Title: Logcrypt: Forward security and public verification for secure audit logs
Author: Jason Holt ⟨*isrl@lunkwill.org*⟩

1 Introduction

The popular application Tripwire keeps cryptographic fingerprints of all files on a computer, allowing administrators to detect when attackers compromise the system and modify important system files. But Tripwire is unsuitable for system logs and other files that change often, since the fingerprints it creates apply to files in their entirety. Several people have proposed cryptographic systems which allow each new log entry to be fingerprinted, preventing attackers from removing evidence of their attacks from system logs. Logcrypt makes significant improvements to existing work by Bellare, Yee, Schneier and Kelsey.

In the systems described first by Bellare and Yee and then Schneier and Kelsey, a small secret is established at log creation time and stored somewhere safe, such as on a slip of paper locked in a safe or on a separate, trusted computer. The secret stored on the computer is the head of a hash chain, changing via a cryptographic one-way function every time an entry is written to the log. This secret is used to compute a cryptographic message authenticaion code (MAC) for the log each time an entry is added, and optionally to encrypt the log as well.

If the system is compromised, the attacker has no way to recover the secrets used to create the MACs or decryption keys for entries in the log which have already been completed. He can delete the log entirely, but can't modify it without detection. Later, the administrator can use the original secret to recreate the hash chain and check whether the logs are still intact. To keep an attacker from interfering with this process, this should happen on a separate, secure machine.

MACs may also be sent to another machine as they're written. Then they can serve as commitments to log entries. A radiologist, for instance, could send MACs for each MRI image she creates to an auditing agency. Later, she could produce the images in court and the auditor could vouch that the images she presented match the MACs she sent out. But otherwise, the auditor would have no way of knowing what the images were. The radiologist is protected from accusations of fraud, and the patient's privacy is protected.

Logcrypt builds on the Schneier and Kelsey system, and adds several significant improvements to their system. The most significant is the ability to use public key cryptography with Logcrypt. Using the symmetric techniques just described, any entity who wishes to verify a log must possess the secret used to create the MACs. This secret gives the entity the ability to falsify log entries as well, which could be a major drawback in many applications. Public key cryptography allows signatures to be created with one key and verified with a different one. Such signatures can be used in place of MACs to allow verification of a log without the ability to modify it.

Such publicly verifiable logs could be used for systems which need to be publicly audited, such as financial books for publicly held companies and voting systems in democratic countries. When such logs are properly created and their initial public keys sent to external auditors, not even their creators can go back and change entries entered before the creator decides that the information is unfavorable. For example, an honest system administrator could set up a log to record all financial bookkeeping entries, sending the initial public key to external auditors before the first entry arrives.

After each entry is recorded, the private key used to create it is destroyed automatically. Later, the CFO approaches the administrator and demands that certain entries be replaced to hide poor quarterly results. But the administrator has no power to do so – the private keys for the existing entries have been deleted, and the auditing agency will be able to detect any missing or modified entries if it ever verifies the log. Of course, the CFO can prevent *future* entries from being recorded properly, but existing entries are irrevocably tamper-evident.

Other improvements we describe include a method of securing multiple, concurrently maintained logs using a single initial value, and a method of aggregating multiple log entries to reduce latency and reduce computational load.

2 Related Work

In 1997, Bellare and Yee [2] proposed the fundamental technique on which Logcrypt is built. Their system closely relates to the simple system we present in section 4. They mentioned the idea of forward security using signatures, and in 1999 proposed a forward secure signature scheme [3], but did not further discuss its application to secure audit logs.

Schneier and Kelsey proposed a very similar system in [6], [7] and [8]. Their system also closely relates to the simple system we present in section 4, but neither system addresses public verifiability, metronome entries, multiple concurrent logs, or high load conditions.

Chong, Peng and Hartel discussed the possibilities offered by tamper-resistant hardware in conjunction with such a system in [9], and implemented Schneier and Kelsey's system on an iButton.

Waters et al described how identity-based encryption can be used to make audit logs efficiently searchable in [10].

3 Overview

Assuming Logcrypt's design, construction and underlying cryptographic primitives are sound, then if a system is secure from tampering at a time t, and the computational overhead from Logcrypt's cryptographic operations takes l milliseconds, then log entries created until time $t - l$ will have forward secrecy in the sense that tampering will be detectable with overwhelming probability.

Three elements of Logcrypt form the foundation of its security:

1. Logs begin in a known state which is recorded in a secure external system.
2. The security of an earlier state can be used to verify the integrity of a later state, assuming the system is secure in both states.
3. Once a secret is used to secure a log entry, it is erased from memory as soon as possible.

Hash chains make it easy to fulfill these requirements. In a hash chain, a secret s is hashed by a cryptographically strong one-way function to produce the next links in the chain, $s_1 = h(s)$, $s_2 = h(s_1)$, etc. One-wayness means it is assumed to be computationally infeasible to find s_1 given s, even though calculating $s_1 = h(s)$ is generally quite efficient.

The simple system we propose is essentially a simplification of the system described in [6]. We first define our simple system, then give the public-key variant, and then describe several other features relevant to real systems.

We have to be careful in describing the assurances our system makes. Once an attacker gains complete control over a system, he can control virtually everything that happens from that point. Consequently, our system provides tamper-evidence by removing secrets from the system as soon as possible: once a secret has been destroyed, that secret can effectively protect information through cryptography.

4 Simple System

Our fundamental system is illustrated in figure 4. An initial secret is used to start a hash chain in which each link is used to derive keys for a single log entry.

As soon as a key is used, it is erased from memory. Likewise, the link in the hash chain used to create each entry must be erased as soon as it has been used. Consequently, only the bottom link in the hash chain and the keys it generates exist in memory while the logging system awaits a new entry.

Figure 4 illustrates how Logcrypt can provide confidentiality as well as integrity. A second key is derived from each link in the hash chain and used to encrypt the entry.

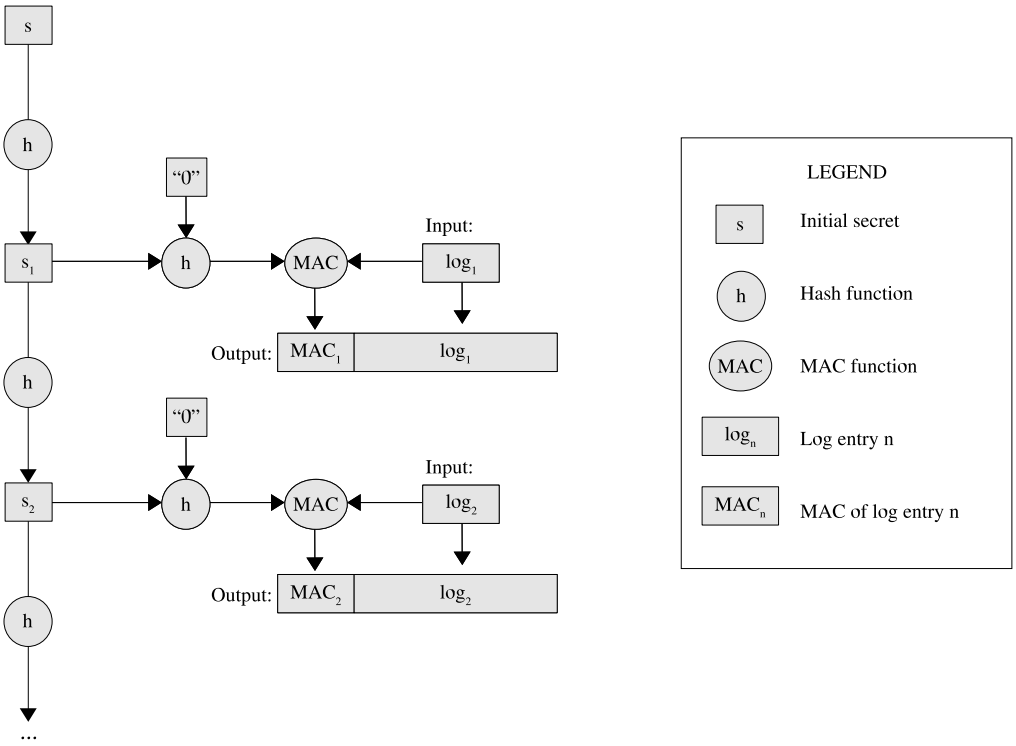

Figure 1: Simple forward security using Message Authentication Codes.

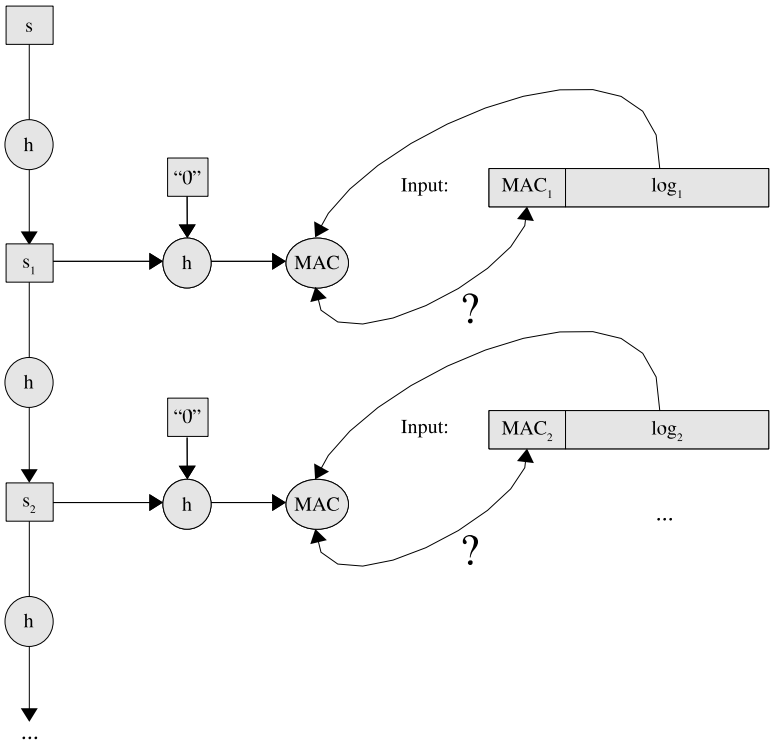

Figure 2: Verifying entries in the simple scheme.

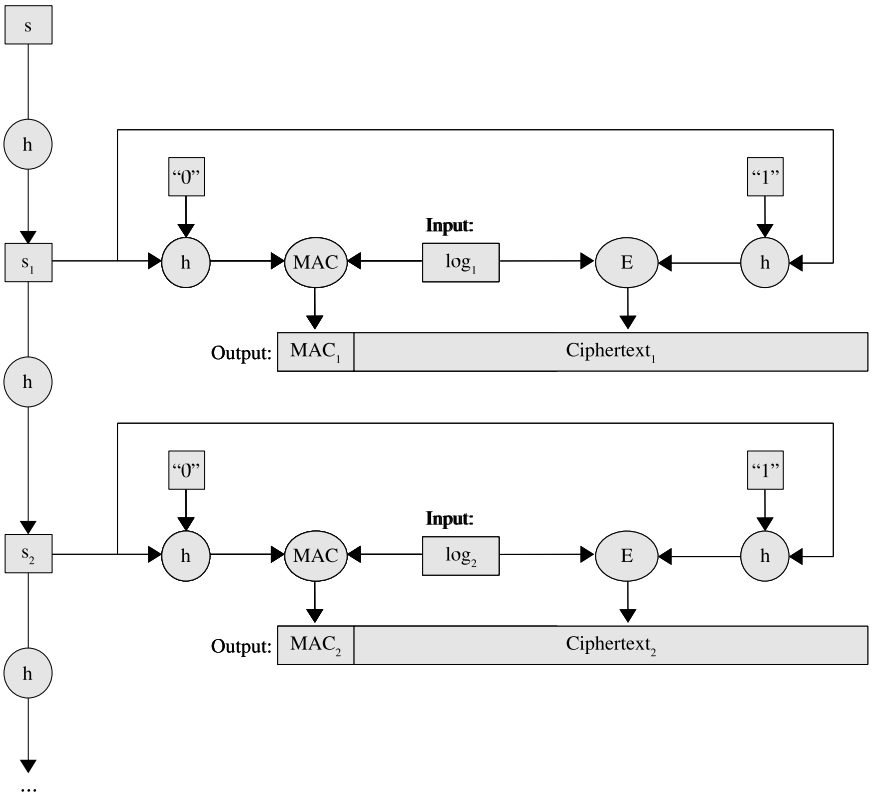

Figure 3: Forward security plus secrecy.

More formally, the Logcrypt algorithm for MAC-based integrity protection with optional encryption is as follows:

Given

A secure cryptographic hash function $h(input)$

A secure message authentication code $MAC(secret, plaintext)$

A secure encryption function $E(secret, plaintext)$

Begin

Randomly generate or accept as input an initial unique secret s

Store s securely (generally accomplished by sending it to a separate machine)

Loop

Ensure that the next 3 steps completely destroy the prior values:

Calculate the next link: $s = h(s)$

Derive the MAC key: $mkey = h(0|s)$

Derive the encryption key: $ekey = h(1|s)$

Wait for the next log entry: log_n

Let $MAC_n = MAC(mkey, log_n)$

If encrypting,

$ciphertext_n = E(ekey, log_n)$

Output $\langle MAC_n, ciphertext_n \rangle$

Else

Output $\langle MAC_n, log_n \rangle$

Verifying a log later proceeds as follows:

Given

The initial secret s

The decryption function D corresponding to E

If encryption was used, a list of log entries such that $L_i = \langle MAC_n, ciphertext_n \rangle$

Otherwise, $L_i = \langle MAC_n, log_n \rangle$

Begin

Loop for each L_i:

Calculate the next link: $s = h(s)$

Derive the MAC key: $mkey = h(0|s)$

Derive the decryption key: $dkey = h(1|s)$

If encryption was used, set $log_i = D(dkey, ciphertext_i)$

Abort unless $MAC_i == MAC(mkey, log_i)$

(optionally output log_i)

Indicate success.

5 Security Proofs

Theorem 5.1 *Assume h is a random oracle, MAC is a secure message authentication code. Assume all secrets $s_i, mkey_i$ corresponding to log entries L_i have been securely deleted at time $t_j, j > i$, and that no adversary has information about s_0. Then with overwhelming probability, no adversary who gains access to the system after t_j can modify any entry L_i without detection.*

Proof: In the random oracle model [11], $h(x)$ provides no information about x. Then knowledge of $s_i, i \geq j$ provides no information about any $s_k, k < j$ since $\forall i > 0, s_i = h(s_{i-1})$. Assuming $MAC()$ is secure, then knowledge of $L_i = \langle MAC_i, log_i \rangle$ where $MAC_i = MAC(mkey_i, log_i)$ provides no information about $mkey_i$, and with overwhelming probability no valid MAC for $log'_i \neq log_i$ can be created without $mkey_i$ □

Theorem 5.2 *Given the assumptions in the previous theorem, and additionally assuming that E is a semantically secure encryption function, then no adversary who gains access to the system after t_j can distinguish $E(ekey_i, log_i)$ from $E(ekey_i, log'_i)$ where $|log_i| = |log'_i|$ in any log entry $L_i = \langle MAC_i, E(ekey_i, log_i) \rangle, i < j$.*

Proof: If E is semantically secure, then any adversary without knowledge of $ekey_i$ cannot distinguish encryptions of equal length plaintexts by definition, and cannot gain information about $ekey_i$ from $E(ekey_i, log_i)$. The previous theorem shows that the adversary cannot learn any $s_i, i < j$, so knowledge of all $s_i, i \geq j$ also provides no information about $ekey_i$ □

Note that these theorems do not address the situation in which adversaries delete or truncate a log, effectively making values of L_i unavailable to the verifier. We address this issue in section 10.

Proofs for the public key variants of Logcrypt are easy to construct for secure signature schemes with protection against existential forgery, and are sufficiently similar to the ones presented here that we omit them.

6 Public Key System

The primary disadvantage of the symmetric system just described is that verification of a MAC requires the same key that was used to create it. This means that anyone with the ability to verify a particular log entry could create arbitrary alternative entries which would also appear correct.

Public key cryptography provides the ability to separate signing from verification and encryption from decryption. This section describes how the signing/verification separation can be used to create logs which can be verified by anyone. We omit discussion of creative applications of the encryption/decryption separation, although several such applications are possible, particularly when using identity based encryption.

Bellare and Miner proposed a public key counterpart to hash chains in [3]. Here we propose a less elegant construction which can nevertheless be used with any signature scheme, and then in section 8 consider a scheme which more closely relates to Bellare and Miner's system.

Figure 6 shows the public key variant of Logcrypt. A signature replaces the MAC, and we add a special log meta-entry listing the next n public keys which will be used for signing.

More formally, the Logcrypt algorithm for signature-based integrity protection is as follows:

Given

A public-key signature function $Sign(private, message)$

A value n describing how many public/private keypairs will be stored in memory.

Begin

Generate an initial random public/private keypair, $(pub_0, private_0)$

Store pub_0 securely (generally, by sending it to a separate machine)

Loop

Create random keypairs $\langle (pub_1, private_1)..(pub_n, private_n)\rangle$

Create the meta-entry listing the public keys: $meta = \langle pub_1..pub_n\rangle$

Generate the signature on the meta-entry: $sig_0 = Sign(private_0, meta)$

Securely delete $private_0$. (pub_0 may also be removed).

Output $\langle meta, sig_0\rangle$

Set $i = 0$

Loop

Increment i

If $i == n$, exit the inner loop

Wait for the next log entry: log_i

Calculate $sig_i = Sign(private_i, log_i)$

Securely delete $private_i$. (pub_i may also be removed).

Output $\langle log_i, sig_i\rangle$

Set $pub_0 = pub_n$

Set $private_0 = private_n$

Recall that the second principle listed as foundational to Logcrypt's security in section

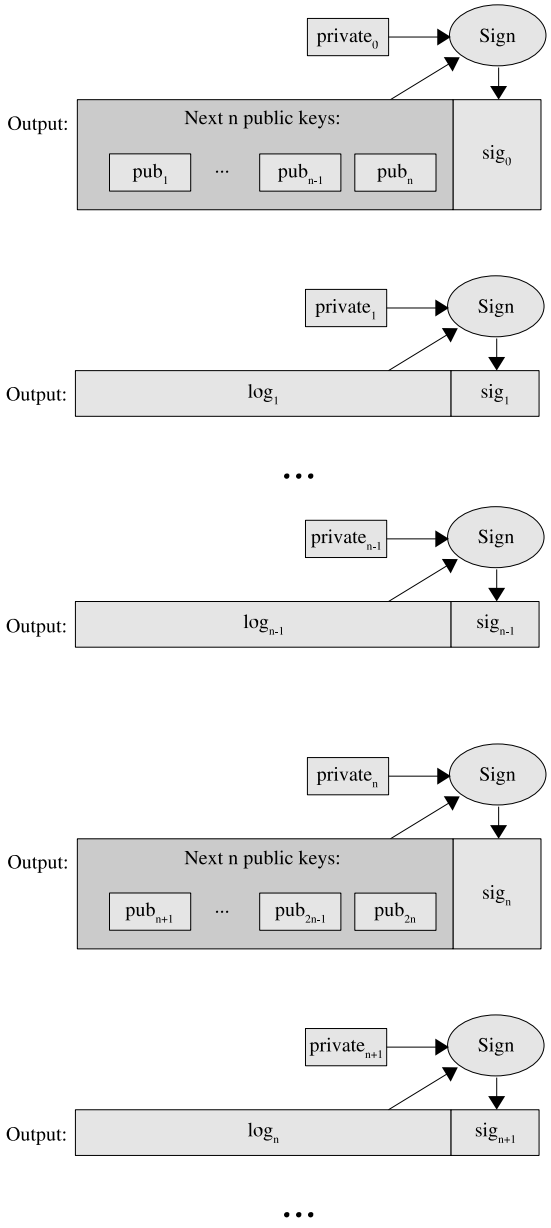

Figure 4: Forward security with public verifiability.

3 is the ability to validate a later log state using the state at an earlier entry. Hash chains actually *derive* later secrets from earlier secrets, even though all we need is validation. Consequently, it suffices to sign the public key that will be used at a later time using an earlier key, then throw away the signing key to fulfill the requirement that secrets be erased after they're used.

Verification proceeds as follows:

Given

The initial public key pub_0

A public-key signature verification function $Verify(pub, sig, message)$ which returns true iff sig is a signature for $message$ made with pub

A list of log entries such that $L_i = \langle log_i, sig_i \rangle$

Begin

Set $i = 0$

Loop

Set $meta = log_i$

Abort unless $Verify(pub_0, sig_0, meta)$ returns true

Extract public keys $pub_1..pub_n$ from $meta$

Set $j = 0$

Loop

Increment i and j

If $j == n$, exit the inner loop

Abort unless $Verify(pub_j, sig_i, log_i)$ returns true

Set $pub_0 = pub_n$

Indicate success.

7 Elliptic Curve Cryptography

Elliptic curve cryptography (ECC) is becoming increasingly popular as an alternative to cryptosystems like RSA because its structure allows shorter key lengths to provide an equivalent degree of security. For example, an RSA key of 1620 bits is estimated by [5] to have the same security as an ECC key with only 256 bits, while more recent estimates specify even longer RSA key lengths.

This property can be particularly useful for Logcrypt, since a new key must be generated and stored for each log entry. When log entries are short, this overhead can consume more than 50% of the total space required for the log. Since the specification in the last section makes no distinction between traditional and ECC cryptosystems, ECC-based signature algorithms can be used without modification to the algorithm. However, ECC is commonly used for identity-based encryption, which has further advantages in storage space which we consider in the next section.

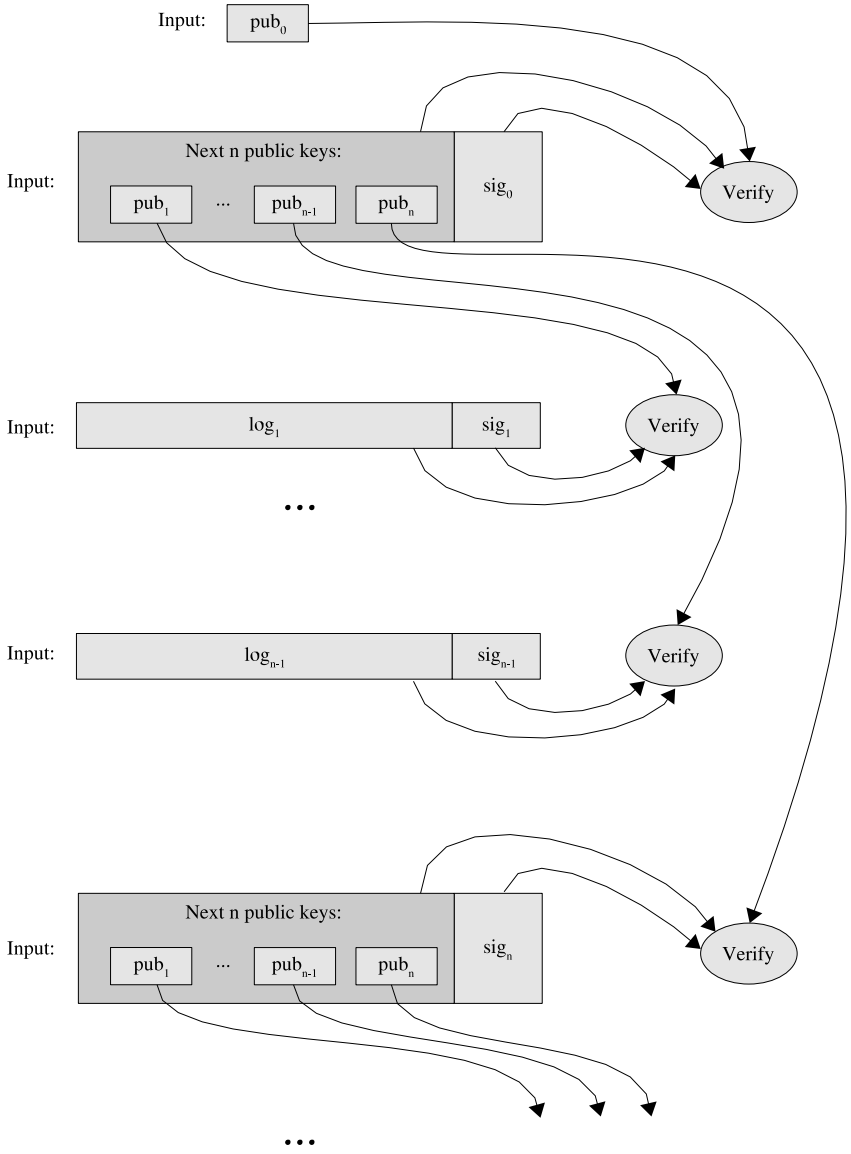

Figure 5: Verifying entries in the public key scheme.

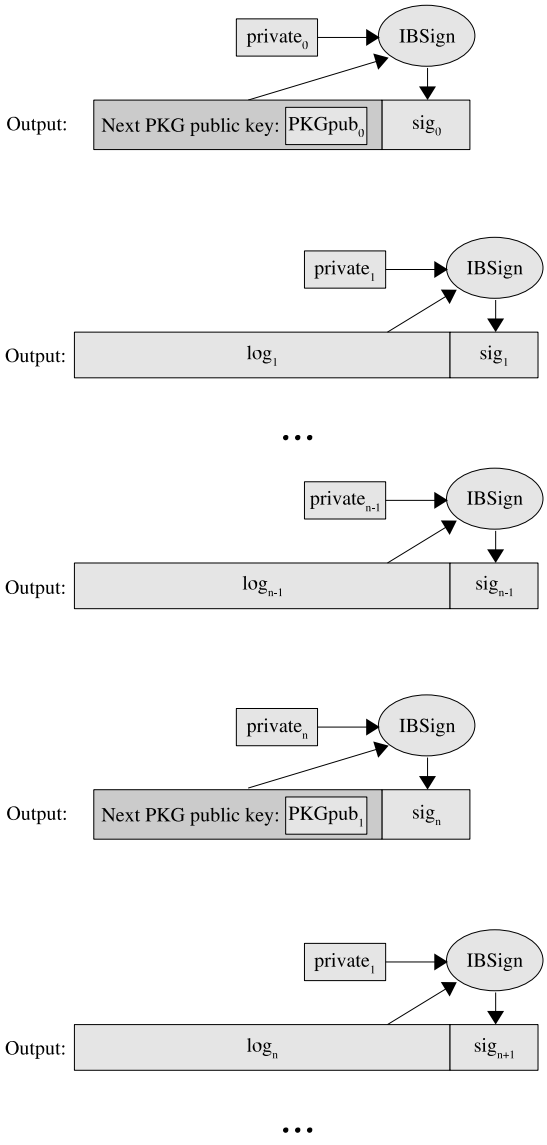

Figure 6: Forward security with public verifiability using Identity Based Signatures.

8 Identity Based Signatures

Identity-based Signatures (IBS) allow private keys to be derived from arbitary bit strings. The construction given by Cha and Cheon uses elliptic curves as the underlying mathematical construction, retaining their advantage of relatively short keys while eliminating the need to list the public keys to be used for upcoming log entries. That is, public keys 1 through n could simply be the strings "1", "2", etc., while the corresponding private keys are cached in memory and deleted after use. A new private key generator (PKG) key is generated for each key block, since it is used to generate all the private keys, and must therefore be erased as soon as all n private keys have been created. This scheme could also be adapted to work with Bellare and Miner's scheme [3], although it would require the longer key lengths used in their system.

Formally, here is the Logcrypt algorithm for IBS integrity protection:

Given

An IBS function $IBSign(private, message)$

Begin

Generate an initial random PKG keypair, $\langle PKGpub, PKGprivate \rangle$

Store $PKGpub$ securely (generally, by sending it to a separate machine)

Loop

Generate private keys for the strings 1..n using $PKGprivate$: $\langle private_1..private_n \rangle$

Securely delete $PKGprivate$.

Set $i = 0$

Loop

Increment i

If $i == n$, exit the inner loop

Wait for the next log entry: log_i

Calculate $sig_i = IBSign(private_i, log_i)$

Securely delete $private_i$. (pub_i may also be removed)

Output $\langle log_i, sig_i \rangle$

Generate a new PKG keypair, $\langle PKGpub, PKGprivate \rangle$

Generate the meta-entry listing the new PKG key: $meta = \langle PKGpub \rangle$

Generate the signature on the meta-entry: $sig_n = IBSign(private_n, meta)$

Securely delete $private_n$.

Output $\langle meta, sig_n \rangle$

Since the public keys are always simply the strings 1 through n, they don't need to be stored in the meta entry. Verification proceeds as follows:

Given

The initial PKG public key $PKGpub$

An IBS verification function $IBVerify(PKGpub, ID, sig, message)$ which returns true iff sig is a valid signature for $message$ using the private key derived from $PKGpub$ and the string ID

A list of log entries such that $L_i = \langle log_i, sig_i \rangle$

Begin

Set $i = 0$

Loop

Set $j = 0$

Loop

Increment i and j

If $j == n$, exit the inner loop

Abort unless $Verify(PKGpub, j, sig_i, log_i)$ returns true

Set $meta = log_i$

Abort unless $Verify(PKGpub, n, sig_i, meta)$ returns true

Extract the new $PKGpub$ from $meta$

Indicate success.

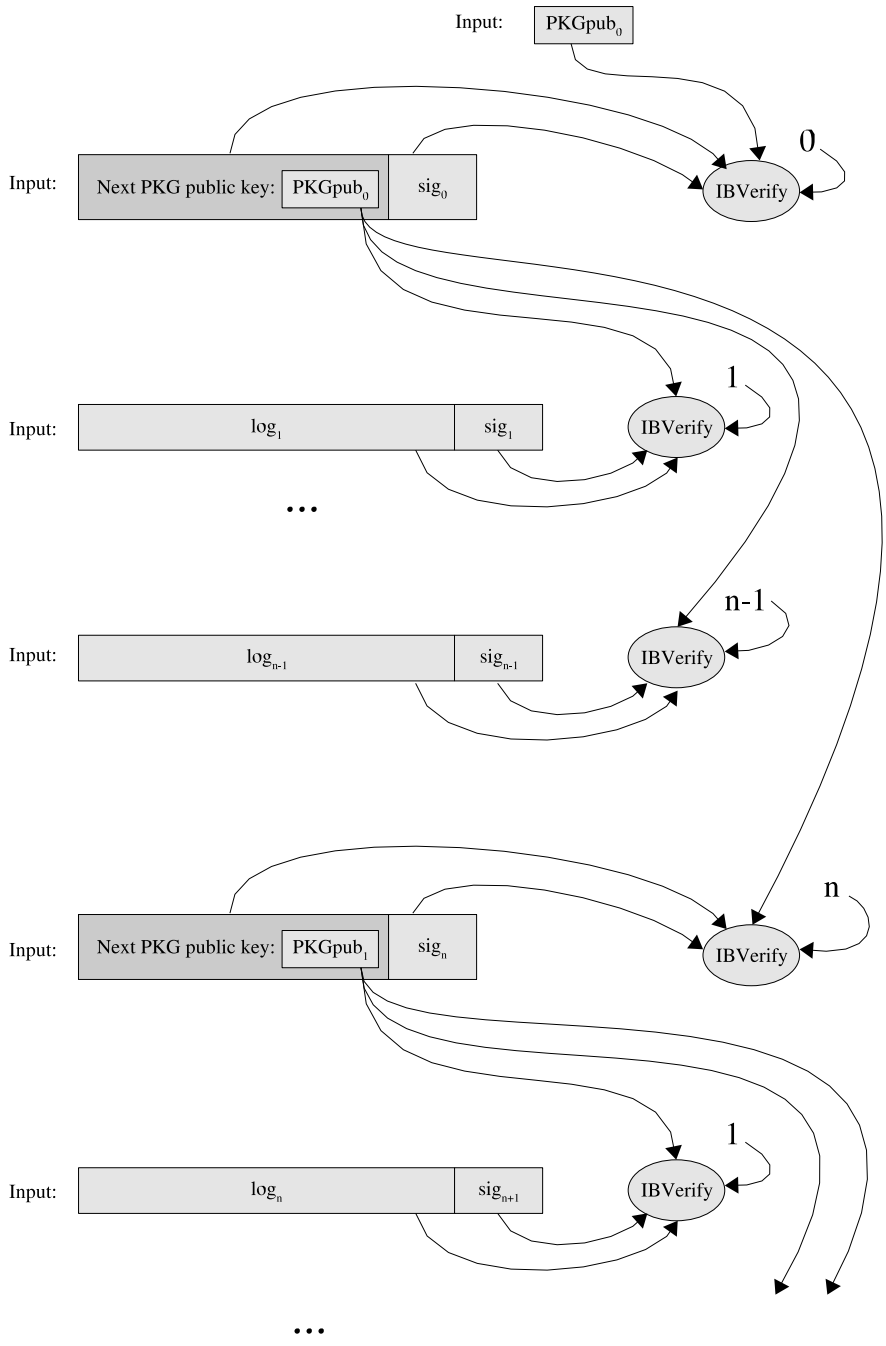

Figure 7: Verifying entries in the IBS scheme.

9 Cumulative Verification

In the construction given in [8], verification values for log entries validate the current log entry as well as the verification value for the previous entry. This is an advantage in that verifying an entry ensures that all prior entries were correct; the last value in such a log can be sent to an external auditor as a commitment to the entire log up until that point. In fact, the last value is the only value which needs to be stored – rather than storing each entry with its signature or MAC, log entries could be kept in one file and another could store only the most recent verification value.

This feature can trivially be added to the simple Logcrypt algorithm by adding the MAC of the previous entry as a third parameter to the $MAC()$ function, and to the other two algorithms by including the cumulative hash of all prior entries with the current entry for the signing process. However, we chose to omit this feature in our specification because it can hamper forensic analysis in some situations.

Consider an attacker who deletes some number of log entries (not including a public key meta-entry) from the middle of a Logcrypt log which uses public key or identity based signatures and stores the signature for each entry. The verification process will detect that the log has been modified in either case. However, if cumulative hashes are being maintained, intact entries after the deletion cannot be verified since the cumulative hash of prior entries cannot be reconstructed without knowledge of the missing entries. Without cumulative hashing, the later entries can still be checked for validity. Of course, if a meta-entry is lost, then there will be no public key available for verifying the later entries.

Logcrypt logs using MACs are not so heavily impacted by this drawback, and may find it worthwhile to use cumulative MACs by default. Entries after a deleted block can still be checked once the number of deleted entries are known as long as the MAC for each entry is kept (rather than storing only the last one), except for the entry immediately after the deleted block. That entry cannot be verified in a system using cumulative MACs since the previous MAC is unknown and thus prevents calculation of the current MAC.

Of course, an attacker aware of how Logcrypt works will tend to remove a Logcrypt log entirely, giving the analyst no information about the log, or leave it unchanged hoping that administrators won't notice any incriminating entries. But especially in situations where verification values are kept separately from a traditional-looking log, attackers may be unaware that Logcrypt is in use, and may remove only a few incriminating entries from the log. In such cases, forensic analysis can benefit from the finer granularity offered by not using cumulative verification.

10 Detecting Truncation

Thusfar, we have only described our system in terms of its abstract requirements, and have not considered other ways in which an attacker might attempt to subvert a system using Logcrypt.

In particular, we have not addressed what happens if an attacker chooses to simply delete or truncate a Logcrypt log rather than attempting to modify existing entries without detection. Of course, no new valid entries can be added once a log has been truncated, since intermediate secrets will have been lost, and this will be detected during verification. But in the case of a log which only

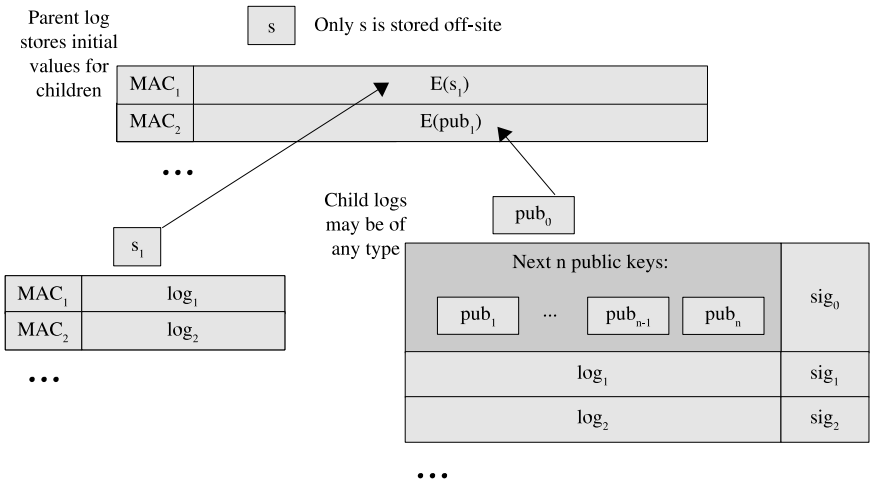

Figure 8: Maintaing concurrent logs with a single initial value.

records breakin attempts, for example, a lack of new entries suggests that the system is still secure.

Logcrypt cannot prevent an attacker in control of a system from deleting and truncating files. But it can be used to let an administrator know when the log is no longer functioning normally by using what we metronome entries. Metronome entries are simply special log entries which are made at regular intervals to indicate that the log is still accepting new data. If an attacker truncates the log just before an incriminating entry was made, he also truncates any metronome entries entered after that entry, and must prevent any future entries from being recorded, since they will fail verification. The verification process can be augmented to ensure that all metronome entries are present at the time of verification. If any are missing, then the last valid entry indicates the earliest time at which the log could have been truncated.

A related idea was described in [8]. They describe how a log can be securely closed by creating a special final entry and storing the final verification value off-site. Such a technique would be easy to use with Logcrypt in situations where this is needed.

11 Multiple Logs

Perhaps the most inconvenient part of Logcrypt is the requirement that initial secrets be securely stored at log creation time. But most systems maintain logs for multiple services concurrently, and regularly prune out older entries by rotating log files. Unless logs can be automatically, securely shipped to an external machine via a network, this quickly becomes an unwieldy task for system administrators.

To simplify this key distribution issue, we create a treelike structure of logs in which parent nodes store the initial secrets for their children. New children can be added at any time, and only the initial value created for the root node needs to be kept securely off the machine.

Figure 11 shows a simple case of a single encrypted master log which maintains the secrets for other system logs. Since the first child uses MACs instead of signatures and therefore has a secret initial value, the parent must encrypt all its entries. However, if all the children of a node

use public key or identity based signatures, all the initial values will be public keys and need not be encrypted. Such a subtree can then be verified by anyone who can verify the integrity of the initial value of the root node.

Storing multiple logs in a tree structure has other advantages as well. If all system logs were merged into a single log instead of being kept separately and maintained in a tree, then the entire log will need to be traversed in order to check the validity of the last entry. With a tree structure, however, any individual log can be verified by starting at the root node and iteratively verifying the entries corresponding to the nodes which lead to the log in question.

12 High Volume Logging

One way an attacker might try to compromise Logcrypt is to generate a large number of spurious events to be logged by the system or otherwise bog down the system's logging facility. If the attacker can generate the events more quickly than the system can process them, the system could end up with a backlog of entries all vulnerable to attack since they haven't yet been signed.

Recall that Logcrypt only offers strong protection for events which occur before time $t - l$, where t is the time of attack and l is the time required for Logcrypt to use and then destroy the secret corresponding to a particular entry. Increasing the demands on the system allows the attacker to increase the overhead l, which defines the window in which he can compromise entries which have already been received. To some extent, this cannot be avoided, particularly if we assume an attacker which can thoroughly overwhelm the system.

However, we can improve the system's resilience to such attacks as well as improving its overall capacity tremendously by taking advantage of the fact that some of the cryptographic operations which Logcrypt uses are much more efficient than others. In particular, public key and IBE signature operations have a relatively high overhead which varies very little with the size of the log entry being signed. This is due to the fact that signatures are performed on a constant-sized hash of the input, so that the only extra cost to lengthening the input comes from the hash function. For example, our 1.4Ghz Pentium workstation running OpenSSL requires 56ms to perform an RSA signature using a 2048-bit key, whereas it can process in excess of 100 bytes of input to SHA-1 per *microsecond*.

Consequently, if multiple new log entries arrive while the present entry is being processed, it makes sense to create a single signature for all of them combined rather than creating one signature for each entry, decreasing the average l across all the entries. Of course, this requires changes to the output format so that the entries can still be identified as distinct, and the verification process will have to consider the entries as a single unit.

This creates new possibilities for attackers which have the ability to overwhelm even the hash function, since they will be able to create increasing numbers of entries which the system will try to process before creating any signatures at all, whereas a system which processes entries one at a time would still create signatures on individual entries even as the queue filled up. Consequently, an upper limit could be set on the number of entries which may be aggregated into a single signature, providing an upper limit on latency for the head entry in the queue while still allowing much higher performance in high-load situations. However, in many systems the hash function will have higher

throughput than the network and disk subsystems, so that external limitations will be reached before hash function performance even becomes an issue.

13 Conclusion and Future Work

In this paper, we showed several innovative ways of achieving forward security for logs. We showed how to allow forward secrecy as well as tamper evidence in the simple system, and public verifiability without the ability to forge entries in the public key variants. We showed how multiple logs can be maintained concurrently and verified using a single initial value. We suggested optimizations which resist flooding attacks and dramatically improve performance under high load conditions. We described how logs can be made resistant to truncation, and closed to further additions. These features all address significant needs in systems administration as well as other disciplines such as finance and medicine which deal with tamper-sensitive data.

Future work may address further improvements to the public key variant of Logcrypt. Hierarchical IBS systems and meta-entries storing multiple PKG public keys can be used to further improve performance while keeping overhead low. Bellare and Miner's contribution [3] also provides an obvious avenue for space-efficient public verification.

An initial implementation of several of the features described here is available under the GPL, and future work will include performance refinements and increased convenience features in the form of both library functions and sample applications.

References

[1] R. Anderson, B. Crispo, J.H. Lee, C. Manifavas, and R. Needham. A New Family of Authentication Protocols. Operating Systems Review, 32(4):9-20, October 1998.

[2] M. Bellare and B. Yee, "Forward Integrity for Secure Audit Logs," Technical Report, Computer Science and Engineering Department, University of California at San Diego, November 1997.

[3] M. Bellare, S. Miner. A Forward-Secure Digital Signature Scheme. In Proc. of Crypto, pp. 431–448, 1999. http://citeseer.ist.psu.edu/bellare99forwardsecure.html

[4] J. Cha, J. Cheon, "An ID-based signature from Gap-Diffie-Hellman Groups," Proc. of PKC 2003, Lecture Notes in Computer Science, Vol. 2567, pp. 18-30 (2003). http://www.math.snu.ac.kr/~jhcheon/Published/2003_PKC/PKC03_CC.pdf

[5] A Cost-Based Security Analysis of Symmetric and Asymmetric Key Lengths, RSA Laboratories Bulletin #13, April 2000. http://www.rsasecurity.com/rsalabs/node.asp?id=2088

[6] B. Schneier, J. Kelsey. Cryptographic Support for Secure Logs on Untrusted Machines. In Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, Jan. 1998.

[7] B. Schneier, J. Kelsey. Secure Audit Logs to Support Computer Forensics. ACM Transactions on Information and System Security 2(2): 159-176, 1999.

[8] J. Kelsey, B. Schneier. Minimizing Bandwidth for Remote Access to Cryptographically Protected Audit Logs. Recent Advances in Intrusion Detection, 1999. http://citeseer.nj.nec.com/kelsey99minimizing.html

[9] C. N. Chong, Z. Peng, and P. H. Hartel. Secure audit logging with tamper resistant hardware. Technical report TR-CTIT-02-29, Centre for Telematics and Information Technology, Univ. of Twente, The Netherlands, Aug 2002. http://citeseer.nj.nec.com/chong02secure.html

[10] B. R. Waters, D. Balfanz, G. Durfee, D. K. Smetters. Building an Encrypted and Searchable Audit Log. ACM Annual Symposium on Network and Distributed System Security, 2004.

[11] Mihir Bellare and Phillip Rogaway, Random Oracles are Practical: A Paradigm for Designing Efficient Protocols, ACM Conference on Computer and Communications Security 1993, pp62-73.