FINDING THE EVIDENCE IN TAMPER-EVIDENT LOGS[1]

Daniel Sandler Kyle Derr
Scott Crosby Dan S. Wallach

Rice University

{dsandler,derrley,scrosby,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, the problem of how to automatically detect violations remains open. Low-level inconsistencies, such as gaps in the hash chain, are detectable without knowledge about the application, but existing research stops short of extracting or verifying application-specific log properties.

In this paper we contribute the design and implementation of a system for discovering this kind of evidence. We first propose a logical language for applications to express concisely the constraints that apply to their logs and the evidence that can be extracted from them. We offer several algorithms for efficiently and incrementally evaluating these rules. Finally, we present Querifier*, a log analysis package that implements our proposed techniques. It can be used offline as an analyzer for static logs, or online during the runtime of a logging application. Given validity rules and available log data, it presents evidence of correctness and offers counterexamples if desired. We describe* Querifier*'s implementation and offer early performance results: for a rule set developed for a distributed voting application, we observed that our system could incrementally verify a realistic election-day log at 50 events per second.*

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. 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 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.

[1] Rice University Department of Computer Science technical report TR08-01, January 23, 2008.

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; we would rather not implement separate analysis tools for each one. 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 present an approach that makes this possible. Applications may express their unique invariants and queries in terms of predicate rules over their own log structure. The logical language in which these rules are written can be applied to log records of any shape, size or format. It makes no assumptions about the source of the logs, nor even their completeness; partial data sets are perfectly valid under the logic, as are groups of logs collected from many nodes in a distributed system.

We also offer Querifier: to our knowledge, the first general-purpose tool for analyzing secure logs. Given a set of rules and a finite log, it will determine whether the log conforms to the rules and, if desired, 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. This partial evaluation technique applies to any situation in which log data may be incomplete, e.g., in a distributed system in which the verifier only has access to some nodes' logs.

Our paper continues with background material on secure logs in Section 2. We then explore in Section 3 our logical language for secure log properties, and offer algorithms for interpreting this logic in Section 4. These algorithms find use in our Querifier implementation, which is the subject of Section 5; we evaluate its performance in the context of realistic log data and rules in Section 6. Finally, we share our conclusions and future directions in Section 7.

2 Background

2.1 Secure logs

Digital signatures offer proof of authenticity and resistance to repudiation. Once revealed, a signed statement cannot be altered by anyone, even its author. But what of the *time* at which such a statement is made? Authenticity alone does not bind actors to a consistent, unchanging view of the past. We seek proof that the historical record has not been modified in any way, including addition, omission, or re-ordering of events.

Secure time-stamping services [13] can be used to bind arbitrary statements to time stamps from a trusted clock, but something more sophisticated is required in order to prove that original entries are not *absent* from forensic records. This assurance is afforded by *secure logs*, which force an actor or process to commit to its entire history with each new message. Secure logs borrow from authenticated data structures [21] the insight that the integrity of a large data set can be represented compactly with an *authenticator* (a "hash total" in Merkle's original articulation [20]) that cryptographically summarizes the contents.

Secure logs are typically realized with hash chains [27]: each new log entry includes the hash of a prior one, and the hash of the log head serves as the current authenticator. Such an append-only data structure locks down the content of historical records so that no individual entry may be altered without also requiring every subsequent one to also be altered (to restore the integrity of

the hash chain). A log whose entries are linked together in this way represents a *secure timeline:* a tamper-evident total order on events.

An attempt to retroactively alter those events is only detectable, however, if there already exists some commitment to the correct, *present* state of the log. For example, the owner of a log might periodically publish his log's authenticator to a public medium, such as a bulletin board. In distributed systems where no such medium exists, mutually distrustful parties may exchange authenticators with one another. By incorporating hashes from (untrusted) remote logs into a (trusted) local log, a *hash lattice* [27] is formed, spanning all logs; each timeline becomes *entangled* [19] with the timelines of other service domains. In such a system, any unilateral attempt to alter the past is identifiable as it violates commitments already made to (and retained by) by other parties. As described by Maniatis [18], these commitments allow participants to map events in others' timelines to intervals in their own, creating a partial order over all events, in the manner of Lamport clocks [17]. The more frequent these exchanges of commitment, the more dense the entanglement, increasing the effective temporal resolution of the system, and reducing each party's freedom to make changes to his own log.

2.2 Integrity checking in secure logs

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. . . . After that, the details are completely dependent on the particular log entries. [28]

Current research in secure and entangled logging is content to leave the discussion here, having successfully argued that buried within such a secure log is proof of its order and integrity. But how might we go about unearthing this proof? For a log with many entries, how do we go about locating, for example, a hole in the record left by the omission of an incriminating statement?

This low-level idea of validity—namely, that a secure log is tamper-free if its hash chains are valid and complete—can be verified mechanically. It is easy to envision a program that combs such a log, computing message digests and validating signatures. The program will likely differ for each application, owing to the idiosyncrasies of log formats, but will perform the same essential computations.

2.3 Application-specific properties

It is reasonable to assume that an application which makes use of such logs might want to enforce more sophisticated constraints on the behavior of its participants (as evidenced by entries in the log). Some examples include:

- **In a banking system:** We can ask questions about transactions and balances. For example, did Alice receive a certain amount of money before spending it? This could help her prove that she never went below a minimum account balance.
- **In a stock trading system:** Ordering issues are critical to the fair execution of stock trades. For example, the SEC has charged several stock traders with eavesdropping on large institutional stock orders and trading ahead of them at better prices [29]. This evidence naturally extends to more typical communications (e.g., email, instant messaging), permitting someone accused of insider trading to prove that they issued a stock order before they acquired any insider information.

- **In a multiplayer game:** Modern online games typically rely on correct behavior of client software, and are therefore vulnerable to cheating by modified clients. Such a constraint might involve a player completing some action (e.g., a payment) before being allowed to begin another action (e.g., accessing a certain area). Violation of this constraint indicates buggy or malicious software.
- **In a file server:** Yumerefendi et al. [33] describe a networked storage service that uses secure logs as evidence when the correctness of the server is challenged. Clients "trust but verify" the server, periodically requesting from it proofs of correct behavior. Clients must check these proofs for *semantic* correctness: beyond merely being well-formed and authentic, they must conform to the operational semantics of a file server.

For a deeper and more concrete example, let us consider an electronic voting system similar to VoteBox [26], which connects voting terminals ("booths") in a polling place together. Every booth announces its critical state changes (e.g., "I have come online" or "I just cast a ballot") in the form of broadcast messages. Also present in this network is a "supervisor" terminal, used by poll workers to start and stop the election, as well as to assign voters to booths; these operations also result in broadcasts. Each broadcast message includes one or more hashes of recent messages, and is recorded redundantly by every connected machine in its own secure log. The logs captured by each booth, taken together, yield a reliable account of the election that survives the failure of individual machines and attests to the validity of the election results.

In order to make this claim, we must be able to automatically examine these logs not just for breaks in the hash chain, but for violation of the "rules" of a valid election in this system. For example, each ballot to be tallied must first be shown to have been *authorized* to be cast. The authorizations appear in the log as authorized-to-cast messages (sent by the supervisor when it assigns a new voter to a specific booth); one such message must therefore precede every single cast-ballot message.

Clearly, this rule—asserting an invariant in the secure log—makes sense only in the context of the electronic voting problem, and is in fact intimately tied to this particular system design. We may reasonably assume that other systems that employ secure logs will have their own particular constraints, above and beyond state replication, that must be mechanically verified.

Ideally, we ought to be able to develop a single log analysis tool that, given a log, can reach any such conclusion that an application might need. The tool should need no intrinsic understanding of the structure of the log, nor of the application's needs; nor should it achieve this generality by *imposing* those things on an application. Rather, we would like to allow applications to be able to *specify* what behavior is considered correct, and to do so in terms of their own log entry structure and format.

Finally, this verification tool must not rely on deterministic or repeatable behavior on the part of nonetheless correct applications. The properties we wish to articulate (indeed, those we have already described) are not limited to matters of faithfully replicating some piece of state across a system. That is, a system free of Byzantine faults (e.g., a buggy implementation, or a correct system that is incorrectly operated) may still violate a rule that we have set out for the system. Therefore, techniques such as PBFT [4]—as well as the recently proposed PeerReview [14], which relies on deterministic replay to identify failures—are insufficient to the task we have proposed.

3 A language for log properties

We begin by proposing an abstract language for an application to articulate its constraints and queries over logs, including secure logs, to some general-purpose rule evaluator. This language

is based on the well-understood constructs of first-order predicate logic. We have found that useful properties of logs are naturally expressible in such a system. In this section, we detail the semantics of this language; discussion of how to interpret this language for concrete inputs is the subject of the following section.

3.1 Domain of expression

In practice, a log is a finite set of entries. An entry might take the form of a single datum, such as a character string; a list or array of strings or other data; or a more general recursive structure, possibly including other entries. Our language must generalize over all such logs.

We therefore propose the following *domain* for expressions in our language. The abstract set of all *tuples* (denoted $\mathbf{T}^*$) has as elements all tuples whose entries are either other tuples or elements of the set of *characters* (denoted $\mathbf{T}$). (As a shorthand, we will write *strings*—flat tuples of only characters—as text in "quotes".) In addition, $\mathbf{T}^*$ contains the *null tuple* (ε) and the *wildcard tuple* (**?**).

Finally, $\mathbf{T}^*$ is an infinite set, so we define $L \subset \mathbf{T}^*$ to be the finite corpus of *log entries under scrutiny*. We will use L to refer to a set of concrete log entries that have been produced by an application and that may therefore be considered by logical rules.

3.2 Relations

The following relations (functions over tuple space) are defined:

Equality ($\mathbf{T}^* \times \mathbf{T}^* \to \{\text{TRUE}, \text{FALSE}\}$) We define $T_1 = T_2$ to be true if both are ε, or if they have the same length and are recursively equal: that is, the i^{th} sub-element of T_1 and T_2 are equal for all $1 \leq i \leq |T_1|$.

Pattern matching ($\mathbf{T}^* \times \mathbf{T}^* \to \mathbf{T}^*$) We say that a tuple T_1 *matches* the tuple pattern P (written $T_1.P$) if they are recursively equal as described above, with the following relaxation: we also consider any member of $\mathbf{T}^*$ to match the special *wildcard* tuple. This permits recursive pattern-matching of tuples. For example, the tuples ⟨"All", "Men", "Are", "Mortal"⟩ and ⟨"All", "Cats", "Are", "Animals"⟩ both match the pattern ⟨"All", **?**, "Are", **?**⟩.

We define the result value of $T_1.P$ in the affirmative case to be a tuple containing those sub-elements of T_1 that correspond to the wildcards in the pattern P. Therefore, the result of the match
⟨"All", "Men", "Are", "Mortal"⟩.⟨"All", **?**, "Are", **?**⟩ is the tuple ⟨"Men", "Mortal"⟩.

When T_1 and P do not recursively match, that is, for some subexpression of T_1, it is not equal to the corresponding (non-wildcard) subexpression of P, the result of $T_1.P$ is defined to be ε.

Ordering ($\mathbf{T}^* \times \mathbf{T}^* \to \{\text{TRUE}, \text{FALSE}\}$) A function that is absolutely essential to rules for secure logs—and which we therefore include in the core logic—is the "precedes" relation. We define $T_1 \lll T_2$ as true if $T_1 \in L$ and $T_2 \in L$ and T_1 provably precedes T_2 in time. (Though this relationship is represented in secure logs by a a hash chain path between T_2 and T_1, we avoid discussing concrete implementations here. It is sufficient to make this operation available to the logic. The problem of finding this path is discussed in Section 4.2.)

Other functions ($(\mathbf{T}^*)^n \to \mathbf{T}^*; n > 0$) Arbitrary relations over tuple space may be necessary in order to express interesting rules. (For example, an application may wish to write rules that involve cryptographic operations, such as hash computation or signature verification.) We explicitly allow such functions to be used in the language.

3.3 Logical expressivity

We express sentences involving these relations in a bivalent predicate logic whose universal domain is $\mathbf{T}^*$.

Truth-functional connection For any formulæ Φ and Ψ, we include the truth functions: negation ($\neg\Phi$), conjunction ($\Phi \wedge \Psi$), disjunction ($\Phi \vee \Psi$), and material conditional ($\Phi \rightarrow \Psi$).

Quantification The universal and existential quantifiers are also defined over $\mathbf{T}^*$, and have the form $(\forall\alpha)\ \Phi$ and $(\exists\alpha)\ \Phi$ respectively, where only the variable α is allowed to be free in Φ.

(We make frequent use of the shorthand $(\forall\alpha \in S)\ \Phi$, which represents the conditional expression $(\forall\alpha)\ ((\alpha \in S) \rightarrow \Phi)$. The set $S \subset \mathbf{T}^*$ can be defined using set-builder notation of the form $S = \{\alpha\ :\ \alpha \in L \text{ and } \Psi\}$, where the variable α may be free in the predicate Ψ.)

4 Algorithms

4.1 Evaluation of tuple logic expressions

To guarantee our tuple logic's decidability, we disallow quantification over infinite sets. Intuitively, this restriction does not limit an application in the practical sense; the set of constraints an arbitrary application should wish to verify will only govern L (the logs under scrutiny), which is, by definition, a finite set. We now consider the problem of *evaluation*, that is, computing the results of expressions in this logic.

4.1.1 Logical formulæ

Each of the truth-functional connectives can be evaluated by a constant-time lookup in a small fixed truth table. A single quantification $(\forall\alpha \in S)\ \Phi$ or $(\exists\alpha \in S)\ \Phi$ can be evaluated by doing an exhaustive search of S for a witness which makes Φ false (in the former case) or true (in the latter case). At each iteration in the search, Φ must be evaluated once. If we assume Φ has only connectives and relations, then the un-optimized evaluation of a single quantifier is $O(n \cdot c)$, where n is the cardinality of S and c is the cost of evaluating the formula Φ (which may involve relation evaluation, described below). More generally, the complexity of any expression involving quantification is $O(n^d \cdot c)$, where d is the maximum depth of quantifier nesting.

4.1.2 Relations

Evaluating the pattern-match relation can be done using a recursive pairwise comparison algorithm mirroring the description given in Section 3.1. Any atom (character or ε) matches itself; the wildcard matches any tuple or atom. Two tuples T_1 and T_2 match if their lengths are the same and each respective pair of elements recursively match. This algorithm's complexity is $O(s)$, where s is the size of the pattern (more specifically, the number of sub-tuples and atoms in the pattern). Equality can be treated as a special case of match in which no wildcards are present.

4.1.3 Finding counterexamples and witnesses

Beyond merely determining whether a log is valid given a rule set, we must also consider how to identify the specific entries that are responsible for violations. This task is challenging because it is unclear, given an arbitrary expression, whether any individual subexpression's truth value is

Technique	Prep	Lookup	Space
Graph search	—	$O(e)$	$O(n)$
Full precomputation (Warshall's)	$O(n^3)$	$O(1)$	$O(n^2)$
Memoized graph search	—	$O(e)$	$O(n^2)$
Graph search with pruning	—	$O(e)$	$O(n)$
Precompute with timelines	$O(k \cdot e)$	$O(1)$	$O(n \cdot k)$

Table 1: Summary of graph-of-time search algorithms. "Prep" is the running time for the startup phase of the algorithm, and "Lookup" is the running time for each test of $A \ll B$. "Space" is the storage cost of the algorithm over all operations. (Complexity is given in terms of n log entries among k hosts and e edges in the graph of time.)

exceptional and therefore of interest. If a negation governs a quantifier, its meaning is inverted, making it impossible to automatically determine if, when a witness is found in the exhaustive search, it is evidence of *good* or *bad* behavior.

If automatic isolation of these witnesses is desired, we may transform rules (preserving truth) such that negations only govern predicates, not quantifiers. We do this translation by repeated application of De Morgan's law for truth-functions $[\neg(\Phi \wedge \Psi) \Leftrightarrow (\neg\Phi \vee \neg\Psi)\,;\, \neg(\Phi \vee \Psi) \Leftrightarrow (\neg\Phi \wedge \neg\Psi)]$ and Quantifier Negation $[\neg(\forall\alpha)\Phi \Leftrightarrow (\exists\alpha)\neg\Phi\,;\, \neg(\exists\alpha)\Phi \Leftrightarrow (\forall\alpha)\neg\Phi]$. If, in the course of evaluating any universally quantified formula, a member of the domain set is found to make the quantified formula false, then this member serves as a *counter-example*. Likewise, a member which makes an existentially quantified formula true serves as a *witness*. Because quantifiers can be nested, such a witness or counter-example of a nested quantifier must be reported along with any bindings in effect from parent quantifiers for it to be meaningful.

4.2 Algorithms for ordering log entries

4.2.1 The graph of time

The hash chains in secure logs provide irrefutable evidence of order, so we turn now to the problem of determining the temporal relationship between any pair of entries in the log.

Determining order 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. (A corollary of the "happened before" relationship is potential causality: A may have affected B [17].) If instead a path exists from A to B, then B precedes A in time.

If neither of these directed paths exists, yet A and B are still a member of the same graph of time, they are contemporaneous events. They may not actually have happened simultaneously in "real" time, but the graph of time cannot establish their relative ordering; the best we can do is discover either a common precedent, a common successor, or both.

In a system where concurrent events are impossible, the DAG degenerates to a list and such queries can be made very fast, but any interesting system will allow two events A and B to be contemporaneous and we must perform a more general graph search.

The algorithms we present here represent various points in the time/space efficiency spectrum; space efficiency can be traded for time efficiency in varying degrees when solving this problem. Table 1 summarizes the complexity of our four techniques.

In this analysis we will frequently refer to the following variables: the number of hosts k, the number of log messages in the system n, and the number of edges e. In all cases, we assume the partial ordering of log messages is organized as a DAG representing the graph of time.

4.2.2 Graph search algorithms

Full graph search To determine if A precedes B, we perform a conventional graph search of the DAG starting at B and ending at A. $O(n)$ storage is required for the intermediate state (e.g., the frontier set) and the worst case time complexity is $O(e)$. While either depth-first or breadth-first search will give correct results, if the graph of time is structured as a set of long timelines with few intersections, depth-first search may consume a great deal of time searching the wrong branch for a desired entry.

Full precomputation Warshall's algorithm for all-points shortest paths [31] can be used to calculate the pairwise ordering relationship for all pairs of messages in the DAG. This takes $O(n^3)$ time to compute, and storing this table takes $O(n^2)$ space, but subsequent tests for order reduce to table lookup and take constant time. Note that the table becomes invalid (and must therefore be recomputed) when new log entries are introduced; therefore, this technique is only recommended for static logs.

Memoized search A variant on full graph search, this technique optimizes for situations in which a few entries of interest are compared with many others. When evaluating $A \ll B$, the algorithm traverses the graph from B in search of A. Noting that every node x ever entering the frontier set precedes B, we store the relationship $x \ll B$ for future queries. Depending on the structure of the logs and query order, lookups may now return in constant time, though they still have a worst case running time of $O(e)$ as in conventional graph search.

We now consider a particular subset of all graphs of time: namely, those that comprise a small number of *timelines*. This occurs in distributed systems in which each participant's secure log establishes a total order on its entries. Such a per-host ordering may be efficiently verified by ensuring that each new message from a given host includes the hash of the previous message from the same host.

If this property holds for all hosts in the system, we have the opportunity for some novel optimizations when searching the graph of time. In the following two algorithms, we consider a system with k hosts (and therefore k timelines) and assume that k is substantially less than the total number of log entries n.

Graph search with pruning First, we assume that for each message x, we know its host $h = \mathbf{host}(x)$ and its integer index in that host's log, $\mathbf{idx}(h, x)$. As such, to compute $A \ll B$, we begin as usual by searching the DAG from B with the intent of finding A. Note once again that for each x discovered during this search, $x \ll B$. If any message x is discovered such that $\mathbf{host}(A) = \mathbf{host}(x) = h$, we can then take advantage of the total ordering of the messages from h and compare $\mathbf{idx}(h, A)$ and $\mathbf{idx}(h, x)$:

- If $\mathbf{idx}(h, A) < \mathbf{idx}(h, x)$, then A precedes x in h's log. Because $x \ll B$, we conclude $A \ll B$ and terminate the search.
- If $\mathbf{idx}(h, A) = \mathbf{idx}(h, x)$, then $x = A$ because messages on h are totally ordered. Therefore, we terminate, having shown $A \ll B$.
- If $\mathbf{idx}(h, A) > \mathbf{idx}(h, x)$, x is older than A. We have not yet shown definitively whether $A \ll B$; a path might still exist from B to A. What we *have* shown is that, because x precedes A and time is acyclic, x cannot be on any such path. We may therefore *prune* this part of the search tree and continue searching from other nodes in the frontier set.

In the worst case, no vertices are pruned, resulting in the same space and time complexity as graph search, but in graphs of time that result from very dense entanglement and few extended divergences, most search paths are pruned very quickly.

Precompute with timelines While a full precomputation of the pairwise ordering relationship consumes $O(n^2)$ space, some of this information is redundant when we assume that the DAG is composed of timelines. If we know that $A \ll B$ for some entries A and B, we know also that A precedes all subsequent entries in B's timeline.

This means that for every entry x, on every host h there exists a unique index $i \in [0, \infty]$ such that, for any y in h's timeline, if $i \leq \mathbf{idx}(h, y)$ then $x \ll y$. (An index of ∞ indicates no such entry exists.) This index is the least upper bound of the *projection* of x onto the timeline of host h; that is, it is the "latest" that it may have happened from the perspective of h.

We can precompute the full table $\mathbf{P}$ of these projections: $\mathbf{P}(x, h) \leftarrow i$. Using this table, $x \ll y$ is true when $\mathbf{P}(x, h) \leq \mathbf{idx}(h, y)$, where $h = \mathbf{host}(y)$. Lookups in this table cost $O(1)$ and storage is $O(k \cdot n)$.

We now offer two approaches to pre-computing the table $\mathbf{P}$ such that it maintains the invariant that $\mathbf{P}(x, h) \leq \mathbf{P}(y, h)$ for all $x \ll y$.

Naïve computation of P. Our first approach exploits the fact that $\mathbf{P}(x, h) \leq \mathbf{idx}(h, y)$ iff $x \ll y$ and $h = \mathbf{host}(y)$. We begin by initializing $\mathbf{P}(y, h) \leftarrow \infty$ for every y on every host h; this will be an initial estimate which we will relax during the precomputation. For every entry y at index $i = \mathbf{idx}(h, y)$ on its own timeline, we perform a DFS through the entire subgraph of preceding events x reachable from y. Each $x \ll y$, so we relax our estimate: $\mathbf{P}(x, h) \leftarrow \min(\mathbf{P}(x, h), i)$. Once we have performed this search for every y on host h, $\mathbf{P}(x, h)$ will be equal to the smallest $\mathbf{idx}(h, y)$. The cost of this precomputation is $O(n + n \cdot e) = O(n \cdot e)$.

Improved computation of P. This technique operates similarly to the naïve approach, but reduces the cost of precomputation to $O(k \cdot e)$ by exploiting two insights:

- By iterating over all messages for each host in timeline order—that is, in order of *increasing* $\mathbf{idx}(h, y)$—we avoid having to relax our estimate more than once: $\mathbf{P}(x, h)$ will be set to its lowest possible value immediately.
- We can prune nodes during the traversal when they cannot cause any further updates to $\mathbf{P}$. This happens whenever $\mathbf{P}(x, h) \leq \mathbf{idx}(h, y)$; that is, we encounter a region of the graph that we have already identified as projecting to a point on the timeline of h that already precedes y.

Using this algorithm, each $\mathbf{P}(x, h)$ entry is only updated once. For each host h, then, each edge in the graph of time must therefore be followed exactly once; any attempt to follow an edge $x \rightarrow x'$ a second time would result in relaxing $\mathbf{P}(x, h)$ a second time. Therefore, the total number of edges visited per host is $O(e)$, and the total precomputation cost is $O(n + k \cdot e) = O(k \cdot e)$.

4.3 Incremental verification

An application may wish to use a rule evaluator not just as a *post facto* auditing tool but as a runtime watchdog. Such usage necessitates applying the verification engine "online," on a growing log created by a running application. As an example, a supervisor console for a network of electronic voting machines could use incremental rule verification on its event log while the election is ongoing. A runtime rule violation might be cause to alert a poll worker to take the offending machine out of service for examination.

Unfortunately, so far we have only described algorithms for a verifier that operates "offline": all at once, on a log considered to be complete. A straightforward approach to online verification

is to periodically re-start the verifier from scratch, using as input the entirety of the log so far. This approach can prove quite costly; each time the verification process is begun, it will re-consider earlier entries that are unchanged from the last run. Clearly, a great deal of redundant computation may result, which, given complex rules involving multiple nested quantifications, will be polynomial in the number of log entries. If we avoid this performance penalty by evaluating rules more rarely, however, we weaken our ability to detect constraints violations as they occur.

4.3.1 Limitations of all-at-once evaluation

There is a more subtle limitation here: A post facto verifier makes no accommodation for potential future changes to the log. In particular, it cannot distinguish the difference between the case where an entry is missing from a partial log because it hasn't yet appeared and the case where it never will. Let us consider, again in the electronic voting context, the following constraint:

$$(\forall b \in L)\ (\ b.\textsc{cast\_ballot} \neq \varepsilon\ )\ \rightarrow (\exists z \in L)\ (\ z.\textsc{polls\_closed} \neq \varepsilon\ \wedge\ b \ll z\ )$$

This rule confirms that all ballots were cast before the polls closed on election day (and were not added later). The basic verifier, presented with an incomplete log (that is, one in which the polls-closed message has not yet appeared), will erroneously deduce a violation.

It is tempting to sidestep this particular complication by taking the additional manual step of segregating our rules into two categories: those that may be evaluated at any time, and those (such as the foregoing) that must wait until the log is complete. But this is unsatisfying; certainly, we would prefer a unified approach to rules and queries. Moreover, the verifier has enough information to know when a part of the evaluation is final and when it is not; it should be able to produce definitive answers as soon as they are available, rather than blindly waiting until the log is complete for these tricky cases.

4.3.2 Performing partial evaluation

Recall that $L \subset \mathbf{T}^*$ is the set of all log messages under scrutiny. We observe that L grows monotonically—log entries, once added, are never taken away from the log. (If this were possible, any secure logging scheme would be faulty.) For this reason, we term $S \subseteq L$ to be *open* if new elements may appear later, and *closed* if S contains all the elements that it will ever contain. The evaluation of a quantification over a closed S, then, behaves precisely as before. The evaluation of a quantification over an open S, however, behaves differently. In the case where, while evaluating $(\exists \alpha \in S)\Phi$, a witness is not found, it cannot be assumed that one will never be found. Likewise, in the case where, while evaluating $(\forall \alpha \in S)\Phi$, a counter-example is not found, it cannot be assumed that one will never be found.

To represent this, we say that any particular evaluation which involves quantification over an open set 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 evaluated, no computation will be repeated in the search for truth.

In the case where no witness is found for an existential quantifier over an open set, the reduction returned will only represent the computation *yet to be done* (i.e., the evaluation of the governed formula in the case where the variable is bound to any *future* set member.) In the case where no counter-example is found for a universal quantifier over an open set, the reduction returned will similarly only represent computation regarding future entries. In all other cases, evaluation of quantification over an open set behaves exactly as it does over a closed set (e.g., if a witness is found in an existential over an open set, there is no reason not to evaluate this as true, even

though the set is open). Now, rather than $O(n \cdot c)$ for each quantifier evaluation (given cost c for the quantified expression), this becomes $O(c)$ per incremental evaluation in the worst case.

Because we've introduced these reductions as placeholders for truth values, we must adjust how truth-functions and quantifiers deal with their sub-formulæevaluating to reductions (i.e., the cases where truth-functions govern quantifiers or when quantifiers are nested). For the truth-functions, we convert the truth tables used for the evaluation of each truth function to the corresponding truth table which a three-valued system (where the third value is the unknown value) would use. In the case where the evaluation of some truth-function which connects two formulæcannot be known because reductions are evaluated from either or both of the connected formulæ, these reductions are connected using this truth-function and returned. Similarly, the reductions which are returned by the evaluation of nested quantifiers will remember which of its evaluations returned reductions, and therefore need to be reevaluated. Concisely stated, our incremental evaluation only saves work on the inner-most nested quantifier which governs the open set in question. In the case where d quantifiers are nested over the same open set, the run time for each incremental evaluation is $O(n^{d-1} \cdot c)$ per incremental evaluation in the worst case, with $O(n^{d-1})$ space needed to save the reduction.

4.4 Related Algorithms

Secure logs are a finite object and audit rules are assertions over those finite objects. The simplicity and declarative nature of these assertions make them amenable to automatic checking by theorem prover systems, logical programming systems, deductive database systems, and relational database systems. These systems can all potentially check audit rules.

We did not consider using theorem proving systems because of the quantity of log entries and the simplicity of first-order audit rules. Model checking [6] tools help in verifying that a finite implementation or design satisfies a finite specification, but can not prove that a remote participant, with an unknown implementation, is not experiencing byzantine failure [30]. Validation is a structured way of testing an implementation for required properties [10], but it is not intended to handle situations where a system and its co-located testing infrastructure experience byzantine failures.

Prolog [25] is a Turing-complete logical programming language that performs computation as a side effect of proof search [7] to satisfy a goal clause. Being Turing-complete, it can implement much more sophisticated audit rules than first order logic at the cost of being able to express undecidable audit rules.

Several deductive database systems featuring recursion and supporting a decidable subset of Prolog have been envisioned [1]. Datalog¬ can be evaluated in polynomial time, and by supporting recursion it is expressive enough to directly implement many complex functions (such as the matching and ordering relations defined in Section 3) while also supporting more expressive audit rules than first order logic. Evaluation strategies include top-down approaches [22] and bottom-up approaches [2].

A standard relational database may be used to store log entries and check audit rules. Audit rules represented in first order logic may be directly translated into the relational calculus [5] or standard SQL. Current production SQL languages do not allow recursive queries, so the "precedes" relation must be implemented with one of the precomputation algorithms described in Section 4.2.

The relational calculus has a deep theoretical basis, and is essentially first order logic without function symbols [1]. Because of its widespread use, database query optimization is well-studied [9, 32]. For example, merge-join may be used in evaluating queries such as $(\forall A)(\forall B)(A.x = B.x) \wedge \Phi$, where instead of an $O(n^2)$ iteration over all A and B tuples, it instead sorts A and B based on x, and merges the the two sorted lists in time $O(2n \log n + R)$ where n is the number of tuples

and R is the size of the result set. A second optimization includes the use of indices, which are secondary data structures that record, for each value of a particular field, which records have that value. In queries such as $(\forall A)(\exists B)(A.x = B.x) \Rightarrow \Phi$, indexing B on x permits quickly finding the matching B entries for any A.

At a high level, database optimization involves query rewriting strategies that convert a query into a semantically equivalent one with higher performance. Query rewriting strategies apply to the correlated and nested queries that would occur in SQL translations of nested quantification [16, 8, 23]. Query rewriting also attempts to find strategies permitting the use of the two low-level optimizations described above.

Two databases problems are similar to the problem of incremental verification. The first is the *materialized view maintenance* problem, which deals with efficient algorithms for recomputing the results of a static query on a changing dataset [11]. If an audit rule is translated into a query that generates counterexamples, then checking the audit rule is equivalent to determining if the resulting view is empty. Complexity classes for the costs of incremental materialized view maintenance have been defined [15]. Approaches to incremental view maintenance exist [12], but would be ineffective for the types of log auditing queries we expect to write because of correlated nested subqueries. These occur when the predicate in an inner quantifier contains references bound in the outer quantifier and so cannot be evaluated independently of the outer quantifier, such as when finding counterexamples to queries like $(\forall A)\,(\exists B)\,\Phi$.

The second related problem is integrity maintenance. Database implementations have mechanisms to prevent semantically invalid data from being inserted through the use of constraints. Integrity constraints differ from view maintenance in that integrity constraints are assumed to always hold; the system disallows the addition of any violating entries. Many audit rules can be translated by an algorithm by Bernstein and Blaustein permitting the automatic checking of a large class of constraints in $O(\log n)$ time per update [3].

5 Implementation: Querifier

5.1 Introduction

Owing to our need, discussed in Section 4.3, for an online verifier that can be embedded into an application, we eschewed solutions involving dedicated relational database engines or theorem provers. Instead, we developed Querifier, a log verification tool comprising approximately 2700 semicolons of Java source (including tests and performance measurement code). Its key advances are:

Expressivity. Any rule expressible in the predicate logic of Section 3 is also directly expressible in the rule language understood by Querifier; converting from one to the other is a straightforward syntactic transformation (from infix logical connectives to prefix S-expression notation).

Incremental evaluation. Querifier implements the incremental evaluation algorithm described in Section 4.3, and is therefore able to offer partial results with low overhead per query.

5.2 Operation

The structure of Querifier is summarized in Figure 1. The core Querifier implementation comprises a rule parser and evaluator. Rules and log data are represented in a format based on Rivest's canonical S-expression encoding [24]. We chose this format due to its compact representation, low scanning complexity, and ability to encode arbitrary recursive data structures (particularly tuples, as defined in Section 3). The canonical encoding of an S-expression is unambiguous and therefore

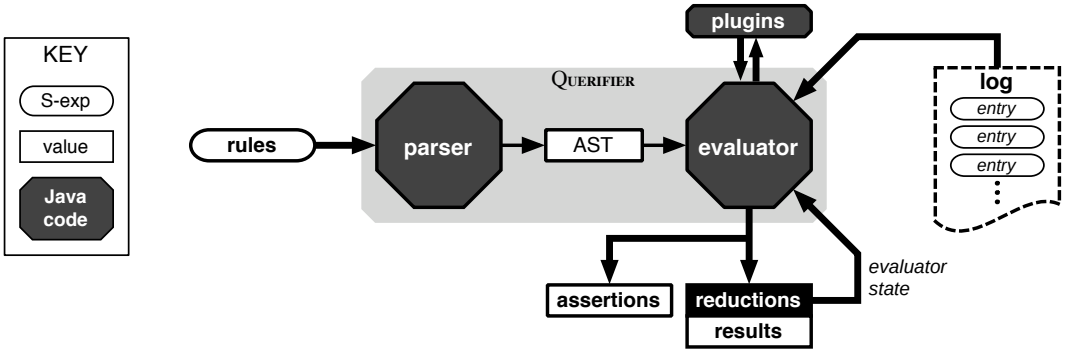

Figure 1: QUERIFIER components and operation. Applications supply rules in the form of S-expressions; the verifier parses rule expressions into an AST suitable for evaluation. Log entries, also S-exps, are fed to the verifier, which recursively interprets the AST for each, finally yielding a result value 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.

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

```
(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: 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.

suitable for digital signatures and hashing. It is straightforward to marshal structured log data into and out of S-expressions, and applications may even choose to directly use S-expressions when representing secure logs.

Clients of QUERIFIER initially supply *rules* in the form of a single S-expression representing a sentence in predicate logic. The sentence can be arbitrarily complex; typically, sets of rules are conjoined to form a single rule expression. The complete grammar is omitted here for space, but an example transformation is given in Figure 2. Rules are parsed, according to the grammar, into an abstract syntax tree (AST).

Applications can also supply *plugins*, small pieces of Java code that define additional functions for use in rules (see Section 3.1). A common function of all plugins is to identify which portions of each log message correspond to hash chain pointers so that the DAG of time may be constructed and the "precedes" operation can be computed during rule evaluation.

Thus initialized, QUERIFIER is ready to consume log data (as S-expressions) and generate results. When invoked, the rule *interpreter* recursively navigates the AST, using the algorithms described in Section 4 to compute a result value for the rule.

While log data is still being introduced by the application, the log is considered "open" and some quantifiers may return, instead of truth values, *reductions* as defined in Section 4.3. These objects contain sufficient state to resume computation at any time without duplicating effort. The log is eventually "closed" by the application, signaling to QUERIFIER that no more log data will appear; from this point forward, the final verification result is computable and no reductions will ever be returned.

In the next section we present our early performance evaluation of QUERIFIER on realistic data and rules. Our experiments cover variations on the evaluator implementation, including all-at-

once versus incremental evaluation, as well as several graph search algorithms for determining order.

6 Evaluation

6.1 Application: Electronic voting

Our discussion has made frequent reference to electronic voting, a problem domain in which secure log auditing is both necessary and complex. Even assuming correct voting system implementations, any amount of poor record-keeping or procedural errors can easily cast doubt on the correctness of the final tally. Electronic voting systems in use today create audit logs that would in theory be useful for forensic analysis in this situation. In practice they prove useless: they offer no security features of any kind; their time-stamps are derived from each machine's hardware clock (akin to that found on a consumer PC, and just as likely to be set incorrectly); and, critically, each machine is exclusively trusted to preserve its own logs.

In prior work [26], we have proposed that all voting machines in each polling place keep tamper-evident logs, exchanging events in those logs with one another to form an entangled record. The result is a believable global picture of all critical election events. In particular, for any ballot, it should be possible to prove that it was authorized by a poll worker and cast by a voter while the polls were open on election day. These are application-specific rules that are a natural fit for the predicate logic we propose here, and as such we use this system design as a framework for our evaluation.

6.2 Experimental setup

We evaluated the performance of Querifier on a synthetic election log and a set of rules that we believe to be representative of a realistic deployment of the Auditorium polling place [26].

Voting simulation. 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 Sandler and Wallach's VoteBox 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 Auditorium voting messages.
2. There exists a polls-open message beginning the election.
3. There exists a polls-closed message 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.

Other properties and constraints may be of interest, but this set is both interesting and meaningful (affirming the essential ingredients of a valid election) and will suffice for our evaluation.

Rule 1, expressed in predicate logic, is quite simple:

$$(\forall x \in L)\ (a.\textsc{auditorium\_message} \neq \varepsilon)$$

Rules 5, 6, and 7 may be represented as three separate logical expressions or combined into the expression:

$$\begin{aligned}
&(\forall b \in L)\ (b.\textsc{cast_ballot} \neq \varepsilon) \rightarrow (\\
&\quad (\exists a \in L)\ (\ a.\textsc{vote_auth} \neq \varepsilon \ \wedge\ a.\textsc{auth_nonce} = b.\textsc{ballot_nonce} \ \wedge\ a \ll b\ )\\
&\wedge\ (\exists r \in L)\ (\ r.\textsc{vote_receipt} \neq \varepsilon \ \wedge\ r.\textsc{receipt_nonce} = b.\textsc{ballot_nonce} \ \wedge\ b \ll r\ )\\
&\wedge\ \neg(\exists x \in L)\ (\ x.\textsc{cast_ballot} \neq \varepsilon \ \wedge\ x.\textsc{ballot_nonce} = b.\textsc{ballot_nonce} \ \wedge\ x \neq b\ )\ )
\end{aligned}$$

A straightforward translation from the above to S-expression syntax (as described in Section 5) yields rules that Querifier can directly evaluate. Note that the maximum nesting depth of quantification is 2; by our reasoning in Section 4, a naïve implementation will perform $O(n^2)$ computations.

Equipment. The machine running Querifier was a lightly-loaded dual 2 GHz PowerPC G5 workstation (Mac OS X 10.4) with 4 GB RAM; the Java runtime in use was the Sun Java HotSpot Client VM (1.5.0).

6.3 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). We are interested, however, in a solution that can be used many times during our hypothetical election, because we would like to know about obvious violations within a short amount of time (that is, before the end of the day).

A naïve approach is to re-run Querifier anew after every message. The cost, of course, is unacceptable: the last run takes the same 220 seconds, resulting in an overall computation time of just over 30 thousand seconds, or about 8 ½ hours—longer than the election itself!

Much of this computation is redundant, and as described in Section 4.3, an incremental approach is vastly preferable. To demonstrate this, we simulated an online verification scenario with a persistent instance of incremental Querifier using reduction-based incremental evaluation. We fed entries from our synthetic log to this instance with different batch sizes ranging from one (invoking the verifier for every entry) through the entire log (essentially the offline verification case). For comparison, we also attempted to use a non-incremental Querifier with each partial log at the same event intervals (simulating the effect of using an offline, all-at-once verifier in an online context). Figure 3 shows the dramatic difference between the performance of full vs. incremental evaluation at various intervals.

Graph search. We also compared three of our algorithms for computing order between entries in the graph of time: BFS, memoized BFS, and BFS with timeline pruning (see Section 4.2). 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 4.

The memoized search algorithm can be seen to perform substantially better when fed large amounts of new log data at once; this is because the first precedence test in the new dataset frequently ends up pre-computing other ordering relationships that will soon be requested. Pruned

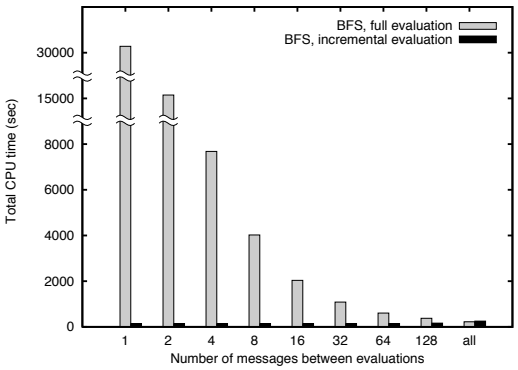

Figure 3: Incremental evaluation. Bars indicate total time to consume and evaluate the entire log. The rightmost bar represents an interval equal to the length of the input, in which case the two approaches are equivalent; as the intervals get shorter, the costs of re-verifying from scratch become obvious.

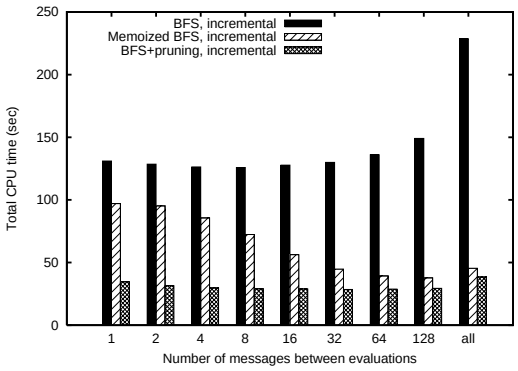

Figure 4: Graph search. Incremental verification performance across several graph search algorithms (described in in Section 4.2). As in Figure 3, we re-run the verifier at various intervals to quantify the overhead associated with each invocation. Bars indicate total time to verify the entire log.

BFS performs better still, and does so consistently for any size of input data; we attribute this to the few (9) distinct timelines that make up the overall graph of time, and systems with a greater number of participants will generate logs that cannot be pruned as quickly. (Note that if there is no total ordering of each participant's messages, this algorithm is not applicable.)

Summary of implementations. We use a log scale in Figure 5 to collapse the many orders of magnitude separating these implementation variants. Optimized graph search algorithms improve upon all-at-once verification, but cannot fundamentally change the problem's complexity as our reduction-based incremental approach does. Finally, note that at an increment of 763 events (the full log), incremental and full verification have almost identical performance (there is a small amount of overhead involved in the additional data structures necessary for incremental operation, though they are hardly used). These evaluations show that optimization in the verifier can yield excellent gains, to the point that the entire log in our example voting scenario can be processed in about 30 total CPU seconds (0.04s per entry).

Ruleset comparison. Returning to our voting scenario, the only nodes that need perform "online" verification are the supervisors, and they can be provisioned appropriately. If performance is an issue, the verification interval can be increased and results obtained less frequently. Likewise, not every rule must necessarily be evaluated every time a new message arrives. We compare the performance of two smaller rule sets to our full rule set in Figure 6. Finally, the algorithmic complexity of the "online" ruleset can be reduced (for example, to singly-nested quantification) so that some simpler rules (for example, "has the election begun?"), allowing simple verifications to be executed even on underpowered devices or devices with much higher rates of log activity.

7 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 contributes the following advances in this domain:

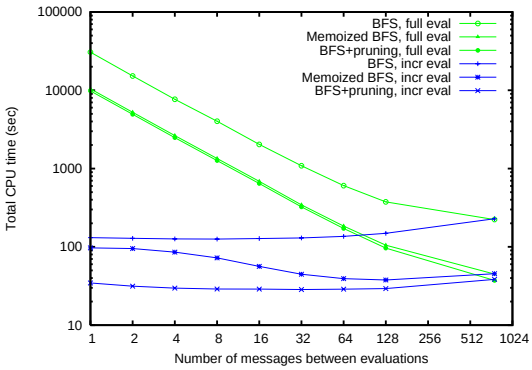

Figure 5: Summary of implementations. As before, when the batch size equals the entire log, full and incremental evaluation are equivalent, but when results are requested after every entry, incremental evaluation is two orders of magnitude less expensive than full evaluation.

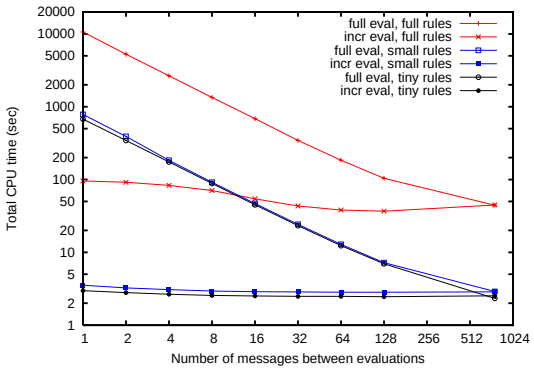

Figure 6: Ruleset comparison. The "full" set of rules is the same as in earlier figures. The "small" set includes only rules 1–4; the "tiny" set comprises 1 and 2. (All evaluators used the memoized graph search algorithm for precedence queries.)

- An exploration of using **predicate logic** with pattern-matching to express properties over secure logs of arbitrary shape and complexity
- Consideration of the **complexity** of interpreting these logical statements, including several useful **algorithms** for incremental operation and graph-of-time search
- A concrete **implementation** of a rule verifier that applies the above advances to logs and rules from

In particular, we have found Querifier to be an essential 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, unambiguous 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. The optimizations described in Section 4.4 are applicable to certain subsets of the entire space of possible logical statements, 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] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *PODS '86: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–15, Cambridge, MA, 1986.

[3] P. A. Bernstein and B. T. Blaustein. Fast methods for testing quantified relational calculus assertions. In *SIGMOD '82: Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data*, pages 39–50, Orlando, Florida, 1982.

[4] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 173–186, New Orleans, LA, Feb. 1999.

[5] E. F. Codd. Data models in database management. In *Proceedings of the 1980 Workshop on Data Abstraction, Databases and Conceptual Modeling*, pages 112–114, Pingree Park, Colorado, 1980.

[6] J. Edmund M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[7] M. H. V. Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, 1976.

[8] R. A. Ganski and H. K. T. Wong. Optimization of nested SQL queries revisited. *SIGMOD Rec.*, 16(3):23–33, 1987.

[9] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, 1993.

[10] J. Grundy, G. Ding, and J. Hosking. Deployed software component testing using dynamic validation agents. *J. Syst. Softw.*, 74(1):5–14, 2005.

[11] A. Gupta and I. S. Mumick. Maintenance of materialized views: problems, techniques, and applications. *Materialized views: techniques, implementations, and applications*, pages 145–157, 1999.

[12] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 157–166, Washington, D.C., 1993.

[13] S. Haber and W. S. Stornetta. How to time-stamp a digital document. In *CRYPTO '90: Proceedings of the 10th Annual International Cryptology Conference on Advances in Cryptology*, pages 437–455, Santa Barbara, CA, 1991. Springer-Verlag.

[14] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, Oct. 2007.

[15] H. V. Jagadish, I. S. Mumick, and A. Silberschatz. View maintenance issues for the chronicle data model (extended abstract). In *PODS '95: Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 113–124, San Jose, California, 1995.

[16] W. Kim. On optimizing an SQL-like nested query. *ACM Trans. Database Syst.*, 7(3):443–469, 1982.

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

[18] P. Maniatis. *Historic Integrity in Distributed Systems*. PhD thesis, Stanford University, Stanford, CA, Aug. 2003.

[19] P. Maniatis and M. Baker. Secure history preservation through timeline entanglement. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, Aug. 2002.

[20] R. C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO '87: Proceedings on Advances in Cryptology*, pages 369–378, 1988.

[21] M. Naor and K. Nissim. Certificate revocation and certificate update. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, Jan. 1998.

[22] T. C. Przymusinski. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5(2):167–205, 1989.

[23] J. Rao and K. A. Ross. Reusing invariants: a new strategy for correlated queries. *SIGMOD Rec.*, 27(2):37–48, 1998.

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

[25] P. Roussel. *Prolog: Manuel de Reference et d'Utilisation.* University d'Aix Marseille, Marseille, France, 1975.

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

[27] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. In *USENIX Security Symposium*, pages 53–62, San Antonio, TX, Jan. 1998.

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

[29] Securities and Exchange Commission, Washington, D.C. *SEC charges four brokers and day trader in fradulent "squawk box" scheme*, Aug. 2005. http://www.sec.gov/news/press/2005-114.htm.

[30] W. Visser, K. Havelund, G. Brat, and S.-J. Park. Model checking programs. In *Proc. of the 15th IEEE International Conference on Automated Software Engineering*, 2000.

[31] S. Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, 1962.

[32] K. Youssefi and E. Wong. Query processing in a relational database management system. In *VLDB 1979: Proceedings of the Fifth International Conference on Very Large Data Bases*, pages 409–417, Rio de Janeiro, Brazil, 1979.

[33] A. R. Yumerefendi and J. S. Chase. Strong accountability for network storage. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST '07)*, pages 77–92, San Jose, CA, Feb. 2007.