Finding the Evidence in Tamper-Evident Logs

Daniel Sandler
Rice University
dsandler@cs.rice.edu

Kyle Derr
Rice University
derrley@cs.rice.edu

Scott Crosby
Rice University
scrosby@cs.rice.edu

Dan S. Wallach
Rice University
dwallach@cs.rice.edu

Abstract

Secure logs are powerful tools for building systems that must resist forgery, prove temporal relationships, and stand up to forensic scrutiny. The proofs of order and integrity encoded in these tamper-evident chronological records, typically built using hash chaining, may be used by applications to enforce operating constraints or sound alarms at suspicious activity. However, existing research stops short of discussing how one might go about automatically *determining whether a given secure log satisfies a given set of constraints on its records.*

In this paper, we discuss our work on QUERIFIER*, a tool that accomplishes this. It can be used offline as an analyzer for static logs, or online during the runtime of a logging application.* QUERIFIER *rules are written in a flexible pattern-matching language that adapts to arbitrary log structures; given a set of rules and available log data,* QUERIFIER *presents evidence of correctness and offers counterexamples if desired. We describe* QUERIFIER*'s implementation and offer early performance results.*

1. Introduction

Modern communication, computation, and commerce are notorious for generating a tremendous amount of logging data. Every transaction or conversation, public and private, leaves a permanent record, and such records are beginning to find use in forensic investigations and legal proceedings.

The evidence one seeks in these sorts of investigations often takes the form of statements of *existence* and *order*. Put another way, we wish to discover, "What did Alice know, and when did she know it?" Critically, we must be able to *prove* the veracity of our findings in the face of accidental erasure and deliberate manipulation of the "permanent" record.

Recent research offers a powerful tool in the form of *secure logs:* data structures that use hash chains to establish a provable order between entries [6]. The principal benefit of this technique is tamper-evidence: any commitment to a given state of the log is implicitly a commitment to all prior states. Any attempt to subsequently add, remove, or alter log entries will invalidate the hash chain. A log whose entries are linked together in this way represents a *secure timeline:* a tamper-evident total order on events. We may additionally *entangle* the timelines of different service domains [2] by periodically cross-pollinating entries among their logs, further restricting a party's ability to unilaterally alter the past without leaving evidence.

A secure log is therefore an excellent ingredient for any system which must yield evidence about the order of past events. A messaging application, for example, should be able to tell us what Alice knew and when by performing some sort of query on Alice's secure log. Such a system may even wish to use the log to enforce operational constraints; if Alice is only allowed to communicate with a fixed set of people, her log will contain proof of any misbehavior.

The question, however, of how exactly to *find* evidence in secure logs remains unanswered. We can hardly expect human auditors to pore over raw log data, but neither does there exist a ready solution for *automatically* probing logs for correctness or inconsistency. This problem is exacerbated by the diversity of potential secure logging applications, whose log queries and constraints are likely to vary widely. The goal of a general-purpose facility for domain-specific log verification, therefore, has been thus far avoided by secure logging research.

In this paper we share our work on QUERIFIER: to our knowledge, the first general-purpose tool for analyzing secure logs. It allows auditors or application software developers to articulate properties of interest using a flexible pattern-matching predicate language that can generalize to any log or record format. Given a set of predicate rules and a finite log, QUERIFIER will determine whether the log conforms to the rules and optionally emit a list of counterexamples. QUERIFIER can also be used "online," embedded in a live application: it will perform queries and verification while operating *incrementally* on a growing log. In this case, QUERIFIER will return what results it can, and its computation may be resumed as more data becomes available, improving performance greatly compared to re-starting the computation from scratch..

2. Querifier

According to Schneier and Kelsey:

> *Audit logs are useless unless someone reads them. Hence, we first assume that there is a software program whose job it is to scan all audit logs and look for suspicious entries.* [6]

It is unclear how to automatically identify a "suspicious entry." It is easy to envision a program that merely validates all the hash pointers and signatures in a log, but this only tells us if someone has tampered with the record. How do we go about discovering violations of high-level constraints, such as "Alice may only talk to certain people?" Rather than developing custom log-analysis software for every situation, we attempt here to solve the problem more generally, allowing an application to *specify* its constraints of interest (viz., what is "suspicious") to a verification tool.

2.1. Goals

We seek, then, a general-purpose tool with a number of essential properties: applications and auditors must be able to specify arbitrary properties and rules of interest; it must be able to verify these rules in logs of many shapes; it must be practical to embed within a Java program; it must be able to operate incrementally on a growing log.

These objectives are motivated by our ongoing research into electronic voting [5], in which we use secure logs to capture a tamper-evident record of election-day events. If a suspicious or invalid event occurs on election day, we want to know immediately so that appropriate action can be taken (for example, removing the offending voting machine from service).

Construct	Form	Result
Equality	$T_1 = T_2$	True when both tuples are recursively identical.
Pattern matching	$T_1.P$	Like equality, but tuple P may contain wildcards that match any element of T_1 in the same position.
Ordering	$T_1 \lll T_2$	True when T_1 precedes T_2; that is, if a hash chain path exists from T_2 back to T_1.
Negation	$\neg\, \Phi$	Boolean negation of predicate Φ.
Conjunction	$\Phi \wedge \Psi$	True when both Φ and Ψ are true.
Disjunction	$\Phi \vee \Psi$	True when either Φ or Ψ is true.
Material conditional	$\Phi \rightarrow \Psi$	Equivalent to $\Psi \wedge \neg\Phi$.
Universal quantification	$(\forall \alpha \in S)\, \Phi$	True when Φ is true for every α in the set S.
Existential quantification	$(\exists \alpha \in S)\, \Phi$	True when Φ is true for any α in the set S.

Figure 1. Expressivity of QUERIFIER rules. The logic includes truth-functional connection, quantification, and relations between log records (here termed tuples).

$$
\begin{array}{l}
(\exists x \in L) \\
\quad (\exists y \in L) \\
\\
\quad (\ (x.\text{POLLS\_OPEN\_MSG} \neq \varepsilon) \\
\qquad \wedge\ (y.\text{POLLS\_CLOSED\_MSG} \neq \varepsilon) \\
\qquad \wedge\ (x \lll y)\ )
\end{array}
$$

```
(exists x all-set
  (exists y all-set
    (and
      (match POLLS_OPEN_MSG x)
      (match POLLS_CLOSED_MSG y)
      (precedes x y all-dag))))
```

Figure 2. Example S-expression representation of logical rules. The simple rule here asserts that there exist both a polls-open and polls-closed message in the log, and that the former precedes the latter. The special value `all-set` is the set of the available log messages (corresponding to the finite set L in the logic), and `all-dag` is a DAG of time constructed from `all-set` by an application plugin.

2.2. Rule language

QUERIFIER treats a log as an unordered set of tuples. Each element of a tuple is either a string or another tuple; this recursive data structure is analogous to the well-known LISP S-expression, and is sufficiently general to represent any log record.

A QUERIFIER rule expression is a first-order logical predicate over the entire log. The rule language allows basic logic connectives as well as quantification over sets (such as the log); it also includes pattern-matching and name-binding constructs to allow records of interest to be selected and compared. Finally, it allows for application domains to "plug in" additional language features, such as the "happened-before" relation that is essential to dealing with secure log timelines. Table 1 summarizes the rule language; a more thorough description of the language can be found in a technical report [4]. Our QUERIFIER prototype uses a concrete syntax based on Rivest's S-expression specification [3] to express these logical rules (see Figure 2).

2.3. Algorithms

We briefly describe some of the unusual algorithmic approaches currently used in our QUERIFIER prototype.

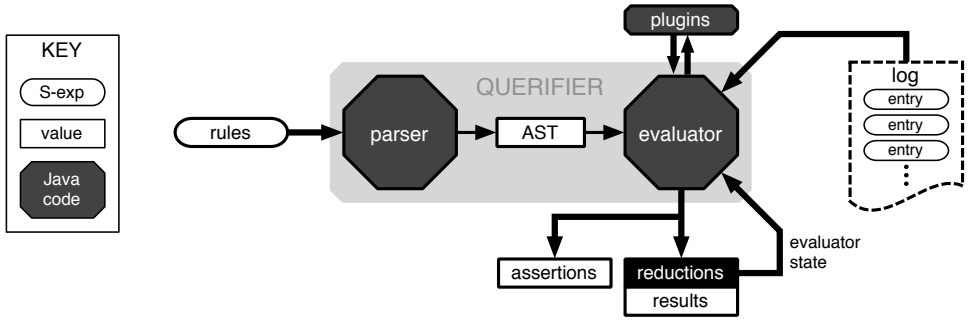

Figure 3. QUERIFIER components and operation. Applications supply rules in the form of S-expressions; the verifier parses rule expressions into an AST suitable for evaluation. Log entries, also S-exps, are fed to the verifier, which recursively interprets the AST for each, finally yielding a result value and a list of assertions (if any). Partial results contain sufficient state to resume the evaluator without performing redundant computation when new log data arrives.

Incremental evaluation using reductions. In the case of a system (such as our voting booth example) where a secure log needs to be examined in near real time for violations, we will re-invoke QUERIFIER on the rules and the log each time a new log record appears. But we want to avoid re-starting the *entire* computation from scratch; much of this computation is redundant. (For example, once we have satisfied our constraint that there exist a "polls open" message, we need not check it again.)

QUERIFIER addresses this by using a kind of partial computation technique when evaluating rules: any particular evaluation which involves quantification over a growing log may result in a *reduction* rather than in a result. This reduction, while its truth value is unknown, is a simplification of the original problem. That is, when the reduction is fed back into the evaluator, no computation will be repeated in the search for truth.

Once the log is closed for good, QUERIFIER runs one more time, reducing the entire computation to a final result. The structure of the QUERIFIER evaluation engine, taking into account this reduction technique, is shown in Figure 3.

Graph search with timeline pruning. Determining the order between secure log entries can naturally be cast as a graph search problem, because hash-chained log entries form a *graph of time*: a directed acyclic graph whose vertices are entries, and whose directed edges represent direct precedence. If a *hash chain path* exists in a log leading from entry B to entry A, then the event described by entry A must have happened before event B, written $A \ll B$. (A corollary of the "happened before" relationship is potential causality: A may have affected B [1].) If instead a path exists from A to B, then B precedes A in time.

Note that in a log captured on a single host, this graph ought to degenerate to a line (Figure 4a). When multiple hosts entangle their logs, however, nearly-simultaneous events from different origins may result in log entries that succeed the same prior entry. The set of hash-chained log entries form a DAG in this case (Figure 4b.)

A conventional breadth-first graph search costs $O(e)$ time for an arbitrary directed graph with e edges; for a log with many entries and many repeated order queries this can highly problematic. Fortunately, we can exploit the special structure of secure logs to make this quite a bit faster. We observe that in a single log, we can precompute a total order on all records in the log, making the

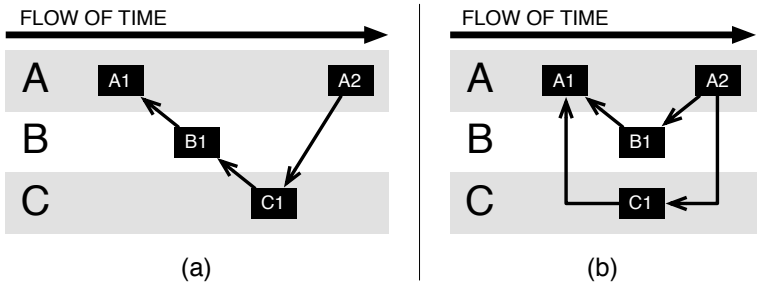

Figure 4. The graph of time. Participants A, B, and C entangle their logs to form an overall timeline. Arrows denote direct ordering due to a hash chain link; for example, in (a), event A1 directly precedes B1, and so forth. Graph search proves that, for example, B1 $\ll$ A2, despite the lack of a direct link. In (b), the timeline becomes a more general graph; events B1 and C1 happened roughly simultaneously, and as a result, they share A1 as a direct predecessor. We do not know which happened first, but we know that they both happened after A1 and before A2 (which includes hash links to both B1 and C1).

precedes relation return in constant time. For a timeline-entangled system with a few timelines, we must still use a BFS, but once our search finds the *timeline* of the destination node, we may now prune the rest of the BFS search tree and use the local ordering instead. A complete discussion of this algorithm can be found in a technical report [4].

3. Evaluation

3.1. Experimental setup

Voting simulation. We evaluated our prototype of QUERIFIER using a synthetic log from an electronic voting system [5]. The log, comprising 763 individual events from 9 nodes (eight voting booths and one supervisor console), was collected during an 8-hour real-time simulation of an election held in a single polling place. The simulation was generated using a modified version of the Java source code to our VoteBox electronic voting system, replacing the supervisor and voter GUIs with automated drivers that behave as follows. After opening the polls, the supervisor authorized a new ballot (simulating a new voter being assigned to a voting machine) every 10 to 120 seconds when voting machines were available. Each "booth" node simulated a voter who completed his/her ballot anywhere from 30 to 300 seconds later. After eight hours, the polls were closed; a total of 127 ballots were cast in that time.

Voting rules. Our experimental rule set contains seven constraints, expressed in English as follows:

1. All messages are correctly formatted.
2. There exists a polls-open record beginning the election.
3. There exists a polls-closed record concluding the election.
4. The polls-open precedes the polls-closed.
5. Every cast-ballot is preceded by an authorized-to-cast, and their authorization nonces match.
6. Every cast-ballot precedes a ballot-received, and their authorization nonces match.
7. Every cast-ballot has a unique authorization nonce.

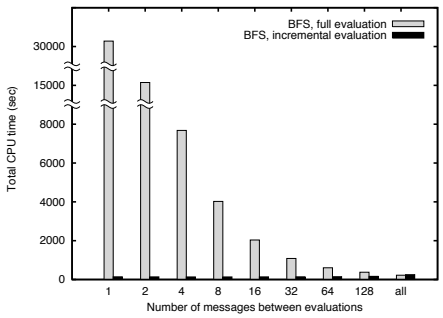

Figure 5. Incremental evaluation. Bars indicate total time to consume and evaluate the entire log. The rightmost bar represents an interval equal to the length of the input, in which case the two approaches are equivalent; as the intervals get shorter, the costs of re-verifying from scratch become obvious.

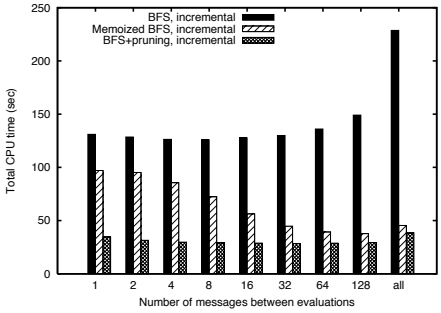

Figure 6. Graph search. Incremental verification performance across several graph search algorithms (described in in Section 2.3). As in Figure 5, we re-run the verifier at various intervals to quantify the overhead associated with each invocation. Bars indicate total time to verify the entire log.

As an example of how these assertions may be realized as QUERIFIER logical rules, we represent rules 5, 6, and 7 together with the following expression:

$$
\begin{aligned}
&(\forall b \in L)\ (b.\textsc{cast\_ballot} \neq \varepsilon)\ \rightarrow\ (\\
&\qquad (\exists a \in L)\ (\ a.\textsc{vote\_auth} \neq \varepsilon \quad \wedge\ a.\textsc{auth\_nonce} = b.\textsc{ballot\_nonce} \quad \wedge\ a \ll b\ )\\
&\wedge\quad (\exists r \in L)\ (\ r.\textsc{vote\_receipt} \neq \varepsilon\ \wedge\ r.\textsc{receipt\_nonce} = b.\textsc{ballot\_nonce}\ \wedge\ b \ll r\ )\\
&\wedge\ \neg(\exists x \in L)\ (\ x.\textsc{cast\_ballot} \neq \varepsilon\ \wedge\ x.\textsc{ballot\_nonce} = b.\textsc{ballot\_nonce}\ \wedge\ x \neq b\ )\quad)
\end{aligned}
$$

A straightforward translation from the above to S-expression syntax (as described in Section 2.2) yields rules that QUERIFIER can directly evaluate.

3.2. Results

Incremental evaluation. When given the entire 763-entry log at once, our basic QUERIFIER implementation completed rule verification in about 220 CPU seconds (0.3s per log entry). When fed the log incrementally, however, as shown in Figure 5, using reductions in the evaluator omits a tremendous amount of redundant work incurred when reevaluating the entire log from scratch with every batch.

Graph search. We also compared three algorithms for computing order between entries in the graph of time: BFS, memoized BFS (like BFS but with pairwise cached results), and BFS with timeline pruning (see Section 2.3). We examined the performance of these algorithms in the incremental verifier, using the same batch-size variation described previously; the results are shown in Figure 6. As expected, the pruning algorithm improves substantially over naïve graph search.

4. Conclusion

Contributions. The burgeoning study of secure logs has much to say about what evidence those logs may potentially yield, but little about how to find that evidence. The work we have described explores using **predicate logic**, combined with order and pattern-matching relations, to express properties over secure logs of arbitrary shape and complexity. We have also described QUERIFIER, a prototype **implementation** of a rule verifier that applies this technique to logs and rules using several novel algorithms.

We have found QUERIFIER to be a useful ingredient in our secure logging applications under development. It allows us to focus on expressing the rules that define correct behavior without reinventing the mechanism for *evaluating* those rules in each unique situation. The rules ultimately represent concise specifications of application semantics, and have even revealed evidence of bugs in our software. QUERIFIER is a first step in bridging the gap between "there exists a proof of misbehavior" and actually finding that evidence.

Future work. We look forward to improving the performance and scalability of QUERIFIER. Optimizations and other techniques from databases may be applicable to the problem, and we hope to identify these situations in an optimized version of the evaluator. Where possible and appropriate, we may also be able to reduce the size of the log under examination by summarizing or pruning obsolescent and unnecessary data. We are also actively exploring techniques for *distributing* the verification task among a number of computation nodes. Our objective is to make QUERIFIER practical for vastly larger data sets collected over very long periods, such as logs from internet-scale applications like instant messaging and email.

References

[1] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[2] Petros Maniatis and Mary Baker. Secure history preservation through timeline entanglement. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, August 2002.

[3] Ronald L. Rivest. S-expressions. IETF Internet Draft, May 1997. http://people.csail.mit.edu/rivest/sexp.txt.

[4] Daniel Sandler, Kyle Derr, Scott Crosby, and Dan S. Wallach. Finding the evidence in tamper-evident logs. Technical Report TR08-01, Department of Computer Science, Rice University, Houston, TX, January 2008. http://cohesion.rice.edu/engineering/computerscience/TR/TR_Download.cfm%?SDID=238.

[5] Daniel Sandler and Dan S. Wallach. Casting votes in the Auditorium. In *Proceedings of the 2nd USENIX/ACCURATE Electronic Voting Technology Workshop (EVT'07)*, Boston, MA, August 2007.

[6] Bruce Schneier and John Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security*, 1(3), 1999.