Reliably Determining Data Leakage in the Presence of Strong Attackers

Riccardo Bortolameotti
University of Twente
r.bortolameotti@utwente.nl

Andreas Peter
University of Twente
a.peter@utwente.nl

Maarten H. Everts
TNO & University of Twente
maarten.everts@tno.nl

Willem Jonker
University of Twente & EIT Digital
w.jonker@utwente.nl

Pieter Hartel
University of Twente
pieter.hartel@utwente.nl

ABSTRACT

We address the problem of determining what data has been leaked from a system after its recovery from a successful attack. This is a forensic process which is relevant to give a better understanding of the impact of a data breach, but more importantly it is becoming mandatory according to the recent developments of data breach notification laws. Existing work in this domain has discussed methods to create digital evidence that could be used to determine data leakage, however most of them fail to secure the evidence against malicious adversaries or use strong assumptions such as trusted hardware. In some limited cases, data can be processed in the encrypted domain which, although being computationally expensive, can ensure that nothing leaks to an attacker, thereby making the leakage determination trivial. Otherwise, victims are left with the only option of considering all data to be leaked.

In contrast, our work presents an approach capable of determining the data leakage using a distributed log that securely records all accesses to the data without relying on trusted hardware, and which is not all-or-nothing. We demonstrate our approach to guarantee secure and reliable evidence against even strongest adversaries capable of taking complete control over a machine. For the concrete application of client-server authentication, we show the preciseness of our approach, that it is feasible in practice, and that it can be integrated with existing services.

Keywords

Data Leakage, Forensics, Applied Cryptography, Distributed Systems Security

1. INTRODUCTION

Recent security incidents, such as those at JP Morgan [23], Adobe [15] and Sony [3], have shown how the leakage of (often personal) data can have a high economical and societal impact. US and European governments have noticed these effects and started to introduce *breach notification laws* [7, 9], where companies are obliged to disclose to the government the details of a data breach, and to notify individuals of security breaches where personal identifiable information was involved. In some states in the US [13] and some EU member states such laws are already being applied, and it will likely involve other countries in the near future. This leaves companies with the problem of determining what data has been leaked after a data breach. In this work we focus on the problem of determining the data leakage for those cases where companies collect and process customer data to provide a specific service, or to improve the quality of an existing service.

Determining the data leakage is a forensic process, and as such, it needs digital evidence that allows the company to retrieve the information about what data has been leaked. Existing work on data leakage focuses on prevention by providing Data Leakage Prevention solutions [25, 24, 36]. However, these solutions cannot be used to determine the data leakage in case the prevention fails. The most promising existing solutions are *provenance-aware* systems [22, 11, 26], which record the entire data life cycle including its origin and where it moves. Unfortunately, these works make strong assumptions on what the attacker cannot do (e.g., the attacker cannot access the OS kernel), which is unrealistic given the occurrences of sophisticated malware in data breaches [20], and the recent increase of rootkit development [19]. This lack of solutions against such powerful attackers, has only been recently addressed by Bates et al. [2] who proposed LPM, a Linux kernel module capable of collecting and protecting evidence against such an attacker. Unfortunately, LPM's security mainly relies on trusted hardware. Trusted hardware, which is invasive because it imposes significant changes to the infrastructure, often is not a feasible option for companies. This leaves companies without a solution to create evidence to determine data leakage.

There exist cryptographic techniques, such as fully homomorphic encryption (FHE) [12] and secure multi-party computation (MPC) [34], which allow data processing in the encrypted domain. Applications using these techniques do not require digital evidence since data is protected by cryptographic means. Therefore, determining the leakage is trivial, because *nothing* leaks to an attacker. However, these techniques are computationally expensive and only suitable in limited practical scenarios. Consequently, most of today's data breach victims have no other option than considering

ACSAC '16 December 05-09, 2016, Los Angeles, CA, USA

ACM ISBN 978-1-4503-4771-6/16/12.
DOI: http://dx.doi.org/10.1145/2991079.2991095

all data to be leaked.

Our main contribution is an approach capable of determining the data leakage that is not just *all-or-nothing*. Our solution is able to create tamper-resistant evidence against adversaries capable of taking complete control of a machine. Our approach does not rely on any trusted hardware. Lastly, we demonstrate for the specific application setting of client-server authentication that our solution is feasible in practice, and that it can be easily integrated with a existing services.

2. EVIDENCE FOR DATA LEAKAGE DETERMINATION

A company that wants to determine what data has been leaked needs digital *evidence* that describes what happened in the system. We consider data to be leaked if an attacker could have had access to it. The evidence is a digital data structure, for instance a log file, that records all the accesses to the data. When the company knows when and what has been compromised in its system, for instance through an Intrusion Detection System (IDS), it can determine from the evidence what data has been accessed in that period of time by specific machines or users. This represents the data leakage. However, the evidence can be targeted by an attacker who wants to cover his traces. For example, an attacker often wants to modify the evidence to remove any indication that he had access to the data. This would result in an imprecise determination of data leakage. Therefore, the evidence should meet certain requirements in order to be considered a reliable source of information. We define such requirements as follows: *1) Tamper-Resistance*, any unauthorized modification to the evidence should be detectable; *2) Availability*, the company should be able to access untampered evidence at any point in time; *3) Trustworthiness*, the evidence should not be forgeable, so it should not be possible to append fake records to the evidence; *4) Completeness*, the evidence should contain records for all the accesses on plaintext data that happened in the system.

In case there is no evidence or the evidence does not meet the requirements, the company *must* consider *all* data to be leaked. This holds for all scenarios where compromised machines process data in plaintext. The company cannot prove without trusted evidence that only specific data items have been leaked to the attacker. In scenarios where access to data is restricted by cryptographic techniques (e.g., FHE [12] or MPC [34]) and data is processed in the encrypted domain, *nothing* can be considered leaked. However, even the results of computations remain encrypted and can be decrypted only by interacting with the client who knows the decryption key. Moreover, these techniques are known to have severe limits in performance and functionalities in practice, especially in presence of *active* attackers that try to cheat during the protocol execution.

We propose an approach that creates evidence of data access and fulfills the above requirements. We demonstrate our approach to be neither *all* nor *nothing*. Our approach avoids the drawbacks of fully encrypted solutions [12] by performing computations on plaintext data, but instead it creates reliable evidence to determine data leakage more precisely than simply stating all data has been leaked.

3. INTUITION OF OUR APPROACH

First let us assume that a central server is responsible for creating and maintaining digital evidence for all accesses to data. If this machine is compromised by a powerful attacker, for instance by using a kernel rootkit, there is no solution to protect the evidence from being tampered with or deleted by the adversary unless trusted hardware is used [2]. Now we assume that there are two machines, A and B, where A sends its data access information to B and B stores the evidence for A. If an attacker compromises A, he can send fake information to B. There is no way for B to distinguish real from fake information, so the evidence stores fake records, which breaks the trustworthiness requirement. This is also true in case there are n machines and A sends its information to $N_1, \ldots, N_{n-1}$. As long as a machine is solely responsible for the content of its own evidence, and this machine can be compromised, it is not possible to guarantee all the requirements discussed in Section 2. All common logging mechanisms work as just described.

Our approach *inverts* this process. It is not A to tell the $n-1$ other machines what it had access to, but the $n-1$ machines tell A what it is going to access. So, the $n-1$ machines store in the evidence that A had access to a specific data item. This approach can be considered a generalized digital analogue of the *four-eyes principle* (n-eyes in our case), which requires two individuals to approve some action before it can be taken. We enforce this approach by requiring data to be encrypted at rest and decrypting it (making it accessible) only when it is needed for processing. This decryption step can be done only through the collaboration of multiple machines that carefully log this process.

4. BACKGROUND

In this section we introduce the building blocks used in our approach and discuss the choice of the specific implementations.

Byzantine Fault Tolerant Consensus Protocol. Byzantine fault tolerant consensus protocols are a form of state machine replication, such that a set of n distributed nodes can behave consistently as a centralized implementation executing operations atomically one at a time. These protocols survive f Byzantine failures. We decided to use PBFT [5] because it is a widely accepted in practice and because it provides strong consistency guarantees despite a communication complexity of $\mathcal{O}(n^2)$. However, other Byzantine fault tolerant consensus protocols can be used (e.g., [6, 16]). In PBFT there are $n = 3f + 1$ nodes, and they move through a sequence of configurations called *views*. Within each view one node is the *primary* node. The primary node is responsible to forward client requests for operations to other nodes and to propose the order of their execution. A request is executed if at least $2f + 1$ nodes have accepted it. When a request is executed, a *reply* message is sent to the client. When the client receives $2f + 1$ reply messages, he considers the operation to be successfully executed. Moreover, nodes periodically generate *checkpoints*, which are proofs confirming that at least $2f+1$ nodes have executed the same number *seq* of requests. In case the primary node is considered compromised, all nodes start a *change-view* procedure, which elects a new primary node.

t-out-of-n Secret Sharing. Secret sharing (SS) is a cryptographic primitive that allows any party to split a data

item s into a set of n shares, such that any combination of t shares, where $t \leq n$, can be used to reconstruct s. In our case we consider $t = f + 1$ and $n = 3f + 1$. The data item s can be recovered only if at least $f+1$ shares are combined. Therefore, until at least one correct node reveals its share, the attacker learns nothing about s. A secret sharing scheme provides the following algorithms:

- $\mathsf{Split}(s) \rightarrow \{s_1, \ldots, s_n\}$ generates a set of n shares from a data item s.
- $\mathsf{Combine}(\{s_i\}_{i=1}^{f+1}) \rightarrow s$ combines a set of shares to obtain the original data item s (or $\perp$ if there are less than $f + 1$ valid shares).

In this work we propose to use AONT-RS [29] for its performance. Compared to Shamir Secret Sharing [30], AONT-RS provides better storage efficiency[1] and faster split and reconstruction of secrets. AONT-RS is *computationally* secure (based on AES), and does not require a phase to distribute encryption keys.

Commitment Scheme. A commitment scheme is a cryptographic primitive that involves two parties: a *committer* and a *receiver*. The committer can commit to a specific value x, without revealing x to the receiver (i.e., *hiding* property), unless the committer itself reveals it. Once the committer commits to x, he cannot change it (i.e., *binding* property). A commitment scheme provides the following algorithms:

- $\mathsf{Commit}(x) \rightarrow c$ generates a commitment message for a value x.
- $\mathsf{Reveal}(x, c) \rightarrow 1/0$ verifies if x is the committed value for c.

In this work we use the simple hash-based construction that commits to a value x by computing $H(r||x)$ for a random r and a cryptographic hash function $H(\cdot)$ (modeled as a random oracle). However, other schemes might be used as well.

Digital Signature Scheme. A digital signature is a cryptographic primitive that allows a *signer* to authenticate a message, and a *verifier* to verify the authenticity of that message. A digital signature scheme provides the following set of algorithms:

- $\mathsf{KeyGen}(1^\lambda) \rightarrow (pk, sk)$ generates a signing key pair.
- $\mathsf{Sign}(sk, m) \rightarrow \sigma_m$ generates a signature for a message m.
- $\mathsf{Verify}(pk, m, \sigma_m)$ verifies a signature on a message m.

In this work we use RSA. Other digital signature scheme can be used as well.

Hash Chain. A hash chain is a consecutive application of a cryptographic hash function $\mathcal{H}$ to multiple data items that binds all of them to a single value. Given a family of events $\{E_i\}_{i=1}^{l}$, the value stored in the hash chain for each event E_i, is $L_i = \mathcal{H}(L_{i-1}||E_i)$, where $L_{i-1} = \mathcal{H}(L_{i-2}||E_{i-1})$. Each new entry is directly linked with the previous event, and the base hash L_0 is a predefined value.

[1]For example, given a 4 KB data block to divide 10-out-of-16 shares, Shamir Secret Sharing requires 64 KB, while AONT-RS needs 6,45 KB [29].

5. SYSTEM AND ADVERSARY MODEL

Consider a scenario where users can upload data to a company infrastructure which can run computations over this data, either on the user's or system's demand. The company is responsible for the data storage and the computation. Moreover, it wants to deploy a mechanism capable of *reliably* determining the data leakage in case its infrastructure was compromised.

We consider the company's infrastructure to be a distributed asynchronous system composed of n nodes, which log and execute the operations requested by clients. The set of possible operations is determined by the specific application. Each node has an append-only log, which describes all operations ran by the system and can be used to determine the data leakage. The nodes are connected by a network that may: fail to deliver messages, deliver them out of order, duplicate them, or delete them. We assume a Byzantine failure model, where faulty/*compromised* nodes might behave arbitrarily (e.g., a compromised node does not necessarily follow the protocol). Nodes are non-faulty/*non-compromised* if they behave according to the protocol. Our approach guarantees that clients eventually receive replies to their request (*liveness* property) and those replies are correct according to linearizability (*safety* property) if we assume that: the attacker can compromise at most f machines, where $f = \lfloor \frac{n-1}{3} \rfloor$; the attacker is computationally bounded, so he cannot break cryptographic primitives; there is a delay T between the first time a message is transmitted and when it is received by a non-compromised node, where T does not grow indefinitely [10]. The latter assumption appears to be reasonable in practice if network faults are quickly repaired. We also assume the company deploys security solutions that can detect compromised machines and determine the period of compromise (e.g., IDSs).

We consider a strong adversary that can take complete control over the machines he compromises by using a kernel rootkit. However, we assume compromising a machine is time consuming and a difficult task even for a skilled attacker. On these machines the attacker can tamper, disable or permanently delete any possible evidence. The attacker is adaptive, so he is allowed to compromise any machine and his choice may depend on previous information. Finally, the attacker can also coordinate the compromised machines to disrupt or to compromise the service.

Security Requirements for a Reliable Log

A log needs to fulfill the following security requirements to be considered *reliable* to determine the data leakage:

Tamper-Resistance: any unauthorized modification of the log, such as addition, removal or content modifications of a log entry e_i, should be detectable.

Availability: it should not be possible to remove all copies of the log from the system.

Trustworthiness: the log of a non-compromised node should contain only events that have been requested by legitimate events. An attacker should not be able to convince non-compromised nodes to log events that were never requested. The difference to *tamper-resistance* is that the attacker may try to remotely influence the log of *other* nodes by hijacking the protocol.

Completeness: the log should store all the accesses on plaintext data. An attacker should not be able to access any plaintext data item without being recorded by non-compromised nodes.

6. A DISTRIBUTED LOG AS EVIDENCE

In this section we discuss our approach for a *reliable* distributed log to determine data leakage. As discussed above, we combine threshold cryptography and Byzantine consensus to achieve this goal.

As follows from our high level description in Section 3, we see two important requirements: 1) a single node cannot access data without the consent of other nodes, and 2) a group of nodes should agree on when and what data a node can have access to. We address the first requirement using threshold cryptography, which enforces that t shares are needed to access the data. A lower number of shares does not reveal any information about the data. We address the second requirement using a Byzantine consensus protocol, guaranteeing consistency in the execution of operations among distributed nodes, despite (at most) f of them may be compromised. With this protocol, all nodes can agree on which data items a specific node can access, send the correct shares to the same node, and log the access to the data consistently with the other nodes.

As a consequence of the liveness and safety properties of a Byzantine consensus protocol (see Section 5) this setup requires $n = 3f + 1$ machines, where f is the number of compromised nodes. Considering this strict requirement, we define the threshold for Secret Sharing as $t = f + 1$. This is the lowest value that guarantees that the attacker cannot read the data, since the system assumes at most f compromised machines.

6.1 Creating Evidence of Data Access

We assume a finite universe of clients that can request to upload data, or to perform computations over data stored in a distributed system $\mathcal{N} = \{n_1, \ldots, n_n\}$ composed of n nodes, where $n = 3f + 1$, and f is the maximum number of nodes that can be compromised by an attacker. All clients and nodes have a public-private key pair $(pk, sk) \leftarrow \mathsf{KeyGen}(1^\lambda)$. All public keys pk are known to each node. Node keys are securely shared when the system is deployed, while client keys are securely shared at the time of client registration to the system. We assume the attacker is not present at registration time, and we assume clients to be *trusted*. We note that this is a simplified setup that considers clients to be trusted and authenticated. We also discuss our approach for scenarios with *untrusted* clients in Section 6.1.4.

6.1.1 Requests Need to be Accepted and Logged

Clients interact with the distributed system by sending *upload requests* or to perform *query requests*. These requests have to be accepted and logged by each node before being executed. This process is performed in the three-phase protocol of PBFT, as shown in Figure 1a.

In both cases, a client generates a request $\texttt{REQ}$, which describes the operation the client would like to run. He generates its signature $\sigma_{\texttt{REQ}} \leftarrow \mathsf{Sign}(sk, \texttt{REQ})$, and he sends the signed request $\langle \texttt{REQ}, \sigma_{\texttt{REQ}} \rangle$ to the *primary* node, which then forwards it to all nodes (*pre-prepare*). Each node accepts $\langle \texttt{REQ}, \sigma_{\texttt{REQ}} \rangle$ if $\mathsf{Verify}(pk, \texttt{REQ}, \sigma_{\texttt{REQ}}) = 1$ and if the message is formatted as specified by the protocol (see below Section 6.1.2 and Section 6.1.3), otherwise it is discarded. Please note that PBFT has additional checks during the execution of its protocol, which are independent from our solution. We refer to the original work for further details [5].

If $\langle \texttt{REQ}, \sigma_{\texttt{REQ}} \rangle$ is accepted, nodes broadcast their decision to other nodes (*prepare* and *commit* phases). When a node receives $2f + 1$ responses during the *commit* phase, it logs $\langle \texttt{REQ}, \sigma_{\texttt{REQ}} \rangle$. Each log entry $e_i = (i, \langle \texttt{REQ}, \sigma_{\texttt{REQ}} \rangle, L_i)$ has a sequence number i, the content of a request along with its signature $\langle \texttt{REQ}, \sigma_{\texttt{REQ}} \rangle$ and a recursive hash value $L_i = \mathcal{H}(L_{i-1} || \texttt{REQ})$. L_i links $\texttt{REQ}$ to its previous request.

Once e_i is stored, each node sends a signed *reply* message back to the client. The client considers $\texttt{REQ}$ to be successfully logged if it receives at least $2f+1$ signed reply messages from distinct nodes. At this point, $\texttt{REQ}$ is executed.

6.1.2 Upload Requests

Clients can store new data items on the nodes or update existing items. An *upload request* $\texttt{REQ}_U = \langle \text{ID}, ts, \sigma_{\texttt{REQ}_U} \rangle$ has an identifier ID of the item the client wants to upload, a timestamp ts and a digital signature $\sigma_{\texttt{REQ}_U}$. The request is sent to the nodes for approval and logging, following the process discussed above.

After receiving $2f + 1$ *reply* messages, the client runs $\mathsf{Split}(D) \rightarrow \{s_1, \ldots, s_n\}$ and generates n shares for the item D that has ID as identifier. Finally, as shown in Figure 1b, the client sends to each node n_i a message $\langle \text{ID}, s_i, ts, \sigma_d \rangle$ containing: the identifier ID, a share s_i of D, a timestamp ts, and a signed digest σ_d where $d = \mathcal{H}(\text{ID}, s_i, ts)$.

Each node accepts the share s_i if the following verifications steps succeeds (otherwise, it discards it): 1) compute d from the received message and check that $\mathsf{Verify}(pk, d, \sigma_d) = 1$, and 2) if a request $\texttt{REQ}_U$ with same ID and from same client was already logged (Section 6.1.1). If another share s_i is stored with the same ID, it is overwritten with the new share. Otherwise s_i is stored as a new item with identifier ID.

Although the information stored in $\texttt{REQ}_U$ does not influence the determination of data leakage, it is stored in the log to avoid possible data inconsistency in case of concurrent updates. For instance, if two clients are concurrently trying to update the same data item, all non-compromised nodes can decide, based on the log, to keep only the shares of the latest request they executed.

Replay attacks are possible, but do not affect the determination of data leakage and can be easily prevented by integrating orthogonal techniques in the protocol (e.g., nonce).

Upload by Nodes. A non-compromised node uploads data only if a client specified it in a *query* request $\texttt{REQ}_Q$ with a $\mathsf{write}(\text{ID})$ command (see Section 6.1.3). An upload request $\texttt{REQ}_U$ is generated and processed through PBFT as a client's request. The identifier ID for the new item is the one specified in the $\mathsf{write}(\text{ID})$ command.

A node accepts the request if: 1) after computing d from the received message, $\mathsf{Verify}(pk, d, \sigma_d) = 1$; 2) there exists a query request $\texttt{REQ}_Q$ that specifies a $\mathsf{write}(\text{ID})$ command such that ID matches the identifier in $\texttt{REQ}_U$; and 3) the node that generated $\texttt{REQ}_U$ is the same node that executed $\texttt{REQ}_Q$, and no upload requests have been accepted yet for that write command. The rest of the protocol follows as in the client case.

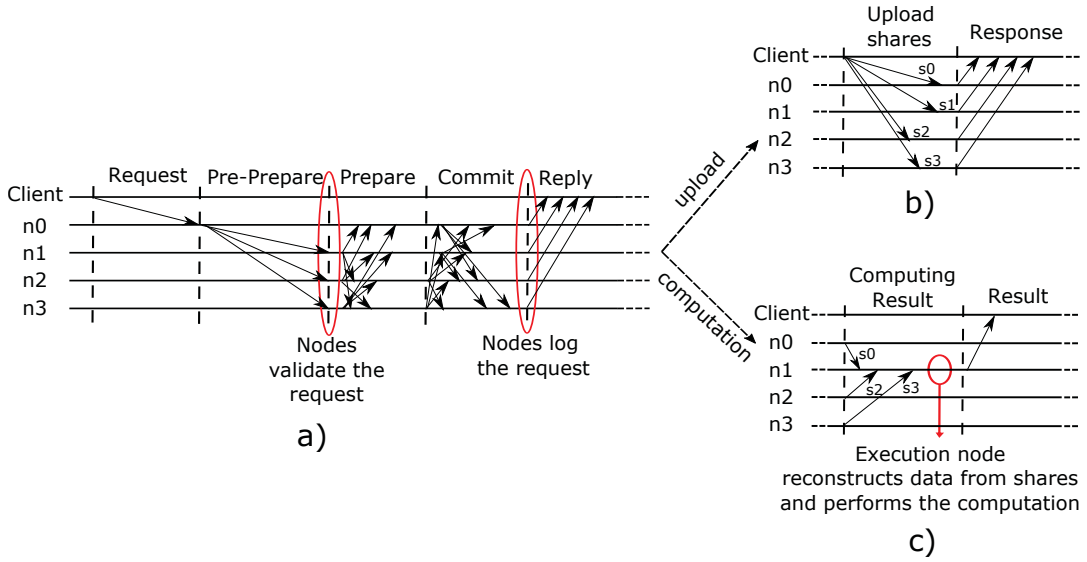


Figure 1: Overview of the interaction between a client and four nodes: a) (§ 6.1.1) shows the common process for each request, which is validated and logged through the PBFT protocol [5], where n0 is the primary node; b) (§ 6.1.2) shows how data is uploaded once the request has been logged, where the client t-out-of-n secret shares a secret s, and sends each share s_i to each node; c) (§ 6.1.3) shows how the computation is performed once the request has been logged, where the execution node (n1) receives the shares needed to reconstruct the data, performs the computation and sends back the result to the client.

Upload failure. It is possible that $\texttt{REQ}_U$ is logged, but the shares are not sent or delayed. Once $\texttt{REQ}_U$ is stored, each node starts a timer. All following requests that involve the uploaded item are logged and queued. Once the shares have been successfully received, requests are executed on the new item. If the timer reaches a globally set timeout value, the log is "rolled-back". $\texttt{REQ}_U$ is removed from the log, the hash chain is updated and the following requests are executed on the old item.

Timeouts can be exploited by an attacker to degrade the performance of the system (e.g., wait for shares and delay computations), however these denial of service attacks do not influence the determination of data leakage.

6.1.3 Query Requests

Clients can perform computations over data stored on the nodes. A *query request* $\texttt{REQ}_Q = \langle \mathcal{D}, O, r, ts, \mathsf{command}, \sigma_{\texttt{REQ}_Q} \rangle$ has a set $\mathcal{D}$ of unique data item identifiers, an operation O to apply to the data, a uniformly distributed random number r, where $0 \leq r \leq n-1$, a timestamp ts, a command read or write(ID), and the signature $\sigma_{\texttt{REQ}_Q}$.

$\mathcal{D}$ is the set of items requested for the computation. O is any possible computation allowed by the application. r is a random number used to select the *execution node*. This node is responsible for reconstructing the data from the shares, performing the requested computation and returning the result to the client. The execution node for a request is the node $n_i = r$. The command write(ID) is used by a client when he wants to retrieve the result of the computation and then store it in the system with the specified identifier ID. A client uses a read command if he only wants to retrieve the result.

$\texttt{REQ}_Q$ is validated and logged as discussed in Section 6.1.1. If accepted, each node retrieves the set of requested shares $\mathcal{S} = \{s_1, \ldots, s_m\}$ where $m = |\mathcal{D}|$. Each node sends $\mathcal{S}$ to the execution node. The execution node waits until it receives at least $f+1$ set of shares (including its own). It reconstruct each requested item in $\mathcal{D}$ such that $D \leftarrow \mathsf{Combine}(\{s_i\}_{i=1}^{f+1})$. It executes the requested operation O over the reconstructed set of items. Lastly, it sends the result of the computation to the client. This process is shown in Figure 1c. The execution node deletes all the received set of shares and the reconstructed data. Finally, it verifies whether a write(ID) command is specified in query request or not. If so, the results need to be uploaded through an *upload* request.

It is worth noting that if k shares are generated and uploaded instead of n, where $k < n$, the attacker may be able to block the execution of a query request by not sending the share he stores. We choose to generate n shares to guarantee the functionality of the system in a Byzantine model.

Execution Node Unavailable. In case the execution node is not available at the time the shares are being transmitted, all nodes run an instance of the protocol discussed in Section 6.1.5, which allows distributed nodes to agree on selecting a random execution node. The shares are then retransmitted to the new execution node.

The execution node might also fail to deliver the result of the computation to the client after it received the shares. For instance, the attacker controls the execution node of a specific request and wants to disrupt the service. Therefore, a client expects a response from the execution node within a predefined timeout. If the timeout is exceeded, the client generates a new query request with a different execution node. A compromised execution node can send wrong computation results to the client. Overcoming this problem is not the goal of this work. However, this does not affect the determination of leakage provided by our solution, because the nodes already logged all the data the (in this case) compromised execution node had access to. This happens independently from a successful or unsuccessful execution.

6.1.4 Untrusted Clients

In certain application scenarios, we cannot assume clients to be trusted. When assuming clients to be *untrusted*, we cannot rely on the authentication based on cryptographic keys.

This affects our approach in three ways: 1) a compromised primary node can successfully execute requests that were never generated by a client, because nodes cannot verify a request's authenticity; 2) a compromised node can generate fake client requests, and append them in the log while maintaining a consistent hash chain; 3) if an attacker compromises one node of the system, he can pretend to be any client and generate fake requests. He can modify r in $\texttt{REQ}_Q$, such that $r = n_i$ and n_i is the node he controls, so it gets always selected as execution node. These issues are related to the verification and logging process of requests, shown in Figure 1a.

The first problem breaks the *Trustworthiness* requirement discussed in Section 5. To solve this problem, clients broadcast their requests `REQ` to all nodes, instead of forwarding only to the primary node. Then the primary continues the PBFT protocol as usual. Non-compromised nodes accept `REQ` if they have received two copies of it, one from the primary node and one from the client. Thus, if a compromised primary node forwards fake requests, they are rejected by all non-compromised nodes.

The second problem breaks the *Tamper-resistance* requirement. This problem is solved by including the *checkpoints* generated by PBFT (see Section 4) in the log. A checkpoint is included in a log entry along a hashed value that links it to the previous request `REQ`. Checkpoints are generated periodically after z requests. An attacker cannot forge a checkpoint because he cannot forge the signature of $2f + 1$ nodes. He can forge at most $z - 1$ requests without being detected. However, the attack is detected as soon as the z-th request is stored and the checkpoint is created.

The third problem affects the random number r in $\texttt{REQ}_Q$. Manipulating this value can guarantee to an attacker the access to the data. We solve this problem by removing r from a query requests, and forcing nodes to agree on a random number *without* involving the client. After accepting `REQ`, all nodes run a distributed protocol to randomly select an execution node. The details of this protocol are discussed in Section 6.1.5. As consequence, even if an attacker controls a node and can impersonate an untrusted client, he cannot deterministically analyze data in plaintext on machines he controls. However, there is no way to prevent a compromised client to access the result of the computation.

A collusion of an *untrusted* client and a compromised node, can lead an attacker to attempt accessing any other users data, and the lack of digital signatures impede the nodes to determine whether those requests are originated by the owner of the data or not. However, the attacker has no control over the random selection of the execution node, which is the only node that can access the plaintext of a request. Therefore, even in case of collusion, the attacker has no guaranteed access to the data. The attacker can send (or replay) multiple requests to increase his chances to have his compromised node selected as execution node. This misuse can be limited by enforcing security policies such as limiting the number of requests on certain data.

Lastly, although the attacker can access more data by compromising clients, the log is still able to *reliably* determine the data leakage, and it is not affected by the collusion.

6.1.5 Collaborative Selection of the Execution Node

This protocol allows distributed nodes to agree on a specific random number, which is used to select the execution node for *query* requests. The selection is guaranteed to be random as long as the attacker cannot compromise more than f machines. This protocol is used in two occasions: in presence of untrusted nodes (see Section 6.1.4), or in case the execution node of a request is not available and a new node should be selected (see Section 6.1.3).

Each node n_i generates a value x_i uniformly distributed at random. x_i is the value used to select the execution node. Each node computes a commitment $c_i \leftarrow \mathsf{Commit}(x_i)$. In our specific scenario $\mathsf{Commit}(x_i) = H(R_i||x_i)$ for a cryptographic hash function H and some random value R_i. Each node sends c_i to the primary node. The primary node selects a set $\mathcal{F} = \{c_1, \ldots, c_{f+1}\}$ of commitments, and forwards it to each node. Each node n_i check if its commitment c_i is in $\mathcal{F}$. If so, it sends x_i and R_i to the other nodes. Each node verifies the commitments $\mathsf{Reveal}(x_i, c_i) = 1, \forall c_i \in \mathcal{F}$. If they are all correctly verified, the execution node is computed as $r = \sum_{i \in \mathcal{F}} x_i \mod n - 1$.

In case the primary node does not forward $\mathcal{F}$, it is considered compromised and all nodes start an instance of the *change-view* protocol (cf. Byzantine Fault Tolerant Consensus Protocol in Section 4) to select a new node as primary node. Nodes selected to reveal their committed value can also be compromised and do not reveal x_i. For this reason, there exists a timeout that limits the waiting time of revealing the commitments. In case the timeout is exceeded of which the duration is application-dependent, the primary node sends another set $\mathcal{F}'$ of commitment, that differs from $\mathcal{F}$ only for those commitments that were not revealed within the timeout.

The Benefit of using a Commitment Scheme. Suppose every node n_i in the system broadcast a random value x_i to all other nodes, and the sum of all those values is the execution node. A compromised node can wait until all other $n - 1$ nodes broadcasted their value. At this point, the attacker can compute the sum S of these values and he can choose a value x_i such that $x_i + S = n_i$, where n_i is a compromised node. This attack could not be detected given the asynchrony of the network.

The commitment scheme impede a compromised node learning what other nodes have committed to, unless they purposely reveal it, or allowing an attacker to change its value after it has been already committed. Therefore, the attack just described would not work. Our approach also resists at the adversary model described in Section 5, because even though an attacker controls f nodes, including the primary node, he has to select a set of $f + 1$ commitments. Therefore, there is always at least one value that is uniformly distributed at random, which guarantees r to be random as well.

6.2 Determining Data Leakage using the Distributed Log

As discussed in Section 5, we assume the company knows the set $\mathcal{M}$ of compromised machines, and the period of compromise (T_{begin}-T_{end}). This information is often the result of a breach investigation. Typically, Intrusion Detection Systems are deployed in order to detect whether a machine is

compromised. The detection of attacks is an independent research direction, and it is outside the scope of this work.

Algorithm 1 describes how to determine the data leakage from a log V. First, it verifies the integrity and authenticity of the content of each entry e_i (Line 4). Then, it verifies the integrity of the position of e_i (Line 5). If any of these two steps fails, V is considered tampered and discarded. Otherwise, Algorithm 1 checks if e_i describes a computation over a compromised node within the period of compromise in Line 7. If so, the set $\mathcal{D}$ of items is added to a list of leaked items. This process is repeated recursively on each entry in the log. Lastly, Algorithm 1 returns the list of leaked items. The data leakage is determined by identifying all data items, within the period of compromise, accessed by a compromised nodes.

For scenarios where clients are trusted, it is also possible to determine the data leaked to a specific compromised client. This can happen in case of malicious *insiders*, which is also a severe threat scenario in data breaches.

Algorithm 1 Determination of Data Leakage

Require: $\mathcal{M}$ is the set of malicious machines, where $|\mathcal{M}| \leq f$; T_{begin} is the time when the breach started; T_{end} is the time when the breach finished; a log V from a non-compromised machines. Let us consider j as the client who signed the request REQ.

```
1: procedure DATA LEAKAGE(V, M, T_begin, T_end)
2:     leaked_items ← [ ]
3:     for all e_i ∈ V do                ▷ e_i = (i, ⟨REQ, σ_REQ⟩, L_i)
4:         if Verify(pk_j, REQ, σ_REQ) = 1 then
5:             if L_i = H(L_{i-1}||REQ) then
6:                 if REQ.r ∈ M and
7: T_begin ≤ REQ.ts ≤ T_end then
8:                     leaked_items.add(REQ.D)
9:                 end if
10:            else
11:                return ⊥
12:            end if
13:        else
14:            return ⊥
15:        end if
16:    end for
17:    return leaked_items
18: end procedure
```

7. ANALYSIS

In this section we analyze the security of our approach and its communication and computational complexity. We also analyze the quality of leakage determination of our construction to show that it is not an all-or-nothing approach.

7.1 Security

The distributed log satisfies the security requirements discussed in Section 5.

7.1.1 Tamper-resistance

Each log entry $e_i = (i, \langle \texttt{REQ}, \sigma_{\texttt{REQ}} \rangle, L_i)$ contains a signature $\sigma_{\texttt{REQ}}$ and a hashed value L_i that links the ith entry with the $(i-1)$th. An attacker is not able to break cryptographic primitives, therefore he cannot modify REQ and generate a valid signature $\sigma_{\texttt{REQ}}$. Any addition or removal of entries in the log is detected by the hash chain values $L_0, \ldots, L_{i-1}, L_i$, where L_0 is a root value, because an attacker cannot generate a new $\texttt{REQ}'_i$ such that $\mathcal{H}(L_{i-1}||\texttt{REQ}'_i) = \mathcal{H}(L_{i-1}||\texttt{REQ}_i)$. Lastly, an attacker cannot *forge* the history of the log (e.g., providing a log with a consistent hash chain but containing *fake* entries), because each entry e_i contains the signature of a client and the attacker cannot forge it. The only entry that does not require a client signature is the upload of an execution node. However, this operation is considered valid only if a client request containing a write(ID) has been accepted, which cannot be forged by the attacker. This attack is also detectable in case of *untrusted* clients as discussed in Section 6.1.4.

7.1.2 Availability

Our solution is based on PBFT and it survives Byzantine failures when the total number of nodes is $3f+1$. Therefore, even if an attacker compromises f machines (i.e., worst-case scenario) and additionally f other machines fail (e.g., hardware failure), it is still possible to access $f+1$ *untampered* logs from the remaining *non-faulty* machines.

7.1.3 Trustworthiness

An attacker wants to remotely forge the log of non- compromised nodes by convincing them to append requests that have never been requested by clients. The attacker can achieve only by being the *primary node* and if he fulfills one of the following requirements: 1) forge a client's private key or 2) control $2f+1$ nodes. In the former case, an attacker cannot forge a client signature, because he has no access to his private key sk and he cannot break cryptographic primitives. In the latter case, the attacker can compromise at most f nodes, which is not enough to convince all non-compromised nodes that fake events really happened (the *safety* property of PBFT).

7.1.4 Completeness

The distributed log is complete if all the accesses to plaintext data are recorded by *non-compromised nodes*. Every time a request is processed and data is reconstructed in plaintext to an execution node, all *non-compromised* nodes log the event. If an attacker wants to access data in plaintext without being recorded, he needs to collect enough shares s_i in order to reconstruct the data. He can access at most f shares for each item in the system. However, $f < f+1$, where $f+1 = t$, so $\mathsf{Combine}(\{s_i\}_{i=1}^{f}) \rightarrow \perp$. Therefore, he cannot access any data item without any help from at least a non-compromised node, which logs every time it sends a share and it does that only upon valid requests.

7.2 Communication and Computation Complexity

Our approach describes two types of requests: the upload of data and the computation over data. Both request types use PBFT to verify and to log requests, so their communication complexity is $\mathcal{O}(n^2)$ per request. This is caused by the nodes broadcasting the request in the *prepare* and *commit* phase, as shown in Figure 1a. The remaining phases for both query types, shown in Figure 1b and Figure 1c, are responsible for transmitting shares over the network and their communication complexity is $\mathcal{O}(n)$. The computational complexity to split a data item in t-out-of-n shares is $\mathcal{O}((n-\ t)t)$ [29]. In our case, we have $t = f+1$. The

communication complexity of the distributed protocol used to select the execution node is $\mathcal{O}((f+1)n)$, because $f+1$ (or t) nodes broadcast their values to every node once their commitment has been selected.

The above complexity analysis only considers the additional overhead imposed by our approach when used on top of an existing service. The plaintext operations of the specific underlying service that are performed by the execution node are excluded in our analysis.

7.3 Quality of Leakage Determination

Recall that we have shown in Section 7.1 that our approach *reliably determines* data leakage, which was the goal of this work. So, in principle, we could stop here as we have achieved our goal. However, it does *not prevent* leakage because data is processed in the clear by the execution node (which might be compromised). Looking at our construction, we notice that it can still happen that *all* data leaks or that *nothing* leaks to an attacker. For instance, assume that an attacker manages to access all data in our system. In this case, our mechanism accurately and reliably determines the data leakage as 100%. Therefore, in order to evaluate where our approach lies between "*all* data has leaked" as opposed to "*nothing* has leaked", we would like to evaluate the expected data leakage in our system. More precisely, we want to evaluate the *quality of leakage determination*, which we define as the expected leakage in the system in the *worst-case* attack scenario. Recall that, following our adversary model (as defined in Section 5), our worst-case attack scenario is met when an attacker has fully compromised exactly f (out of n) nodes.

While the worst-case attack scenario is fixed in our adversary model, the expected leakage clearly depends on the concrete application setting we deploy our system in. Due to this dependence, evaluating the quality of leakage determination is not feasible in general terms. However, there is a specific category of applications for which the quality of leakage determination can be accurately evaluated without looking at individual applications within this category. We do our analysis of the quality of leakage determination for this category of applications which can be described by the following representative setting.

Consider a setting in which users are associated with a specific set of data items. The users can run computations only on their own dataset, and the computations are independent events. A very important example is a client-server authentication as discussed in Section 8. Moreover, let us assume an attacker controls f nodes during the whole system lifetime, which is the worst-case scenario of our approach. For this setting, we can derive a formula that allows us to compute the expected data leakage. Specifically, we can compute the number of distinct clients/users that would have their data leaked. We assume each user i requests m_i computations on his data during the whole system lifetime. A user's dataset is considered leaked if at least one computation out of m_i is reconstructed on a compromised node. Since the execution node is selected uniformly at random, a user's dataset is leaked with probability $p = 1 - (1 - \frac{f}{n})^{m_i}$, where n is the total number of nodes and f is the number of compromised nodes. The leakage of a user's dataset is an event (X) with two possible outcomes: either it is leaked (1) or it is not leaked (0). The expected data leakage for a user's dataset matches with the probability of its dataset to be leaked. So, if M is the entire population of users in the system, the expected data leakage is:

$$E[X] = \sum_{i=1}^{M} 1 - (1 - \frac{f}{n})^{m_i} \quad (1)$$

Overall, this shows that our approach is, on average, better than just saying that all or nothing has leaked. In Section 8.2, we evaluate a specific real-world application scenario using equation (1).

8. APPLICATION: CLIENT-SERVER AUTHENTICATION

In this section we evaluate the efficiency and the quality of our leakage determination approach in the setting of authentication. We have chosen this type of application because credentials are often a major target of data breaches. A specific example is the recent case of LinkedIn that, despite knowing they have been compromised in 2012, they were not able to determine the data leakage. Their estimate was around 7 million records. Only in 2016, when the attacker sold the data online, they realized that the leakage affected around 160 million records [17]. In biometric authentication, the data is even more sensitive because it cannot be changed, and determining its leakage is essential for customers to prevent identity theft.

We evaluate password-based, biometric-based and PUF-based (Physical Unclonable Function) authentication. These authentication mechanisms represent the three authentication categories: something we know, something we are and something we have, respectively. All three types of authentication involve a registration phase, where users or devices register their authentication token. In the specific case of biometrics, the token is a biometric template of a physiological or behavioral user characteristics, whereas in the PUF case, the token is a challenge-response pair [32].

8.1 Efficiency of Our Approach

Our approach is based on two main building blocks, AONT-RS [29] and PBFT [5], whose performance and functionalities have been already verified in literature and are adopted in practice [29, 4]. We believe *an implementation would not improve the understanding of our solution.* However, we estimate the efficiency of our approach to show its feasibility. Our analysis considers the case of normal operations, where there are no compromised nodes. We consider nodes to run operations in sequence and we do not consider any protocol optimizations, especially for PBFT. We are aware that this estimate does not reflect the real efficiency. However, we believe the estimate suffices to give an indication on the feasibility of integrating our approach.

We estimate the efficiency our approach in a biometric-based authentication such as iris recognition [8]. Let us consider a scenario where iris scanners (on company premise) are trusted clients, which have a RSA 2048-bit key pair and communicate through a 100Mbits network link with n nodes. Nodes are also connected with a 100Mbits network link. A biometric-based authentication, with our approach, would work as follows: first the client uploads the iris feature vector, then it requests a comparison between the recently uploaded vector and the one uploaded at registration time. If the feature vector would not be uploaded in a separate step, but sent in the clear within the query request, it would be

automatically leak to any node. Thus, each phase of the PBFT consensus mechanism (Figure 1a) is run twice, while phases of data upload (Figure 1b) and computation (Figure 1c) are run only once. We calculate the estimate by summing the time to transfer each message from one party to another, and the time to validate the signatures. We consider all operations to be in sequence, for instance if four nodes have to validate a signature, we sum four times the validation time.

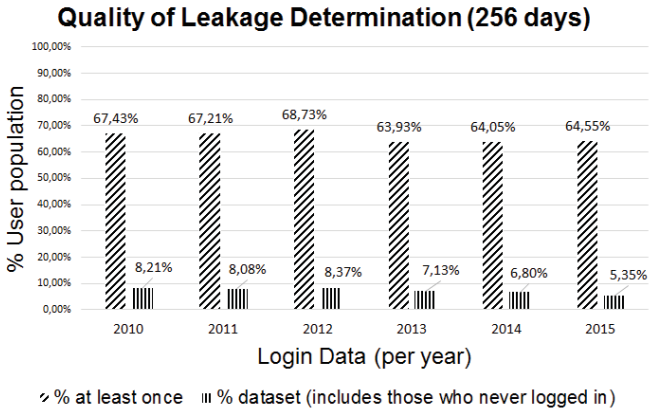
Table 1 shows that PBFT is the bottleneck of our approach. When there are four nodes, during one instance of PBFT 29 messages are transmitted over the network. Messages are upload or query requests. Since an authentication attempt needs two requests, the number doubles[2]. Due to the communication complexity of $\mathcal{O}(n^2)$, the performance of PBFT quadratically degrades as the number of nodes increases. The upload and computation phases, which send shares[3] over the network, do not influence the performance much and their overhead linearly increases with the number of nodes. The low overhead is a consequence of the small number of messages, the small size of the reconstructed item (an iris feature vector of 256 bytes [8]), and the encoding performance of AONT-RS [29].

Considering the performance values shown in Table 1, we believe this approach is efficient enough to be integrated in existing biometric-based authentication applications. Moreover, our estimation does not take into account PBFT optimizations, the parallelization of operations, and the fact that only $f + 1$ shares are needed to perform computations and not n. This approach is also practical in password-based and PUF-based solutions given the fact that they run the same number of requests and transmit smaller shares.

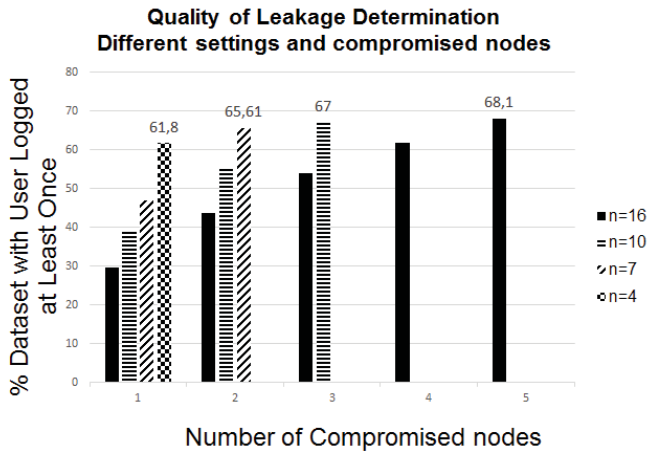
8.2 The Quality of Leakage Determination in a Real-World Application

In this section we show by using a real-world example that our approach is not an *all-or-nothing* approach. Undoubtedly, a *nothing* approach is the ideal scenario. However, this is not (yet) achievable in practice, since required cryptographic techniques (e.g., FHE or MPC) are not efficient enough in most practical applications. The remaining option is to consider *all* data to be leaked, which is the other extreme case.

Our approach explores a middle-ground between the two approaches, where the company might still suffer a leakage of data but it can *precisely prove* to the authorities, through reliable evidence, what has been leaked. We show that our approach is not *all-or-nothing* by determining the data leakage using equation (1) in the setting of a real-world application. The same equation can be used by operators to understand the behavior of their application under the proposed distributed systems and understand its possible benefits a priori.



(a) Quality of leakage determination with $n = 4$ and $f = 1$ in 5 years, for a compromise time of 256 days [1] .



(b) Quality of leakage determination in 2013 with number of nodes: $n = 4, 7, 10, 16$.

Figure 2: Representation of our quality of leakage determination in the setting of the ePrint service of CS faculty.

8.2.1 *Real-world Example*

We consider a currently centralized real-world application such as the internal ePrint web service of the entire faculty of Computer Science at an international university. We evaluate the quality of leakage determination if this service would be integrated with our solution. We analyze the authentication log, which contain over 10 years of login information. We consider the time of compromise to be 256 days, which, according to a recent study on data breaches [1], has been identified as the average time to identify a data breach.

We apply the model discussed in Section 7.3 to our password-based authentication dataset, where we consider the user's dataset as the user's password.[4] A user computation is a login, in which the user requests the system to compare whether the provided password matches the stored password

[2]For this use case, we assumed the upload requests to be 276 bytes, and the query requests to be 288 bytes. In both requests, 256 bytes represents the digital signature. The different size between the two request is due to the larger amount of arguments in the query requests.

[3]The data that we secret share is composed by 256 bytes of the iris feature vector, plus additional 64 bytes needed for AONT-RS [29]. For this use case shares have the following size: 160 bytes (4 nodes), 106 bytes (7 nodes), 80 bytes (16 nodes), because the size of shares is dependent to the threshold value t, such that $\frac{\text{data}}{t}$ [29].

[4]The term password can be substituted with other authentication tokens such as fingerprints for a biometric-based scenario, or the one-time challenge-response pair for the PUF-based scenario.

Nodes	PBFT protocol (2x)	Upload	Computation	Total time	Op/s
n=4, f=1	1.5 ms	0.1 ms	0.1 ms	1.7 ms	588
n=7, f=2	3.7 ms	0.2 ms	0.2 ms	4.1 ms	243
n=16, f=3	21.2 ms	0.5 ms	0.4 ms	22.1 ms	45

Table 1: Estimates of the additional overhead (per authentication) caused by our approach for an iris recognition application. The column 'PBFT protocol' represents the protocol steps in Figure 1a, while 'Upload' and 'Computation' represent the steps depicted in Figure 1b and Figure 1c, respectively. Their time estimates are computed for different number of nodes.

or not. We extracted the amount of logins (m_i) for each of the 2330 users, and we computed the overall expected data leakage with equation (1).

Figure 2a represents the average quality of leakage determination in case an attacker would have compromised f nodes (i.e., $n = 4$ and $f = 1$) for a period of 9 months (i.e., 256 days). We computed this value over the last 5 years data and it is possible to see how the leakage of the entire user base is determined to be lower than 9%. If we consider only the set of users that logged in at least once, the leakage is at most 68%. In both cases, it is clear that our solution is not an *all-or-nothing* approach.

We have also analyzed the *unrealistic* worst-case scenario where we consider an attacker to compromise f nodes for the entire 10 years. In this case, 32,5% of the entire dataset is leaked, while if we consider only users who logged in at least once, the leakage is 86,4%. Therefore, even in the unrealistic setting of 10 years of compromise, our solution proves to not be all-or-nothing.

Figure 2b depicts the quality of leakage determination from the log data of 2013 considering a different numbers of nodes. As expected, a lower number of compromised nodes results in less data leakage. Finally, it is worth noting that the determination of leakage in the worst-case scenario, where f nodes are controlled by the attacker, slightly increases with the number of nodes. This is due to the different ratio of $\frac{f}{n}$ (e.g., $\frac{1}{4} = 0.25$, $\frac{5}{16} = 0.31$).

This evaluation holds also for biometric-based and PUF-based authentication scenarios, because also in these cases one token is reconstructed for each authentication. In passwords and PUFs cases it is possible to update of authentication tokens after a fixed period of time, such that leaked data becomes useless. This computation can help in defining such policy. For instance, an operator can learn from the computation the subset of users whose passwords are more prone to be leaked in case of compromise. With this very specific information, the operator can define a policy to enforce a change of password for this specific set of users, so that the leaked data can become useless for the attacker.

9. DISCUSSION

Other Applications. Our approach is not limited to authentication applications. *Any* service running computations over data, can be integrated with our approach. It is important to notice that our solution moves the data over the network every time a computation is requested, therefore applications that are often computing on big data items (e.g., hundreds of megabytes) can suffer severe performance issues. A possible solution for those application could be to encrypt the data with a symmetric encryption scheme and secret share the secret key. The storage requirements would increase (e.g., an encrypted copy of each item on each node), but the network performance would significantly improve.

Practicality. Compromising a central server containing sensitive data is very often a complicated task and it requires time, because the infrastructure is often well protected. Nonetheless, central servers are still being compromised. A distributed solution, protected with the same degree of attention, makes it harder for an attacker to achieve his goal, because compromising one machine is not enough. This preserves the functionality of the system for a longer period (i.e., determination of leakage), and it exposes the attacker to higher risk of detection because it has to try additional attacks. This holds if the system is heterogeneous.

The major limitations of our system is inherent to the quadratic overhead of PBFT. Scaling to a huge number of nodes would make it incredibly difficult for an attacker to compromise the entire system, making the system very secure. Unfortunately, despite the fact that there are more efficient Byzantine fault tolerant consensus protocols than PBFT, there are still scalability issues. The new approaches to consensus used in cryptocurrencies, such as Proof-of-Work [28] and Proof-of-Stake [27], might be an alternative that can solve the problem of scalability. However, these techniques introduce new problems (e.g., consumption of resources) that can limit the practicality of our solution. Nonetheless, investigating these alternatives is a topic for future research.

Finally, considering the performance estimate of our approach, we believe our system is practical for a small number of nodes. We also consider it practical when it comes to keeping the system heterogeneous, because it is possible to install different OSs, instantiate different root passwords etc. for few machines.

10. RELATED WORK

Most of the work on the topic of data leakage focus on prevention. One of the most obvious approaches for leakage prevention is cryptography [12, 34]. Data Leakage Prevention (DLP) [25, 24, 36] is another category of techniques. All these approaches are not capable of determining the data leakage in case the prevention fails (in case they prevent it, the leakage is nothing).

Other research areas such as secure storage and data provenance are closer to achieve the determination of data leakage. Strunk et al. [31] proposed a self-securing storage that tracks and logs all the modifications to the data stored in the system, unless the attacker can compromise the host operating system. However, to determine the data leakage is important to log the access to all files and this information is not logged. If an attacker opens a file and reads its content, the operation is not logged as evidence. Other secure versioning solutions such as [33] have the same problem, and it is not known how much this would impact the system performance, especially on platforms performing a

lot of computations.

Provenance-aware systems are more promising solutions for data leakage determination because they aim to collect, store and manage the history of every single data object in the system. The term *provenance* refers to the history of ownership of an object. The vast majority of works on data provenance [18, 14, 22, 21, 11] are not designed to resists malicious adversaries, which makes them unusable in case of system compromise (one of our main assumptions). Pohly et al. [26] proposed Hi-Fi, a provenance-aware system that collects provenance through the Linux Security Module, which mediate any data manipulating operation on kernel objects. Hi-Fi protects the integrity provenance data against adversaries in user-space by using additional hardware such as read-only storage. Although this allows for the investigation or detection of a compromised system, it does not help the determination of data leakage because an attacker can append new fake entries to the provenance log. Additionally, protecting from user-space attacker is a weaker model. Zhou et al. [35] introduced secure network provenance (SNP), a technique that provides forensic capabilities to an untrusted networked system, where a subset of nodes can be fully compromised. SNP allows an administrator to identify faulty nodes and to explain any unexpected state change on any node. SNP focuses more on tracking faulty nodes and provide evidence for it, rather than determining what data has been accessed and possibly leaked. Bates et al. [2] have recently presented the only solution capable of collecting data provenance against a strong adversary, which could be used to determine what data has been leaked. The Linux Provenance Module (LPM) allows a secure provenance collection of processes, network activities and the kernel itself. Through the analysis of the provenance graph, it would be possible to determine the data leakage. However, the security of the whole system mainly relies on trusted hardware, and it is known that the storage and analysis of months (or years) of collected provenance data is still a severe practical problem.

11. CONCLUSION

We presented an approach to determine data leakage using a reliable distributed log as digital evidence. Our work reliably determines data leakage even in the presence of powerful attackers that are capable of taking complete control of one or more machines. In contrast to previous work in this strong attacker model, we do not require any trusted hardware. We have demonstrated the quality in leakage determination of our solution on real-world data; showing that it is not an *all-or-nothing* approach.

Given that we use well-known building blocks that have been demonstrated to work in practice [29, 4] and based on the values obtained from a worst-case estimation of our protocol, we believe our solution can be integrated with existing (authentication) services.

Acknowledgment

The authors would like to thank the anonymous reviewers for their insightful comments on the work. A special thanks to Tim van de Kamp, Bence Bakondi, Marco Caselli, and Chris G. Zeinstra. This work was supported by the THeCS project as part of the Dutch national program COMMIT/.

References

[1] 2015 Cost of data breach study: Global Analysis. URL: http://www-03.ibm.com/security/data-breach/.

[2] A. M. Bates, D. Tian, K. R. B. Butler, and T. Moyer. Trustworthy whole-system provenance for the linux kernel. In *24th USENIX security symposium, USENIX security 15, washington, d.c., usa, august 12-14, 2015*. J. Jung and T. Holz, editors. USENIX Association, 2015, pp. 319–334.

[3] BBC. The Interview: a guide to the cyber attack on Hollywood. URL: http://www.bbc.com/news/entertainment-arts-30512032.

[4] A. N. Bessani, J. Sousa, and E. A. P. Alchieri. State machine replication for the masses with BFT-SMART. In *44th annual IEEE/IFIP international conference on dependable systems and networks, DSN 2014, atlanta, ga, usa, june 23-26, 2014*. IEEE, 2014, pp. 355–362.

[5] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM trans. comput. syst.*, 20(4):398–461, 2002.

[6] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX symposium on networked systems design and implementation, NSDI 2009, april 22-24, 2009, boston, ma, USA*. J. Rexford and E. G. Sirer, editors. USENIX Association, 2009, pp. 153–168.

[7] 1. Congress. S.177 - Data Security and Breach Notification Act of 2015. URL: https://www.congress.gov/bill/114th-congress/senate-bill/177.

[8] J. Daugman. How iris recognition works. *IEEE trans. circuits syst. video techn.*, 14(1):21–30, 2004.

[9] Epic.org. EU Data Protection Directive. URL: https://epic.org/privacy/intl/eu_data_protection_directive.html.

[10] M. J. Fischer, N. A. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[11] A. Gehani and D. Tariq. SPADE: support for provenance auditing in distributed environments. In *Middleware 2012 - ACM/IFIP/USENIX 13th international middleware conference, montreal, qc, canada, december 3-7, 2012. proceedings*. P. Narasimhan and P. Triantafillou, editors. Vol. 7662. In Lecture Notes in Computer Science. Springer, 2012, pp. 101–120.

[12] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st annual ACM symposium on theory of computing, STOC 2009, bethesda, md, usa, may 31 - june 2, 2009*. M. Mitzenmacher, editor. ACM, 2009, pp. 169–178.

[13] J. Halpert and M. J. Anderson. State breach notification laws - updates from the 2015 legislative sessions, 6 action steps for companies. URL: https://www.dlapiper.com/en/us/insights/publications/2015/07/state-breach-notification-laws/.

[14] 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, february 24-27, 2009, san francisco, ca, USA. proceedings*. M. I. Seltzer and R. Wheeler, editors. USENIX, 2009, pp. 1–14.

[15] A. Hern. Did your Adobe password leak? now you and 150m others can check. URL: http://www.theguardian.com/technology/2013/nov/07/adobe-password-leak-can-check.

[16] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. L. Wong. Zyzzyva: speculative byzantine fault tolerance. *ACM trans. comput. syst.*, 27(4), 2009.

[17] LinkedIn lost 167 million account credentials in data breach. URL: http://fortune.com/2016/05/18/linkedin-data-breach-email-password/.

[18] P. Macko and M. Seltzer. A general-purpose provenance library. In *4th workshop on the theory and practice of provenance, tapp'12, boston, ma, usa, june 14-15, 2012*. U. A. Acar and T. J. Green, editors. USENIX Association, 2012.

[19] McAfee. Rise of rootkits. URL: http://www.mcafee.com/us/security-awareness/articles/rise-of-rootkits.aspx.

[20] D. McMillen. Wiper malware analysis. URL: http://www.ibm.com/developerworks/library/se-wiper-analysis/index.html.

[21] K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. L. MacLean, D. W. Margo, M. I. Seltzer, and R. Smogor. Layering in provenance systems. In *2009 USENIX annual technical conference, san diego, ca, usa, june 14-19, 2009*. G. M. Voelker and A. Wolman, editors. USENIX Association, 2009.

[22] K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX annual technical conference, boston, ma, usa, may 30 - june 3, 2006*. A. Adya and E. M. Nahum, editors. USENIX, 2006, pp. 43–56.

[23] E. Munoz. JP Morgan hack exposed data of 83 million, among biggest breaches in history. URL: http://www.reuters.com/article/us-jpmorgan-cybersecurity-idUSKCN0HR23T20141003.

[24] I. Papagiannis and P. R. Pietzuch. CloudFilter: practical control of sensitive data propagation to the cloud. In *Proceedings of the 2012 ACM workshop on cloud computing security, CCSW 2012, raleigh, nc, usa, october 19, 2012*. T. Yu, S. Capkun, and S. Kamara, editors. ACM, 2012, pp. 97–102.

[25] V. Pappas, V. P. Kemerlis, A. Zavou, M. Polychronakis, and A. D. Keromytis. CloudFence: data flow tracking as a cloud service. In *Research in attacks, intrusions, and defenses - 16th international symposium, RAID 2013, rodney bay, st. lucia, october 23-25, 2013. proceedings*. S. J. Stolfo, A. Stavrou, and C. V. Wright, editors. Vol. 8145. In Lecture Notes in Computer Science. Springer, 2013, pp. 411–431.

[26] D. J. Pohly, S. E. McLaughlin, P. McDaniel, and K. R. B. Butler. Hi-Fi: collecting high-fidelity whole-system provenance. In *28th annual computer security applications conference, ACSAC 2012, orlando, fl, usa, 3-7 december 2012*. R. H. Zakon, editor. ACM, 2012, pp. 259–268.

[27] Proof of Stake. bitcoin wiki. URL: https://en.bitcoin.it/wiki/Proof_of_Stake.

[28] Proof of Work. bitcoin wiki. URL: https://en.bitcoin.it/wiki/Proof_of_work.

[29] J. K. Resch and J. S. Plank. AONT-RS: blending security and performance in dispersed storage systems. In *9th USENIX conference on file and storage technologies, san jose, ca, usa, february 15-17, 2011*. G. R. Ganger and J. Wilkes, editors. USENIX, 2011, pp. 191–202.

[30] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[31] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: protecting data in compromised systems. In *4th symposium on operating system design and implementation (OSDI 2000), san diego, california, usa, october 23-25, 2000*. M. B. Jones and M. F. Kaashoek, editors. USENIX Association, 2000, pp. 165–180.

[32] G. E. Suh and S. Devadas. Physical unclonable functions for device authentication and secret key generation. In *Proceedings of the 44th design automation conference, DAC 2007, san diego, ca, usa, june 4-8, 2007*. IEEE, 2007, pp. 9–14.

[33] J. Wires and M. J. Feeley. Secure file system versioning at the block level. In *Proceedings of the 2007 eurosys conference, lisbon, portugal, march 21-23, 2007*. P. Ferreira, T. R. Gross, and L. Veiga, editors. ACM, 2007, pp. 203–215.

[34] A. C. Yao. Protocols for secure computations (extended abstract). In *23rd annual symposium on foundations of computer science, chicago, illinois, usa, 3-5 november 1982*. IEEE Computer Society, 1982, pp. 160–164.

[35] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. Secure network provenance. In *Proceedings of the 23rd ACM symposium on operating systems principles 2011, SOSP 2011, cascais, portugal, october 23-26, 2011*. T. Wobber and P. Druschel, editors. ACM, 2011, pp. 295–310. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043584. URL: http://doi.acm.org/10.1145/2043556.2043584.

[36] D. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. TaintEraser: protecting sensitive data leaks using application-level taint tracking. *Operating systems review*, 45(1):142–154, 2011.