Privacy-Preserving Logical Vector Clocks using Secure Computation Techniques

Florian Kerschbaum
SAP Research CEC Karlsruhe
Vincenz-Prießnitz-Straße 1
D-76131 Karlsruhe, Germany
Florian.Kerschbaum@sap.com

Julien Vayssière
SAP Research CEC Brisbane
Level 12, 133 Mary Street
Brisbane QLD 4000, Australia
Julien.Vayssiere@sap.com

Abstract

Systems of logical clocks are commonly found in distributed systems for establishing causality between events occurring in concurrent communicating processes. Vector clocks are a popular type of logical clocks which require processes to attach to each message a logical timestamp that contains information about the sender process's view of the state of the distributed computation at the time of message sending. Causality between two events can then be determined by comparing their two logical timestamps. However, by doing so, processes leak potentially sensitive information about the advancement of their computation and the computations performed by the processes they communicate with. The contribution presented in this paper is a protocol, based on secure computation techniques, which preserves the privacy of each process's local logical clock while being strictly equivalent to regular vector clocks.

1 Introduction

In this section we first introduce the use of logical clocks for the purpose of determining causality in distributed systems. We then point out the privacy issues raised by logical vector clocks and lay out the structure of the paper.

1.1 Use of logical clocks for establishing causality in distributed systems

Distributed systems can be modeled as collections of processes which communicate by means of asynchronous message passing. The behavior of a given process can be described as an ordered sequence of local events. Some of these events are internal to the process, while others have to do with either sending or receiving messages.

An example of such a system is shown in Figure 1 where events are represented by circles placed on the horizontal timeline of the process they occur in. Arrows between events in two separate processes denote a message being sent from one process to the other one. In Figure 1, for example, a message m is sent from process $p1$ to process $p3$ by means of the sending event a and the receiving event b.

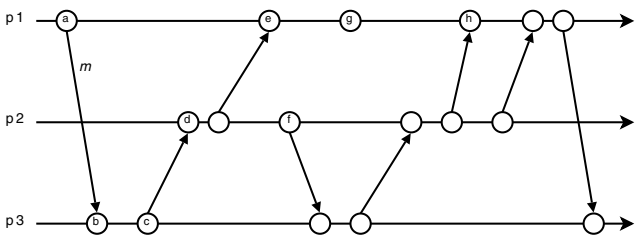

Figure 1. A distributed system composed of three concurrent communicating processes

The distributed and asynchronous nature of this model, however, introduces added complexity when it comes to determining causality between events that occur in separate processes. Since causality is closely linked to the passage of time, the causality relationship in distributed systems is usually referred to as the *happened before* relationship. The two terms will be used interchangeably in this paper.

An attractive, yet flawed, approach for determining causality would be to rigorously timestamp each event using an accurate global clock, or an accurate local synchronized copy of a global clock. It has however been proven that timestamps that use real (physical) time cannot be relied upon in a distributed system. This argument was first presented by Lamport [9] and contains both technological aspects, such as the difficulty to keep clocks in different locations synchronized, and more fundamental aspects linked to the finite speed at which information can be transmitted.

Causality can however be trivially established for events that happen within the same process, since the sequential execution of the process guarantees their total ordering. Pairs of communication-related events are also eas-

978-1-4244-1890-9/07/$25.00 ©2007 IEEE

ily ordered since the sending of a message always precedes the reception of the same message. However, determining causality between two arbitrary events that happen in two separate processes requires more work. For example, with event a occurring in process $p1$ and event d occurring in process $p2$, how can one determine which event happened before the other one? In this case, one can simply follow the chain of causality between a and d through b and c and conclude that a happened before d. In the case of a pair of events such as f and g however, no causality chain can be found and these two events are therefore called concurrent.

The first solution to the problem of establishing causality between events in distributed systems in the general case was proposed by Lamport in the form of scalar logical clocks [9]. The main idea is that each process maintains a local logical clock, which is a counter that the process can increment at will. When sending a message, the process attaches the current value of its logical clock to the message. On reception of the message, the receiving process updates its own logical clock according to a set of rules which guarantee that, if event x caused event y, then the timestamp of x will have a value lower than the timestamp of y.

The main limitation of scalar logical clocks is that the reciprocal of the above property is not true: comparing two timestamps does not allow one to infer which event happened first, or if the two events happened concurrently. A major improvement on scalar clocks therefore came with the introduction of vector clocks [10]. With vector clocks, comparing timestamps does in fact allow one to infer whether x happened before y, y happened before x, or if x and y are concurrent events that are not causally related. Logical vector clocks are presented in more details by means of an illustrated example in section 2.1.

1.2 Privacy issues raised by logical vector clocks

Confidentiality of message exchange is a desirable property of secure distributed systems: no other party than the intended recipient is able to read the content of a message. A stricter definition would also include the confidentiality of the existence of the message exchange itself: no other parties than the sender and the intended recipient should know that a message exchange actually took place.

Confidentiality of the message content can be ensured by well-known cryptographic techniques for secure and authenticated channels [17]. Such techniques are routinely used on the Internet for securing Web or email traffic, or on corporate networks for securing remote login sessions and access to databases. As such, ensuring the confidentiality of the message content is an orthogonal concern to that of determining causality in a distributed system using logical clocks. This is because systems of logical clocks work by piggybacking logical clocks on existing messages, and thus are not actually concerned with the content of the message.

This paper will therefore focus on preserving the privacy of the existence of the message exchange. Even without access to the content of messages, there are situations where sensitive information can be inferred from the mere existence of a message exchange between two parties. In an online bidding protocol, for example, observing communications provides information about the advancement of the negotiation between the bidder and potential suppliers.

Vector clocks can unfortunately be a great tool for attackers to learn about other communications. With vector clocks, each message exchanged between two processes carries a vector timestamp that contains the latest known value (to the sending process) of the local scalar clocks of every other process. Careful analysis of vector timestamps can therefore provide the receiving process with information about message exchanges that are taking place between the other processes. This breach of privacy will be explained in more details in 2.2.

The work of privacy-preserving vector clocks presented in this papers prevents this kind of information leakage in vector clocks based on secure computation techniques, i.e. purely in software and without using a trusted third party.

The remainder of the paper is structured as follows: We first review some background material on vector clocks and the cryptographic techniques of secure computation in Section 2. In Section 3 we describe the protocols for the basic operations of vector clocks. We validate the achievements of the protocols in Section 4 before reviewing related work in Section 5. Conclusions are summarized in Section 6.

2 Background on vector clocks and secure computation

This section introduces logical vector clocks in more details and presents the three operations on logical vector timestamps required to operate the protocol. We then explain what are the possible privacy breaches in the standard implementation of logical vector clocks. We finish by introducing the secure computation techniques that will be used in section 3 to provide a privacy-preserving version of the logical vector clock protocol.

2.1 Logical vector clocks

We will now introduce the operation of a system of logical vector clocks in sufficient details for the reader to understand what are the three basic operations that we propose to implement in a privacy-preserving manner.

Just like with the scalar logical clocks of Lamport [9], each process in a logical vector clock system keeps a local scalar clock whose value is modified upon the occurrence

of either an event internal to the process or the reception of a message from another process. In addition, logical vector clocks require that each process keeps an estimate of the value of the local scalar clock of each of the other processes.

As a result, each process ends up containing a number of scalar clocks equal to the total number *n* of processes: one local scalar clock and *n-1* estimates of the local scalar clocks of the other processes. If we arrange these scalar clocks in a vector we get the so-called vector clock of the process. The structure and protocol of operation of logical vector clocks are presented in all necessary details in [10]. Figure 2 presents an example of vector clocks with three processes. The local scalar clock of process *i* is, by convention, the i^{th} component of the vector clock of each process.

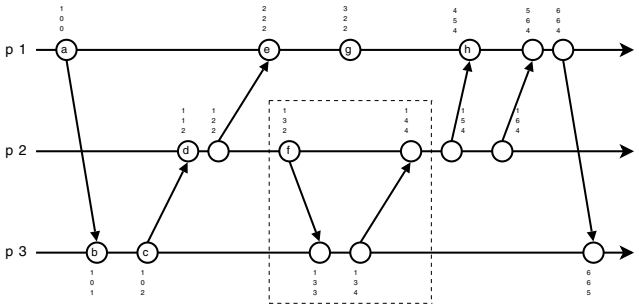

Figure 2. Breach of privacy with regular logical vector clocks

Two operations are required for operating a system of logical clocks, while a third operation is required for comparing two arbitrary timestamps and determining what the causality relationship between them is.

The INC operation increments the value of the local scalar clock by a fixed value[1] upon the occurrence of an event, be it an internal event or the reception of a message. In Figure 2, the INC operation is applied to the vector clock of $p3$ upon the occurrence of event c.

On reception of a message, the local vector clock is updated according to the vector clock that the sending process piggybacked to the message. Each component of the vector clock takes the value of the maximum of that component and the corresponding component of the vector clock received from the sender. We call this operation MAX.

Once logical vector timestamps have been produced for a set of events, there comes a time when someone, either a process or an external agent, will want to compare two logical timestamps in order to establish whether two events are causally connected, and if so which one happened first, or if the two events are concurrent. This operation uses a component-wise comparison of the values of the logical vector clocks, which we call COMP.

[1]The value of the increment is irrelevant and is usually set to 1. We will use this value throughout the remainder of this paper

2.2 Privacy breaches in regular vector clocks

With reference to Figure 2, we can now understand how privacy breaches may occur with regular vector clocks.

Let's consider the two events e and h which take place in process $p1$. Both events relate to the reception by process $p1$ of a message from process $p2$. Now, looking at the bottom two timelines of the diagram, we can see that, in-between sending these two messages to $p1$, process $p2$ engaged in a message exchange with process $p3$. The existence of this message exchange between $p2$ and $p3$ is something that both $p2$ and $p3$ may want to prevent $p1$ from learning about.

Unfortunately, by looking at the values of the vector clock associated with events e and h in $p1$, one can see how easy it is for $p1$ to conclude that a message exchange took place between $p2$ and $p3$: the two lower components of the vectors have both increased, respectively by 3 and 2. This cannot only be due to internal events at $p2$ and $p3$, otherwise the vector clock transmitted by $p2$ and received at event h would not show an increase in its bottommost component. Therefore, $p1$ now knows about the exchange of messages between $p2$ and $p3$, which is represented as a dashed box in Figure 2.

2.3 Secure Computation Techniques

Secure (multi-party) computation is a cryptographic technique that allows n parties to compute any function of m inputs ($m \leq n$) without revealing the inputs to each other or to a trusted third party. The final results of the computation will be available to the all the corresponding parties. However, a given party will only be able to compute what can be inferred from the final results and its own input. It was proven more than 20 years ago that theoretically any function can be computed with two parties [19], with multiple parties and computational security [6] or information-theoretic security [1]. It was however suggested in [7] that for important problems special solutions should be sought.

Our protocols work in the semi-honest model, where each participant is expected to follow the protocol, but keeps a record and tries to compute as much information as possible from this record. This matches a real implementation where timestamps are usually recorded in log files and one can access them any time he wishes, but the software for performing the protocol is often deeply buried in the stack, such that its modification is not easy. In addition, the legal context of the operation of a distributed system may make it acceptable for a party to try to gain any advantage they can from the information that they legitimately have

access to, but unacceptable to tamper with the operation of the distributed system for attaining the same goal.

Secure computation protocols build on top of two cryptography techniques: homomorphic encryption and oblivious transfer. We will now introduce these in turn, together with a solution to Yao's millionaires protocol. All these results will be used later in the design of our secure version of privacy-preserving logical vector clocks.

2.3.1 Homomorphic Encryption

Homomorphic encryption allows one operation on the cipher text to map to one homomorphic operation on the plain text. In particular, we require that multiplication (modulo a constant) of the cipher text results in addition (modulo a – potentially different – constant) of the plain text. Let $E_A(x)$ denote the encryption of x under Alice's (public) key. Thus

$$E_A(x) \cdot E_A(y) = E_A(x + y)$$

From this the following operation follows

$$E_A(x)^y = E_A(x \cdot y)$$

We furthermore require the encryption system to be public-key and semantically secure.

The properties of public-key systems [3] are common knowledge in the security community due to the wide acceptance of such systems, e.g. the RSA system [16].

A semantically secure system ensures, loosely speaking, that no information is leaked by a cipher text (and the public key) about its plain text. This is important for small domains in public key systems where an attacker could otherwise enumerate all possible plain texts and encrypt them. Here, the plain text domain is composed of the possible values of scalar logical clocks, i.e. integers. Semantic security is usually achieved by randomizing the encryption, such that one plain text can result in many different cipher texts.

Another important operation on homomorphic encryptions is rerandomization. Rerandomization allows the modification of a cipher text, such that the plain text is not changed, but the cipher texts cannot be linked any more, i.e. presented with two rerandomized ciphertexts one cannot tell whether they are from the same plain text or not without the decryption key. In homomorphic, semantically secure encryption systems, even without a special rerandomization operation, rerandomization can be achieved by multiplying with an encryption of the neutral element of the homomorphic operation, which amount to adding 0 to the plain text.

Furthermore, since we blind by adding a random number (secret share) in the homomorphic domain, this number needs to be uniformly chosen in the domain of the homomorphic operation. Therefore we require the modulus of the homomorphic domain to be public, i.e. it must not reveal the private key. This requirement rules out many popular homomorphic encryption systems, such as Paillier [13] or its modification by Damgard, Jurik [2]. An encryption system with the required properties is e.g. Naccache-Stern [11].

2.3.2 Oblivious Transfer

Oblivious Transfer (OT) is a cryptographic protocol that allows a receiver to receive one (a_b) of two values ($a_b \in \{a_0, a_1\}$) from a sender. The receiver will learn nothing about the other value ($a_{\neg b}$) and the sender will not learn which value (a_b or even just b) was picked.

OT was introduced by Rabin [14] and extended to 1-out-of-2 OT by Even et al [4]. In 1-out-of-2 OT the receiver can determine by a choice bit b which of the two values he wants to receive. The fastest implementation of OT (which we recommend to use) is described by Naor-Pinkas [12].

This implementation can also be very practically extended to 1-out-of-n OT (where $n > 2$). In this case the sender has n values for the receiver to choose from.

2.3.3 Yao's Millionaires' Protocol

In [19] Yao introduce the problem of computing the maximum with the following problem: Two millionaires want to compare their riches, but do not want to reveal to each other which is the exact amount of their wealth. He presented a first solution for this problem. A faster solution to the problem was introduced in [8]. We will recapitulate this solution before adapting it to the needs of the protocols of this paper.

Assume Alice has a value a and Bob has b. They want to compute $a \leq b$ without revealing their values to each other. If they have exchanged public keys $E_A(\cdot), E_B(\cdot)$ in the required homomorphic encryption systems, they can do so using the following protocol:

$$\begin{array}{lll} A \longrightarrow B & E_A(a) & \\ B & 0 \leq r' < r & \\ & E_A(c) = & (E_A(a) \cdot E_A(-b))^r \cdot E_A(-r') \\ & = & E_A(r \cdot (a-b) - r') \\ B \longrightarrow A & E_A(c) & \end{array}$$

It is easy to verify that $c \leq 0 \Leftrightarrow a \leq b$ holds. In this version of the protocol, only Alice has access to the answer.

2.3.4 Split Yao's Millionaires' Protocol

We will now modify the above protocol, such that Alice no longer obtains the result c, but such that the result is shared between Alice and Bob. Alice will have a bit c'' and Bob will have c', such that $c'' \oplus c' \Leftrightarrow a \leq b$ (where $\oplus$ denotes the "Exclusive-Or" operation).

Bob chooses a bit c' ($c' \in \{0, 1\}$) in step 2 above and computes

$$E_A(c) = E_A(-1^{c'} \cdot r \cdot (a-b) - r')$$

Alice computes $c'' = (c \leq 0)$ and then $c'' \oplus c' \Leftrightarrow a \leq b$ holds.

3 Proposed protocol for privacy-preserving vector clocks

A participant in the protocol is a process which keeps a virtual vector clock. For simplicity we will refer to the three participant processes as Alice, Bob and Charlie, abbreviated as usual as A, B and C. Each participant or process has its own public, private key pair, e.g. $E_A(\cdot), D_A(\cdot)$. During set-up before the processes begin to run and the protocols are executed the participants have exchanged public keys: $E_A(\cdot), E_B(\cdot), \ldots$. We furthermore assume secure and authenticated pair-wise channels.

Our protocols extend to an arbitrary number of parties but for the sake of explanation we will have a running example with three participants. Extensions to four or more parties follow simply from the given protocols.

A privacy-preserving vector clock timestamp is represented as an encrypted tuple of a regular vector clock timestamp:

$$E_A(t_A), E_B(t_B), E_C(t_C)$$

Each party's timestamp is encrypted under this party's public key, such that it is only accessible to itself, i.e. no timestamp value is being leaked to the other party.

3.1 Secure Increment

An increment of a vector clock is done locally at either party's site. Since this party possesses the decryption key, they can simply decrypt, add 1, and then encrypt again. For performance reason it is recommendable to operate on the cipher text directly using the homomorphic operation. A secure INC would then be performed as, e.g. at Bob's

$$E_A(t_A), E_B(t_B) \cdot E_B(1) = E_B(t_B + 1), E_C(t_C)$$

3.2 Secure Maximum

Each time a party send a message to another party, the receiving party needs to update its vector clock. Without loss of generality, assume Alice is sending a message to Bob. Let

$$A : E_A(t'_A), E_B(t'_B), E_C(t'_C)$$

denote the encrypted vector clock timestamp of Alice at the time of sending the message, and

$$B : E_A(t_A), E_B(t_B), E_C(t_C)$$

denote Bob's timestamp. Then the update is

$$A \longrightarrow B : E_A(\max(t'_A, t_A)), E_B(t_B), E_C(\max(t'_C, t_C))$$

Bob can simply take $E_B(t_B)$ from his timestamp and Alice does not send Bob's logical time in his vector clock timestamp, but a placeholder $E_B(0)$ instead, since it may reveal unwanted causal information between two messages otherwise. This section will focus on how Bob can compute $E_A(\max(t'_A, t_A))$ and $E_C(\max(t'_C, t_C))$.

Consider the first case of $E_A(\max(t'_A, t_A))$. Bob has received $E_A(t'_A)$ already and privately holds $E_A(t_A)$. He can engage in a split Yao's millionaires' protocol from Section 2.3.4 with Alice for computing $t'_A \leq t_A$:

$$A \longleftrightarrow B : (c' \oplus c'') = (t'_A \leq t_A)$$

Bob now has c' and Alice has c''. Bob uniformly chooses a random number r in the encryption domain of $E_A(\cdot)$ and prepares $E_A(t'_A) \cdot E_A(r) = E_A(t'_A + r)$ and $E_A(t_A) \cdot E_A(r) = E_A(t_A + r)$. He numbers them according to c':

$$A : \theta_{c'} = E_A(t_A + r) \qquad \theta_{\neg c'} = E_A(t'_A + r)$$

Alice and Bob engage in an 1-out-of-2 Oblivious Transfer protocol (see Section 2.3.2) with Bob as the sender of the pair $\{\theta_0, \theta_1\}$. Alice chooses $\theta_{c''}$ according to c''.

$$B \xrightarrow{\text{OT}} A : \{\theta_0, \theta_1\}_{c''}$$

It is not hard to verify that it follows that

$$B : \theta_{c''} = E_A(\max(t'_A, t_A) + r)$$

Note, that Alice does not know which the larger timestamp is, although she can decrypt $E_A(\max(t'_A, t_A)+r)$, since she is blinded by the secret sharing using r. Alice rerandomizes $\theta_{c''}$ to $\theta'_{c''}$, such that Bob cannot guess her choice and c'', and then sends $\theta'_{c''}$ to Bob.

$$A \longrightarrow B : \theta'_{c''} = \theta_{c''} \cdot E_A(0)$$

Finally, Bob can obtain the desired (encrypted) maximum:

$$B : E_A(\max(t'_A, t_A)) = \theta'_{c''} \cdot E_A(-r)$$

Bob repeats the same protocol with Charlie (replacing all occurrences of Alice by Charlie) and would do so for any other party completing the required vector clock timestamp.

Our protocol supports networks with message reordering where messages, e.g. from Alice to Bob, are not ordered FIFO. This can be the case over the Internet for UDP messages or multiple TCP connections in parallel. When the network guarantees FIFO delivery the first maximum protocol with Alice can be replaced by simply using $E_A(t'_A)$ as the maximum $E_A(\max(t'_A, t_A))$ and Bob has to execute only one maximum protocol with Charlie. This is because Alice, by construct of the logical vector clock protocol, always sends to Bob the newest version of her own logical scalar clock, which has a higher value than the value that Bob has as its own estimate of Alice's clock.

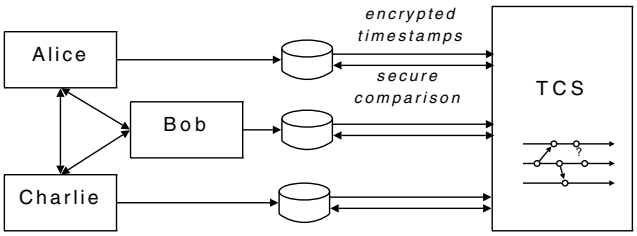

Figure 3. Secure creation and comparison of encrypted timestamps

3.3 Secure Comparison

In a deployment of a distributed system for the secure creation and comparison of timestamps (see Figure 3), we introduce a component called the Timestamp Comparison Service (TCS) whose responsibility is to reconstruct a view of the events that took place in a distributed computation and the causality relationship between them. Note that the TCS is purely here for the sake of convenience, it is not a trusted third party since it does not have access to private keys and requires the cooperation of all the participants to carry out its duties. In order to determine, for example, the causality relationship between the two events linked by a question mark in Figure 3, the TCS receives two encrypted vector clock timestamps:

$$TCS : E_A(t'_A), E_B(t'_B), E_C(t'_C)$$

$$TCS : E_A(t_A), E_B(t_B), E_C(t_C)$$

Again, we assume that Alice sent the first timestamp and Bob the second. For Bob's event to be caused by Alice's event, the following must hold

$$t'_A \leq t_A \wedge t'_B \leq t_B$$

The TCS engages in a split Yao's millionaires' protocol with Alice for $t'_A \leq t_A$. Let c'_A be Alice's part of the result and c''_A be TCS's.

$$A \longleftrightarrow TCS : (c'_A \oplus c''_A) = (t'_A \leq t_A)$$

The TCS does a similar protocol for $t'_B \leq t_B$ with Bob. Note that there is a variation of Yao's millionaires' protocol above to compute $t_B < t'_B$, where r' is chosen as a negative number ($0 \geq r' > -r$). Bob then just negates his partial result and the combined result equals $t'_B \leq t_B$. Let c'_B be Bob's part of the result and c''_B be the TCS's.

$$B \longleftrightarrow TCS : (c'_B \oplus c''_B) = (t'_B \leq t_B)$$

Alice forwards her partial result c'_A to Bob. Bob and the TCS need to compute the following formula

$$c'_A \oplus c''_A \wedge c'_B \oplus c''_B$$

This formula is similar to the main formula computed in [5]. We will use a similar protocol except for the output secret sharing. Bob prepares four values assuming each possible combination of the TCS's values.

$$B : \lambda_0 = c'_A \oplus 0 \wedge c'_B \oplus 0$$

$$B : \lambda_1 = c'_A \oplus 1 \wedge c'_B \oplus 0$$

$$B : \lambda_2 = c'_A \oplus 0 \wedge c'_B \oplus 1$$

$$B : \lambda_3 = c'_A \oplus 1 \wedge c'_B \oplus 1$$

Bob and the TCS now engage in 1-out-of-4 OT protocol with Bob as the sender of the four bits $\{\lambda_0, \lambda_1, \lambda_2, \lambda_3\}$. The TCS chooses according to c''_A and c''_B the element with the number $2c''_B + c''_A$.

$$B \xrightarrow{\text{OT}} TCS : \{\lambda_0, \lambda_1, \lambda_2, \lambda_3\}_{2c''_B + c''_A}$$

This completes the computation of the timestamp comparison. If the result is $false$ (0), i.e. Alice's event did not cause Bob's event, then the TCS computes with the help of Alice and Bob the formula below for the inverse comparison to check whether Bob's event caused Alice's event.

$$t_A \leq t'_A \wedge t_B \leq t'_B$$

If the result of this formula is $false$ as well, then the two events are concurrent.

4 Validation

In this section we perform an evaluation of the protocol described in the previous section. The protocols' achievements can be stated as two theorems.

Theorem 1. *Our protocols prevent any process from learning any causal relationship between any two distributed events e and f, i.e. either event e caused event f ($e \rightarrow f$) or event e did not cause event f ($e \nrightarrow f$).*

Proof. (Sketch) Theorem 1 follows from the semi-honest security of the protocols. In our ideal model the processes only know their local logical time and no information about other processes' logical clocks is transferred. We sketch the security proof in Section 4.1. □

Theorem 2. *The TCS can compute any causal relationship, i.e. event e caused event f ($e \rightarrow f$), event f caused event e ($e \leftarrow f$) or event e and f are concurrent ($e \sim f$).*

Proof. (Sketch) Theorem 2 follows from the correctness of the protocols. The correctness of our protocols follows from the correctness of vector clocks, and the reduction of vector clocks to the three operations (INC, MAX, COMP), and our protocols correctly implementing those three operations. The proof that our protocols correctly implement the three operations follows form their construction in Section 3. □

4.1 Privacy

The privacy of the protocols rests on four pillars, i.e. properties of the building blocks used.

The first pillar is semantic security of the homomorphic encryption system. Semantic security has been proven equivalent to cipher text indistinguishability and loosely speaking states that an adversary should not be able to compute any more function on the plain text given its cipher text and the encryption key than without those two. This prevents the cipher text from leaking any information, e.g. in the case of small encryption domains where the input can be guessed and then encrypted. The recipients of such cipher texts in our system are all parties or processes including the TCS, since all parties or processes have a timestamp which contains cipher texts of logical times of other parties.

The second pillar of our protocols is multiplicative blinding. We assume it is feasible to effectively hide a number (n) by multiplying it with a larger ($O(n^2)$) number. Furthermore we subtract another, smaller, random number to prevent factoring. One party in both versions of our Yao's millionaires' protocol will receive such a blinded number.

The third pillar is perfect secret sharing. Before choosing the maximum via OT in the Secure Maximum Protocol (Section 3.2) Bob builds two secret shares. One share he keeps securely with him and the other is chosen by Alice in the OT protocol. As the maximum is shared using this secret sharing scheme, neither Alice nor Bob know at this point of time in the protocol the maximum value. The semantic security of the homomorphic encryption scheme does not prevent Alice from getting her share, since she has the decryption key. The combination of the shares on Bob's side is protected by the homomorphic encryption scheme again, so that Bob only obtains an encrypted version of the maximum, but never the maximum itself.

The fourth and last pillar of our protocols is the security of the Oblivious Transfer protocol. Our assumption rests on the proofs that the protocols conform to the description from Section 2.3.2 even in the malicious case given in [4, 12]. In particular we recommend the implementation of [12].

The combination of these four security assumptions completes the security and privacy analysis of our protocol. Our protocols work in the semi-honest model where we assume the participants follow the protocols. Our focus is on the confidentiality of the timestamps, such that no eavesdropping on other connection is feasible.

4.2 Denial and Forgery

In [15] Reiter and Gong propose a method to prevent forgery of causal relationships, i.e. faking logical times when using vector clock timestamps. Note, that logical times might still be forged even though they are private. An attacker has access to all public keys and can enter and encrypt any value he intends to. If he guesses a logical time correctly, he can forge a causal relationship.

To prevent this each logical time is signed by its originator before sending it to any other party. Let $S_A(\cdot), S_B(\cdot), S_C(\cdot)$ denote the signed envelopes (i.e. signature and message) by Alice, Bob, and Charlie, respectively. A signed privacy-preserving vector clock timestamp is

$$S_A(E_A(t_A)), S_B(E_B(t_B)), S_C(E_C(t_C))$$

The signatures are verified before the parties engage in a secure maximum protocol, but there are no changes to the secure maximum protocol.

Other techniques by Reiter and Gong, such as the causality server or the conservative approach, are also complementary to prevent denial of causal relationships. The piggybacking algorithm works independently to (privacy-preserving) vector clocks.

4.3 Communication Cost

Using the Oblivious Transfer implementation of [12] our secure maximum protocol can be implemented as a two-round protocol. Alice sends the encrypted vector clock timestamp to Bob and then Bob and Charlie engage in a two-round protocol. One protocol has constant communication cost ($O(1)$), i.e. linear in a security parameter k, but the receiver of a message (Bob) needs to execute $n-1$ ($O(n)$) such protocols: one with each process except himself. Fortunately these protocols can be executed in parallel to any computation at Bob's side that does not involve sending or receiving messages. The overall round complexity remains constant therefore with a linear communication cost.

5 Related work

In [15] methods to prevent denial and forgery of causal relationships is presented. As mentioned in Section 4.2 these methods are complementary to our protocols. We therefore add privacy to the vector clocks preventing additional attacks to the methods in [15].

In [18] privacy of vector clocks has been achieved, but it comes at the cost of hardware support. It uses secure coprocessors to execute the code for vector clock timestamps wrapping the timestamps in encryption envelopes accessible only within the secure coprocessor. Our work does not protect in the case of malicious parties, since we do not have a secure execution environment, but we do not rely on hardware. Our protocols can be implemented in software alone similarly to secure and authenticated channels.

6 Conclusion

This paper presented a solution to the problem of preserving the privacy of the existence of message exchanges in the context of distributed systems where causality is determined by the use of logical vector clocks.

We first explained how some participants in a distributed computation may, by observing the vector clocks they receive from other processes, infer information about the existence of message exchanges they are not part of. We then broke down the vector clock protocol into three basic operations. For each of these operations, we proposed an implementation using secure computation techniques which prevents leakage of information while being strictly equivalent to the original protocol.

The solution we propose can be implemented at constant complexity per participant with an overall linear cost in communications. The closest known piece of previous art achieves a similar result but requires the use of specialized hardware. Our solution, by contrast, can be implemented in software only over existing communication channels.

Future work will address the optimization of the protocol for a number of well-known interaction patterns between processes in a distributed system. In addition, we are interested in investigating how this protocol can be introduced in a non-intrusive manner into existing distributed systems for the purpose of auditing and compliance.

References

[1] M. Ben-Or, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. *Proceedings of the 20th ACM symposium on theory of computing*, 1988.

[2] I. Damgard, and M. Jurik. A Generalisation, a Simplification and some Applications of Pailliers Probabilistic Public-Key System. *Proceedings of International Conference on Theory and Practice of Public-Key Cryptography*, 2001.

[3] W. Diffie, and M. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory 22(6)*, 1976.

[4] S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. *Communications of the ACM 28(6)*, 1985.

[5] O. Goldreich. Secure Multi-party Computation. Available at *www.wisdom.weizmann.ac.il/˜oded/pp.html*, 2002.

[6] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. *Proceedings of the 19th ACM conference on theory of computing*, 1987.

[7] S. Goldwasser. Multi party computations: past and present. *Proceedings of the 16th ACM symposium on principles of distributed computing*, 1997.

[8] F. Kerschbaum, and O. Terzidis. Filtering for Private Collaborative Benchmarking. *Proceedings of the International Conference on Emerging Trends in Information and Communication Security*, 2006.

[9] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21(7)*, 1978.

[10] F. Mattern. Virtual Time and Global States of Distributed Systems. *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, 1989.

[11] D. Naccache, and J. Stern. A New Public-Key Cryptosystem Based on Higher Residues. *Proceedings of the ACM Conference on Computer and Communications Security*, 1998.

[12] M. Naor, and B. Pinkas. Efficient Oblivious Transfer Protocols. *Proceedings of the symposium on data structures and algorithms*, 2001.

[13] P. Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. *Proceedings of EUROCRYPT*, 1999.

[14] M. Rabin. How to exchange secrets by oblivious transfer. *Technical Memo TR–81, Aiken Computation Laboratory*, 1981.

[15] M. Reiter, and L. Gong. Preventing Denial and Forgery of Causal Relationships in Distributed Systems. *Proceedings of the IEEE Symposium on Security and Privacy*, 1993.

[16] R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM 21(2)*, 1978.

[17] B. Schneier. Applied Cryptography, 2nd Edition. *John Wiley & Sons*, 1996.

[18] S. Smith, and D. Tygar. Security and Privacy for Partial Order Time. *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, 1994

[19] A. Yao. Protocols for Secure Computations. *Proceedings of the IEEE Symposium on foundations of computer science* 23, 1982.