Forward Integrity For Secure Audit Logs

Mihir Bellare* Bennet S. Yee†

November 23, 1997

Abstract

In this paper, we define the *forward integrity* security property, motivate its appropriateness as a systems security requirement, and demonstrate designs that achieve this property. Applications include secure audit logs (e.g., `syslogd` data) for intrusion detection or accountability, communications security, and authenticating partial results of computation for mobile agents. We prove security theorems on our forward integrity message authentication scheme, and discuss the secure audit log application in detail.

Keywords: Audit logs, MACs, forward integrity.

*Dept. of Computer Science & Engineering, Mail Code 0114, University of California at San Diego, 9500 Gilman Drive, La Jolla, CA 92093, USA. E-mail: `mihir@cs.ucsd.edu`. Web page: `http://www-cse.ucsd.edu/users/mihir`. Supported in part by NSF CAREER Award CCR-9624439 and a 1996 Packard Foundation Fellowship in Science and Engineering.

†Dept. of Computer Science & Engineering, Mail Code 0114, University of California at San Diego, 9500 Gilman Drive, La Jolla, CA 92093, USA. E-mail: `bsy@cs.ucsd.edu`. Web page: `http://www-cse.ucsd.edu/users/bsy`. Supported in part by a 1996 National Semiconductor Faculty Development Award and the U. S. Postal Service.

Contents

1 Introduction and Motivation

We introduce a novel security property that we call *forward integrity* (FI). What is it? View it as an integrity analog of "forward privacy" (FP, also known in the literature as "perfect forward secrecy" or PFS[1]) [13, Chapter 12]. Informally, we would like to generate message authentication codes (MACs) (e.g., for audit logs) in such a way that even if the MAC generating machine is later *completely* compromised — meaning the key under which MACs are generated becomes known to the attacker — it is not possible to forge data pertaining to the past.[2] For audit logs, this means that the attacker can erase log entries, but cannot otherwise modify an existing entry or create new bogus entries with logging time preceding his break-in time.

1.1 Mail Sorter Fraud

The motivation for this work was the "mail sorter fraud problem." The U. S. Postal Service (USPS) pays third-party companies called "service bureaus" to collect mail from other organizations (e.g., hospitals, universities, local governments, etc), apply bar-codes to their outgoing mail, and pre-sort it according to the bar-coded ZIP+4 information using "Multi-Line Optical Character Recognition" machines (MLOCRs). The amount of payment to a service bureau depends on the granularity of the sort, with limits on various parameters such as the number of letters per bin. Such outsourcing, while improving the USPS's efficiency, entails some risks: some service bureaus have falsified the documentation generated by the MLOCRs to defraud the USPS.

The MLOCRs are usually leased from their manufactuers, though they are sometimes purchased; they are certified by the USPS to operate properly, that they can correctly recognize addresses and apply bar-codes. Thus, one avenue for improving security is to upgrade the MLCOR software to include fraud prevention / detection techniques.

The fraud threat is real. Litigation, past and pending, involve multi-million dollar losses. The total amount of fraud is unknown, but is estimated by the USPS to be anywhere from \$20–\$60 million per year.

Existing techniques used to deter fraud include paper-based audit trails and sampling of the sorted mail. The paper-based audit information is simply printed output from the MLCORs, and has no authenticity guarantees: the formats are easily duplicated by users familiar with easy-to-use commercial software such as Microsoft Excel. One might imagine that statistical sampling techniques would work quite well in detecting this kind of fraud. Unfortunately, due to processing throughput requirements, in practice the USPS must inform the service bureaus of the destination zip code(s) to be sampled at the beginning of the business day — because the identity of the "random" sample is known in advance by the service bureaus, it can only serve to detect accidental rather than intentional problems with the mail sort.

1.2 Threat Model

In the mail sorter fraud problem, the day-to-day operators of the MLOCRs are not necessarily trustworthy. This means that the best that can be hoped for is detection during periodic security inspections, which can be expensive. Unlike normal computer systems, MLOCRs software are not programmed by their users (except for installing field upgrades from manufacturers); their operators use a specialized interface to enter data and to optimize the mail sort for revenue generation (assigning zip code ranges to output bins), but do not need the ability to program the machines: a

[1] "Perfect Forward Secrecy" is a misnomer: most FP schemes are not "perfect" in any sense.

[2] One might ask why this is called "forward" integrity, since security pertains to the past. We are simply following the terminology used in the case of privacy. One possible explanation for the terminology is that security holds from the time of the event forward, because you can't reach back and compromise it.

MLOCR is more like an embedded processor than a workstation or PC, even though workstations or PCs may serve as components.

Compared to the threat model for normal computer systems, the threat model here in some ways places greater limits on the attackers' and in other ways gives them greater freedoms. The adversaries are not well funded compared to the USPS and are unlikely to be able to mount research projects to determine undetectable methods of attack — the motivation for fraud is economic in nature, and must minimally provide a reasonable return on investment: the cost of mounting the attack cannot exceed the (expected) returns. Neither can defrauding service bureaus operate in complete secrecy: the USPS knows where the MLOCRs are located due to various certification requirements, special laws protect the USPS, the USPS's Inspection Service (a specialized police force) specializes in mail fraud of various kinds, and disgruntled employees of service bureaus are in the position to observe many kinds of overt fraud. On the other hand, the MLOCRs operate in a hostile environment: they reside on service bureau premises and are subject to limited physical tampering by service bureau personnel. Using removable media, such as WORM drives, for logging is not a viable alternative, since the MLOCR operators can easily swap the logging media.

In this work, we do not address the problem of secure boot [2]. This is an orthogonal problem. We merely assume that the MLOCR controlling software is initially installed properly, and until the attackers succeed in penetrating the system, the audit log software (and the relevant parts of the underlying operating system) runs correctly. Note that FI alone cannot — theoretically — prevent fraud: a well funded adversary can completely replicate the state to another machine (e.g., temporarily remove and copy the hard disks) and search for any stored keys at leisure. Using some physically protection component [16] would be required to protect against that sort of attack. In this paper, we assume that state duplication is too expensive for the attacker, and that the attacker will attempt to discover some other "short cut" method of penetrating the system — short cuts which are (initially) detectable by basic intrusion detection / auditing techniques. It is in this operating environment that we strive to protect the security audit data against tampering.

In somewhat more detail, our forward integrity scheme works like this. We divide time into "epochs". A FI audit log system ensures that if a machine is first compromised at time T_c during epoch $\mathcal{E}_j$, i.e., $T_c \in \mathcal{E}_j = \{t : T_j \leq t < T_{j+1}\}$, then the attacker cannot forge log entries to appear to be generated at any times $t < T_j$. No guarantees are provided for any log entries produced after T_j.

MACs that enjoy the FI property may be used in communications context as well as in the logging context. In the rest of the paper, we will use secure audit logs as the motivating application and in the description of the notation; the notation does not change for other applications, of course, so the same analysis applies.

The technical challenge will be to design MAC schemes with the FI property. We propose the *prf*-chain FI MAC. It uses block ciphers (modeled as pseudorandom function families) and standard MACs (e.g. HMAC [3]) to achieve FI via a chaining process. The cryptographic security of this scheme is modeled and analyzed in the style of works like [4], meaning a concrete security analysis in the "provable security" paradigm reduces the security of our FI scheme to that of the underlying primitives.

Now, let us look at all this more closely. We discuss secure audit logs and the notion of FI in this context, in Section 2 below. Then in Section 3 we provide our FI schemes and their analysis. In Section 4 we discuss stronger schemes which prevent deletion of log entries as well. The cryptographic routines have been implemented, and in Sections 5 and 6 we describe implementation concerns and performance results. In Section 7 we discuss other applications of FI MACs, and conclude in Section 8 with some directions for future work.

2 Audit Log Systems

Computer audit logs contain descriptions of noteworthy events — crashes of system programs, system resource exhaustion, failed login attempts, etc. Many of these events are critical for post-mortem analysis after a break-in. The first target of an experienced attacker will be the audit log system: the attacker wishes to erase traces of the compromise, to elude detection as well as to keep the method of attack secret so that the security holes exploited will not be detected and fixed by the system administrator(s). To make the audit log secure, we must prevent the attacker from modifying the audit log data.

To crystallize what we mean by audit log security, let us first define more precisely what we mean by an audit log system and the sort of attacks to which an audit log system might be subject. First, we review the standard techniques used to protect audit logs. Next, we define the notation that we will use in the rest of this paper to talk about audit log security. We also discuss some requirements and non-requirements for secure audit logs.

2.1 Existing Audit Log Techniques

In the Orange book [14], audit log security is defined informally in Requirement 4:

> Audit data must be protected from modification and unauthorized destruction to permit detection and after-the-fact investigations of security violations.

As long as the Trusted Computing Base (TCB) — this is the system component responsible for logging — retains its integrity, the audit log should, of course, be secure. Unfortunately, TCBs are not always bug free, and intruders can often gain the necessary privileges to modify audit log data. This is especially important with so-called Common/Commercial Off The Shelf systems (COTS), where functionality and compatibility with previous versions are often more important than operating system security in the design.

To further enhance the integrity of audit log data, some systems use *remote logging*, which sends the audit data to remote networked hosts, typically configured similarly to the logging host. The additional security hinges upon the hopes that an attacker will not know to, or be able to, compromise the remote host as well. Additionally, *log replication* may be used [10], where the audit data is copied to multiple remote logging hosts in order to to force the attacker to break into all or most of the remote logging hosts in order to erase evidence of the original intrusion. Often, “outmoded” old machines may be reconfigured to be the physically secure logging host; typically the machine will not provide any other network services, and thus should be immune to network-based attacks.

The classic method for protecting audit log integrity is writing the audit log data to a continuous-feed printer. Unlike remote logging and log replication, logging to such a (electronically) write-only media is only appropriate for low-volume logging — the audit log system must *a priori* filter the log entries so that the audit log printer would not be overwhelmed. Additionally, print logging makes it impossible to electronically process the audit log to scan for suspicious activity, and physical access to the printers by potentially untrustworthy operators diminishes its usefulness. All of this limits the effectiveness of post-mortem analysis.

Another similar write-once logging employs Write Once Read Multiple (WORM) drives. These are optical disks with a write bandwidth that, while lower than magnetic disks, is usually more than sufficient for logging. WORM drives write to removable media; in the case where attackers have limited physical access to a machine, (optical) disk substitution can completely compromise the integrity of the audit log data.

2.2 Using Message Authentication Codes

It might seem natural to use message authentication codes (MACs) to protect the integrity of audit log data. Normally, MACs are used in a communications context, where the sender and the recipient share a secret MAC key. The sender uses this key to generate a message's MAC, and attaches it to the message; the receiver, who knows the secret MAC key and thus can regenerate the MAC, would accept as genuine only those messages for which the regenerated MAC matches the transmitted MAC. The security of MACs rely on the fact that it is computationally infeasible for a network-based adversary who does not know the secret MAC key to modify the messages and the MACs so that the receiver would accept them as genuine with non-negligible probability. Since audit logs are simply messages that are read and verified by a recipient later in time rather than (necessarily) across a network, perhaps we can simply attach MACs to the the audit log entries to protect them.

Unfortunately, this scheme fails in the absence of immediate, continuous remote logging. When the audit log is stored locally or when the audit data must be buffered (over long time periods) before being sent to the remote log repository, the audit log data is at risk: an attacker who penetrates the system will gain knowledge of the secret MAC key, allowing the attacker to change the log data at will. Furthermore, note that immediate continuous remote logging is no panacea: the security of the remote logging host becomes critical, and security concerns about the log-generator to logging host communications remain.

2.3 FI for Audit Logs

We introduce a new way to maintain the security of audit logs. This avoids remote logging, continuous remote logging or log replication. We do use MACs, but in a new way.

A log entry consists of a date (time) and event description. As indicated above, an experienced attacker will, upon break-in, try to compromise log data pertaining to the past: he wishes to alter, or erase, any entries pertaining to his current or past login attempts. Our goal will be prevent this, even when the past log entries are available to the attacker and the latter has control of the entire system.

Our system will be MACing log entries as it makes them. The danger, as indicated above, is that upon break-in an attacker gets the MAC key and can then forge all log entries. In our system, however, possession of the key at a certain point in time does not enable the attacker to forge any log entries with a date (time) that is in the past. Thus the attacker cannot alter the log contents. (He can still erase entries, but gaps will be visible in the log, and also, occasional transmission of the log to a remote source mitigates the effect of erasure.)

How is this forward integrity possible? Our system does not always MAC under the same key. Rather, the key evolves over time periods. The key K_i in epoch i is obtained as a non-reversible function of the key K_{i-1} of the previous epoch, and upon start of epoch i, the key K_{i-1} is erased. Thus if the attacker breaks in during epoch i, he gets K_i, but cannot know K_j for $j < i$. Thus, he cannot forge log entries pertaining to previous epochs. (Meanwhile, the base key K_0 serves to verify the MAC of all log entries regardless of epoch. This may be held by a secure, remote verifier.)

The technical challenge will be to design an efficient MAC scheme with this FI property. We will first pin down the requirements more precisely and then present the construction.

We introduce the following notation to capture the above. An audit log system LOG accepts a time-ordered sequence of log messages $\{m_1, m_2, \ldots, m_l\}$ from log producers. For notational convenience, we also define t_i (lower case) to be the time at which message m_i is given to the audit log system ("logged"). Time is divided into epochs $\mathcal{E}_j$, for $j = 1, \ldots, n$; the start of an epoch $\mathcal{E}_j$ is denoted T_j, and thus $\mathcal{E}_j = \{t : T_j \leq t < T_{j+1}\}$. Epoch boundary times (T_i) and the time of compromise (T_c) (upper case) do not have associated log producer messages. Epochs are nonoverlapping and sequentially ordered in time; they partition the message stream: for each epoch $\mathcal{E}_i$, there is a first message $m_{first(i)}$ and a last message $m_{last(i)}$ logged within the time period defined by the epoch, and no message is ever logged on an epoch boundary. Given a time t, let $epoch(t)$ denote the epoch containing t. Each log message is contained in exactly one epoch. We will sometimes abuse our notation to denote by $epoch(m_i)$ or $epoch(i)$ the epoch during which message m_i is given to LOG.

The log producers generate the messages m_i; they may or may not have any control over when the logger decides to change epochs. For each message m_i received in epoch $\mathcal{E}_j$, LOG generates a log entry $log\_fn_j(m_i)$ and stores it in the audit log (some form of non-volatile memory — a local disk, or remote dedicated log storage accessed through the network), which can be later verified. The verifier is assumed to be uncompromised.

In this paper, we suggest audit log systems where $log\_fn_j(m_i) = (m_i, FIMAC_j(m_i))$, where $FIMAC_j(m_i)$ is an authentication code appended to the messages — no transformation is done on the messages themselves. A verifier *verif*, knowing some secrets, can efficiently compute the predicate

$$valid(m, j, x) = \begin{cases} \text{true} & \text{if } x = FIMAC_j(m) \\ \text{false} & \text{otherwise} \end{cases}$$

The logger may also independently generate additional data to be written to the audit log; the amount of such data should be small, linear on the total message, or epoch count. In one of the schemes discussed below, an additional audit log entry is made at the end of every epoch.

The MAC $FIMAC_j(\cdot)$ is generated under a key K_j that depends on the epoch j. As indicated above, in our solution, we will obtain this key via $K_j = f(K_{j-1})$ for some one-way function f, and erase K_{j-1} from the system at this point.

In building the MACs, we consider the most powerful attacks against them, namely chosen message attacks. Furthermore the attacker can control the epoch change decision. At some time T_c, the system is compromised, and any secrets in the system at that time become known to the attacker. We want that past log entries can nonetheless not be compromised.

Definition 1 *A log system LOG is said to be* (q, t, L, ϵ) FI secure *if there is no algorithm $\mathcal{A}$ which after running in at most time t, generating at most q log messages (MAC queries) of length at most L and obtaining LOG's output, then makes an "open" request at time T_c to obtain all secrets in the log system and successfully outputs (m, j, σ), a false log entry (the pair (m, σ)) for some earlier epoch $(\mathcal{E}_j)$ with probability at least ϵ, i.e., $j < epoch(T_c)$, $\forall i : epoch(m_i) = j \implies m_i \neq m$, and $Pr[valid(m, j, \sigma) = true] \geq \epsilon$.*

As is customary in many exact security analyses, the time t parameter also includes the size of the program $\mathcal{A}$ to take table-lookup algorithms into account (think of this as the time needed to write down the algorithm).

In this definition, we are concerned only with the (in)ability of the attacker to create forged log messages — we do not yet deal with the deletion of log entries, which is a form of denial-of-service attack that can only be detected, not prevented.

For audit logs, being able to detect the deletion of log messages made in an earlier epoch is just as important as being able to detect alterations: the attacker may selectively erase tell-tale audit log entries and pretend that no such entries were made. Thus, we need slightly stronger security properties; informally:

Definition 2 *A log system is said to be* (q, t, L, ϵ) deletion detecting-FI secure *(DD-FI) if it is $(q, t, L, \epsilon) - FI$ secure and the log verifier can determine, given the output of the log system and* *the epoch in which the log was compromised, whether any log entries has been deleted from earlier epochs.*

We will later see how adding sequence numbers and epoch change markers can protect the log from having entries deleted in an undetectable fashion. Note the reliance on being able to know the epoch in which the compromise occurred: if this is not the case, then the attacker can still erase a suffix of the log entries and pretend that an earlier epoch is the current one.

2.4 Efficiency Requirements

Efficiency is an important concern for practical secure audit log systems. The first target of an experienced attacker will be the audit log system, and the speed at which the attacker can penetrate a system and attempt to modify the audit log data depends on various factors such as network latencies (if it is a networked-based attack), etc, the most important of which is the level of automation employed in the attack — well funded adversaries will have duplicate hardware/software against which attack scenarios have been practiced and/or automated.

In order to protect against well funded, expert attackers, a FI audit log system need to be able to quickly change epochs. The design options include changing epochs frequently, say every 100 mS in a deterministic fashion; changing epochs after every N log entries (an important special case being $N = 1$); or categorizing the audit log entries by severity and changing epochs immediately after logging entries of a certain level or higher.

2.5 Relationship with Standard Techniques and other work

Our FI audit log system model does not address remote logging nor log replication. Not all applications are networked, and of those that are networked, for some the networking/replication overhead is too expensive.

For the mail sorter fraud scenario, some MLOCRs are not networked or are only networked to other MLOCRs, all of which are physically located in the service bureau premises: no physically secure host is available, and thus remote logging is not a viable technique.

While replication and remote logging may be effective in many cases, sometimes the underlying assumptions are falsified; for example, there may be common-mode failures that can allow all of the logging hosts to be compromised (e.g., operating system level bugs). Furthermore, when remote logging is employed, authenticating the identity of the remote log generators is needed, since otherwise bogus log generators can mount a denial of service attack by flooding the log with bogus entries. This is a design issue that is often ignored. Such log-generator to logger authentication must be inexpensive, since denial of service can still be achieved if the logger can be overloaded with authenticity verification of bogus log submissions. For efficiency, a MAC-based scheme is usually advisable, and employing a DD-FI MAC scheme compares well with a plain MAC in authenticating log generators at the same time as providing FI security for log entries. Similarly, for efficiency audit log entries are often buffered and only periodically flushed to disk (perhaps an append-only file, e.g., 4.4 BSD's `chflags(2)` [11, 12]) or to a remote logging host, perhaps on a daily or even weekly basis: a DD-FI log system can prevent an attacker from undetectably modifying these in-memory or in-transit copies of the log.

Ross Anderson has suggested the idea of forward integrity for signature schemes [1].

2.6 Non-requirements

Log entries, whether still in memory buffers or sent over the network, may have associated privacy requirements. For example, often `syslogd` entries contain information about failed log-in attempts, which, when associated with honest typing errors, leak the identity of valid accounts on a system.

Ignoring covert channel and traffic analysis style privacy leaks, the privacy of audit log entries may be provided by encrypting the log entries prior to submitting them to the logger. Additionally, forward privacy for log entries may be provided by the same key generation scheme outlined below for integrity.

3 FI Log Schemes

This section describes audit log systems that possess the FI property. The key observation is that the audit log system has the ability to forget.

3.1 Random-key FI-MAC

Let the audit log system LOG_0 be initialized for n epochs, $\mathcal{E}_1, \ldots, \mathcal{E}_n$. For each epoch $\mathcal{E}_j$, LOG_0 has a randomly chosen MAC key k_j. For each message m_i set $FIMAC_j(m_i) = MAC_{k_j}(m_i)$. At the end of the epoch (time T_{j+1}), we erase key k_j from memory.

The verifier *verif* is assumed to be separated from the system in which the audit log is computed, so it retains all of the keys ($k_j, j = 1, \ldots, n$) and thus the ability to verify the MACs; we also assume that *verif* maintains its privacy and integrity: its keys are never compromised.

Clearly, this scheme has the FI property. If the audit log is compromised during epoch $\mathcal{E}_c$, MAC keys $k_1, \ldots, k_{c-1}$ are not available; to forge audit log entries that should have appeared at an earlier epoch, the attacker must compromise the MAC scheme.

However, this requires the system, and the verifier, to store a number of keys equal to the number of epochs, which is not desirable. We want instead that a single key is stored, and evolves with time in pre-determined way.

3.2 *prf*-chain FI-MAC

Instead of a statically determined number of epochs where preselected keys occupy storage, the following scheme is cheaper and less awkward to use.

3.2.1 Description of *prf*-chain FI-MAC

Let *prf* denote a family of pseudo-random functions. (The general notion is due to [6]. In our context, a block cipher can be used in this role, as per [4].) Within any epoch $\mathcal{E}_j$, the logger LOG_1 possesses a secret values s_j. All log entries m_i received within the epoch receives the log authentication code $FIMAC_j(m_i) = MAC_{k_j}(m_i)$, where the MAC key $k_j = prf_{s_{j-1}}(0)$. To change to the next epoch $\mathcal{E}_{j+1}$, the secret value is transformed: $s_{j+1} = prf_{s_j}(1)$, a new MAC key $k_{j+1} = prf_{s_j}(0)$ is computed, and the secret and MAC key from the previous epoch is then erased from memory.

The actual MAC under some key k is computed via a fixed MAC scheme whose tagging algorithm is denoted $Tag_k(\cdot)$. (For example, one could use HMAC [3] or the CBC MAC.) This MAC function is assumed secure against chosen message attack in the sense of [4], meaning it is computationally infeasible for an adversary to forge a correct tag for a new message even after a chosen-message attack.

3.2.2 Security Analysis of *prf*-chain FI-MAC

First, some notation specific to this scheme. We define our epoch key transformation scheme:

$$\begin{tikzcd}
s_0 \arrow[r, "0"] \arrow[dr, "1"'] & s_1 \arrow[r, "0"] \arrow[dr, "1"'] & s_2 & \cdots & s_{n-1} \arrow[r, "0"] \arrow[dr, "1"'] & s_n \\
& k_1 & k_2 & \ddots & k_{n-1} & k_n
\end{tikzcd}$$

where $x \overset{0}{\rightarrow} y$ denotes $y = prf_x(0)$, and $x \overset{1}{\rightarrow} y$ denotes $y = prf_x(1)$. Within each epoch $\mathcal{E}_i$, k_i is used to MAC log messages.

We define experiments $Expr_i$, for $i \in \{0, 1, \ldots, n\}$, as follows:

$$\begin{tikzcd}
s_0 & s_1 & \cdots & s_i \arrow[r, "0"] \arrow[dr, "1"'] & s_{i+1} & \cdots & s_{n-1} \arrow[r, "0"] \arrow[dr, "1"'] & s_n \\
& k_1 & \cdots & k_i & k_{i+1} & \ddots & k_{n-1} & k_n
\end{tikzcd}$$

Here, $s_0, \ldots, s_i$ and $k_0, \ldots, k_i$ are randomly chosen, and $s_{i+1}, \ldots, s_n$ and $k_{i+1}, \ldots, k_n$ are determined by the $\overset{0}{\rightarrow}$ and $\overset{1}{\rightarrow}$ mappings. Note that $Expr_0$ is precisely our LOG_1 *prf*-chain FI-MAC scheme, and $Expr_n$ is precisely our LOG_0 random-key FI-MAC scheme with n independently chosen keys ($s_0, \ldots s_n$ are unused). We define an algorithm "breaking $Expr_i$" in the natural way — generating a forgery within the appropriate resource bounds.

We first analyze the security of $Expr_n$. Suppose an adversary's algorithm $\mathcal{A}$ can break the LOG_0 FI-MAC scheme in time t with probability at least ϵ using at most q queries per epoch, each MAC query of at most length L. Let the particular MAC function that we wish to break be denoted $Tag_k(\cdot)$. It is a black box — we do not know the hidden key k, but are able to compute MACs with it using the $Tag_k(\cdot)$ oracle.

Claim 1 *Given $\mathcal{A}$ we can break the MAC function $\mathrm{Tag}_k(\cdot)$ in time $t' = t + c_1 n + c_0$ with probability $\epsilon' > \frac{\epsilon}{n}$ making at most $q' = q$ queries, where c_1 and c_0 are (small) constant.*

Proof: We construct our MAC breaking algorithm $F^{\mathrm{Tag}_k(\cdot)}$ as follows:

- Pick $i \in_R \{1, \ldots, n\}$.
- Pick $k_1, \ldots, k_{i-1}, k_{i+1}, \ldots, k_n$ MAC keys at random.
- Run our adversary's algorithm $\mathcal{A}$:
  - In epochs $j \neq i$, use k_j to MAC messages.
  - In epoch i, use $Tag_k(\cdot)$ to MAC. If $\mathcal{A}$ asks to open the secret at epoch i, we abort ($F^{\mathrm{Tag}_k(\cdot)}$ fails).
- Eventually $\mathcal{A}$ outputs a forgery (m, j, σ) for epoch $\mathcal{E}_j$.
- If $j \neq i$, we abort.
- Output (m, σ).

Since the probability that $\mathcal{A}$ will happen to generate its forgery for epoch i is $\frac{1}{n}$, the probability of breaking $Tag_k(\cdot)$ is $\epsilon' > \frac{\epsilon}{n}$. The runtime of this MAC-breaking algorithm is $t' = t + c_1 n + c_0$, where $c_1 n + c_0$ is the overhead setting up to run $\mathcal{A}$. ▌

From this we can conclude that the probability that an adversary can break $Expr_n$ is less than $\delta \triangleq n\epsilon$, since otherwise we'd be able to break the MAC function with probability at least ϵ, violating the security assumption for the MAC function.

Theorem 1 *If the underlying MAC function is $(q_m, t_m, L_m, \epsilon_m)$-secure, then LOG_0 is (q, t, L, δ) FI secure, where $q = q_m$ and $t = t_m - c_1 n - c_0$ and $L = L_m$ and $\delta = n\epsilon_m$.*

Next, we examine *Expr*$_0$. Assume that the probability that an algorithm $\mathcal{A}$ can break *Expr*$_0$ is $\epsilon = \delta + \gamma$.

Claim 2 *The pseudo-random function family can be broken with probability $\frac{\delta}{n}$ using 2 queries.*

Proof: To see this, we look at the hybrid experiments.

We denote by D^f to be *prf*-distinguisher algorithm as follows. f is either a pseudo-random function *prf*$_s$ for some random s or a random function. D^f uses $\mathcal{A}$, an algorithm that breaks the *prf*-chain FI-MAC, as a subroutine. D^f is given f as an oracle, and it must output 1 or 0 ("*prf*" or "random") to determine whether f is *prf*$_s$ or a random function:

- Pick $i \in_R \{1, \ldots, n\}$
- Pick $\begin{array}{c} s_0, s_1, \ldots, s_{i-1} \\ k_1, \ldots, k_{i-1} \end{array}$ at random.
- Set $s_i = f(0)$, $k_i = f(1)$ (the two queries to f).
- Set $s_j = \mathit{prf}_{s_{j-1}}(0)$ and $k_j = \mathit{prf}_{s_{j-1}}(1)$ for $j = i+1, \ldots, n$.
- Run $\mathcal{A}$ for epochs $j = 1, \ldots, n$. In $\mathcal{E}_j$, reply to MAC queries using k_j. If an "open" request is generated, reply with (s_j, k_j).
- Eventually $\mathcal{A}$ outputs a forgery (m, j, σ). Verify it using k_j. If it is valid, output 1 (i.e., "*prf*"); otherwise output 0 (i.e., "random").

This runs in $t + c_3 n + c_2$ time.

Now we return to our series of n experiments. Consider applying the algorithm $\mathcal{A}$ to them. Let $P_i = \Pr[\mathcal{A} \text{ breaks } \mathit{Expr}_i]$. We know that $P_n \le \delta$ from theorem 1. By assumption, $P_0 = \delta + \gamma$, so

$$P_0 - P_n = \sum_{i=0}^{n-1} P_i - P_{i+1} > \gamma$$

When f is a random function, its output is indistinguishable from a randomly chosen number, so the situation is equivalent to randomly choosing s_i and k_i. Thus,

$$\begin{array}{lcl} q_1 & \triangleq & \Pr[D = 1 | f = \text{random}] & = & \sum_{i=1}^{n} \frac{P_i}{n} \\ q_2 & \triangleq & \Pr[D = 1 | f = \mathit{prf}] & = & \sum_{i=1}^{n} \frac{P_{i-1}}{n}, \end{array}$$

and so the advantage that D has is:

$$\mathrm{adv}_D = q_2 - q_1 = \frac{1}{n} \sum_{i=1}^{n} P_{i-1} - P_i = \frac{1}{n}(P_0 - P_n) > \frac{\gamma}{n}.$$

So with 2 queries to f, we determine whether it is pseudo-random with probability $\frac{\gamma}{n}$. ∎

Theorem 2 *Assume the MAC function family is $(q_m, t_m, L_m, \epsilon_m)$-secure and the pseudo-random function family is (q_p, t_p, ϵ_p)-secure. Then LOG_1 is (q, t, L, ϵ) FI secure, where*

$$\begin{aligned} q &= q_m \\ t &= \min(t_m - c_1 n - c_0, t_p - c_3 n - c_2) \\ L &= L_m \\ \epsilon &= n\epsilon_m + n\epsilon_p = n(\epsilon_m + \epsilon_p) \end{aligned}$$

Since the security of the *prf*-chain FI-MAC degrades only linearly with the number of epochs, the amount of truly random key material consumed can be very low in practice.

4 DD-FI MAC Schemes

We obtain a DD-FI MAC scheme by a simple modification of a given FI MAC scheme. To make a FI log system *LOG* DD-FI, we include sequence numbers for the log entries and generate a change-of-epoch log marker prior to an actual epoch change. All log entries m_j received within the epoch $\mathcal{E}_i$ receives the log authentication code $\mathit{FIMAC}(m_j, i) = \mathit{MAC}_{k_i}(0|\mathit{rel}(j)|m_j)$, where $\mathit{rel}(j)$ denotes the position number of message m_j within the epoch, i.e., $j - \mathit{first}(\mathit{epoch}(j)) + 1$, and $|$ denotes concatenation. To change to the next epoch $\mathcal{E}_{i+1}$, an "end-of-epoch" message is first generated and logged: $\mathit{MAC}_{k_i}(1|\mathit{rel}(\mathit{last}(i)))$. Then the secret value is transformed as before, and the keys from the previous epoch is then erased from memory. The end-of-epoch marker prevents the attacker from deleting messages from previous epochs undetectably. We denote by $\mathit{LOG}_{\mathrm{seq}}$ this modification to *LOG*.

Theorem 3 *If LOG is* (q, t, L, ϵ) *FI secure, then* LOG_{seq} *is* $(\min(q - n, 2^r), t, L - r - 1, \epsilon)$ *DD-FI secure, where* r *is the length of the sequence number field, and* n *is the number of epochs.*

We omit the proof. Here, the maximum number of messages is additionally bounded by 2^r to avoid replay risks arising from overflowing the sequence number field. Furthermore, since an epoch change generates a log entry, epoch changes are treated as if they are additional MAC queries, lowering the total number of DD-FI MAC queries allowed.

We denote by LOG_2 the DD-FI version of LOG_1, i.e., $\mathit{LOG}_{1\mathrm{seq}}$.

5 Implementation Notes

This section gives suggestions for implementing our FI scheme. It discusses the difficulty of providing the erasure property — forgetting secrets — and some traps to avoid in the initial set up with the log verifier.

5.1 Providing the Erasure Property

It can be extremely difficult to faithfully erase a value from memory. Whether it can be accomplished at all depends on the type of memory used (RAM versus disk files), the ability of the programming language to allow low-level, machine specific operations, and the underlying operating system semantics.

For traditional procedural languages, local variables contained in stack frames — especially those for cryptographic routines — often hold partial information about the value of a hope-to-be-erased secret key, and such data should be erased from long-lived programs or other long-lived storage. Even short-lived programs could be a source of weakness for the attacker to exploit: many operating systems do not erase memory pages until other processes (or the kernel) allocate new memory pages or until the system has idle CPU time to "scrub" memory pages; some operating systems never scrub memory pages at all (improper object reuse). Suppose a short-lived process had access to a secret key, and it terminated just prior to an attacker gaining access to a machine. If the CPU has been fully loaded but the machine's processes had allocated few or no new memory pages, memory pages from that process could remain intact, and the attacker would thus gain knowledge of the key. It is usually preferred to erase memory under program control rather than relying on the operating system, even for short-lived programs.

To have procedures erase local variables prior to returning can be a little tricky, especially in the presence of an optimizing compiler: erasures are "useless" assignments to dead variables, and thus are dead code. Techniques for bypassing the dead code elimination optimization depend on the compiler; for many compilers, it suffices to call a separately compiled routine to clear the values (the standard ANSI-C library routine `memset` works for most compilers). In any case, the generated assembly code should be examined to ensure that memory erasure is actually being accomplished.

The problem of erasing values is largely hopeless for most modern languages such as Scheme, ML, or Java: the garbage collector may, in the course of memory compaction, have left many copies of critical data all over memory. While it may be impossible to obtain access to these copies at the language level, an attacker that has gained supervisor privilege can easily examine raw memory in most operating systems (`/dev/mem` access in Unix systems), or use process debugging interfaces to examine a running program's memory.

At the operating system level, if the memory pages containing secret keys or partial information about their values (e.g., stack pages) are allowed to be paged out to disk, those memory pages are readable by an attacker (via low level device driver accesses, or installation of a new OS kernel). Almost no OSes will erase released virtual memory backing storage; instead, they simply rely on the OS's normal object reuse and access control policies to prevent user-level access to the released backing storage. Attackers that obtain system-level access or physical access to a computer, however, *are* able to read these released virtual memory pages.

Furthermore, because we would like our secure audit log scheme to work even if extremely well funded, motivated attacker has complete physical access to the machine, we must be very careful to prevent private data from being written to magnetic media: even overwritten magnetic media can usually be read using magnetic force scanning tunneling microscopy techniques. [7, 8, 15] Even if our paging partition (or paging file) is overwritten by new data, there's little guarantee that the old data would then be unreadable by a sophisticated attacker.

In many operating systems, it is possible to prevent memory from being paged out to disk.[3] We recommend that implementations of our algorithms lock the memory regions used to store the FI secrets and the scratch memory used during the epoch change calculations.

5.2 The Log Verifier

The secrets used by the log systems ($k_1, \ldots, k_n$ for LOG_0, and s_0 for LOG_1 and LOG_2) must be shared with the log verifier. This "sharing" could be delayed: the logger could generate random secrets and cryptographically protect them (e.g., using the verifier's public key), storing the protected versions on disk.

Such cryptographic protection must be done extremely carefully. Simply encrypting with the public key of the verifier is *not* enough: attackers can simply erase the encrypted (and unknown) secrets, generate their own new secrets, encrypt those in the same way, and proceed to generate a false history to be placed into the log using their own, newly generated secrets. In addition to encrypting these secrets, the logger must authenticate their use at the start of the log: this may be done using a timestamping protocol [9], which would require some network communications, or a public key based signature scheme which has the forward non-repudiability property [1].

Rather than generating the secrets in the logger, the key material could be generated by an external agency, e.g., the log verifier, and securely delivered to the logger using a cryptographic protocol with forward privacy. Forward privacy is critical, of course: if an encryption system without forward privacy is used, it would allow an attacker to recover secrets from previous epochs, since the logger's keys would permit recovery of the secrets from logged network messages containing it, destroying the forward integrity property for the audit log. Other than the one-time pad, the authors know of no cryptographic primitives with forward privacy.

[3] For Solaris and Linux, use the `mlock` and `munlock` system calls. For Windows, use the `VirtualLock` and `VirtualUnlock` calls.

6 Performance

This section discusses the performance of our sample implementation of our DD-FI MAC system. The source code is available via the Web at `http://www.cs.ucsd.edu/users/bsy/pub/fi-ensemble.tar.gz` in `gzip` compressed form. This implementation uses HMAC-MD5 as the MAC function, and uses the IDEA cryptosystem as the pseudo-random function family. Note that using a secret key cryptosystem — a pseudo-random permutation — instead of a pseudo-random function, in general, reduces security: by taking advantage of the birthday paradox, pseudo-random permutations can be distinguished from pseudo-random functions (and thus random functions) by sampling the domain for collisions using $\sqrt{N}$ time and space, where N is the size of the domain (in the case of IDEA, $N = 2^{64}$). Fortunately, the security of our *prf*-chain DD-FI MAC do not depend on the number of queries on the pseudo-random function (q_p).

Our sample code is not optimized for performance; instead, it is structured for clarity, to make changing cryptographic primitives (e.g., HMAC-SHA1 instead of HMAC-MD5, or DES instead of IDEA) as well as security code reviews easier. The timing data below was generated on a 150MHz Pentium machine with 40 MB of RAM. Memory size is not an issue, since the code is quite small (12K). The code only generates the DD-FI MACs and does not log the generated entries to disk nor to the network, avoiding the non-cryptographic overhead in the performance measurement. Each log entry was 80 bytes in length. The code included clearing the stack at every epoch change, so that if the underlying pseudo-random function implementation was not careful about erasing scratch memory locations, our sample DD-FI implementation will still erase deallocated stack memory.

Our unoptimized sample code achieves a sustained throughput rate of 21,709 log entries per second if 1024 log entries are made between epoch changes (averaging 0.046 mS per entry). At a more realistic rate of 8 log entries per epoch change (approximation from "burstiness" of `syslogd` data), the rate is 14604 entries per second (0.068 mS per entry). Even if we changed epochs immediately after every log entry, our code achieves a throughput of 4471 log entries per second (0.22 mS per entry); this is still much faster than the estimated 100 mS rate needed to guard against automated attacks – more than 99.7% of the CPU is available for non-FI log computation (i.e., we incur less than a 0.3% overhead).

7 Other Applications

Forward integrity schemes are useful for other applications. For example, consider the problem of mobile software agent security. In mobile agent systems, a software entity autonomously migrates among servers — to visit large databases, for example, much more efficiently than using remote procedure calls. There are two security problems here: the more commonly considered one of protecting the agent server from the incoming, possibly malicious agent, and the converse one of protecting the agent from the possibly malicious server on which it executes. It is the latter, more difficult problem where forward integrity has an application.

Some suggested applications for mobile software agents are in electronic commerce, and here the security of the agent becomes important. We would like to prevent a potentially malicious agent server, which has complete access to a mobile agent's state, from corrupting any partial results computed while the agent was at a previous host[17]. If, for example, the agent is performing price comparisons on the behalf of the user, a malicious server would wish to alter the agent's memory of the best price seen so far, so that the malicious server would get the user's business instead of some previously visited server. This is very important for the acceptance of remote execution – if the users of mobile agents cannot have trust that the result returned is correct, there would be little incentive to use such systems in the first place.

To help protect a mobile agent's state, we view the migration of an agent from one agent server to the next as a change of epoch. Partial results that will be eventually communicated back to the user are authenticated using a DD-FI MAC, and prior to traveling to the next agent server an epoch change occurs: because all computation done at the next server will be in a different epoch, that server cannot undetectably modify the partial results contained in the agent's memory. When the agent goes home with its list of partial results, the user can verify non-tampering using the original secret key.

8 Future or related Work

The key generation scheme that we described above can be used to provide forward privacy as well as forward integrity, effectively "stretching" any forward privacy initial secret transfer, amortizing its cost over many private-key forward privacy epochs. Here, the need for an initial forward private secret transfer is the same as that in the forward integrity above. This motivates the development of cryptosystems with forward privacy. While interactive protocols with forward privacy exist (e.g., Diffie-Hellman key exchange), for applications where the logger does not have a good source of randomness available or where the communications round-trip overhead is prohibitive, the availability of such a cryptographic primitive is critical for allowing the log verifier to safely provide the initial secret value to the logger.

A technically more challenging problem is to design digital signature schemes with the forward security property. This has been done by Bellare and Miner in [5]. Such schemes provide an attractive alternative to the secret-key forward integrity system described here. While the efficiency of public key schemes is unlikely to rival that of private key schemes, the elimination of the need to maintain privacy for the verifier can be extremely attractive in many applications: forward secure non-repudiable signatures can simplify, for example, the construction of some kinds of electronic contracts.

References

[1] R. Anderson, B. Crispo, J.-H. Lee, C. Manifavas, and R. Needham. The Guy Fawkes protocol, 1997. Preprint.

[2] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Proceedings 1997 IEEE Symposium on Security and Privacy*, pages 65–71, Oakland, CA, May 1997. IEEE, IEEE Computer Society Press.

[3] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In N. Koblitz, editor, *Advances in Cryptology: Crypto '96 Proceedings*, volume 1109 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[4] M. Bellare, J. Kilian, and P. Rogaway. The security of cipher block chaining. In Y. Desmedt, editor, *Advances in Cryptology: Crypto '94 Proceedings*, volume 839 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[5] M. Bellare and S. Miner. Digital signatures wth forward security. 1997.

[6] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the ACM*, 33(4):210–217, 1986.

[7] R. Gomez, A. Adly, I. Mayergoyz, and E. Burke. Magnetic force scanning tunnelling microscope imaging of overwritten data. *IEEE Transactions on Magenetics*, 29(6):2380, November 1993.

[8] P. Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the Sixth USENIX Security Symposium*, San Jose, CA, July 1996.

[9] S. Haber and W. S. Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 3(2), 1991.

[10] M. P. Herlihy and J. D. Tygar. How to make replicated data secure. In *Advances in Cryptology, CRYPTO-87*, pages 379–391. Springer-Verlag, August 1987.

[11] M. K. McKusick, K. Bostic, and M. J. Karels. *The Design and Implementation of the 4.4 BSD UNIX Operating System.* Addison-Wesley, 1996.

[12] M. K. McKusick, M. J. Karels, S. J. Leffler, W. Joy, and R. Fabry. *Berkeley Software Architecture Manual, 4.4BSD Edition.* Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, Berkeley, California.

[13] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997. ISBN 0-8493-8523-7.

[14] U. S. Department of Defense, Computer Security Center. Trusted computer system evaluation criteria, December 1985.

[15] V. Veeravalli. Detection of digital information from erased magnetic disks. Technical report, Carnegie-Mellon University, 1987. MS Thesis.

[16] B. S. Yee. *Using Secure Coprocessors.* PhD thesis, Carnegie Mellon University, 1994.

[17] B. S. Yee. A sanctuary for mobile agents. Technical Report CS97-537, University of California at San Diego, La Jolla, CA, April 1997. An earlier version of this paper appeared at the DARPA Workshop on Foundations for Secure Mobile Code.