Automatic Event-Stream Notarization Using Digital Signatures

Bruce Schneier John Kelsey

Counterpane Systems, 101 East Minnehaha Parkway, Minneapolis, MN 55419
{schneier,kelsey}@counterpane.com

Abstract. Some digital signature algorithms (such as RSA) require messages to be padded before they are signed. Secure tokens can use these padding bits as a subliminal channel to embed auditing information in their signed messages. These auditing bits simplify protecting against lost and stolen tokens, breaks of specific protocols, hash functions, and ciphers, and attacks based on defeating a token's tamper-resistance.

1 Introduction

We present a signature format which simplifies the task of designing strong protocols for tamper-resistant tokens, like smart cards. The basic idea embeds auditing information within the block to be signed. Packets signed by RSA are typically 512- to 1024-bits long, sometimes even longer; hashed messages are only 128- or 160-bits long. These hashed messages are padded to the length required by RSA, generally with fixed bits. Some fixed bits are required to prevent arbitrary bit strings from being valid signatures and other cryptanalytic attacks against RSA, but far more bits are available than are needed to prevent these attacks. These bits can be more usefully used for other things.[1]

Many protocols add auditing information to messages before they are hashed and signed, but that increases the length of the message and often cannot be enforced from within a token. Our signature format is essentially a subliminal channel [Sim84], under the control of the token—one that cannot be affected by any party in the protocol—and allows the token to embed auditing information in everything it signs. The token has this ability regardless of the protocols that use it, even if it does not create or hash the actual messages.

The most generally useful part of the auditing information included is the hash chain [And95]: a cumulative hash of everything the token has signed to date. Every signed message depends on every previous signed message; this makes it very difficult for an attacker to make any valid-looking changes in previous transactions, even if he manages to defeat the token's tamper-resistance. This idea is used in [And92,HS91].

[1] Jean-Jacques Quisquater and Louis Guillou suggested embedding message bits within an RSA signature [QG95], which was the impetus for this idea.

Our protocols attempt to prevent as many potential attacks as possible. For those attacks which we cannot prevent, we attempt to minimize their damage and increase their cost to the attacker. Because token systems of this kind may be implemented in many different legal environments, we do not generally assume that the law will be helpful in tracking down incidents of fraud or misuse [And93,And94].

We also attempt to increase the potential evidence available in the even of fraud. Fraud does not happen in isolation: someone does not steal $1M and then disappear. He is likely to spend some of the money. While our protocols might not prevent someone from stealing $1M, they will lead investigators to potential suspects for that theft.

1.1 Token Resources Needed

The tokens discussed in this paper require the following resources for most or all protocol steps:

a. Sufficient storage to record all protocol messages in some kind of log. This is needed to provide an audit trail. Some implementations can store this log on the token itself, while others may require the user keep his own audit log and surrender it in case of a dispute.
b. Sufficient non-secure storage to maintain a current public-key certificate for the token's internal key.
c. An internal, secure private/public key pair, with the private key not known by any entity except the token, and the public key certified by the token manufacturer or certification authority. (This is essentially the "Verify-Me" key in National Semiconductor's CAKE proposal [Swe95].)
d. Several internal, securely stored values: a counter, key ID, token ID, and chained hash value.
e. Secure facilities for performing digital signatures and message hashing.
f. An internal, secure source of random bits.

Additionally, some protocols will also require:

g. Secure facilities for performing public key and symmetric encryption.
h. An internal, secure clock.
i. A non-secure link with some external device, such as a hard drive controller or a LAN or modem card.
j. Some small additional secure memory, and the ability to do very simple operations upon it, such as subtraction and comparison with zero.

2 Building the Signature Packet

This section defines a specific format for signature packets. This signature format is incompatible with existing formats: e.g. PKCS [RSA93]. A signature packet is a relatively large block of data which is digitally signed with message recovery by the card. The best known digital signature scheme which allows signatures with recovery is RSA. The DSA algorithm cannot be used for this, since it does not allow message recovery. Digital signature schemes based on the discrete logarithm problem which allow message recovery are discussed in [NR96]. For the remainder of this paper, we will assume RSA.

The signature packet has the following fields:

a. Signature packet version
b. Token ID
c. Signing key ID
d. Packet sequence number
e. Hash of hash of most recently signed packet
f. Hash of hash of most recently received packet
g. Optional 64 or 128-bit data field
h. $hash(hash(message), hash(a--g))$

Each of these fields is useful for preventing or increasing the difficulty of some kinds of attack.

Signature packet version The signature packet type and version tells the receiver of the packet how the packet is to be processed. (Many protocols will also require a message-type byte in the message, rather than the signature packet.) This is a 24-bit field, with the following structure:

bit 0	– always zero
bits 1..7	– reserved—set to all zeros for version 1.x
bits 8..15	– major version byte
bits 16..23	– minor version byte

A given version number corresponds to a single signature algorithm, key size, packet format (including use and meaning of optional fields), hash function, symmetric and asymmetric encryption algorithm, as well as other things. As an example, Version 01.00 might correspond to signature and asymmetric encryption algorithm RSA with a 768-bit modulus, hash function SHA1, symmetric encryption function two-key triple-DES, and no defined optional fields. Version 01.01 might correspond to the same thing, with a 64-bit optional field defined as a token-generated random number. A recipient of a signature packet with version 01.01 would then know that, if the sender's tamper-resistance hadn't been defeated, the random number that appeared in the optional field was not under the control of the sending token's owner.

By including the signature packet version, we prevent some kinds of replay attack which involve trying to get a system to use an older packet version. We also make explicit our packet version, so that it is simple for the card to internally scan the version and accept or refuse to accept it. We allow backward compatibility if later versions add more fields or change their width, since the first 24 bits of the packet can always be easily found. Note that the high-order bit must always be a 0 in RSA-based versions.

Token ID Each token has a unique 64-bit ID, possibly further subdivided into 16 bits defining a specific manufacturer and 48 bits identifying a unique token. By including the token ID, we dramatically simplify the problem of tracing lost or stolen tokens. In most cases, lost or stolen tokens are dealt with using only the token ID, key ID, and sequence number.

Signing key ID Each key has a unique 64-bit key ID. This key ID is always included in key certificates, and can easily be fit into an X.509 certificate. The purpose of the key ID is mainly to make easy to find a unique identifier of the key being used to sign the signature packet. (Note that some tokens may have more than one signing key.) In some applications, 16 bits may signal a specific certification authority, and 48 bits this key's specific ID. As long as the key ID is unique, this does not present a problem.

Packet sequence number The packet sequence number is 32 bits long, and is incremented by 1 every the token signs a message. Its purpose is to frustrate most replay and insertion attacks on protocols, and also to make it easier to trace lost or stolen tokens by issuing a "stop transaction" order on all transactions after sequence number X. Additionally, this ensures that there are never two identical signature packets generated by the same properly- functioning token. Finding two packets with the same sequence number, key ID and token ID is evidence that something very bad is happening with that token.

Hash of hash of most recently signed packet This field is included so that the sequence of signatures from a given token will form a hash chain. This ensures that, even when a token's private key is compromised, it won't be possible for an attacker to modify previously-completed transactions. The audit trail created by this hash chain is as difficult to bypass as it is to find collision values for the hash function. A similar hashing chain is used to build a digital timestamping service in [HS91].

We include the hash of the previous hash, instead of the hash itself, to avoid any chance of an attack based on an untrusted application providing this value to the token.

Hash of most recently received message This field ensures that the sequence of messages in a protocol form a hash chain. This makes replay and "cut and paste" attacks impractical, and makes every step in each protocol dependent on every previous step. We include the hash of the previous hash, instead of the hash itself, to avoid any chance of an attack based on an untrusted application providing this value to the token.

Optional 64- or 128-bit data field This field can be included in the packet to carry internally generated random values or timestamps, often with the guarantee that if the token's tamper resistance hasn't been defeated, these values come directly from the token and not from the user.

$\boldsymbol{Hash(message, a--g)}$ This is the hash of the rest of the signature packet and the current message. Note that by including the hash of the other party's most recent message in the current message, we get every protocol message cryptographically dependent upon every previous protocol message, in an auditable chain that extends through every transaction performed by each token. This is intended to make recovery from various kinds of attack as easy as possible and to leave a very strong audit trail. As a side-effect, this chain of hashes frustrates insertion and replay attacks in any protocol where the chained hashes are checked and all messages in the protocol are signed.

Note that prepending (rather than appending) a–g would leave the hash chains vulnerable to some message-extending attacks after a token's private key had been compromised.

2.1 Size of the Packet

The packet's minimum size is determined by the size of the hash function's output and by whether or not the optional 64- or 128- bit data field is used. For a 160-bit hash, the minimum packet size is 664 bits. For a 128-bit hash, the packet size is 568 bits. For most practical implementations of digital signatures, there is sufficient room for both the signature packet and additional redundant information to frustrate cryptanalysis.

3 Protocol building blocks

There are three protocol building blocks (subprotocols) which, along with the signature packet format, simplify the design of robust and secure protocols. We assume that, while executing these protocols, no other signing operations are performed with the token. In other words, the token must complete all the steps of one protocol before starting another.

3.1 Interlock

The interlock subprotocol is intended to convince two tokens (referred to in this section as Alice and Bob) that they are communicating with one another in real time [DP89]. If all messages passed after the interlock subprotocol is performed are signed using the specified signature packet formats, then attacks based on insertion, alteration, or replay of messages should be impossible.

If C_A is Alice's certificate, and C_B is Bob's certificate, the interlock protocol works as follows:

(1) Alice:
- (a) Generates a random number, R_0.
- (b) Forms message $M_0 = (R_0, C_A)$.
- (c) Sends to Bob $M_0, Sign(M_0)$.

(2) Bob:
- (d) Verifies C_A.
- (e) Verifies $Sign(M_0)$.
- (f) Generates a random number, R_1.
- (g) Forms $M_1 = hash(M_0), R_1, C_B$.
- (h) Sends to Alice $M_1, Sign(M_1)$.

(3) Alice:
- (i) Verifies C_B.
- (j) Verifies $Sign(M_1)$.
- (k) Verifies $hash(M_0)$.
- (l) Forms $M_2 = (hash(M_1), FirstProtocolMessage)$.
- (m) Sends to Bob $M_2, Sign(M_2)$.

What is "FirstProtocolMessage"? Is it M_0?

(4) Bob:
- (n) Verifies $Sign(M_2)$.
- (o) Verifies $hash(M_1)$.

At the end of step (3), Alice has seen enough to verify that she is getting a response by someone who knows Bob's private key, as specified in C_B, because he has returned a signed message which includes a hash of her first (partially random) message, and a key certificate. After step (4), Bob can verify that he's getting a response from someone who knows Alice's private key, as specified in C_A, because he, too has gotten an appropriate response to his message. In both cases, Alice and Bob each know that the other party received their entire certificates intact. This may guard against some obscure attacks, such as where Alice is talking to Bob legitimately but simultaneously Bob is pretending to be Alice to a third party.

Note that verifying the certificates means verifying the signatures, the valid dates (possibly the issued-date of the other party's certificate against the issued-date of the token's own certificate), and possibly checking the certificates against a list of known stolen or invalid key IDs or token IDs.

It's important to note that while Alice the token knows whom she's dealing with (Bob, whose token ID and key ID are clearly noted inside C_B), there isn't any cryptographic way to ensure that Alice's human owner knows which Bob she's interacting with. (This is known as the "Chess Grandmaster's Problem.") For applications in which this is a problem, it is a good idea to equip the token with some kind of display, and to show some human-readable identification from C_B. This gives the owner of Alice the token an opportunity to end a transaction in which she doesn't want to be involved, or at least the knowledge that she has been involved in this unwanted transaction.

3.2 Interlock With Privacy

The interlock subprotocol can easily be extended to support a secure key exchange and encrypted communications. The protocol is identical for substeps (a) through (k).

(1) Alice:
  (a) Generates a random number, R_0.
  (b) Forms message $M_0 = (R_0, C_A)$.
  (c) Sends to Bob $M_0, Sign(M_0)$.
(2) Bob:
  (d) Verifies C_A.
  (e) Verifies $Sign(M_0)$.
  (f) Generates a random number, R_1.
  (g) Forms $M_1 = hash(M_0), R_1, C_B$.
  (h) Sends to Alice $M_1, Sign(M_1)$.
(3) Alice:
  (i) Verifies C_B.
  (j) Verifies $Sign(M_1)$.
  (k) Verifies $hash(M_0)$.
  (l) Generates a random number, R_2.
  (m) Forms $KE = (PKEncrypt(R_2, key = PK_{Bob}), PKEncrypt(R_2, key = PK_{Alice}))$.
  (n) Forms $M_2 = (hash(M_1), KE)$.
  (o) Sends to Bob $M_2, Sign(M_2)$.
  (p) Forms session key $KS = hash(R_0, R_1, R_2)$.
(4) Bob:
  (q) Verifies $Sign(M_2)$.
  (r) Verifies $hash(M_1)$.
  (s) Forms session key $KS = hash(R_0, R_1, R_2)$.

In all messages after this one, Alice and Bob sign the plaintext messages, then encrypt them and their signature packets under a symmetric algorithm. (The specific symmetric algorithm should be specified by the signature packet version.) Note that KE is encrypted under both Bob and Alice's public keys, so that either token can reproduce the session key, and thus present the plaintext that was originally signed for audit. In some systems, it may also be necessary to encrypt R_2 under an auditor's public key.

3.3 Trusted Values from the Card

Some protocols benefit from having the card generate some internal, trusted value, such as random numbers or timestamps. In this case, the purpose of the interlock operation is going to be for Alice to get this trusted value from Bob. The simplest way to do this is to go through the interlock process, with Alice's FirstProtocolMessage set to a request for a random number or a timestamp. Bob's response is an acknowledgement message, signed with a signature packet which also includes a random value. Note that the signature packet version is used to indicate that this packet's random number or timestamp emerged from the token. Some tokens may always include random numbers in their signatures. (It's important to note that this kind of token would be especially vulnerable to attacks based on a subliminal channel in the random number stream; it would be trivial for a tamper-resistant device to leak 64 bits of private key per signature. This needs to be guarded against.) For applications which need these trusted values to be private as well, we use the Interlock-with-Privacy subprotocol, and the next message requests a random number or timestamped signature packet. Note that after Interlock-with-Privacy, all further communications, including signature packets, are encrypted.

4 Sample applications

In this section, we give several sample applications using this signature packet and the above subprotocols. These applications aren't meant as finished products, but instead as examples of the usefulness of this kind of construction.

4.1 The Digital Timestamping Proxy

Haber and Stornetta have designed some clever methods for timestamping digital documents [HS91]. Some variations of their methods can be used by a tamper-resistant token, to allow it to function as a proxy for a master digital timestamp server. For example, the card may sit on an internal network, and be available for users of the network to timestamp their documents. Perhaps once per week, the card interacts with the master timestamp server, leaving the hash of its chain of hashes with the server. So long as the hash function is secure, this chain of hashes can't be altered even if the tamper resistance of the card is defeated somehow. In this protocol, additionally, Bob is assumed to have a tamper-resistant clock. (The protocol can be done less elegantly without the clock.)

Alice, Bob, and Trent are the players in this protocol. Alice is a user's card, Bob is the timestamping service proxy, and Trent is the timestamping service. (As discussed below, it turns out that Trent doesn't need to be just one device; it can be a network of cooperating devices.)

When Alice has a hash value to get timestamped, she interlocks with Bob, and sends her hash value as her first protocol message. After Alice and Bob have completed steps (1) through (4) in the interlock protocol, Bob performs the following:

(p) Forms $M_3 = hash(M_2)$.
(q) Sends to Alice $M_3, SignWithTimestamp(M_3)$.

At this point, Alice has a verification of her hash, timestamped by Bob the tamper-resistant card.

When Bob interacts with Trent, things work the same way. Bob fills the part of Alice in the interlocking protocol, and Trent takes the part of Bob. Bob ends up with a timestamped verification of his hash value, which is the hash of his chain of hashes since his last interaction with Trent. Bob must also send his chain of hashes to Trent, as his FirstProtocolMessage.

If someone steals Bob, the users can get together to recreate their interactions with Bob and can (with Trent's help) produce authentication of the order of their timestamps since Bob's last interaction with Trent. Of course, every interaction before that has been saved by Trent, and can be authenticated with a consistent hashing chain and with newspaper-published hashes.

If someone hacks into Bob, or there are other reliability problems with Bob, we can use similar techniques to maintain the integrity of our timestamps. It is important to note that for some applications, Alice would need to send an entire file to Bob, so that she could not conveniently lose it later if it held some incriminating information about her. Even if Trent is lost or subverted, we can use all of the proxies to reproduce complete, internally-consistent hash chains. In a real-world system, these hash chains would be backed up off-site on a regular basis.

4.2 Auditable Applications

Many applications need a strong audit trail. In particular, security-relevant operations should generally be logged, and it should be very difficult to delete or alter the logs, and impossible to do so without being detected. The most obvious applications for this kind of thing are for key certification and key escrow agencies; in both cases, it should be infeasible to perform some operation (certify public keys, recover private or secret keys) without leaving an audit trail.

Any auditable application can be attacked in at least one simple way, which cannot be prevented by cryptographic means. We will call this the "end-run" attack. The end-run attack happens when a group of users gets together and sets up their own separate system which isn't audited. To the extent that most or all of the resources controlled by an auditable application can be acquired outside the system, this attack is effective. The obvious example of an end-run attack is a key-escrow system whose escrowed keys are kept in a carefully-audited database for police use, and in a "black" unaudited database for intelligence agency use. In real-world systems, this attack needs to be guarded against, but we cannot provide much protection from this at the cryptographic level. About the only thing we can do is to ensure that all messages to and from the auditable application are encrypted. This makes it significantly harder for a rogue organization with some, but not all users cooperating with it to keep its black system in synch with the legitimate system.

Another potential attack exists any time we leave authenticated logs with a user who could be prosecuted for whatever information is on the logs. The user may decide he's better off to "accidentally" delete the logs, and perhaps wind up in jail for it, than to certainly go to jail for the evidence that exists on the logs. We will call this the "Watergate attack." This can't be guarded against by cryptography, but it can be guarded against by good system design.

These are the players in this protocol: Alice is the auditor's card, Bob is the user's card, Carol is the card that's controlling the audited application. We also have Mallory, the possibly malicious network owner. He can insert, delete, change, replay, and observe messages between Carol and Alice and Bob. He has a card that he's hacked open, which may or may not be on Carol's list of acceptable Bob users.

Bob uses the application by interlocking with privacy with Carol, and then by sending a request for some operation or information. If Bob is authorized to do this operation (Carol must know Bob's identity at the end of the interlock-with-privacy protocol), Carol carries it out. In any case, the protocol messages are kept in a log. Additionally, Carol encrypts the session key used under Alice's public key as well as its own in substep (m) of the protocol. Any response data is sent back to Bob, encrypted.

Alice regularly interacts with Carol. First, they Interlock- with-Privacy, and then Alice requests copies of all logs since her last interaction. Carol verifies that Alice is authorized for this, and if she is, sends her the logs, encrypted and signed. Alice can use this and her knowledge of the signature packet format and Carol's public key to verify all transactions that have occurred in the time covered by the logs.

Bob the card may have his tamper-resistance defeated, but while this allows Bob's human owner to hand out his private key to his friends, it doesn't keep him from being audited. Even defeating Carol's tamper-resistance and recovering her private key doesn't allow repudiation of old logs, only of future ones.

Mallory can prevent messages from getting into and out of Carol. He can even alter her log. However, by doing this, he can't frame anyone: any alterations he makes are detected by Alice when she interlocks with Carol (and thus has the correct ending chained hash), and notices that there is an inconsistency. Mallory can destroy the logs, but only for the short period of time between Alice's interactions with Carol. In addition, if Mallory wants to reverse engineer Carol, he has to somehow convincingly interact with Alice and all of the Bobs out there while he's doing it; otherwise everyone will know something is going on.

4.3 The Guaranteed Checking Account

Many secure payment protocols have been published. Ours is relatively simple, and intended to demonstrate the ease with which we can build good protocols from these signature packets and starting protocols.

This system is meant to allow users to have a guaranteed checking account, where identity is verified by both possession of a smart card and by a PIN, and where sufficient funds to cover each "check" is guaranteed by the tamper-resistant card. It is important to note that the account that these cards draw on must be frozen while the cards are active; otherwise they can't possibly know how much money they are allowed to spend. This system protects its users' privacy by encrypting transactions, no anonymity is supported. This is a design feature: recovery from many kinds of problems involves the ability to trace a given user's transactions.

Any two tokens, Alice and Bob, can transfer money freely between them, so that if Alice has \$500 and Bob has \$200, it's possible for them to interact to distribute that \$700 in any way they choose. This works just like a checking account. However, there should be no way for them to interact in such a way that their total money after their transactions is more or less than \$700. Hence, they are allowed neither to overdraw, nor to burn money in their fireplace.

Three interesting things can happen in this system: Alice can transfer some money to Bob, Alice can reconcile with Dave the banker, and Trent the auditor and Dave the banker can try to recover from some attack.

Spending Alice and Bob are tokens. The purpose of this protocol is to allow Alice to transfer some money to Bob. The first part of this is that Alice's human owner requests a transfer of $\$X$ to Bob. Alice verifies that her current internal balance is more than $\$X$. If not, she refuses to initiate the protocol. This is not seen as an attack, it's seen as a standard operating mode of the system, protecting the user from an embarrassing miscalculation.

Assuming she has the money, things proceed as follows: First, Alice and Bob interlock-with-privacy. For this application, the KE value in substep (m) of the protocol includes the session key encrypted under the bank's public key. Also for this application, note that the certificates contain valid date and sequence number ranges. Each token's certificate has an issued-date. If the issued-dates of the two tokens' certificates are more than T days apart (the smaller T is, the more often tokens must interact with the bank, but also the less time a hacked token has to write bad checks), then the tokens refuse to accept the certificates as valid, and the interlock-with-privacy protocol fails. This is used to limit the total amount of time that a rogue card (one which has been reverse-engineered) can possibly write bad checks.

Next, Alice sends Bob an encrypted and signed "check" which says something like "Alice's account number transfers $\$X$ to Bob's account number." To prevent attacks based on spoofed account numbers, their account numbers are the hashes of their public keys. These have been exchanged with the certificates in the interlock-with-privacy protocol, as described above. At this point, the protocol ends with an acknowledgement message from Bob. If Alice doesn't receive the acknowledgement message, she flags it as an error condition and assumes that the money has been transferred to Bob. This should be relatively rare, but it needs to be defined to prevent some classes of attacks. Bob knows whether he's got the money, because he can assume he has money as soon as he has received Alice's signed check. Now, Bob and Alice each adjust their internal balances, and go on about their business.

Note that if Alice the token has been reverse-engineered or hacked, she can write bad checks to Bob. The bank has guaranteed these, and we can't assume that the authorities most places will be terribly helpful, so the bank has to have some faith that tamper-resistance is hard to defeat, so that this occurs only very rarely, and also that it can recover quickly from these rare events, and freeze the hacked token before its owners can make much of a profit. (Losing $100,000 or more on their attempted scam is more likely to deter them from trying again than having the police after them.)

Reconciling with the Bank Alice is a user's token and Dave is a bank's token. The purpose of reconciliation is for Alice to send her accumulated logs of transactions to Dave, and then for Dave to send her a new certificate, and a new list of invalid keys or certificates. This list should be fairly short, since the certificates themselves are pretty short-lived.

The T parameter defines how often each user must reconcile with the bank, because the bank issues certificates. If T is set to 20 days, then each account owner must reconcile with the bank every 20 days, or their token can't operate with anyone. If T is set to 5 days, then we wind up with only a very short window for a user with a hacked token and write bad checks, before his token is permanently frozen. Similarly, if Alice complains to the bank that she was mugged by some guy who'd just watched her punch in her PIN to buy something, they won't re-issue a valid certificate to the token, so the robber only has a few days of spending left.

There may even be discounts for high-volume users to operate with a lower T, perhaps reconciling once per day. Additionally, all tokens that reconcile after the robbery is reported or the bounced checks are noticed have this token ID and key ID on their reject list, so it should quickly become difficult for a robber to use his stolen token, or for the user of a hacked token to write bad checks. Reconciliation should be possible by telephone so long as the token's certificate hasn't lapsed. If it has lapsed, then the token should need to be brought into a branch office of the bank. This gives us some chance of noticing physically hacked tokens, and also leaves us with pictures of some of the people involved on our security cameras.

The reconciliation protocol works as follows: First, Alice and Dave interlock-with-privacy. Next, Alice sends Dave her entire transaction log since her last reconciliation, encrypted and signed. Dave verifies that her transactions don't disagree with other information available to him, and that all the logs are internally consistent. Alice may also request some additional transactions, such as moving money into or out of this account. Dave either performs these, refuses, or forwards them to some other party which can decide whether to perform them. Dave sends Alice her new current balance and her new certificate in the same message, and Alice verifies it, and sends Dave a receipt. Dave signs it and sends it back to Alice. If she doesn't get this receipt, she must call back and interact with Dave again to deal with this. If Alice has had some transactions end without proper receipts, she notes this in her transaction logs, and Dave reconciles this as well as possible: If he hasn't yet heard from the other parties in that, he leaves her balance as she has calculated it, but notes that if the transaction doesn't show up in those parties' logs, Alice should get the money back.

As soon as Dave learns of overdrawn checks or stolen or lost tokens, he adds the token's certificate (with its key and token ID) to the bad certificate list, and refuses to issue that token another certificate until problems have been resolved. This should require intervention from another token, Trent.

Audit Dave routinely interacts with Trent, the auditor and timestamp service, at least once per day. He Interlocks-with-Privacy and sends encrypted, signed logs and running hash chains that have come in since the last interaction, and gets back a timestamped receipt. These, along with the chained hashes available in Dave's logs, protect the logs from someone defeating Dave's tamper resistance and changing previous logs. In addition, Dave needs to be under good physical security, including continuous surveillance camera coverage.

If a problem arises, things have to be handled by humans using legal and accounting, rather than cryptographic, methods. However, it should be possible to verify that the logs are correct up to some point.

Trent also is able to (traceably) re-authorize frozen key and token IDs. Dave will take no other source for these orders. Trent may well be several tokens that must act together to authorize this.

4.4 Distributed Secure Applications

All of the schemes above can be implemented with the "overseer" role (Trent the timestamping service, Alice the auditor, Trent the auditor) performed by a network of cards interacting. We will outline this using the timestamping service as an example.

Instead of having one Trent, we have some large number of time- stamping proxies. Each is implemented in tamper-resistant hardware, with a tamper-resistant clock. Instead of publishing hashes, the network of proxies continuously interacts. It's fairly easy to draw a network in which each proxy connects to four others: every so often, proxy A backs up its logs with an interaction with proxy B, getting a timestamp from B in the process. This kind of design can make it arbitrarily difficult to spoof a digital timestamp. Similar designs can work for key certification or key escrow, audited applications, payment protocols, etc.

5 Further Work

Other audit fields could be embedded into the signature packet, which may be useful in some applications: the identity of the signer (in the case where multiple signers share the same token), the identity of the application producing the signature, and the intended recipient of the signature. These fields are different than the ones described above in that they must be explicitly told to the token by the user or application, and hence can be forged.

6 Conclusions

We have presented a signature packet format which automatically includes the key ID, token ID, a sequence number, and the current value in a running hash chain. It is hoped that this format can be used to develop token-based protocols which are more robust than was previously possible. Additionally, we feel that these techniques could be useful design principles for software implementations for abstract protocols.

7 Acknowledgments

The authors would like to thank Paul Kocher and David Wagner for their helpful comments, and the National Semiconductor iPower Business Unit for helping fund this research.

References

[And92] R. Anderson, "UEPS — A Second Generation Electronic Wallet," *Computer Security — ESORICS '92*, Springer-Verlag, 1992, pp. 411-418.

[And93] R. Anderson, "Why Cryptosystems Fail," *Communications of the ACM,* v. 37, n. 11, Nov 1994, pp. 32-40.

[And94] R. Anderson, "Liability and Computer Security: Nine Principles," *Computer Security — ESORICS '94,* Springer-Verlag, 1994, pp. 231-245.

[And95] R. Anderson, "Robustness Principles for Public Key Protocols," *Advances in Cryptology — CRYPTO '95,* Springer-Verlag, 1995, pp. 236-247.

[DP89] D.W. Davies and W.L. Price, *Security for Computer Networks,* Second Edition, John Wiley & Sons, 1989.

[HS91] S. Haber and W.S. Stornetta, "How to Time-Stamp a Digital Document," *Journal of Cryptology,* v. 3, n.2, 1991, pp. 99-112.

[NR96] K. Nyberg and R. Rueppel, "Message Recovery for Signature Schemes Based on the Discrete Logarithm Problem," *Advances in Cryptology — EUROCRYPT '94, Springer-Verlag,* 1995, pp. 182-193.

[QG95] J.-J. Quisquater and L. Guillou, "DSS and RSA,"presented at the rump session of Eurocrypt 1995.

[RSA93] RSA Laboratories, "Public Key Cryptography Standards #1: RSA Encryption Standard," version 1.5, 1 November 1993.

[Sch96] B. Schneier, *Applied Cryptography, 2nd Edition,* John Wiley & Sons, 1996.

[Sim84] G.J. Simmons, "The Prisoner's Problem and the Subliminal Channel," *Advances in Cryptology: Proceedings of CRYPTO '83,* Plenum Press, 1984, pp. 51-67.

[Swe95] W.B. Sweet, "Commercial Automated Key Escrow (CAKE): An Exportable Strong Encryption Proposal, Version 2.0," National Semiconductor iPower Business Unit, 4 June 1995.