Securing Logs in Operation-based Collaborative Editing

Hien Thi Thu Truong
INRIA Nancy-Grand Est
hien.truong@inria.fr

Claudia-Lavinia Ignat
INRIA Nancy-Grand Est
claudia.ignat@inria.fr

Pascal Molli
Nantes University, France
pascal.molli@acm.org

ABSTRACT
In recent years collaborative editing systems such as wikis, GoogleDocs and version control systems became very popular. In order to improve reliability, fault-tolerance and availability shared data is replicated in these systems. User misbehaviors can make the system inconsistent or bring corrupted updates to replicated data. Solutions to secure data history of state-based replication exist, however they are hardly applied to operation-based replication. In this paper we propose an approach to secure log in operation-based optimistic replication system. authenticators based on hash values and digital signatures are generated each time a site shares or receives new updates on replicas. authenticators secure logs with security properties of integrity and authenticity. We present in detail algorithms to construct and verify authenticators and we analyse their complexities.

Author Keywords
secure log, authenticator, auditing, operation based optimistic replication

ACM Classification Keywords
C.2.4 Computer-Communication Networks: Distributed Systems

[Distributed applications];

D.4.6 Operating Systems: Security and Protection

[Authentication, Information Flow Controls];

K.6.5 Management of Computing and Information Systems: Security and Protection

[Authentication]

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. In order to improve reliability, fault-tolerance and availability, shared data is replicated in these systems. As shared data is mutable, consistency among the different replicas must be ensured. Rather than adopting a pessimistic replication mechanism that blocks access to a replica until it is provable up-to-date, these systems adopt an optimistic replication approach [15] that allows replicas to diverge during a short time period. The consistency model for optimistic replication is called eventual consistency, meaning that replicas are guaranteed to converge to the same value when the system is idle.

Optimistic replication can be classified into state-based and operation-based [6]. In state-based replication each site applies updates to its replica without maintaining a change log. Usually in systems adopting a state-based replication, every site sends its local state to other replicas that can merge the received state with their own state. Systems that use operation-based replication keep modifications performed on the replica in a log which is then sent to the other replicas. Operation-based approaches are used when cost to transfer state is high such as databases, mobile systems and when operation-semantics is important. In this work we target systems that use operation-based replication.

However, in operation-based replication systems users can misbehave and propagate incorrect contents to other users such as falsifying the causal order of the operations or removing some contents which are not allowed to be removed. Some mechanisms for securing operation-based logs exist [8, 3], but they are not adapted for optimistic replication where concurrent modifications can be done in parallel on the replicas. A mechanism for securing changes in optimistic replication was proposed in [5], but it is applicable only for state-based replication.

In the domain of data security, security properties [11] are defined including confidentiality, integrity, authenticity, access control, availability. For the purpose of preventing log tampering, in this paper we consider two of these security properties: *integrity* and *authenticity*. We informally consider such security properties as follows:

- *Integrity.* Adversaries cannot forge an event of document such as modify content or introduce new forged events to document log without being detected. Integrity is the most important required property for securing operation-based replication. Securing the individual document can be done 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.

- *Authenticity.* Any user in a system can verify the validity of events in a log. Adversaries are made accountable for unauthorized actions. This property can be done by using non-repudiation digital signatures such as RSA signature scheme [14], ECDSA scheme [4].

CSCW'12, February 11–15, 2012, Seattle, Washington, USA.

Traditionally, hashes are used to ensure integrity of any data object. However, in replication systems this data object changes due to the contributions of distributed users. Thus a tamper-evident update history is maintained to mitigate security challenges. Such histories containing record information of the updates that have been performed on the replicas along with the causal orderings of these updates are constructed in the manner that any tampering to previous version of replicas will be detected at other replica sites.

In this paper, we present scenarios that expose vulnerabilities of operation-based replication and then present an approach named to create authenticators that prevent users from tampering logs. authenticators detect if original data was modified or operations were reordered. Our solution ensures the integrity of shared documents in an operation-based replication system and the authenticity of user changes.

The structure of the paper is organized as follows. Section summaries related work in the domain of securing logs in distributed systems. Section introduces operation-based optimistic replication and conceptual operation-based model. Section presents algorithms to secure logs by authenticators along with their construction and verification. And Section 25 concludes the paper.

RELATED WORK

Logging systems and audit logs are very useful and widely used in distributed systems, however existing approaches with log auditing and authenticator do not work for replication systems, particular for operation-based replication. Among these approaches, Peterson et al. presents an approach to secure version history in a versioning file system [13]. 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 our system we do not rely on a third party and any user can perform a log auditing mechanism.

Secure audit logs based on hash chains and signatures have been widely used. Most of existing proposals are not suitable to apply directly to operation-based replication. Kang et al. proposed SHH [5] for optimistic replication using hash values in Merkle tree as revision identifiers to protect causality of version history. 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 contents to verify update is good or forged. The sender signature can not be included in generating summary hash since that makes summary hashes different even if the merged versions are identical, thus it makes replicas divergence. Thus it cannot provide the authenticity property. Similar approaches to SHH in which hashes are used as identifiers are implemented in distributed version control systems such as Git history [7] and Mercurial history [12]. In other related works such as secure provenance [3], secure timeline [9], secure logging [8], secure cooperative XML updates [10], the proposals address problem of handling information flow in highly distributed manner. However they only consider one-path of updating process rather than multi-paths with reconciliation as in collaborative editing.

OPERATION-BASED REPLICATION

In this section, we first start by presenting operation-based model in the context of our system. Then we will look into the vulnerabilities that could arise in replicated phases of a shared object.

System model

We consider a system of N *sites* of users which can operate independently on *replicas*. Each site keeps a document (replicated object) as a log of operations that have been performed during collaborative process. An object can be as large as a database (i.e. Bayou) or as small as a single file (i.e. WinFS file). An object is created at one site and propagated to other sites which contain a replica of the object. A site may replicate an object to any other sites. When a site which keeping a replica receives the updates, a process called *replica synchronization* will be performed. During synchronization, it requires to detect the concurrency and causality between updates of different sites. Also, it requires to resolve conflict if it occurs between concurrent updates. Replicas are consistent if their states are identical when they have the same set of operations. Two replicas are consistent after synchronization if each replica is maintained correctly. To ensure this we use *authenticator* after each sharing of replicated object. Operation-based replication maintains a history of operations and with operation transfer, only missing operations are sent to bring other replicas up-to-date (i.e. anti-entropy propagation with minimal traffic overhead). Receiving replicas updates new operations to their logs. From the log it is possible to see what are user contributions to different parts of the document and when these contributions were generated. Two users can make progress independently on the shared document. The changes are propagated in weakly consistent manner. It means user can decide when, with whom and what data to be sent and synchronized.

Definition 1. *An operation-based replicated object in a system of N sites is a log of operations which were performed to transfer a document D from an initial version V_0 to a final version V_k. Operations can be seen in different granularity, such as, lines or paragraphs in document, or deltas between revisions. Each operation recorded in the log is called an event. A shared object is considered as a log of events $L = [e_1, e_2, ..., e_k]$. Each event is parameterised by systems or applications.*

In the operation-based replication system, the input of synchronization is a common initial state and logs of operations that were performed on each replica. The system merges the logs in some fashion, and replays the operations in the merged log against the initial state to yield a reconciled version. Logs provide the history of site operations which can help to infer information about user's intents and therefore potentially more powerful.

For instance, in Fig.1 each site has an initial version V_0. At a later time, sites create or synchronize new updates on

each single replica to create new versions as logs of operations by manipulating with lines or paragraph of the document such as $V_1 = [op_1, op_2]$, $V_2 = [op_1, op_3, op_4]$, $V_3 = [op_1, op_2, op_5]$, $V_4 = [op_1, op_2, op_5, op_3, op_4]$, $V_5 = [op_1, op_2, op_5, op_4, op_3]$.

Users store operations in their logs in an order that is consistent with generation order. The collaboration of users involves logs reconciliation. When a user *u* receives a remote log L from a remote user *v* through anti- entropy propagation [1], *u* elects new events from *L* to append to his log *L*. Log synchronization of any two replicas ensures duplicate elimination.

Causality and Concurrency

As users perform updates locally, replicas diverge from one another. Operations isolated on one site are propagated to other sites and are integrated or replayed there to make sites convergent. However replicas might not converge if they execute operations in different orders. As the Lamport definition of causality, two operations $\mathtt{op_a}$ and $\mathtt{op_b}$ generated at sites `i` and `j` are in causal dependency $\mathtt{op_a} \rightarrow \mathtt{op_b}$ if: $\mathtt{op_a}$ happened before $\mathtt{op_b}$ and $\mathtt{i} = \mathtt{j}$; or the execution of $\mathtt{op_a}$ at site `j` happened before the generation of $\mathtt{op_b}$. Otherwise they are concurrent ($\mathtt{op_a} \,||\, \mathtt{op_b}$), that means neither $\mathtt{op_a} \rightarrow \mathtt{op_b}$ nor $\mathtt{op_b} \rightarrow \mathtt{op_a}$.

Since the log is an operation-based replicated document, it is possible to replay operations to get a state-based document. In order to guarantee convergence, it requires techniques to maintain and execute operations in correct orders, such as enforcing total order of operations before execution or using operational transformation, or using commutative replicated data type (CRDT) such as Logoot [17] in which concurrent operations can be replayed in variant orders. Note that as concurrency can occur in the replication, the orders of concurrent operations might vary in logs. Thus authenticators are designed to protect causality while allow variant orders of concurrent operations.

Reconciliation

A sequence of events either isolated or reconciled from other users are added to the log. We assume reconciliation process modifies one log so that it becomes the union of the two logs by adding new operations to the end of the log with respect the order as in original logs. As we mentioned previously that replica diffusion uses anti-entropy mechanism which preserves causality orders of operations and the reconciliation ensures causality of operations as well as allows that concurrent operations appear in logs in variant orders.

When a sending site sends a document to a receiving site, it creates an authenticator for its log to attach with the sent document. The receiving site creates a new authenticator when it receives the document. We assume each site involved in the update process possesses a cryptographic *public key pair* that is corresponded to a unique site identifier and that all the other users can retrieve the public key of each other. This assumption is reasonable in practice [16, 10]. The key pair is used to sign entries of log that prevent malicious sites modifying events on behalf of other users. Though sites can choose a public key pair on their-own, to limit Sybil attacks [2] we can require each site possesses a digital certificate from trusted certification authority or have an offline channel (such as email) to identified the owner of public keys. In either case the certification authority plays no role in creating authenticator process, and it is used only during initial phrase when a site joins the system. We also use cryptographic hash function H with properties collision- and preimage-resistant. The collision-resistant property can be used to establish the uniqueness of logs at a certain moment when an authenticator is created.

Vulnerabilities to document

In optimistic replication, the correctness of document history is based on the trustworthiness of users who maintain its own ordering of versions correctly such as when a user generates new updates, the order of new version should follow all previous versions. Unfortunately a malicious user can always generate phony updates to forge the history or alter the order of versions. This attacks make the document inconsistent among replicas. Users can update corrupted data from malicious site and discard data from valid site. Replicas with corrupted update might never converge with other valid replicas while victim site whose updates were lost and this is really problematic in replication system. In Fig.1 we present an example of a malicious site that generates phony updates without being detected.

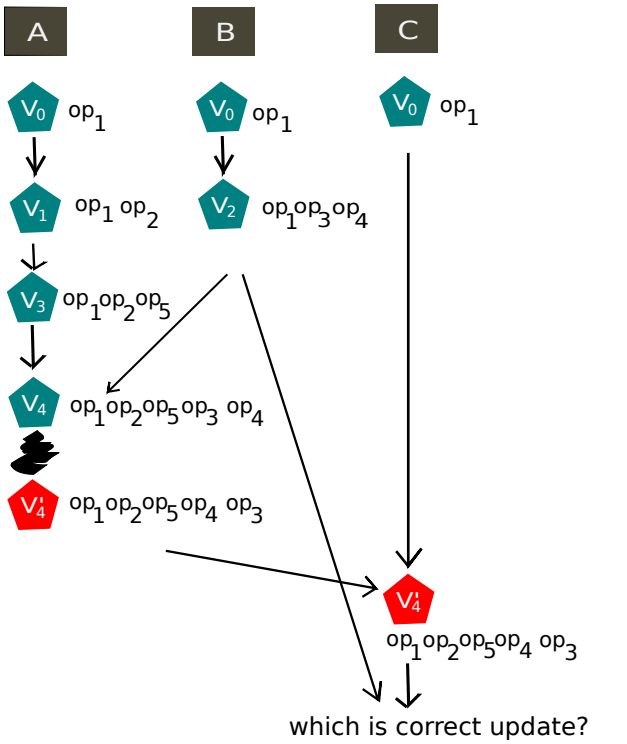

Figure 1. An attack to document log. User A modifies version V4 to create V'4 by changing causal order of op_3 and op_4, and then shares it with user C. C will never know that V'4 contains corrupted update based on data received from user B.

There are two types of malicious users that are insider and outsider. We consider in this paper an inside adversary who

has full rights to access to replicated object. Such an adversary might want to alter the process including actions performed on data of authorized contributors. For example when Alice provides a document to Bob who can perform and contribute new updates to the document, he should not be able to modify history of actions that Alice performed which were recorded in document log. The main goals of an adversary include: alerting existing records or adding forged information to logs (this might include involve forging other individual events, changing the order of events, or adding forged events into the log).

Our work assumes only adversaries who act inside of a system since we cannot stop an adversary from copying data to create a new document and claim at a later time as owner. It might be possible if an adversary remove completely the log of operations, however in such case it means the document will be removed. We can deal with other types of adversary (insider or outsider) with the support of trusted platform or system; however we do not consider this assumption in our system model.

The replication-based collaboration the secure of data items during the document collaborative workflows has gained increasing importance. As an example, if Bob receives a document and processes a part thereof and then forward to Claude, the operation-based document should include the chronology of actions that each user (Alice, Bob, Claude) performed on the document.

AUTHENTICATOR

In this section we are going to present our approach to create authenticator $\mathcal{A}$ in details which can ensure integrity and authenticity and suffer Byzantine attacks for operation-based optimistic replication.

Authenticator

In our context, we define authenticator as follow:

Definition 2. *An authenticator is a log tamper-evident denoted as $\mathcal{A}$ which captures a sub sequence of one or more events (operations) of the log in one updating session to protect their integrity and authenticity; where an updating session at one user's site is considered as a duration before the user sends or receives subsequent updates to/from other users.*

For illustration this definition, let consider at a site of user *u* at certain writing session the log is $[e_1,\ e_2,\ e_3,\ e_4,\ e_5]$. The user *u* creates an authenticator $\mathcal{A}$ for these events with willingness that when the log is synchronized at other replicas, it cannot be tampered by receiver, for example, the receiver reorders e_1 and e_2 to change the causal-dependency of e_1 and e_2.

Definition 3. *Each authenticator $\mathcal{A}$ as a tuple* $\langle$`ID, SIG, POS, IDE, PRE, SYN`$\rangle$ *holds:*

- `ID`*: identifier of authenticator including site identifiers U (and V) who commits (and receives) the authenticator and an event identifier that the authenticator is linked to;*
- `SIG`*: the value (signature signed by a public key PK of a user U to verify the authenticator later) of a certain state of document D based on operations;*
- `IDE`*: a list of identifiers of events which are used in the SIG;*
- `PRE, SYN`*: identifiers of preceding and receiving authenticators used to compute value of the authenticator;*

Definition 4. *The* `SIG` *of an authenticator $\mathcal{A}$ 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 latest authenticator $\mathcal{A}_n^S$.SIG = $\sigma_S\{H(\mathcal{A}_{n-1}^S \mid\mid E)\}$ with a condition that $E \neq \varnothing$; and the receiver computes $\mathcal{A}_m^R = \sigma_R\{H(\mathcal{A}_{m-1}^R \mid\mid E \mid\mid \mathcal{A}_n^S)\}$ with a condition that $\mathcal{A}_{m-1}^R \neq \varnothing$ or $\mathcal{A}_n^S \neq \varnothing$ or $E \neq \varnothing$;*

where,

- $\mathcal{A}_n^S$*: the last n^{th} authenticator committed by sender;*
- $\mathcal{A}_m^R$*: the last m^{th} authenticator committed by receiver;*
- $E = [e_{i_1} \mid\mid e_{i_2} \mid\mid ... \ e_{i_r}]$*: subsequent updates generated after preceding authenticator;*
- *($\sigma_u\{H(\odot||\ldots||\odot)\}$ denotes the concatenated arguments are hashed and signed with user u's private key).*

When a user shares a document by sending the whole log, he creates an authenticator for log events computed based on preceding authenticator, new updated events. The authenticator is signed by his private key and linked to the last event of the log. At the receiving site, the receiver performs reconciliation and creates a new authenticator up on reception.

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

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

As an example, we consider three sites A, B and C who collaboratively work through a shared object as a document with the initial version $V_0/\{op_1\}$. Each site performs isolated updates on the initial version to contribute to the object. In detail, site A, B create new versions $V_1/\{op_1, op_2\}$ and $V_2/\{op_1, op_3, op_4\}$ independently. At a later time site A reconciles with updates from site B to achieve new version $V_4/\{op_1, op_2, op_5, op_3, op_4\}$.

Fig. 1 shows one of vulnerabilities to forge versions. In this scenario, site A changes the content of reconciled version V_4 to forged version V_4' for her purpose. Later when site A shares

the forged version V_4' to site C, site C cannot know this version contains corrupted data that site A performed on the updates of site B. Also if site C then receives version V_2 from site B, site C cannot know which version either V_2 or V_4' contains correct content.

Our proposal with authenticators is one solution to avoid this forge. When user B shares his replica first time with user A, he creates an authenticator $\mathcal{A}_1^B$. As definition 4 to compute authenticator for a sender, $\mathcal{A}_1^B$ is computed with $\mathcal{A}_1^B$.SIG = $\sigma_{PK_B}\{H(op_1 \,||op_3\, ||op_4)\}$. An the authenticator is described fully as $\mathcal{A}_1^B$ = $\langle$ ID:$\{B, 1\}$, $\mathcal{A}_1^B$.SIG, POS:op_4, IDE:op_1, op_3, op_4, PRE:$\varnothing$, SYN:$\varnothing$ $\rangle$. When user A receives updates from user B, she performs synchronization and updates only new operations to the log, and creates an authenticator for new version of document, $\mathcal{A}_1^A$ where $\mathcal{A}_1^A$.SIG = $\sigma_{PK_B}\{H(op_1||\ op_2||\ op_5||\ \mathcal{A}_1^B)\}$, and $\mathcal{A}_1^A$ = $\langle$ ID:$\{A, 1\}$, $\mathcal{A}_1^A$.SIG, POS:op_4, IDE:op_1, op_2, op_5, PRE:$\varnothing$, SYN:$\mathcal{A}_1^B$ $\rangle$. These authenticators are linked to event op_4. When C receives the document from A, he updates new changes to his replica and creates a new authenticator $\mathcal{A}_1^C$. Later on, C receives the document from B but since no events are new to C no new authenticator is created.

Now we discuss that the authenticators created on this example can guarantee the integrity and authenticity of replicas. Let consider the case site A wants to forge version V_4 which is reconciled from V_2 and V_3. Since site A receives op_3 and op_4 from site B with authenticator $\mathcal{A}_1^B$, and then A changes the order of op_3 and op_4, this tampering will be detected by C when A sends the forged version to C. Authenticator $\mathcal{A}_1^B$ guarantees that A cannot forge operations on behalf of B. Similarly authenticator $\mathcal{A}_1^A$ guarantees that C cannot forge operations on behalf of A.

Algorithms

We present in details algorithms to create and verify authenticators in the following subsections.

Constructing authenticator

Since an authenticator is computed from preceding authenticator of the log, the current events and the authenticator of synchronized log, the input data to compute authenticator are two logs at site U and V, two sets of authenticators Z^U and Z^V. If it is the first authenticator, one of these parameters can be empty. In Algorithm 1, if L' = $\varnothing$ then there is no remote log to synchronize and it is the case when a sender computes an authenticator before sending his updates. The condition $E \neq \varnothing$ ensures that an authenticator is created only if the sender has generated new updates; otherwise the sender sends the log without computing any new authenticator.

Note that when synchronizing replicas, their authenticators that are linked to operations must be also synchronized so that all events from remote replica are authenticated when they are appended to other replicas.

For the complexity analysis, we assume the size of logs L and L' are treated as constants. We consider *time* and *space* complexities of $\mathcal{A}$ in algorithms of constructing authenticator and verifying authenticator.

Input:
A log L of site U and an authenticator set Z^U;
A log L' of a site V and an authenticator set Z^V;
The last authenticator of L: $\mathcal{A}_{n-1}^U$;
The last authenticator of L': $\mathcal{A}_m^V$;
Output: new authenticator created by U: $\mathcal{A}_n^U$

begin
  $E \leftarrow [e_{i_1}, ..., e_{i_r}]$ as new events added by U to L from last $\mathcal{A}_{n-1}^U$;
  if $L' = \varnothing$ **then**
    if $E \neq \varnothing$ **then**
      $\mathcal{A}_n^U$.ID $\leftarrow$ $\langle$ U, e_{i_r} $\rangle$;
      $\mathcal{A}_n^U$.SIG $\leftarrow \sigma_u\{H(\mathcal{A}_{n-1}^U||e_{i_1}||...||e_{i_r})\}$;
      $\mathcal{A}_n^B$.IDE $\leftarrow e_{i_1}, ..., e_{i_r}$;
      $\mathcal{A}_n^B$.PRE $\leftarrow \mathcal{A}_{n-1}^U$.ID;
      $\mathcal{A}_n^B$.SYN $\leftarrow \varnothing$;
    end
  end
  else
    if $\mathcal{A}_{n-1}^U \neq \varnothing$ *or* $\mathcal{A}_m^V \neq \varnothing$ *or* $E \neq \varnothing$ **then**
      $\mathcal{A}_n^U$.ID $\leftarrow$ $\langle$ U, e_{i_r} $\rangle$;
      $\mathcal{A}_n^U$.SIG $\leftarrow \sigma_u\{H(\mathcal{A}_{n-1}^U||e_{i_1}||...||e_{i_r}||\mathcal{A}_m^V)\}$;
      $\mathcal{A}_n^B$.IDE $\leftarrow e_{i_1}, ..., e_{i_r}$;
      $\mathcal{A}_n^B$.PRE $\leftarrow \mathcal{A}_{n-1}^U$.ID;
      $\mathcal{A}_n^B$.SYN $\leftarrow \mathcal{A}_m^V$.ID;
    end
  end
  $\mathsf{Z}^U \leftarrow \mathsf{Z}^U \cup \mathsf{Z}^V$;
  Add($\mathsf{Z}^U, \mathcal{A}_n^U$);
end

Algorithm 1: Authenticator construction

An authenticator construction in Algo.1 is $O(1)$ in time, and $O(|\Delta|)$ in storage, where Δ is the maximum number of events that are linked to $\mathcal{A}$.IDE to keep a list of events.

Since authenticator is created each time a site sends or receives (so-called interaction) updates, the number of authenticators on a replicated object created by a site *u* is the total number of interactions the site performed during the development process of the object. Let $|\Gamma|$ be the total number interactions of one site, each site needs $O(\Gamma \cdot |\Delta|)$ space. In synchronization, one log is updated to become the union of two logs, and the new log shall result with $O(\Gamma \cdot |\Delta_U| + \Theta \cdot |\Delta_V|)$ space, where Δ_U and Δ_V are maximum number of events authenticated in any authenticator of *U* and *V*. We can see the storage complexity depends on the number of interactions and the number of events between two interactions.

Auditing authenticator

The algorithm algo. 2 presents a mechanism to verify authenticators. When a user receives a document as a log of operations together with authenticators, he verifies these authenticators against to events in the log. The main idea of the verification is to check the authenticity of events with valid authenticators. Thus a log with either events are not authenticated or they are authenticated by invalid authenticators is unauthorized. An authenticator is checked by verifying its digital signature. In the verification not only the content but also the order of events are taken into account. If the corrupted data or falsified order of updates were introduced, authenticators will not be valid corresponding to the events and

Input:
A log L of site U;
A list of authenticators AL^U;
Output:
A boolean result AUTH, misbehaving site M (if found)

begin
  Let Q be a queue storing authenticators to be verified;
  $Q \leftarrow \mathcal{A}_n^U \in AL^U$;
  verified $\leftarrow$ TRUE;
  while $Q \neq \varnothing$ **do**
    Get authenticator $\mathcal{A}$ from Q;
    Extract ID, U, PK, SIG, POS, IDE, PRE, SYN from $\mathcal{A}$;
    Verify SIG by public key PK on signed data of H(PRE||IDE||SYN) ;
    if *order/content of events in IDE and order of PRE,IDE valid* **then**
      Marked events in L with identifiers $\in$ IDE;
      Add(Q, PRE);
      Add(Q, SYN);
    end
    else
      verified $\leftarrow$ FALSE;
    end
  end
  if *exist event* $e \in L$ *not marked) or (verified = FALSE)* **then**
    AUTH $\leftarrow$ FALSE; M $\leftarrow$ U;
  end
  else
    AUTH $\leftarrow$ TRUE; M $\leftarrow \varnothing$;
  end
  return AUTH, M;
end

Algorithm 2: Authenticator verification

the verification algorithm returns negative result. Authenticators help to aware attacks and once the log is detected as unauthorized, the user who sent the log is made accountable for the misbehavior.

Authenticator verification has $O(1)$ complexity in space and $O(\Gamma)$ in time, where Γ is the total number of authenticators in the log of replica. Since authenticators in a log are linked as a chain in which an authenticator is linked to a preceding one, it is enough to authenticate the log by checking the last authenticator in a log. This checking process requires to pass through all preceding authenticators, and that makes the time complexity depends on the total number of all authenticators.

CONCLUSION

In this paper we addressed security challenges in operation-based optimistic replication systems. To ensure integrity and authenticity of logs of operations during collaborative process, authenticators are computed and attached to logs as replicated objects. We argue that authenticators work for operation-based replication systems while existing approaches cannot be directly applied to such systems. While tamper-resistance is impossible to be ensured in replication systems, tamper-detection should be guaranteed. We presented an approach for securing logs that made misbehaving users accountable in a peer-to-peer environment where there is no central authority. We plan to apply our approach on real traces of collaboration such as on available projects developed using Mercurial and measure the complexity of our approach in this case.

REFERENCES

1. 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.
2. 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.
3. R. Hasan, R. Sion, and M. Winslett. The case of the fake picasso: Preventing history forgery with secure provenance. In *FAST*, pages 1–14, 2009.
4. D. Johnson, A. Menezes, and S. A. Vanstone. The elliptic curve digital signature algorithm (ecdsa). *Int. J. Inf. Sec.*, 1(1):36–63, 2001.
5. B. Kang, R. Wilensky, and J. Kubiatowicz. The hash history approach for reconciling mutual inconsistency. In *ICDCS*, pages 670–677, 2003.
6. E. Lippe and N. van Oosterom. Operation-based merging. *SIGSOFT Softw. Eng. Notes*, 17:78–87, November 1992.
7. J. Loeliger. Collaborating with Git. *Linux Magazine*, June 2006.
8. D. Ma and G. Tsudik. A new approach to secure logging. *TOS*, 5(1), 2009.
9. 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.
10. G. Mella, E. Ferrari, E. Bertino, and Y. Koglin. Controlled and cooperative updates of xml documents in byzantine and failure-prone distributed systems. *ACM Trans. Inf. Syst. Secur.*, 9:421–460, November 2006.
11. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001.
12. B. O'Sullivan. *Mercurial: The Definitive Guide*. O'Reilly Media, 2009.
13. 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.
14. R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.
15. Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37:42–81, March 2005.
16. K. Walsh and E. G. Sirer. Experience with an Object Reputation System for Peer-to-Peer Filesharing (Awarded Best Paper). In *NSDI*, 2006.
17. 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.