aiHandle = aiHandle
        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 Create new TPM_HASHTREE_EXEC_CERT execCert
   c. if (hts.mode has EXECUTE bit set)
      then remember hts for immediately following command else erase hts
   d. Return execCert
   e. // 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.
Proposed New Command: TPM_ExecHashTree

**Inputs**
- AIK handle
- mode (Read, Inc, Inc&Exec, Create,...)
- anti-replay nonce
- Counter Blob
- Internal hash tree nodes
- Optional: Wrapped command

**Output**
- “Execution Certificate” signed by AIK
- OR, output of wrapped command

**Relatively Easy to Implement**
- 1 new TPM command
  - plus backward-compatible modification to count-limitable operations and data structures
- 20 bytes (160-bits) of secure NVRAM for root hash
- All internal operations required here are already supported by TPM (e.g., hash)
Read Virtual Counter

- Host feeds TPM
  - Counter blob
  - Internal hashes
    * Sibling hashes on path to root
- TPM computes root hash based on input
  - \( O(\log N_{\text{max}}) \) internal hashing operations
- If computed root hash matches trusted stored root hash,
  - then TPM outputs certificate (signature by AIK) certifying virtual counter blob as being fresh
- Note: If adversary rewinds or modifies leaves or internal nodes
  - root hash will be different
  - TPM will detect and abort

\[ \text{Is Computed root same as stored root?} \]
Increment Virtual Counter

• Same inputs as Read

• Difference: As TPM goes up tree, it computes two sets of hashes based on two counter values
  – The current value
  – The new value
    * (based on counter value + 1)

• If computed root hash based on current value matches trusted stored root hash, then:
  – TPM updates internal rootHash with computed root hash based on new counter value
  – TPM outputs certificate (signature by AIK)
    * certifying that inc was done
    * Indicating and certifying new counter value
Count-Limited Operations

• Same input as above
  PLUS wrapped command
  – Sort of like transport session

• Mode specifies Read or Increment
  – Normally, use increment
  – Read mode allows for objects which can be used unlimitedly until something else increments the same counter
    * e.g., revocable key delegation

• If computed orig root hash does not match stored root, then fail

• If it matches, then
  – perform increment (if desired),
  – verify that (new) current counter value satisfies count-limit conditions of command / object
  – if so, execute command
  – Output output of command directly
    * Optionally, wrap output in exec. cert.

```plaintext
TPMchip
NVRAM
rootHash

TPM_ExecHashTree
( aikHandle, mode, nonce,
  c1101, [ h1100, h111, h10 ],
  { TPM_Sign(...) } )

Output
Of
TPM_Sign
{...}
```
Count-Limited Keys

• Existing TPM feature: wrapped keys
  – Alice can give Bob encrypted blob containing her PK-SK keypair
  – Alice encrypts blob using Bob’s TPM’s storage key’s PK
    * SK of storage keypair is never revealed outside the TPM
    * So, only TPM can decrypt and use Alice’s SK in the blob
  – To use:
    * Use TPM_LoadKey to load blob into TPM → returns key handle
    * Use TPM_Sign, etc. with key handle
  – Note: currently, wrapped keys are NOT count-limited

• Modifications to TPM
  – Add count-limit condition field to wrapped key
    * Includes virtual counter ID, valid range, and allowed/required modes
    * Put in a variable-length field where PCR configuration is now
  – When key is loaded, condition is remembered
  – Upon doing a TPM_Sign using that key, check condition
Using Count-Limited Keys

• Scenario: Alice wants to give Bob a 1-time key

• Issuing (Alice and Bob)
  – Step 1: Alice certifies Bob’s TPM and gets Bob’s storage key
    * e.g., check Bob’s AIK’s PK vs. known/certified value or via DAA
  – Step 2: Alice creates a new virtual counter on Bob’s host
    * Bob executes TPM_ExecHashTree
    * gives new counter ID and exec certificate to Alice who verifies it
  – Step 3: Alice encrypts a key blob using Bob’s storage key containing her keypair
    and gives to Bob
    * include count-limit condition
      • Virtual counter ID, required mode=Increment, and valid range (in this case “1”)

• Use (Bob alone, offline from Alice)
  – Step 1: Bob uses TPM_LoadKey on encrypted key blob
  – Step 2: Bob calls TPM_ExecHashTree with wrapped TPM_Sign/TPM_Unbind/etc
    * gets relevant hash tree nodes from his storage
    * Calls TPM_ExecHashTree
    * Computes and stores new counter value and new hash tree nodes
Applications of Count-Limited Keys

- **n-time authentication / authorization / certification**
  - Authority gives Bob a wrapped count-limited signing keypair PK-SK
    * where SK is unknown to Bob,
    * and PK is certified and verifiable as coming from the Authority
    * count-limited to n
  - When Bob needs to prove certification to Charlie
    * Charlie gives Bob a random nonce
    * Bob uses count-limited signing key to sign nonce
    * Charlie verifies Authority’s signature on nonce
  - *Bob can only do this at most n times*

- **This leads to many potential applications**
  - Offline payment: Authority is Bank, signature has cash value
  - E-tickets (probably more feasible)
  - etc.

* Caveat on privacy and resiliency
Caveats

• Note: Anonymity can be preserved because the final output contains nothing from Bob
  – Only Charlie’s nonce, and Authority’s signature
  – (Note: Charlie does not need to verify/identify Bob, because Authority’s signature is enough proof)

• Caveats
  – #1: If Authority uses single global key, then TPMs must never broken
    * If a single TPM is broken, Authority’s private key is revealed. Very bad!
  – #2: If Authority uses multiple keys, then anonymity may be broken
    * At time of issuing, Authority may give Bob a unique key, and be able to link the key to Bob’s AIK (used by Authority to verify Bob’s TPM)
    * Solution (?): Use DAA at time of issuing so Authority can’t link AIK to Bob

• In the end, probably, the best solution for critical apps (e.g., real e-cash) is to use crypto-based n-time-use techniques, but use virtual monotonic counters to count-limit these in hardware
  – e.g., implement a TPM command implementing Brand’s e-cash scheme [Brands93], but store the e-coin as a count-limited object stored outside the TPM
  – Provides hardware support for immediate prevention of double-spending
    * assuming TPM is not broken
  – AND also provides eventual traceability in case TPM is broken

• However, simple schemes based on straightforward count-limited RSA signing operations may still be useful in non-critical applications (i.e., where the cost of breaking a TPM would be much more than the potential gain one can get by doing so)
  – Advantage is that minimal change is needed in the TPM, and no need to define for special-purpose commands/algorithms for each application
Other Variations on Clobs

- **shared-counter limited-use objects/operations**
  - e.g., Alice generates several *different* wrapped objects depending on the *same* virtual counter ID
  - Possibilities
    * N-times-within-a-group operations
    * Interval-limited operations
      - Can translate to time-limited if trusted clock increments counter

- **n-copy migratable objects**
  - TPM already has a migrate key feature
  - Idea: count-limit the migration
    * Assume that *usage* of key reads but does not increment counter
    * But *migration* of key increments counter
    * If Alice migrates a key to Bob, then Alice’s counter gets incremented, so Alice can’t use her copy anymore
    * On Bob’s side, Bob gets a new key tied to a virtual counter on his TPM
    * Bob can use it until he migrates it to someone else (possibly Alice!)
  - “Lendable” objects → *circulatable* DRM, e-cash, etc.
  - Possible to make n-copy (not just 1-copy) circulatable objects
    * Circulatable but at most only n copies at any given time are usable
  - Challenge: Verification must be done by TPM (not host)
    * Verification key must be included in blob

- **Others**
  - See our MIT CSAIL Technical Report, Sept. 2006
Other Variations on Hash-Tree based scheme

- **Split TPM\_ExecHashTree into 2 commands**
  - Start() command, followed by Step() command for each level of tree
  - Advantage: no need to feed all internal tree node hashes (sibling hashes) to the TPM at once
    * works even if TPM only has small input buffer space
  - **Note:** internal volatile memory requirement of TPM does NOT grow
    * computation of hashes and updating of state is done at each step
    * no need to remember all the node hashes
    * Hash tree state is constant-sized
  - **Note:** Failure before the end is not a security problem
    * TPM state is only changed at the very last step if everything succeeds
  - **However, not clear whether splitting is even necessary**
    * we can handle 32 levels ($2^{32}$ virtual counters) with only $20 \times 32 = 640$ bytes for the sibling hashes
      - Even with other input data, total input size would still be much less than 4K typical input buffer space of TPM 1.2 chips
    * maybe it can be useful for 160-bit (unique) virtual counter ID’s

- **Other Variants**
  - Multiple root hashes (allows independent hash trees, possibly of different depths)
  - Dynamically growing hash trees
  - Caching
  - Have TPM\_ExecHashTree support operations other than increment
    * “mode” field can indicate different kinds of operations
    * e.g., Extend (i.e., one-way hash) can lead to unlimited PCR-like “hash clocks”
    * e.g., Read, Update Virtual Trusted Memory
    * This is why we recommend keeping the command name TPM\_ExecHashTree generic
      * it’s not limited to just monotonic counters
  - Multiple counter operations per TPM\_ExecHashTree invocation
    * e.g., increment several counters with one TPM\_ExecHashTree invocation
    * saves on time for signature operation in the end, and also saves on wear out of root hash NVRAM
  - VMCs and count-limited objects/operations using physical monotonic counters
  - Count-limited wrapped commands
    * Encrypted TPM commands with a count-limit condition field
  - Count-limited general-purpose commands

- See MIT CSAIL TR 2006-064 (Sept. 2006) for details
Related Work

• Of course, general idea of n-time-use operations is an old idea
• Some interesting/relevant related work
  – “Consumable Credentials” (Bauer et al. 2006)
    * Logic for analyzing/modeling systems whose security depend on limited-use credentials
    * Currently, they assume an online trusted third party, though
  – Cryptographic Techniques
    * Classic e-cash, etc.: Chaum82, Brands, etc.
    * Lots of other recent work:
      • E.g., n-time anonymous authentication, etc. (e.g., CHKLM, ACM CCS 06)
  – Using Trusted Component
    * Practically all DRM systems fall under this category today
  – Using combination of Crypto and Trusted Hardware
    * e.g., Brands93 talks about “observer” that stores a special value per e-coin in trusted memory and forgets it after using the e-coin once
    * Our approach can be used with this algorithm, and would allow a much larger number of values to be remembered using very little trusted NVRAM
  – One-time or n-time arbitrary programs using very simple hardware
    * Slightly prior to us, Goldwasser et al. have proposed a theoretical scheme using very simple hardware (not a secure processor like TPM). (Not yet published.)
Ongoing / Future Work

• Applications
  – Virtual Storage, Offline Payments, etc.
  – (We’re starting with what we can do with TPM 1.2)

• CLAMs – counter-linkage modules
  – implement VMCs and clobs/clops mechanisms and ideas using other secure components in general, not just TPM
  – using other trusted hardware (e.g., smart cards, IBM 4758, AEGIS, SecureBlue, etc.)
  – or, potentially even CLAMs using obfuscated software and/or trusted OS
    * less secure but more immediately implementable and useful

• How can having VMCs and clobs/clops as a primitive help improve the design of future trusted modules, platforms, and software?
Conclusions

• Virtual Monotonic Counters and Count-Limited/Linked Objects are small but potentially extremely useful primitives

• We have presented 2 solutions
  – Using TPM 1.2 log-based
  – Hash-tree based scheme (better)

• It would be great if TCG incorporates this functionality into the next TPM
  – Very simple to implement
  – Potentially very powerful
For more info

• Email:
  – Luis Sarmenta (lfgs@mit.edu)
    * http://people.csail.mit.edu/lfgs
  – Marten van Dijk (marten@mit.edu)

• MIT CSAIL TR 2006-064 (Sept. 2006) has some more details
  – http://hdl.handle.net/1721.1/33966