Authenticating Operation-based History in Collaborative Systems

Hien Thi Thu Truong
INRIA Nancy-Grand Est
615 rue du Jardin Botanique
54600 Villers-lès-Nancy,
France
hien.truong@inria.fr

Claudia-Lavinia Ignat
INRIA Nancy-Grand Est
615 rue du Jardin Botanique
54600 Villers-lès-Nancy,
France
claudia.ignat@inria.fr

Pascal Molli
University of Nantes
2 rue de la Houssinière,
BP 92208
44322 Nantes cedex 3, France
pascal.molli@acm.org

ABSTRACT

Within last years multi-synchronous collaborative editing systems became widely used. Multi-synchronous collaboration maintains multiple, simultaneous streams of activity which continually diverge and synchronized. These streams of activity are represented by means of logs of operations, i.e. user modifications. A malicious user might tamper his log of operations. At the moment of synchronization with other streams, the tampered log might generate wrong results. In this paper, we propose a solution relying on hash-chain based authenticators for authenticating logs that ensure the authenticity, the integrity of logs, and the user accountability. We present algorithms to construct authenticators and verify logs. We prove their correctness and provide theoretical and practical evaluations.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: [Authentication]; H.5.3 [**Group and Organization Interfaces**]: [Computer-supported cooperative work]; C.2.4 [**Distributed Systems**]: [Distributed applications]

General Terms

Security

Keywords

authenticating logs, multi-synchronous collaboration, authenticators, optimistic replication, logs, operation-based history

1. INTRODUCTION

Collaboration is a key requirement of teams of individuals working together towards some common goal. In recent years collaborative editing systems such as wikis, GoogleDocs and version control systems became very popular. These systems rely on a multi-synchronous collaboration model [8, 15] that allows users to work simultaneously on shared documents following some cycles of divergence and convergence. Copies of the shared data diverge when users work in isolation and converge later when users synchronize their changes.

GROUP'12, October 27–31, 2012, Sanibel Island, Florida, USA.

The mechanism, that allows replicas to diverge during a time interval ensuring that they will eventually converge at a later time, is called optimistic replication [37]. Optimistic replication can be classified into state-based and operation-based [20]. In state-based replication each site applies updates to its replica without maintaining a change log. Usually in systems adopting state-based replication such as Active Directory in Windows Server and Coda file system [38], every site sends its local state to other sites that can merge the received state with their own states. Systems that use operation-based replication such as Bayou [34], IceCube [18] and GoogleDocs keep modifications performed on a replica in a history which is then sent to other replicas. Operation-based approaches are used when the cost to transfer state is high such as in database systems or mobile systems; and when operation-semantics is important. In this work we target systems that use operation-based replication.

In operation-based replication systems users can misbehave by tampering history for their convenience. For instance, they can remove some content of the history or change the order of some operations from the history. This might be critical for some collaborative systems such as version control systems. It is vitally important to be able to retrieve and run different versions of a software. If the history can be modified, revisions do not correspond to the expected behavior of the software. Moreover, developers cannot be made responsible for the revisions for which they contributed. Furthermore, by modifying the history, a contributor may introduce security holes in the system under the name of another contributor. Therefore, there is a need to ensure integrity of the log, and in case the log was tampered, the misbehaving user should be detected.

Solutions for securing logs can be classified into two main families: non-cryptographic secure logging and cryptographic secure logging. The former approach is based on a secure logging machine such as a write-only medium (e.g CD/DVD), a tamper-resistant-hardware or a trusted hardware to prevent adversary from modifying logs [3]. However, in real-world applications deployed over large scale distributed environments, it is impractical to assume the presence of such devices. The later approach has been investigated deeply with numerously extensive research (namely, [10, 11, 4, 5, 13, 14, 25, 27, 39, 43]). These existing solutions, however, are adapted only for collaboration based on a single global stream of activity over shared data. For instance, floor control policies [10] and locking mechanisms [11] ensure a single global stream of activity by allowing a single user at a time to access objects in the shared workspace.

Multi-synchronous collaboration abandons constructing a single stream of activity out of the history of all user activities. Instead, it maintains multiple, simultaneous streams of activity, and then manages divergence between these streams. Each user maintains therefore different streams of the global history containing activity

of all users. Throughout this paper we call logs standing for these different streams of activity. The main challenge that we address in this paper is how to secure logs in the multi-synchronous collaboration. To our best knowledge, no existing work addressed this issue.

In this paper, we propose a solution relying on hash-chain based authenticators for securing logs in multi-synchronous collaboration. The proposed authenticators ensure the authenticity and the integrity of the logs, i.e. any log tampering is detectable. Moreover, the proposed authenticators provide user accountability, i.e. any user can be made accountable of her misbehavior on log tampering.

The paper is structured as follows. We start with a context for our work in Section 2. We then go on by presenting in Section 3 a threat model and desirable properties our proposal tackles. We next describe in Section 4 our proposed approach based on authenticators including their definitions and examples of how they are constructed. In Section 5, we present our algorithms to construct authenticators and verify logs, and proofs of their correctness. We provide an evaluation with real collaboration histories from projects using Mercurial and an analysis showing the feasibility of our proposal in Section 6. We give an overview of related works in Section 7. In Section 8, we end the paper with some concluding remarks.

2. CONTEXT

Push-Pull-Clone (PPC) is the most general paradigm supporting multi-synchronous collaboration. Users work simultaneously on different streams of activity on the shared data. In the PPC model, users replicate shared data, modify it and redistribute modified versions of this data by using the primitives push, pull and clone. These primitives are used for managing divergence and convergence of different streams of activity. To start, users clone shared data and maintain in a local workspace this data as well as modifications done on this data. Users can then push their changes to different channels at any time they want, and other users that have granted rights may pull these changes from these channels. By using pull primitives, replicas are synchronized. In Figure 1, an instantiation of the PPC model with three users is illustrated. In this figure, *user 1* and *user 2* interact with each other by using push and pull primitives, while *user 3* performs a clone from *user 2*. The PPC collaboration model is very widely used in distributed version control systems such as Git, Mercurial and Darcs. It is a very general collaboration model without a collaboration provider where users share their data only with people whom they trust. PPC model generalizes the collaboration model with a service provider where users interact only with a server that forwards afterward the changes to the other users.

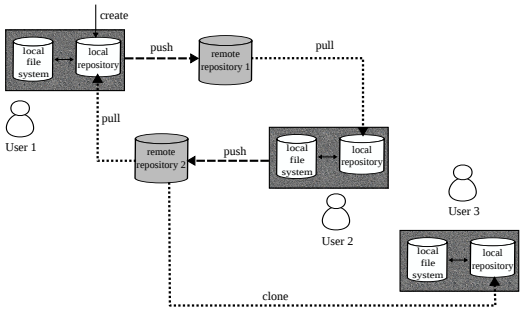

Figure 1: Push-Pull-Clone paradigm

We consider the operation-based collaboration where user changes are kept in a log that is then sent to the other users and logs are merged at synchronizations. We consider a system with a number of sites which can operate independently on replicas of shared documents. Each site keeps the shared document as a log of operations that have been performed during a collaborative process, $L = [op_1, op_2, ..., op_n]$. Each operation is parameterized depending on an application domain. A shared document can be as large as a database (i.e. Bayou) or as small as a single file. Operations can be treated at different granularities ranging from characters, lines or paragraphs in a document to deltas between revisions. A document is created at one site and replicated to other sites by means of push and clone primitives. Sites store operations in their logs in an order that is consistent with the order they were generated.

The order of addition of operations in the log is compatible with the "happened-before" relation between operations [19]. We say that op_a *happened-before* op_b, denoted as $op_a \rightarrow op_b$, if op_b was generated on some site after op_a was either generated or received by that site. The "happened-before" relation is transitive, irreflexive and antisymmetric. Two operations op_a and op_b are said concurrent if neither $op_a \rightarrow op_b$ nor $op_b \rightarrow op_a$.

Changes on the shared document made by users are propagated in weakly consistent manner from one site to another site. Users decide by means of primitives push and pull when, with whom and what data to be sent and synchronized. When a pull is performed by *user 1* from the channel where *user 2* pushed his changes, in order to minimize traffic overhead, an anti-entropy mechanism is used [6]. Only the part of the log of *user 2*, that is new to *user 1* since the last time that two users synchronized, is sent to *user 1*. The remote log from *user 2* is synchronized with the local log of *user 1*. The synchronization mechanism requires to detect the concurrency and happened-before order between changes of different sites. Also the conflicts between concurrent changes must be resolved. Replicas are consistent if their states are identical when they have applied the same set of operations. For our approach, we use the CRDT family of algorithms [36, 42] which design operations to be commutative from the start. When reconciliation is performed, operations from the remote log, that have not been previously integrated into the local log, are simply appended to the end of the local log. The log propagation mechanism uses anti-entropy which preserves happened-before order between operations. Therefore, the reconciliation mechanism ensures happened-before order between operations as well as it allows concurrent operations to appear in logs in variant orders.

We define a *partially ordered set* (poset) $H = (P, \rightarrow)$ where P is a ground set of operations and "$\rightarrow$" is the happened-before relation between two operations of P, in which "$\rightarrow$" is irreflexive and transitive. We call H as an operation-based history in our context. Given a partial order "$\rightarrow$" over a poset H, we can extend it to a total order "$<_t$" with which "$<_t$" is a linear order and for every x and y in H, if $x \rightarrow y$ then $x <_t y$. A linear extension L of H is a relation $(P, <_t)$ such that: (1) for all op_1, op_2 in P, either $op_1 <_t op_2$ or $op_2 <_t op_1$; and (2) if $op_1 \rightarrow op_2$ then $op_1 <_t op_2$. This total order preserves the order of operations from a partial order set H to the linear extensions on the same ground set P.

We call these linear extensions as individual logs observed by different sites. The Figure 2 shows an example of a history and its linear extensions.

In collaborative systems, where multiple sites collaborate on the same shared data object, we can consider that the global stream of activity of all sites is defined by a partially ordered set of operations. Each site, however, can see only operations in his workspace that it generated locally or received from other sites. The site keeps therefore an individual log as a linearization of history built on a subset of a ground set of operations. There are remaining operations

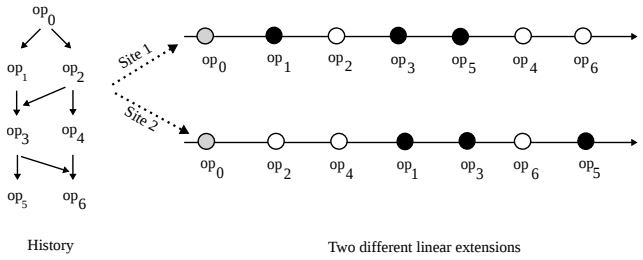

Figure 2: History and its linear extensions

of global history built on entire ground set of operations that are not visible for the site.

3. SECURITY ASPECTS

In operation-based collaborative systems, authentication of data items for collaborative workflows has gained increasing importance. Say for example, if Tom receives a document from Olivia and processes a part of it and then forwards to Pierre, the operation-based document should include the chronological log of actions that each user, Tom, Olivia and Pierre, performed on the document. The correctness of collaboration outcome is based on the trustworthiness of users who maintain the history of versions. Unfortunately, a malicious user can always introduce phony updates to forge history or alter the correct order of versions. This attack raises the threat that honest users might get forged content of shared data. Replicas with corrupted updates might never converge with other valid replicas and this is critical in replication systems. This section presents a threat model followed by desirable properties for dealing with security requirements.

3.1 Threat Model

A threat model, which models the capabilities of attackers, is necessary to analyze the threats that will be addressed by our solution.

There are two types of malicious users: insiders and outsiders. We consider in this paper an inside adversary who has full rights to access a replicated object. Such an adversary might want to alter the history including actions performed on data by authorized contributors. For example, when Tom provides a document to Olivia who can perform and contribute new updates to the document, she should not be able to modify actions that Tom performed which were recorded in log.

We assume users trust each other with their social-based relationship when they start to collaborate. However, trust it not immutable and trusted users once they gained access to the log can always misbehave. Such an active attacker can read, (over)write, delete and change order of log entries. In doing so, an attacker alters existing records or adds forged information to history.

Our work assumes only adversaries who act inside of the system. We cannot prevent outsider attacks where an adversary copies data to create a new document and claims at a later time as being an owner. This might be possible in our system if an adversary removes completely the log of operations, which corresponds to a document removal. We could deal with outsider attacks with the support of a trusted platform, however, we exclude this assumption in our collaborative system.

3.2 Desirable Properties

The following properties are addressed to authenticate operation-based history in collaborative systems.

Integrity. Adversaries are infeasible to forge a log, such as modify its entries or put new forged operations into log, without being detected. Integrity is the most important property required for securing logs in operation-based replication. Ensuring the integrity of a single document can be done easily by using cryptographic signatures or checksums. However, ensuring the integrity of a replicated document represented by a log of operations is more difficult as operations cross multi-contributors and some of them might be adversaries.

Concurrency-collision-freeness. In a history H, some operations might be concurrent, while some others might be in a happened-before relation. If L_i and L_j are different linearizations of the same history H then any authentication mechanism applied to L_i and L_j should yield the same result. The "yielding the same result" is expressed by the concurrency-collision-freeness property: the authentication mechanism holds a function f that $f(L_i) = f(L_j)$ $\forall L_i, L_j \in H$. The "concurrency-collision-freeness" property should be guaranteed in authenticating logs.

In Figure 2, we give an example of a history which is linearized into two logs by two sites. The history H, which is built on the ground set of operations $P = \{op_0, op_1, op_2, op_3, op_4, op_5, op_6\}$ with "$\rightarrow$" relation, is recorded in logs, $L_1 = \{op_0, op_1, op_2, op_3, op_5, op_4, op_6\}$ and $L_2 = \{op_0, op_2, op_4, op_1, op_3, op_6, op_5\}$. They both preserve orders of all operations of the history H. In order to fulfill concurrency-collision-free property, any authentication mechanism applied to L_1 and L_2 should yield the same result. If the authentication results are different, then it means that one of the logs was tampered.

Forward-aggregated authenticity. While logs grow, log verifiers can skip verification of log entries which have been already authenticated. The authentication mechanism should allow accumulation of log verification for a time interval. Not only the integrity of individual log entries but also the integrity of the whole log stream should be preserved. This forward-aggregated authenticity property is similar to forward security and append-only property investigated in many existing works of secure log audit [1, 24, 43].

Public verifiability. This property allows any user in a collaborative system to verify the integrity of logs. Adversaries are made accountable for unauthorized actions. This property can be done by using digital signatures such as RSA or DSA signature scheme. Public verifiability is especially desirable in distributed collaborative systems where logs need to be audited by any collaborator without relying on any trusted central authority.

4. AUTHENTICATORS

In this section, we present our approach to construct authenticators $T_{@site}$ to deter users from log tampering while preserving the above mentioned properties.

When a sending site sends a document to a receiving site, it creates an authenticator for its log. The authenticator is attached to the sent document. The receiving site creates a new authenticator when it receives the document. We assume each site involved in this push-pull communication possesses a cryptographic public/private key pair that is assigned to a unique site identifier and that all users can retrieve the public key of each other. This assumption is reasonable in practice [41, 29]. The private key of the key pair is used to sign entries of log that prevent malicious sites modifying operations on behalf of other sites. Though sites can choose a public key pair on their-own, to limit Sybil attacks [7] we can require that each site possesses a digital certificate from trusted certification authority or has an offline channel (such as email) to identify the owner of public keys. In either case the certification authority plays no role in the process of authenticator creation, and it is used only

during initial phase when a site joins the system. We also use cryptographic hash function with properties collision-resistant (it should be difficult to find two different messages m_1 and m_2 such that $hash(m_1) = hash(m_2)$) and preimage-resistant (with a given hash value h, it should be difficult to find any message m such that $hash(m) = h$). The collision-resistant property can be used to establish the uniqueness of logs at a certain moment when an authenticator is created.

4.1 Definitions

An authenticator is a log tamper-evident which captures a subsequence of operation(s) of a log that were generated in one updating session. An updating session at one user's site is the session between two subsequent push/pull primitives to/from other sites. For example, consider that during a working session, user U generates a log $[op_1, op_2]$ where $op_1 \rightarrow op_2$. When user U pushes his changes, he creates an authenticator for the sequence of operations in the log that their orders should not be tampered by any other user. For instance, a receiver of this log should not be able to re-order op_1 and op_2 to change the happened-before order of op_1 and op_2.

Definition 1. *An authenticator, denoted as $T_{@site}$, is defined as a tuple $\langle ID, SIG, IDE, PRE, SYN\rangle$ where:*

ID: identifier of authenticator which is a tuple $< siteID, opID >$ where siteID is the identifier of the site which creates the authenticator and opID is the operation identifier(s) that the authenticator is linked to;

SIG: the value of signature signed by the private key of the site;

IDE: a list of operation identifiers used to compute SIG;

PRE, SYN: identifiers of preceding and receiving authenticators.

Definition 2. *The SIG of an authenticator $T_{@site}$ at a certain update is computed as a signature of a cumulative hash by a sender S or a receiver R, where the sender computes SIG of the most recent authenticator $T_{m@S}.SIG = \sigma_S(hash(T_{m-1_{@S}}.SIG \,||\, E))$ with condition that $E \neq \varnothing$; and the receiver computes $T_{n@R} = \sigma_R(hash(T_{n-1_{@R}}.SIG \,||\, E \,||\, T_{m@S}.SIG)$ with the condition that there exists new update(s) from S appended to log of R, where:*

$T_{m@S}$: the most recent authenticator committed by sender S;

$T_{n@R}$: the most recent authenticator committed by receiver R;

$T_{m-1_{@S}}$: the preceding authenticator of $T_{m@S}$;

$T_{n-1_{@R}}$: the preceding authenticator of $T_{n@R}$;

$E = [op_{i_1}, op_{i_2}, ..., op_{i_r}]$: subsequent changes generated after preceding authenticator;

$\sigma_{site}(\cdot)$ denotes the signature of site *and $||$ denotes the concatenation of arguments used in hashing, where hashing can be done using any traditional hash function such as SHA-256).*

The structure of an authenticator is illustrated in Figure 3.

When a user shares a document by sending the whole log, she creates an authenticator for log operations computed based on the preceding authenticator and new updated operations. The authenticator is signed by her private key and linked to the last operation of the log. At the receiving site, the receiver performs reconciliation and creates a new authenticator at the reception.

The authenticators of a log of operations are constructed whenever a site sends or receives a new change to/from another site. An authenticator is created in following cases:

- A site sends new changes to other sites. In this case, if a site sends a document without new changes, no new authenticator is needed.
- A site receives new changes from other sites. In this case, the receiving site will check the remote log, detect and resolve conflicts (if there are some conflicts among operations). After these actions, if there are new changes that are added to the receiver's log, a new authenticator is created for this reception.

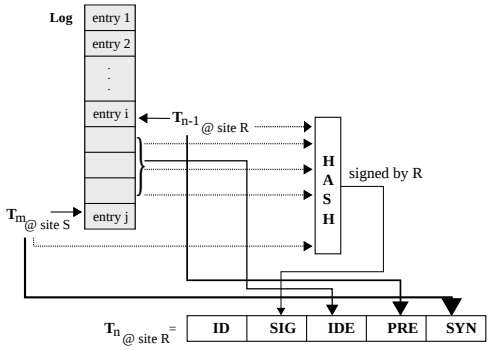

Figure 3: Structure of an authenticator.

4.2 Example

We use the example of history in Figure 2 where two sites collaborate on a shared document having initial version $V_0|\{op_0\}$ to illustrate the construction of authenticators. We assume that the initial version of the document V_0 consisting of operation op_0 was created by some site among collaborating sites. We further assume that all collaborators agreed on this initial version and that the corresponding log of this initial version does not need to be authenticated. Each of the two sites in our example performs parallel contributions based on the initial version of the document. In the example, *site 1* creates the new version $V_1|\{op_0, op_1\}$ and *site 2* creates $V_2|\{op_0, op_2\}$ concurrently. At a later time, *site 1* reconciles with updates from *site 2* and creates the up-to-date version V_3 | $\{op_0, op_1, op_2, op_3\}$. In Figure 4, the two sites, *site 1* and *site 2*, will create authenticators to authenticate their logs each time they do pushing or pulling. In what follows we describe in detail how authenticators are constructed.

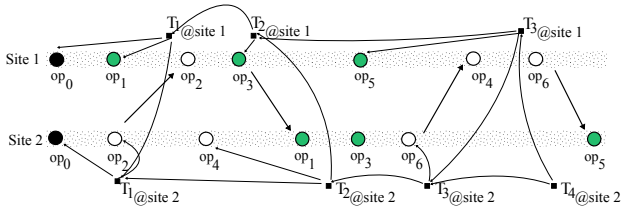

Figure 4: Example of constructing authenticators.

Firstly, when *site 2* pushes his log to *site 1*, it creates an authenticator $T_{1_{@site2}}$ where:

$T_{1_{@site2}}.ID = < site2, op_2 >$ (linked to op_2),

$T_{1_{@site2}}.SIG = \sigma_{site2}(hash(\varnothing \,||\, op_0 \,||\, op_2))$

$T_{1_{@site2}}.IDE = [op_0, op_2]$

$T_{1_{@site2}}.PRE = \varnothing$ (no previous authenticator)

$T_{1_{@site2}}.SYN = \varnothing$ (no received remote authenticator)

When *site 1* pulls changes from *site 2* and receives the log from *site 2*, it creates an authenticator $T_{1_{@site1}}$ where:

$T_{1_{@site1}}.ID = < site1, \{op_1, op_2\} >$ (linked to op_1, op_2),

$T_{1_{@site1}}.SIG = \sigma_{site1}(hash(\varnothing \,||\, op_0 \,||\, op_1 \,||\, T_{1_{@site2}}.SIG))$,

$T_{1_{@site1}}.IDE = [op_0, op_1]$,

$T_{1_{@site1}}.PRE = \varnothing$ (no previous authenticator),

$T_{1_{@site1}}.SYN = T_{1_{@site2}}.ID$

The two sites *site 1* and *site 2* then work concurrently and generate

the new changes op_3 and op_4 respectively. When *site 1* pushes his changes and *site 2* pulls those changes, two authenticators $T_{2_{@site1}}$ and $T_{2_{@site2}}$ are constructed. The structure of $T_{2_{@site1}}$ is given below:

$T_{2_{@site1}}.ID = < site1, op_3 >$ (linked to op_3),
$T_{2_{@site1}}.SIG = \sigma_{site1}(hash(T_{1_{@site1}}.SIG \,||\, op_3))$,
$T_{2_{@site1}}.IDE = [op_3]$,
$T_{2_{@site1}}.PRE = T_{1_{@site1}}.ID$,
$T_{2_{@site1}}.SYN = \varnothing$

Similarly, $T_{2_{@site2}}$ is computed where:

$T_{2_{@site2}}.ID = < site2, \{op_3, op_4\} >$ (linked to op_3, op_4),
$T_{2_{@site2}}.SIG = \sigma_{site2}(hash(T_{1_{@site2}}.SIG \,||\, op_4 \,||\, T_{2_{@site1}}.SIG))$,
$T_{2_{@site2}}.IDE = [op_4]$,
$T_{2_{@site2}}.PRE = T_{1_{@site2}}.ID$,
$T_{2_{@site2}}.SYN = T_{2_{@site1}}.ID$

Again, *site 1* and *site 2* contribute independently to the document, op_5 is generated by *site 1* and op_6 is generated by *site 2*. The two sites then exchange their changes with each other by pushing and pulling other changes. New authenticators are computed at each site. *site 2* computes $T_{3_{@site2}}$ where:

$T_{3_{@site2}}.ID = < site2, op_6 >$ (linked to op_6),
$T_{3_{@site2}}.SIG = \sigma_{site2}(hash(T_{2_{@site2}}.SIG \,||\, op_6))$,
$T_{3_{@site2}}.IDE = [op_6]$,
$T_{3_{@site2}}.PRE = T_{2_{@site2}}.ID$,
$T_{3_{@site2}}.SYN = \varnothing$

site 1 computes $T_{3_{@site1}}$ where:

$T_{3_{@site1}}.ID = < site1, \{op_5, op_6\} >$ (linked to op_5, op_6),
$T_{3_{@site1}}.SIG = \sigma_{site1}(hash(T_{2_{@site1}}.SIG \,||\, op_5 \,||\, T_{3_{@site2}}.SIG))$,
$T_{3_{@site1}}.IDE = [op_5]$,
$T_{3_{@site1}}.PRE = T_{2_{@site1}}.ID$,
$T_{3_{@site1}}.SYN = T_{3_{@site2}}.ID$

In the last step shown in Figure 4, *site 1* pushes his changes to *site 2* and *site 2* pulls these changes from *site 1*. Because there are no new operations since the authenticator $T_{3_{@site1}}$ was created, no new authenticator is created by *site 1*. However, the receiving site *site 2* has to create a new authenticator since a new operation op_5 is added to the log. $T_{4_{@site2}}$ is therefore computed where:

$T_{4_{@site2}}.ID = < site2, \{op_5, op_6\} >$ (linked to op_5, op_6),
$T_{4_{@site2}}.SIG = \sigma_{site2}(hash(T_{3_{@site2}}.SIG \,||\, \varnothing \,||\, T_{3_{@site1}}.SIG))$,
$T_{4_{@site2}}.IDE = []$,
$T_{4_{@site2}}.PRE = T_{3_{@site2}}.ID$,
$T_{4_{@site2}}.SYN = T_{3_{@site1}}.ID$

We will discuss in next section the algorithms supporting the creation and verification of logs as well as prove that the proposed algorithms satisfy desired properties which we mentioned at the beginning of this paper.

5. ALGORITHMS

In this section, we present algorithms to construct authenticators and verify logs based on authenticators. We also provide a proof of correctness of these algorithms.

5.1 Authenticators Construction

Algorithm 1 presents the algorithm for the construction of the authenticator when a sender pushes his changes. An authenticator is computed from the its preceding authenticator and the current generated operations. The algorithm takes as argument the log of the sending site S and generates as output the authenticator computed by the sender site. The condition $E \neq \varnothing$ ensures that an authenticator is created only if the sender has generated new changes; otherwise the sender sends the log without computing a new authenticator.

Algorithm 2 presents the algorithm for the construction of the authenticator when a receiver site pulls changes from a sender site. An authenticator is computed from the preceding authenticator of local log, the current operations generated by the receiver and the most recent authenticator of the remote log. The algorithm takes as arguments the two logs of sending site S and receiving site R and it generates as output the authenticator computed by the receiver site. If there are no new operations sent by the sender that have to be added to the log of the receiver, then the receiver will not compute a new authenticator. Note that in synchronizing logs, authenticators that are linked to operations must be also kept in the local workspace as they authenticate previous operations in the log.

Input: sending site S with its log L_S
Output: $T_{m_{@S}}$
begin
  E ← list of new operations S generates after $T_{m-1_{@S}}$;
  if E ≠ ∅ **then**
    $T_{m_{@S}}$.ID ← <S, identifier of most recently local operation at S>;
    $T_{m_{@S}}$.SIG ← sign(hash($T_{m-1_{@S}}$.SIG ‖ E));
    $T_{m_{@S}}$.IDE ← E;
    $T_{m_{@S}}$.PRE ← $T_{m-1_{@S}}.ID$;
    $T_{m_{@S}}$.SYN ← ∅ ;
  else
    $T_{m_{@S}} \leftarrow \langle\rangle$;
  return $T_{m_{@S}}$;

Algorithm 1: Construct an authenticator for a sender

Input: sending site S, receiving site R and their logs L_S, L_R
Output: $T_{n_{@R}}$
begin
  E ← list of new operations R generates after $T_{n-1_{@R}}$;
  E_S ← list of new operations from L_S added to L_R;
  if $E_S \neq \varnothing$ **then**
    $T_{n_{@R}}$.ID ← <R, $T_{m_{@S}}$.ID.opID ∪ identifier of most recently local operation at R>;
    $T_{n_{@R}}$.SIG ← sign(hash($T_{n-1_{@R}}$.SIG || E || $T_{m_{@S}}$.SIG));
    $T_{n_{@R}}$.IDE ← E;
    $T_{n_{@R}}$.PRE ← $T_{n-1_{@R}}$.ID;
    $T_{n_{@R}}$.SYN ← $T_{m_{@S}}$.ID;
  else
    $T_{n_{@R}} \leftarrow \langle\rangle$;
  return $T_{n_{@R}}$;

Algorithm 2: Construct an authenticator for a receiver

We will consider *time* and *space* complexities of algorithms to construct and verify authenticators. Note that, for the space complexity for verification of authenticators, we exclude the space complexity for maintaining the log. The algorithm to create an authenticator in Algorithms 1 or 2 is $O(1)$ in time, and $O(|\Delta|)$ in storage, where Δ is the set of operations whose identifiers are kept in $T_{@site}.IDE$. Since an authenticator is created each time a site sends or receives changes, the number of authenticators on a replicated object created by site S is the total number of interactions the site has done with other sites. Let Γ be the total number interactions of one site. Then each site needs $O(\Gamma \cdot |\Delta|_{max})$ space for all authenticators, where $|\Delta|_{max}$ is the maximum Δ of all authenticators. In synchronization, one log is updated to become the union of two logs of sites S and R, and the new log shall need $O(\Gamma_S \cdot |\Delta_S|_{max} + \Gamma_R \cdot |\Delta_R|_{max})$ space for all authenticators. We can see that the storage complexity depends on the number of interactions and the number of operations generated by two sites.

5.2 Authenticators-based Log Verification

The Algorithm 3 presents a mechanism to verify log entries based on authenticators. When a site receives a log of operations accompanied by authenticators, it verifies the log based on these authenticators corresponding to entries in the log. The main idea of verification is to check the authenticity of operations preserved by valid authenticators, including checking:

- If authenticators are valid (their signatures are correct). An authenticator is checked by verifying its digital signature using the public key of signer.
- If the log entries are corresponding to these valid authenticators. When an authenticator passes signature checking, the content and the order of operations are taken into account in the verification.

If all of these checkings pass, the log is authenticated. In contrast, a log with either operations not authenticated or authenticated by invalid authenticators is unauthorized. With any detection of the corrupted data or falsified order of changes, authenticators will be not valid and the verification algorithm returns negative result. Authenticators help users being aware of attacks and once the log is unauthorized, the site which sent tampered log is made accountable for the misbehavior.

Input: site R, log L
Output:
1 **begin**
2 Q ← $T_{n_{@R}} \in$ L;
// Q: queue of authenticators to verify
3 verified ← True;
4 **while** Q ≠ ∅ **do**
5 T ← Q.get();
6 check1 ← T.SIG is correct;
7 check2 ← order of operations in L corresponds to T.IDE list;
8 check3 ← T.PRE precedes operations in T.IDE;
9 check4 ← T.PRE and T.SYN precede T.ID;
10 **if** check1 & check2 & check3 & check4 **then**
11 mark operations in *T.IDE* as *checked*;
12 put(Q, T.PRE);
13 put(Q, T.SYN);
14 **else**
15 verified ← False;
16 break;
17 **if** any operation in L is not checked **then**
18 verified ← False;
19 **return** verified;

Algorithm 3: Verify a log

Let us revisit the example in the previous section (see Figure.4). Let us assume one of two sites *site 1* or *site 2*, for instance, *site 2* shares the document by sending its log to another site, say *site 3*. Then *site 3* will verify the log it receives from *site 2*. To verify log, *site 3* has to verify the validity of authenticators and log entries. If it already received one part of log before, *site 3* can skip checking every authenticator linked to that part. We now describe the worst case when *site 3* receives the log from *site 2* for the first time and therefore every authenticator needs to be checked. *site 3* performs the following steps of the log verification procedure.

It starts by checking the most recent authenticator of *site 2* $T_{4_{@site2}}$.

- Verify $T_{4_{@site2}}.SIG$ by using *site 2*'s public key.
- Verify $T_{4_{@site2}}.IDE$. As $T_{4_{@site2}}.IDE = []$ then check2 and check3 can be skipped.
- Verify the order of $T_{4_{@site2}}.ID$ (linked to op_5, op_6) and $T_{4_{@site2}}.PRE = T_{3_{@site2}}.ID$ (linked to op_6) by checking if the log is maintained correctly (if op_6 is logged before op_5). Similarly, the order of $T_{4_{@site2}}$.ID and $T_{4_{@site2}}.SYN = T_{3_{@site1}}.ID$ (linked to operations op_5 and op_6) is checked.
- Since $T_{4_{@site2}}$ was constructed based on $T_{3_{@site2}}$ and $T_{3_{@site1}}$, these authenticators are put into a queue Q in order to be recursively verified.

If every above check passes then the authenticator $T_{4_{@site2}}$ is said valid. For other authenticators in queue Q, the verification is performed recursively and each verification follows steps in Algorithm 3. The verification finishes when queue Q is empty. The final checking result is only positive if all checks return positive result. Otherwise, the log will be not authenticated. Note that in this example, any deletion or re-ordering of operations is detectable. For instance, if *site 2* tries to re-order operations op_2 and op_3, this attack will be detected by authenticating the authenticator $T_{2_{@site1}}$ which is linked to op_3. We can see *site 2* cannot forge this order on behalf of *site 1* since *site 1* signed the authenticator linked to op_3. However, any re-ordering of concurrent operations will not change the verification result. The proof will be presented later.

Authenticators-based log verification has $O(1)$ complexity in space and $O(\Gamma)$ in time, where Γ is the total number of authenticators in the log. Since authenticators of a log are linked as a hash-chain in which an authenticator is linked to its preceding one, and due to the *forward-aggregated authenticity* property, it is enough to authenticate the log by checking only the most recent authenticator of a log. This verification process requires checking of all preceding authenticators. Therefore, the time complexity depends on the total number of all authenticators.

5.3 Proofs of Correctness

The algorithms, that have been presented previously for authenticators construction and logs verification, ensure the desirable properties for authenticating logs which are linearized from operation-based history.

THEOREM 1. *A log is tamper-detectable by using authenticators. A misbehaving site cannot selectively insert, delete or change the happened-before order of other sites' operations from the beginning or the middle of the log without being detected by next audit (Integrity).*

PROOF. Let M be the misbehaving site who receives a log $L = [op_1, op_2, \ldots, op_i, op_{i+1}, op_{j-1}, op_j]$ from site R. Let us assume op_i and op_{i+1} were generated by R and op_{j-1} and op_j were received by R from S. Log L is accompanied with authenticators and the most recent authenticator is $T_{j_{@R}}$ which is linked to operations (op_{i+1}, op_j). Following Definition 1 and Definition 2, $T_{j_{@R}}$ consists of:

$T_{j_{@R}}.ID = <site\ R, \{op_{i+1}, op_j\}>$ (linked to operations op_{i+1}, op_j),

$T_{j_{@R}}.SIG = \sigma_{siteR}(hash(T_{i_{@R}}.SIG \,||\, op_i \,||\, op_{i+1} \,||\, T_{j_{@S}}.SIG))$,

$T_{j_{@R}}.IDE = [op_i, op_{i+1}]$,

$T_{j_{@R}}.PRE = T_{i_{@R}}.ID$,

$T_{j_{@R}}.SYN = T_{j_{@S}}.ID$

There are three cases that M can attack the log as follows.

- *Case 1 - misbehaving site M removes operations at the beginning or in the middle of the log.*

(i) If M selectively removes any operation in range from op_i to op_j from L, i.e. op_i is removed by M but M still keeps op_{i+1}. The authenticator $T_{j@R}$ is then either invalid (missing of op_i) or replaced by $T'_{j@R}$. However, $T'_{j@R}$ is invalid since it should be signed by R and M cannot forge R's signature on $T'_{j@R}$.

(ii) If M removes any operation before op_i, i.e. op_1 is removed, then the authenticator $T_{i@R}$ is invalid and this makes $T_{j@R}$ invalid consequently.

Therefore, if a misbehaving site removes any operation in the middle of the log, the log will not be authenticated by valid authenticators.

- *Case 2 - misbehaving site M changes the happened-before order of operations on behalf of others.*

  If M changes the happened-before order of any operations from op_i to op_j then the operations list $T_{j@R}$.*IDE* will be invalid. When op_{i+1} is generated by site R, op_j is generated by site S, and R receives op_j from S after generating op_i, op_{i+1}, we say op_{i+1} and op_j are concurrent and other users can change the order of op_{i+1} and op_j. In the case of changing order of concurrent operations, the authenticator $T_{j@R}$ is still valid (it passes the check of Algorithm 3 - line 7). However, if R continues to work on the document and adds new operation op_k after op_j, then commits an authenticator $T_{k@R}$, other sites cannot change the order of op_i, op_j and op_k since this misbehavior will make the authenticator $T_{k@R}$ invalid by the checking procedure in Algorithm 3 - line 8.

  Therefore, if a misbehaving site changes any happened-before order, the log will not be authenticated.

- *Case 3 - misbehaving site M inserts an operation at the beginning or into the middle of log.*

  We assume misbehaving site M inserts an operation op_m in the middle of existing log between op_i and op_j.

  If M claims operation op_m was generated by site R then the log will be not authenticated since none of existing authenticators of site R authenticates op_m and it therefore cannot pass the verification process in Algorithm 3 - line 17, 18.

  If M claims op_m was generated by himself, then it needs to commit an authenticator to authenticate op_m. In such case, op_m is considered concurrent with other operations, so it can be inserted into any position in the log L and L is still authenticated.

  Therefore, a misbehaving site only can insert its own operations into its local log. The site cannot claim its insertion as operations on behalf of others because it cannot authenticate such operations.

In summary, it is impossible to forge the integrity of a log without being detected by using authenticators. □

THEOREM 2. *Authenticators preserve concurrency-collision-freeness property.*

PROOF. In the proof of theorem 1, we use a log L of site R, L_R = [op_1, op_2, …, op_i, op_{i+1}, op_{j-1}, op_j]. We assume operations op_i, op_{i+1} are concurrent with op_{j-1}, op_j. Thus the order between them can be interchangeable in any linearization of history. Let us consider that site S maintains a different log of same history L_S = [op_1, op_2, …, op_{j-1}, op_j, op_i, op_{i+1}]. We will prove that the log verification will return the same result on checking L_S and L_R.

When sites S and R share logs with each other, we suppose that $T_{i@R}$ and $T_{j@R}$ are committed by site R before and after receiving the log from site S; $T_{j@S}$ and $T_{i@S}$ are committed by site S before and after receiving log from site R. The log verification by checking $T_{i@S}$ and $T_{j@R}$ yields the same result regardless the order of concurrent operations. Indeed, $T_{i@S}$ and $T_{j@R}$ are valid only if they pass four checks (Algorithm 3, line 6 - 9). Consider check1 and check2 were passed, therefore they must pass check3 and check4 to be completely verified. The check3 only deals with the order of preceding authenticator against operations list IDE ($T_{i@S}$ with op_i, op_{i+1}, $T_{j@R}$ with op_{j-1}, op_j) and these orders are preserved as proved in Theorem 1. The check4 deals with the orders of preceding and synchronized authenticators with respect to the committed authenticator. Since $T_{j@R}$ is committed after $T_{i@R}$ and $T_{j@S}$ (linked to operations op_{i+1}, op_j), the check4 for $T_{j@R}$ passes. Similarly, the check4 for $T_{i@S}$ passes. Therefore, regardless the logging order of concurrent operations, the verification yields same result of checking two logs L_S and L_R. □

THEOREM 3. *Authenticators are forward-aggregated.*

PROOF. This property is achieved by using hash-chain based authenticator, so that $T_{i@site}$.SIG includes $T_{i-1@site}$.SIG in its construction. □

THEOREM 4. *Every site which is in possession of history can verify authenticators by using the public key of the site which committed them. A site which created an authenticator cannot deny having constructed it (public verifiability).*

PROOF. Non-repudiation is an important feature of digital signatures. By this property, a site that has signed authenticators cannot at a later time deny having signed them. Suppose that site S has signed an authenticator for operations $op_1, op_2, ..., op_i$ and shared them with another site. At later time, site S wants to change the history by removing op_i (e.g *insert line X*). In that case, site S should add a new operation op_j (e.g *delete line X*) instead of removing operation op_i since this will make authenticator $T_{i@S}$ invalid. Once a log has been shared with other sites, site S cannot remove its operations due to the using of non-repudiation signature for committed authenticators. Authenticators are linked to operations and replicated together with logs, therefore anyone can authenticate them. □

6. EVALUATION

We are going to present a practical evaluation of our proposed algorithms to authenticate logs. As the time complexity for the creation of authenticators is not significant, we evaluated the time complexity of the algorithm for log verification based on authenticators. Verification is done when a site clones or pulls remote log and it needs to check if the remote log is shared correctly without any tampering.

We carried out experiments on real logs from projects that used Mercurial as a distributed tool for source code management. We chose randomly two projects: Hgview project [22] and one branch of OpenJDK project [31]. The project *Hgview* includes almost 700 committed patches stored in repository gathering contributions from 20 developers with 115 interactions between them. One branch of *OpenJDK* stored about 350 committed patches in repository which were created by 31 developers with 253 interactions between them. A committed patch is a sequence of operations that a user commits. It is also called a log entry. We implemented our experiments by using Python programming language.

In the histories of projects developed with Mercurial or any other distributed version control system, we are unable to know when a user pulls changes. We can only have information about push

operations. We therefore considered the worst case scenario where a pull is performed at each new entry in the repository by an arbitrary user Y that had never interacted before with any other user X that contributed to the project. If previous interactions were taking place between users Y and X, an optimization could be applied.

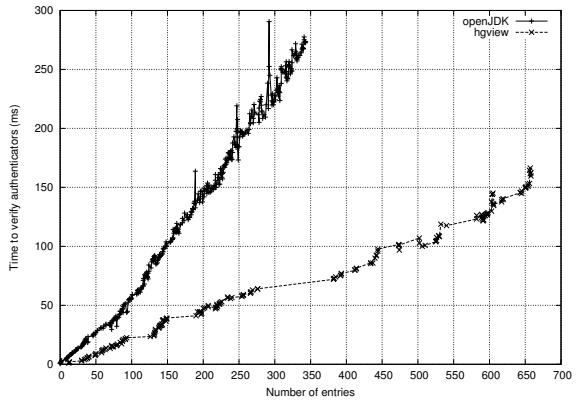

Figure 5: Time overhead to check authenticators created for Hgview and OpenJDK repositories.

In the experiment, we have first traversed the repository to extract entries and user names who contributed to the project. Then we generated for each user one RSA key pair which is later used to sign authenticators. Authenticators are created based on linearized logs of repositories. Finally, verification time is measured for the worst case scenario where a newcomer clones the repository and she has to check all authenticators created by previous contributors. The results are computed by the average values of five run times. Figure 5 presents the experimental results for the worst case behavior. In two experiments with the input data from Hgview project and OpenJDK project, the checking time grows linearly with the increasing number of authenticators. It is observed that it takes less than 50 milliseconds (ms) to verify a log if its size is less than 100 entries (this size is common for the size of all files observed in these projects). However, to check the whole repository, the verifying time depends mostly on the number of interactions between users (this means also the number of authenticators). In Figure 5, we notice that the runtime to verify log of the *Hgview* project is less than that of *OpenJDK* project even though its log size is bigger.

The main conclusion to be drawn from the results is that adding authenticators to secure logs does not create a significant time overhead for collaborative systems even for the worst case. Sites can reduce the time to verify log by skipping authenticators which are already checked when previous pulls were performed.

7. RELATED WORK

In this section, we give a review of existing securing log schemes and highlight their non suitability for securing operation-based history in multi-synchronous collaboration.

Several works introduce a trusted server to be used for verification. This approach makes the system open to a single point of failure. Peterson et al. [35] presents an approach to secure version history in a versioning file system. The approach proposed a design of a system based on generating message authentication codes MACs for versions and archiving them with a third party. A file system commits to a version history when it presents a MAC to the third party and at a later time, an auditor can verify this history. Thus it requires that the system trusts the third party that maintains MACs correctly. In the general model of multi-synchronous collaboration, users have no need to rely on a trusted third party and therefore any user should be able to verify logs.

In a similar direction, Haeberlen et al. designed PeerReview [12] to provide accountability and fault detection for distributed systems. It guarantees the eventual detection of all Byzantine faults. PeerReview framework contains tamper-evident logs and commitment, consistency and audit protocols. However, each node should have an identical log with others. It does not support that each node can keep different orders of operations as in operation-based multi-synchronous collaboration where users maintain different streams of activity on the shared data. The framework offers three applications of overlay multicast, network file system and peer-to-peer email, however, all of these applications do not deal with parallel modifications of data that is the case in multi-synchronous collaboration.

The integrity of audit logs has traditionally been protected through the use of one-way hash functions. There is a line of work that addresses the forward-secure stream integrity for audit logs. Ma et al. proposed a set of secure audit logging schemes and aggregate signatures [24, 23, 25]. Forward security ensures the integrity of log entries in the log stream and no selective deletion or re-ordering to stream is possible. Recently, Yavuz et al. proposed their work to secure audit log such as BAF [43] that was developed to achieve at the same time the computationally efficient log signing and the truncation-attack-resistant logging. This work could be applied only for a particular case of the multi-synchronous collaboration where all users maintain the same linearization of the collaboration history. However, it cannot be applied for the general case of the multi-synchronous collaboration where users work on different streams of activity on the shared data corresponding to different linearizations of collaboration history.

There is another line of work that relies on authenticated data structures to secure logs in distributed systems [9, 28, 27, 33]. While these approaches are computationally efficient, they do not deal with history for collaboration. Maniatis et al. introduced Timeweave [27] that uses a time entanglement mechanism to preserve the history state of distributed systems in a tamper-evident manner. However, Timeweave does not handle the information flows synchronized which is required in optimistic replication where concurrent operations appear in different orders in replicas.

Apart from above approaches, there are works that address securing logs for replication systems. Spreitzer et al. [40] uses hash chain to protect modification orders of a weakly consistent, replicated data system. Kang et al. [17, 16] proposed SHH for optimistic replication using hash values as in Merkle tree [30] for revision identifiers to protect causality of version history. It serves mainly for the purpose of securing version history construction when the log was pruned in limited storage environments such as mobile computing; and for checking distributed replicas' convergence. By ensuring decentralized ordering correctness, SHH can guarantee that all updates are not vulnerable to a decentralized ordering attack. However, SHH cannot ensure the integrity of data in the sense that it is original or forged. Using SHH the sender signature cannot be included in summary hashes since that makes them different even if the merged versions are identical, thus it makes replicas diverge. Without digital signature in summary hashes, SHH cannot protect history from attacks of unauthorized actions and it cannot provide authenticity and accountability. Similar approaches to SHH in which hashes are used as identifiers are implemented in distributed version control systems such as Git history [21] and Mercurial history [32].

Concerning securing document history, Hasan et al. [13] proposed a mechanism of preventing history forgery for a document history where a document refers to a file or database. In [13], the term

"provenance" is used for the history of the ownership of items and actions performed on them. The authors present a provenance-aware system prototype that captures history of document writes at the application layer. To prevent all potential attacks on provenance chain, it requires trusted pervasive hardware infrastructure at the level where tracking is performed. However, contributions to the shared document/database are done sequentially and the approach does not deal with merging of parallel contributions to the shared document.

A different work, Mella et al. [29] proposed a framework to the document control flow in a highly distributed environment. The proposal is aimed at cooperative updates on a document flow with delegation and security policies. However, it considers one stream of update process rather than a multi-way flow of updating with reconciliation as in multi-synchronous model. Moreover, security access control policies are defined at document's attributes level that means each document atomic element is marked with a label containing a set of access control policies that apply to it. The approach described in [29] secures different XML elements, while we aim to secure patches of operations.

In the domain of database security, Mahajan et al. [26] proposed Depot to secure replicated database in the cloud. Among all issues addressed in Depot, we focus on the issues of consistency, integrity and authorization. Depot addresses these issues in the context of database where data is stored in the form of key/value and update is the main operation performed over database. We consider collaborative systems with more operations beyond update, i.e. insert, delete content to/from the shared document. Depot ensures consistency by using version vectors and version history hashes. Each update is signed by authorized node to enforce consistency and integrity. This would be too costly in a collaborative working environment where users produce a huge number of operations on the shared document. Our approach secures logs without requiring that each user signs each operation. Authenticators are created for a patch of operations each time the log is pushed/pulled to/from one user.

The state of the art of secure audit logging research was also surveyed by Accorsi [2]. Though secure audit logging was intensively investigated, we are not aware of any work that ensures secure audit logs for a collaboration history with partial order where users maintain different total ordered logs of the collaboration history corresponding to their activity streams.

8. CONCLUSION

In this paper, we introduced a technique using authenticators to face security challenges in operation-based multi-synchronous collaboration. In multi-synchronous model, users work simultaneously on different streams of activity. As the collaboration progresses, the streams of activity continually diverge and synchronize. Authenticators are used to ensure integrity and authenticity of logs of operations corresponding to different streams of activity during a collaborative process. While tamper-resistance is impossible to be ensured in multi-synchronous collaboration without a central provider, tamper-detection should be guaranteed. We presented an approach for securing logs that made misbehaving users accountable in collaborative systems without the need of a central authority. We provided proofs of correctness of our approach and analyze the complexities of our algorithms. We also conducted a set of experiments testing our proposed approach on real histories of collaboration extracted from real projects using Mercurial. The results show the feasibility of our approach that can be used to provide security, trustworthiness and accountability to distributed collaborative systems.

Acknowledgment

This work is partially funded by the ANR national research grant STREAMS (ANR-10-SEGI-010).

9. REFERENCES

[1] M. Abdalla and L. Reyzin. A new forward-secure digital signature scheme. In *Proceedings of the 6th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology (ASIACRYPT'00)*, pages 116–129, London, UK, 2000. Springer-Verlag.

[2] R. Accorsi. Safe-keeping digital evidence with secure logging protocols: State of the art and challenges. In *Proceedings of the 2009 Fifth International Conference on IT Security Incident Management and IT Forensics (IMF '09)*, pages 94 –110, Stuttgart, Germany, September 2009.

[3] C. N. Chong, Z. Peng, and P. H. Hartel. Secure audit logging with tamper-resistant hardware. In *Proceedings of the 18th IFIP TC11 International Conference on Information Security, Security and Privacy in the Age of Uncertainty (SEC'03)*, pages 73–84, Athens, Greece, May 2003. Kluwer Academic Publishers.

[4] S. A. Crosby and D. S. Wallach. Efficient data structures for tamper-evident logging. In *18th USENIX Security Symposium*, pages 317–334, Montreal, Canada, August 2009.

[5] D. Davis, F. Monrose, and M. K. Reiter. Time-Scoped Searching of Encrypted Audit Logs. In *Proceedings of the 6th International Conference on Information and Communications Security (ICICS'04)*, pages 532–545, Malaga, Spain, October 2004.

[6] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenkcr, H. Sturgis, D. Swinehart, and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the Sixth annual ACM Symposium on Principles of Distributed Computing (PODC'87)*, pages 1–12, Vancouver, British Columbia, Canada, aug 1987. ACM Press.

[7] J. R. Douceur. The sybil attack. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems (IPTPS'01)*, pages 251–260, London, UK, 2002. Springer-Verlag.

[8] P. Dourish. The parting of the ways: divergence, data management and collaborative work. In *Proceedings of the fourth conference on European Conference on Computer-Supported Cooperative Work (ECSCW'95)*, pages 215–230, Norwell, MA, USA, 1995. Kluwer Academic Publishers.

[9] M. T. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. *DARPA Information Survivability Conference and Exposition,*, 2:68–82, 2001. Los Alamitos, CA, USA.

[10] S. Greenberg. Personalizable groupware: accommodating individual roles and group differences. In *Proceedings of the second conference on European Conference on Computer-Supported Cooperative Work (ECSCW'91)*, pages 17–31, Norwell, MA, USA, 1991. Kluwer Academic Publishers.

[11] S. Greenberg, M. Roseman, D. Webster, and R. Bohnet. Human and technical factors of distributed group drawing tools. *Interacting with Computers*, 4(3):364–392, 1992.

[12] A. Haeberlen, P. Kouznetsov, and P. Druschel. Peerreview: practical accountability for distributed systems. *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, 41:175–188, October 2007. Stevenson, Washington, USA.

[13] R. Hasan, R. Sion, and M. Winslett. The case of the fake picasso: Preventing history forgery with secure provenance. In *7th USENIX Conference on File and Storage Technologies (FAST'09)*, pages 1–14, San Francisco, CA, USA, 2009.

[14] J. E. Holt. Logcrypt: forward security and public verification for secure audit logs. In *Proceedings of the 2006 Australasian workshops on Grid computing and e-research - Volume 54 (ACSW Frontiers '06)*, pages 203–211, Darlinghurst, Australia, 2006. Australian Computer Society, Inc.

[15] C.-L. Ignat, S. Papadopoulou, G. Oster, and M. C. Norrie. Providing Awareness in Multi-synchronous Collaboration Without Compromising Privacy. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work (CSCW'08)*, pages 659–668, San Diego, California, USA, November 2008. ACM Press.

[16] B. B. Kang. *S2D2: A Framework for Scalable and Secure Optimistic Replication*. PhD thesis, EECS Department, University of California, Berkeley, Oct 2004.

[17] B. B. Kang, R. Wilensky, and J. Kubiatowicz. The hash history approach for reconciling mutual inconsistency. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS'03)*, pages 670–677, Washington, DC, USA, 2003. IEEE Computer Society.

[18] A.-M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The icecube approach to the reconciliation of divergent replicas. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing (PODC'01)*, pages 210–218, New York, NY, USA, 2001. ACM.

[19] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[20] E. Lippe and N. van Oosterom. Operation-based merging. *Proceedings of the fifth ACM SIGSOFT symposium on Software development environments*, 17:78–87, November 1992. New York, NY, USA.

[21] J. Loeliger. Collaborating with Git. *Linux Magazine*, June 2006.

[22] Logilab.org. hgview. `http://www.logilab.org/project/hgview`.

[23] D. Ma. Practical forward secure sequential aggregate signatures. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security (ASIACCS'08)*, pages 341–352, New York, NY, USA, 2008. ACM.

[24] D. Ma and G. Tsudik. Extended abstract: Forward-secure sequential aggregate authentication. In *IEEE Symposium on Security and Privacy*, pages 86–91, 2007.

[25] D. Ma and G. Tsudik. A new approach to secure logging. *ACM Transactions on Storage (TOS)*, 5(1):1–21, 2009.

[26] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. *ACM Transactions on Computer Systems*, 29(4):12:1–12:38, Dec. 2011.

[27] P. Maniatis and M. Baker. Secure history preservation through timeline entanglement. In *Proceedings of the 11th USENIX Security Symposium*, pages 297–312, Berkeley, CA, USA, 2002. USENIX Association.

[28] P. Maniatis and M. Baker. Authenticated append-only skip lists. *Acta Mathematica*, 137:151–169, 2003.

[29] G. Mella, E. Ferrari, E. Bertino, and Y. Koglin. Controlled and cooperative updates of xml documents in byzantine and failure-prone distributed systems. *ACM Transactions on Information and System Security (TISSEC), Volume 9*, 9:421–460, November 2006. New York, NY, USA.

[30] R. C. Merkle. *Secrecy, authentication, and public key systems.* PhD thesis, Stanford, CA, USA, 1979. AAI8001972.

[31] OpenJDK. OpenJDK. `http://openjdk.java.net`.

[32] B. O'Sullivan. *Mercurial: The Definitive Guide*. O'Reilly Media, 2009.

[33] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables. In *Proceedings of the 15th ACM conference on Computer and communications security (CCS'08)*, pages 437–448, New York, NY, USA, 2008. ACM.

[34] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the sixteenth ACM symposium on Operating systems principles (SOSP'97)*, pages 288–301, New York, NY, USA, 1997. ACM.

[35] Z. N. J. Peterson, R. Burns, G. Ateniese, and S. Bono. Design and implementation of verifiable audit trails for a versioning file system. In *Proceedings of the 5th USENIX conference on File and Storage Technologies*, pages 20–20, Berkeley, CA, USA, 2007. USENIX Association.

[36] N. M. Preguiça, J. M. Marquès, M. Shapiro, and M. Letia. A commutative replicated data type for cooperative editing. In *ICDCS*, pages 395–403, 2009.

[37] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys, Volume 37*, 37:42–81, March 2005. New York, NY, USA.

[38] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, David, and C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39:447–459, 1990.

[39] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. In *Proceedings of the 7th conference on USENIX Security Symposium - Volume 7*, pages 4–4, Berkeley, CA, USA, 1998. USENIX Association.

[40] M. J. Spreitzer, M. M. Theimer, K. Petersen, A. J. Demers, and D. B. Terry. Dealing with server corruption in weakly consistent, replicated data systems. In *Proceedings of the 3rd annual ACM/IEEE international conference on Mobile computing and networking (MobiCom'97)*, pages 234–240, New York, NY, USA, 1997. ACM.

[41] K. Walsh and E. G. Sirer. Experience with an Object Reputation System for Peer-to-Peer Filesharing (Awarded Best Paper). In *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3 (NSDI'06)*, Berkeley, CA, USA, 2006.

[42] S. Weiss, P. Urso, and P. Molli. Logoot-Undo: Distributed Collaborative Editing System on P2P Networks. *IEEE Transactions on Parallel and Distributed Systems*, 21(8):1162–1174, Aug. 2010.

[43] A. A. Yavuz and P. Ning. Baf: An efficient publicly verifiable secure audit logging scheme for distributed systems. In *Proceedings of the 2009 Annual Computer Security Applications Conference, (ACSAC'09)*, pages 219–228, Washington, DC, USA, 2009. IEEE Computer Society.