Distributed runtime verification of metric temporal properties

Ritam Ganguly a,*, Yingjie Xue b, Aaron Jonckheere a, Parker Ljung b, Benjamin Schornstein b, Borzoo Bonakdarpour a,∗, Maurice Herlihy b

a Department of Computer Science and Engineering, Michigan State University, USA
b Department of Computer Science, Brown University, USA

A R T I C L E   I N F O

Keywords:
Metric temporal logic
Partial synchrony
Distributed systems
Runtime verification
Cross-chain protocols

A B S T R A C T

Distributed Systems are often composed of geographically separated components, where the clocks may not be perfectly synchronized. As such verifying the correctness of such system properties are a major challenge and are of utmost importance. In this paper, we describe a centralized runtime monitoring technique for distributed system. First, we propose a generalized runtime verification technique for verifying partially synchronous distributed computations for the metric temporal logic (MTL) by exploiting bounded-skew clock synchronization. Second, we introduce a progression-based formula rewriting scheme for monitoring MTL specifications which employs SMT solving techniques and report experimental results. Third, we also quantify each event according to the possible time of occurrence and calculate the probabilistic guarantee for generating the verification verdict. Lastly, we have implemented the entire procedure and report on extensive synthetic experimental results using UPPAAL, a set of cross-chain transactions implemented on Ethereum and an Industrial Control System of a water treatment plant.

1. Introduction

1.1. Motivation

With more and more applications and solutions going distributed, security and safety of these systems are of utmost importance. A distributed system typically comprises of multiple processes/components that do not share a global clock and while they attempt to accomplish a joint task. Few prominent examples of modern distributed systems include applications in blockchain: a smart contract to transfer assets automatically between two blockchains, industrial control systems (ICS): programmable logic controller (PLC) used to control the operations of a large manufacturing plant, and replicated data centers: maintaining read and write consistency of data. Given the complexity of these systems, deploying exhaustive verification techniques like model checking come at a high cost and there-by does not scale well to analyze the system’s correctness. On the other hand, testing only scrutinizes a subset of behaviors of the system. The inherent uncertainty about the exponential number of ordering of events makes testing distributed systems blind to concurrency bugs.

In this paper, we advocate for a runtime verification (RV) approach, to monitor the behavior of a system of distributed processes with respect to a set of temporal logic formulas. Applying RV to deal with multiple blockchains and multiple components of an ICS can be reduced to distributed RV, where a centralized or decentralized monitor observes the behavior of a distributed system in which processes do not share a global clock. Although RV deals with finite executions, the lack of a common global clock prohibits it from having a total unique ordering of events in a distributed setting. Put it another way, the monitor can only form a partial order of event which may result in different verification verdicts. Enumerating all possible partial ordering of events at run time incurs in an exponential blow up, making the approach not scalable. To add to this already complex task, most specifications for verifying blockchain smart contracts and ICS properties, come with a time bound. This means, not only the partial ordering of the events are at play when verifying, but also the actual physical time of occurrence of the events dictates the verification verdict.

1.2. Contributions

In this paper, we propose an effective, sound and complete solution to distributed RV for timed specifications expressed in the metric temporal logic (MTL) [30]. To present a high-level view of MTL, consider the two-party swap protocol [54] shown in Fig. 1. Alice and Bob, each in
possession of Apricot and Banana blockchain assets respectively, wants to swap their assets between each other without being a victim of a sore loser attack [54]. The protocol involves each party depositing premiums, with Alice and Bob depositing a premium of $p_A + p_B$ and $p_B$ on the Banana and Apricot blockchain, respectively. Once both the parties deposit the premium and a considerable time has passed, Alice and Bob are required to escrow their respective assets ($t_A$ and $t_B$) to the Apricot and Banana blockchains along with the private hash locks. This is followed by the redeem stage of the protocol where each party shares the public hash of the other party to redeem the asset.

There is a number of requirements that should be followed by the conforming parties to discourage any attack on themselves. We use metric temporal logic (MTL) [30] to express such requirements. One such requirement is, where Bob should not be able to redeem his asset before Alice redeems hers within eight time units can be represented by the MTL formula:

$$\varphi_{\text{spec}} = \neg \text{Apr. Redeem}(bob) \land (0,8) \text{Ban. Redeem(alice)},$$

where, $\text{Apr. Redeem}(bob)$ (resp. $\text{Ban. Redeem(alice)}$) are predicates that indicate if Bob (resp. Alice) has redeemed their assets from the corresponding blockchains, Apricot (resp. Banana) and $U_{(0,8)}$ is the until operator that makes sure the left side is continuously true until the right side is true, within the time interval $0, 8$.

We consider a fault proof central monitor which has the complete view of the system but has no access to a global clock. In order to limit the blow-up of states posed by the absence of a global clock, we make a practical assumption about the presence of a bounded clock skew $\epsilon$ between the local clocks of every pair of processes, guaranteed by a clock synchronization algorithm (e.g. NTP [39]). This setting is known to be partially synchronous when we do not assume the presence of a global clock and limit the impact of asynchrony within clock drifts. Such an assumption limits the window of partial orders of events only within $\epsilon$ time units and significantly reduces the combinatorial blow-up caused by nondeterminism due to concurrency. Existing distributed RV techniques either assume a global clock when working with time sensitive specifications [7,53] or use untimed specifications when assuming partial synchrony [22,41].

Our first contribution in this paper is an SMT-based progression-based formula rewriting technique over distributed computations which takes into consideration the events observed thus far to rewrite the specifications for future extensions. Our monitoring algorithm accounts for all possible orderings of events without explicitly generating them when evaluating MTL formulas. For example, in Fig. 2, we observe the events and the corresponding time of occurrence in the two blockchains,

---

1 A sore loser attack is a type of attack in cross-blockchain commerce. It occurs when one party decides to halt participation partway through, leaving other parties’ assets locked up for a long duration.

2 Satisfiability modulo theory (SMT) is the problem of determining whether a formula involving Boolean expressions comprising of more complex formulas involving real numbers, integers, and/or various data structures is satisfiable.
Table 1: Probability calculations for evaluated results.

<table>
<thead>
<tr>
<th>Evaluated Result</th>
<th>Time of occurrence of Deposit($p_3$)</th>
<th>Time of occurrence of Deposit($p_3' + p_1$)</th>
<th>Probability</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_{sync}$</td>
<td>2, 3</td>
<td>3</td>
<td>0.75 × 0.25</td>
<td>0.1875</td>
</tr>
<tr>
<td>$v_{async}$</td>
<td>2, 3, 4</td>
<td>4</td>
<td>1 × 0.5</td>
<td>0.5625</td>
</tr>
<tr>
<td>$v_{sync}$</td>
<td>2, 3, 4</td>
<td>5</td>
<td>0.25 × 0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>

probability of obtaining verdicts true and false are both 0.6875. An interesting note here is that the sum of probabilities for obtaining verdicts true and false is not 1. This is because of the case, when the time of occurrence of both $\text{Ban.Re redeem}(\text{alice})$ and $\text{Apr.Re redeem}(\text{bob})$ was considered to be 6, either order of occurrence of the events is possible. Thereby making, both verdicts, equally likely.

We have fully implemented our technique and report the results of rigorous experiments on monitoring synthetic data, using benchmarks in the tool UPPAAL [35], as well as monitoring correctness, liveness and conformance conditions for smart contracts on blockchains. We put our monitoring algorithm to test, studying the effect of different parameters on the runtime and report on each of them.

1.3. Organization

A preliminary version of this paper appeared in the 42nd International Conference on Distributed Computing Systems (ICDCS) in 2022 [23]. This paper only included the progression-based evaluation approach (our first contribution). This paper extends the ICDCS’22 paper in multiple fronts. First, the notion of evaluating probabilistic guarantee of the verdict is completely new. This includes the notion of estimating the offset distribution and calculating the probability of a verdict using the SMT formulation. Secondly, this technique is also fully implemented and its performance is rigorously analyzed w.r.t. other parameters. Finally, we add another case study where we use our approach to monitor safety properties of a Secure Water Treatment Plant (SWaT).

The rest of the paper is organized as follows. Section 2 presents the background concepts. Formal statement of our RV problem is discussed in Section 3. The formula progression rules, the SMT-based solution and calculating the probabilistic guarantees are discussed in Sections 4, 5 and 6, respectively, while experimental results are analyzed in Section 7. Related work is discussed in Section 8 before we make concluding remarks in Section 9. More details about required preprocessing on the system and our case studies can be found in the Appendix, Section Appendix B and Section Appendix C respectively.

2. Preliminaries

In this section, we present an overview of the distributed computation and the metric temporal logic (MTL).

2.1. Distributed computation

A distributed system (e.g., a system of multiple blockchains) can be modeled as a loosely coupled asynchronous system, consisting of $n$ reliable processes (that do not fail), denoted by $P = \{P_1, P_2, \ldots, P_n\}$. As a system, the processes do not share any memory or have a common global clock. Channels are assumed to be FIFO and lossless. In our model, we represent each local state change and a message activity (send or receive) by an event. Message passing does not change the state of the process and we disregard the content of the message as it is of no use for our monitoring technique. Here, we refer to a global clock which will acts as the “real” time keeper. It is to be noted that the presence of this global clock is just for theoretical reasons and it is not available to any of the individual processes.

We make an assumption about a partially synchronous system. For each process $P_i$, where $i \in [1, n]$, the local clock can be represented as a monotonically increasing function $c_i : Z_{\geq 0} \rightarrow Z_{\geq 0}$, where $c_i(G)$ is the value of the local clock at global time $G$. Since we are dealing with discrete-time systems, for simplicity and without loss of generality, we represent time with non-negative integers $Z_{\geq 0}$. For any two processes $P_i$ and $P_j$, where $i \neq j$, we assume:

$$v \in Z_{\geq 0} | c_i(G) - c_j(G) < \epsilon.$$  

where $\epsilon > 0$ is the maximum clock skew. The value of $\epsilon$ is constant and is known to the monitor. This assumption is met by the presence of a clock synchronization algorithm, like NTP [39], to ensure bounded clock skew among all processes. We denote an event on process $P_i$ by $e_i'$, where $e = c_i(G)$. That is the local time of occurrence of the event at some global time $G$.

Definition 1. A distributed computation consisting of $n$ processes is represented by the pair $(E, \rightarrow)$, where $E$ is a set of events partially ordered by Lamport’s happened-before ($\rightarrow$) relation [33], subject to the partial synchrony assumption:

- In every $P_i$, $1 \leq i \leq n$, all events are totally ordered:

  $$\forall \sigma, \sigma' \in Z_{\geq 0}, (\sigma < \sigma') \rightarrow (e_{\sigma} \rightarrow e_{\sigma'}).$$

- If $e$ is a message sending event in a process and $f$ is the corresponding message receiving event in another process, then we have $e \rightarrow f$;

- For any two processes $P_i$ and $P_j$ and two corresponding events $e_i', e_j' \in E$, if $e + e_i' < e_j'$ then, $e_i' \rightarrow e_j'$, where $e$ is the maximum clock skew, and

- If $e \rightarrow f$ and $f \rightarrow g$, then $e \rightarrow g$. □

Definition 2. Given a distributed computation $(E, \rightarrow)$, a subset of events $C \subseteq E$ is said to form a consistent cut if and only if when $C$ contains an event $e$, then it should also contain all such events that happened before $e$. Formally,

$$\forall e \in E : (e \in C) \wedge (f \rightarrow e) \rightarrow f \in C. \quad \Box$$

The frontier of a consistent cut $C$, denoted by front($C$) is the set of all events that happened last in each process in the cut. That is, front($C$) is a set of $e_i^\text{last}$ for each $i \in [1, |P|]$ and $e_i^\text{last} \in C$. We denote $e_i^\text{last}$ as the last event in $P_i$ such that $\forall e'_i \in C. (e'_i \neq e_i^\text{last}) \rightarrow (e'_i \rightarrow e_i^\text{last}).$

2.2. Metric temporal logic (MTL) [2,3]

Let $\mathbb{B}$ be a set of nonempty intervals over $Z_{\geq 0}$. We define an interval, $I$, to be

$$(\begin{array}{c}
\text{start, end} \\
\text{[start, end]} \triangleq [a \in Z_{\geq 0} | \text{start} \leq a < \text{end}]
\end{array})$$

where $\text{start} \in Z_{\geq 0}, \text{end} \in Z_{\geq 0} \cup \{\infty\}$ and $\text{start} < \text{end}$. We define $\mathbb{A}$ as the set of all atomic propositions, and $\Sigma = 2^{\mathbb{A}}$ as the set of all possible states. A $\text{trace}$ is represented by a pair which consists of a sequence of states, denoted by $a = s_0, s_1, \ldots$, where $s_i \in \Sigma$ for every $i > 0$ and a sequence of non-negative numbers, denoted by $\bar{r} = r_0, r_1, \ldots$, where $r_i \in Z_{\geq 0}$ for all $i > 0$. We represent the set of all infinite traces by a pair of infinite sets, $(2^\Sigma, Z_{\geq 0})$. The trace $s_0, s_1, \ldots$ (resp. $r_0, r_1, \ldots$) is represented by $a^\Delta$ (resp. $\bar{r}^\Delta$). For an infinite trace $a = s_0, s_1, \ldots$ and $\bar{r} = r_0, r_1, \ldots$, $\bar{r}$ is an increasing sequence, meaning $r_{i+1} \geq r_i$, for all $i \geq 0$.
Fig. 3. Different time interleaving of events.

Syntax

The syntax of metric temporal logic (MTL) for infinite traces is defined by the following grammar:

\[
\varphi ::= p | \neg \varphi | \varphi_1 \lor \varphi_2 | \varphi_1 \land \varphi_2 | \varphi_1 U \varphi_2
\]

where \( p \in AP \) and \( U \) is the ‘until’ temporal operator with time bound \( I \).

We also have \( true = p \lor \neg p \), \( false = \neg true \), \( p \lor \neg p = \varphi_1 \lor \varphi_2 = \neg \neg \varphi_1 \lor \neg \varphi_2 \), \( \bigcirc \varphi = true U \varphi \) (“eventually”) and \( \square \varphi = \neg \bigcirc \neg \varphi \) (“always”). The set of all MTL formulas is denoted by \( \Phi_{MTL} \).

Semantics

The semantics of metric temporal logic (MTL) is defined over \( a = s_0s_1 \ldots \) and \( \bar{t} = t_0t_1 \ldots \) as follows:

\[
\begin{align*}
(a, \bar{t}, i) &\models p & \text{if } p \in s_i \\
(a, \bar{t}, i) &\models \neg \varphi & \text{if } (a, \bar{t}, i) \not\models \varphi \\
(a, \bar{t}, i) &\models \varphi_1 \lor \varphi_2 & \text{if } (a, \bar{t}, i) \models \varphi_1 \lor (a, \bar{t}, i) \models \varphi_2 \\
(a, \bar{t}, i) &\models \varphi_1 U \varphi_2 & \text{if } \exists j \geq i. t_j - t_i \in I \land (a, j, \bar{t}) \models \varphi_2 \\
& \quad \land \forall k \in [i, j), (a, k, \bar{t}) \models \varphi_1 \\
\end{align*}
\]

Also, \( (a, \bar{t}) \models \varphi \) holds if and only if \( (a, \bar{t}, 0) \models \varphi \).

In the context of RV, we introduce the notion of finite MTL. The truth values are represented by the set \( B(z) = \{ T, L \} \), where \( T \) (resp. \( L \)) represents a formula that is satisfied (resp. violated) given a finite trace. We represent the set of all finite traces by a pair of finite sets, \( (\Sigma^*, \Sigma_0^*) \).

For a finite trace, \( a = s_0s_1 \ldots s_n \) and \( \bar{t} = t_0t_1 \ldots t_n \), the only semantic that needs to be redefined is that of \( U \) (“until”) and is as follows:

\[
[(a, \bar{t}, i) \models_F \varphi_1 U \varphi_2] =
\begin{cases}
T & \text{if } \exists j \in [i, n]. t_j - t_i \in I \land (a^F, j, \bar{t}) = T \\
L & \text{otherwise} \\
\end{cases}
\]

In order to further illustrate the difference between MTL and finite MTL, consider formula \( \varphi = \bigcirc \varphi \) and a trace \( a = s_0s_1 \ldots s_n \) and \( \bar{t} = t_0t_1 \ldots t_n \). We have \( (a, \bar{t}) \models \varphi \) since \( \varphi \) is satisfied for each \( j \in [0, n] \), we have \( t_j - t_0 \in I \) and \( p \in s_j \), otherwise \( L \). Now, consider formula \( \varphi = \bigcirc \varphi \). We have \( (a, \bar{t}) \models \varphi \) if and only if for some \( j \in [0, n] \), we have \( t_j - t_0 \in I \) and \( p \in s_j \), otherwise \( L \).

3. Formal problem statement

In a partially synchronous system, there are different possible ordering of events and each unique ordering of events [10] might evaluate to different RV verdicts. Let \( (\cdot, \cdot) \) be a distributed computation. A sequence of consistent cuts is of the form \( C_0C_1C_2 \ldots \), where for all \( i \geq 0 \), we have \( 1 \leq |C_i| \leq |C_{i+1}| \) and \( (C_0) = \emptyset \). The set of all sequences of consistent cuts is denoted by \( C \).

We note that in our view, the time interval \( I \) in the syntax of MTL represents the physical (global) time \( G \). Thus, when deriving all the possible traces given the distributed computation \( (\cdot, \cdot) \), we account for all different orders in which the events could possibly occur with respect to \( G \). This involves replacing the local time of occurrence of an event, \( c'_e \), with the set of events \( \{ e'_j, i' \in \max(0, \sigma - e + 1), \sigma + e) \} \). This is to account for the maximum clock drift that is possible on the local clock of a process when compared to the global clock. For example, given the computation in Fig. 3, a maximum clock skew \( e = 2 \) and a MTL formula, \( \varphi = a U_1 b \) one has to consider all possible traces including \( (a, 2)(a, 2)(\forall b)(\forall c) \models \varphi \) and \( (a, 1)(a, 2)(\forall b)(\forall c) \not\models \varphi \).

Given a sequence of consistent cuts, it is evident that for all \( j > 0 \), \( C_{j-1}C_j \) is the last event that was added onto the cut \( C_j \). To translate monitoring of a distributed system into monitoring a trace with guarantees, we define a sequence of natural numbers as \( \bar{x} = x_0x_1 \ldots \), where \( x_0 = 0 \) and for each \( j \geq 1 \), we have \( x_j = \sigma \), such that \( front(C_j) - front(C_{j-1}) = \{ e'_j \} \). To maintain time monotonicity, we only consider sequences where for all \( i \geq 0 \), \( n_i \geq n_{i+1} \).

The set of all traces that can be formed from \( (\cdot, \cdot) \) is defined as:

\[
Tr(\cdot, \cdot) = \{ (front(C_0), front(C)_1, \ldots) \mid C_0C_1 \ldots \in C \}
\]

In the sequel, we assume that every sequence \( a \) of frontiers in \( Tr(\cdot, \cdot) \) is associated with a sequence \( \bar{x} \). Thus, to comply with the semantics of MTL, we refer to the elements of \( Tr(\cdot, \cdot) \) by pairs \((a, \bar{x})\). Thus, we evaluate an MTL formula \( \varphi \) with respect to a computation \((\cdot, \cdot)\) as follows:

\[
[(\cdot, \cdot) \models_F \varphi] = \{(a, \bar{x}) \models_F \varphi \mid (a, \bar{x}) \in Tr(\cdot, \cdot)\}
\]

In reality, the observed clock skew between any pair of processes is typically less than the (worst-case) maximum clock skew. To get a better understanding of the clock skew, we run some diagnostic tests based on the technique in [18] (described in Section Appendix B of the supplementary document) to estimate the CDF defining the clock skew. Here, let assumes that the CDF is represented by a function \( pr(E, \bar{x}) \), such that given an event, \( e'_j \), the function gives us the probability that the event took place at global time \( \bar{x} \). The probability of occurrence of a trace \((a, \bar{x})\) is straightforwardly calculated as follows:

\[
P(a, \bar{x}) = \prod_{j=0}^{\bar{x}} pr(e'_j, \bar{x}).
\]

where \( front(C_j) - front(C_{j-1}) = \{ e'_j \} \). Thus, computing the set of all verdicts, \( [(\cdot, \cdot) \models_F \varphi] \) boils down to having a set of verdicts and the corresponding probability of generating it, since a distributed computation may involve several traces and each trace might evaluate to a different verdict.

The probabilistic guarantee associated with a verdict is calculated as the product of the probability of each event occurring at the time considered when generating the trace:

\[
P \left( (\cdot, \cdot) \models_F \varphi \right) = T \sum_{(a, \bar{x}) \in Tr(\cdot, \cdot)} P(a, \bar{x})
\]

such that \( (a, \bar{x}, 0) \models_F \varphi \) = \( T \). The probability of obtaining verdict \( \bot \) is calculated similarly.

Overall idea of our solution

To solve the above problem (evaluating all possible verdicts), we propose a monitoring approach based on formula-rewriting (Section 4) and SMT solving (Section 5). Our approach involves iteratively (1) chopping a distributed computation into a sequence of smaller segments to reduce the problem size, (2) progress the MTL formula for each segment for the next segment, which results in a new MTL formula by invoking an SMT solver and (3) calculate the sum of the probability of each trace that yield the same MTL formula. Since each computation/segment corresponds to a set of possible traces due to partial synchrony, each invocation of the SMT solver may result in a different verdict.

4. Formula progression for MTL

We start describing our solution by explaining the formula progression technique. We define a function \( Pr \), that takes a finite trace, \((a, \bar{x})\), and a MTL specification, \( \varphi \), as input and rewrites the MTL specification to \( \varphi' \), such that all extensions of the trace, \((a', \bar{x}')\), only need to verify with respect to the rewritten formula, \( \varphi' \).

Definition 3. A progression function is of the form \( Pr : \Sigma^* \times Z^*_0 \times \Phi_{MTL} \rightarrow \Phi_{MTL} \) and is defined for all finite traces \((a, \bar{x}) \in (\Sigma^*, Z^*_0)\), finite traces \((a', \bar{x}') \in (\Sigma^*, Z^*_0)\) and MTL formulas \( \varphi \in \Phi_{MTL} \), such that \((a, a', \bar{x}, \bar{x}') \models \varphi\) if and only if \((a', \bar{x}') \models Pr(a, \bar{x}, \varphi)\).
Algorithm 1 Always.
1: function Pr(a, ⋄, φ)
2: if Istart ≤ τi - τ0 then
3: if Iend ≥ τi - τ0 then
4: return \( \bigwedge_{i \in [0, a]} (\text{Inint}(i) \rightarrow \text{Pr}(a', \bar{t}, \varphi)) \)
5: else
6: return \( \bigwedge_{i \in [0, a]} (\text{Inint}(i) \land \text{Pr}(a', \bar{t}, \varphi)) \land \bigodot_{t < \tau_i - \tau_0} \varphi \)
7: end if
8: else
9: return \( \bigodot_{t < \tau_i - \tau_0} \varphi \)
10: end if
11: end function

Algorithm 2 Eventually.
1: function Pr(a, ⋄, \( \mathcal{O}_i \), φ)
2: if Istart ≤ τi - τ0 then
3: if Iend ≥ τi - τ0 then
4: return \( \bigwedge_{i \in [0, a]} (\text{Inint}(i) \land \text{Pr}(a', \bar{t}, \varphi)) \)
5: else
6: return \( \bigwedge_{i \in [0, a]} (\text{Inint}(i) \land \text{Pr}(a', \bar{t}, \varphi)) \lor \bigodot_{t < \tau_i - \tau_0} \varphi \)
7: end if
8: else
9: return \( \bigodot_{t < \tau_i - \tau_0} \varphi \)
10: end if
11: end function

Compared to the classic formula rewriting technique in [27], here the function Pr takes a finite trace as input, while the algorithm in [27] rewrites the formula after every observed state. When monitoring a partially synchronous distributed system, multiple verdicts are possible as a result of no unique ordering of events, as a result the classical state-by-state formula rewriting technique is of little use. The motivation of our approach comes from the fact that for computation reasons, we chop the computation into smaller segments and the verification of each segment is done through an SMT query. A state-by-state approach would incur a huge number of SMT queries being generated.

Let \( I = [\text{start}, \text{end}] \) denote an interval. By \( I - \bar{t} \), we mean the interval \( I' = [\text{start}, \text{end}', \bar{t}] \), where \( \text{start}' = \max(0, \text{start} - \bar{t}) \) and \( \text{end}' = \max(\text{end}, 0 - \bar{t}) \). Also, for two time instances \( \tau_i \) and \( \tau_0 \), we let Inint(\( i \)) return true or false depending upon whether \( \tau_i - \tau_0 \) is in \( I \).

Progressing atomic propositions. For an MTL formula of the form \( \varphi = \varphi_1 \lor \varphi_2 \), we have \( \text{Pr}(a, \bar{t}, \varphi) = \text{Pr}(a, \bar{t}, \varphi_1) \lor \text{Pr}(a, \bar{t}, \varphi_2) \). Theorem 1. Given a trace, \( (a, \bar{t}) \) and an MTL formula, \( \varphi \), with \(|\varphi| = p \) number of sub-formulas and\( \text{end} = \sum_{i=1}^{p} \text{end}_i \), the sum of all end time intervals of all sub-formulas, \( \varphi_i \) where \( i \in [1, p] \), the time complexity of evaluating the MTL formula is given by \( O(|\varphi|, \max\{\text{end}, |a|\}) \).

Proof. Considering \( |a| > 0 \). Let us divide the proof into multiple cases, each for different kinds of formulas.
Case 1: \( \varphi = \varphi_1, \varphi_2 \) is a disjunction of atomic propositions. Let \( \varphi = \varphi_1 \lor \varphi_2 \), Apart from the trivial cases, the result of progression of \( \varphi_1 \lor \varphi_2 \) is based on progression of \( \varphi_1 \) and/or progression of \( \varphi_2 \):

\[
\text{Pr}(a, \bar{t}, \varphi) = \begin{cases} 
\text{true} & \text{if } \text{Pr}(a, \bar{t}, \varphi_1) = \text{true} \lor \text{Pr}(a, \bar{t}, \varphi_2) = \text{true} \\
\text{false} & \text{if } \text{Pr}(a, \bar{t}, \varphi_1) = \text{false} \land \text{Pr}(a, \bar{t}, \varphi_2) = \text{false} \\
\varphi'_1 & \text{if } \text{Pr}(a, \bar{t}, \varphi_1) = \text{false} \land \text{Pr}(a, \bar{t}, \varphi_2) = \varphi'_2 \\
\varphi'_2 & \text{if } \text{Pr}(a, \bar{t}, \varphi_2) = \text{false} \land \text{Pr}(a, \bar{t}, \varphi_1) = \varphi'_1 \\
\varphi'_1 \lor \varphi'_2 & \text{if } \text{Pr}(a, \bar{t}, \varphi_1) = \varphi'_1 \land \text{Pr}(a, \bar{t}, \varphi_2) = \varphi'_2 
\end{cases}
\]

Always and eventually operators. As shown in Algorithms 1 and 2, the progression for ‘always’, \( \mathcal{Q} \), and ‘eventually’, \( \mathcal{O} \), depends on the value of Inint(i) and the progression of the inner formula \( \varphi \). In Algorithms 1 and 2, we divide the algorithm into three cases: (1) line 4, corresponds to \( I \) is within the sequence \( \bar{t} \); (2) line 6, corresponds to where \( I \) starts in the current trace but its end is beyond the boundary of the sequence \( \bar{t} \); and (3) line 9, corresponds to if the entire interval \( I \) is beyond the boundary of sequence \( \bar{t} \). In Algorithm 1, we are only concerned about the progression of \( \varphi \) on the suffix \( (a', \bar{t}') \) if \( \text{Inint}(i) = \text{true} \). In case, \( \text{Inint}(i) = \text{false} \) the consequent drops and the entire condition equates to true. In other words, equating over all \( i \in [0, |a|] \), we are only left with conjunction of \( \text{Pr}(a', \bar{t}', \varphi) \) when \( \text{Inint}(i) = \text{true} \). In addition to this, we add the initial formula with updated interval for the next trace. Similarly, in Algorithm 2, equating over all \( i \in [0, |a|] \), if \( \text{Inint}(i) = \text{false} \) the corresponding \( \text{Pr}(a', \bar{t}', \varphi) \) is disregarded and the final formula is a disjunction of \( \text{Pr}(a', \bar{t}', \varphi) \) with \( \text{Inint}(i) = \text{true} \).

Progressing the until operator. Let the formula be of the form \( \varphi_1 \mathcal{U} \varphi_2 \). According to the semantics of until, \( \varphi_2 \) should be evaluated to true in all states leading up to some \( i \in I, \varphi_2 \) evaluates to true. We start by progressing \( \varphi_1 \) (resp. \( \varphi_2 \)) as \( \bigwedge_{i \in [0, \bar{t}]} \varphi_1 \) (resp. \( \bigwedge_{i \in [0, \bar{t}]} \varphi_2 \)) for some \( i \in I \). Since, we are only verifying the sub-formula, \( \bigwedge_{i \in [0, \bar{t}]} \varphi_2 \), on the trace sequence \( (a, \bar{t}) \), it is equivalent to verifying the sub-formula \( \bigwedge_{i \in [0, \bar{t}]} \varphi_2 \equiv \varphi_2 \) over the trace sequence \( (a, \bar{t}) \). Similar to Algorithms 1 and 2, in Algorithm 3 we need to consider three cases. In lines 4, 6 and 9, following the semantics of until operator, we make sure for all \( i \in [0, |a|] \), if \( \tau_i < I_{\text{start}} + \tau_0 \), \( \varphi_i \) is satisfied in the suffix \( (a', \bar{t}') \). In addition to this there should be some \( j \in [0, |a|] \) for which if \( \text{Inint}(j) = \text{true} \), then the trace satisfies the sub-formula \( \bigwedge_{i \in [0, \tau_i - \tau_0]} \varphi_i \) and \( \bigwedge_{i \in [0, \tau_i - \tau_0]} \varphi_i \). In lines 6 and 9, we also accommodate for future traces satisfying the formula \( \varphi_1 \mathcal{U} \varphi_2 \) with updated intervals.

5. SMT-based solution
5.1. SMT entities
SMT entities represent the variables used to represent the distributed computation. After we have the verdicts for each of the individual
Algorithm 1: Until

1. function PR(a, f, ρ, U, V)
2. if Istart ≤ f − t_i then
3. if Istart ≤ t_f then
4. return \( \bigwedge_{i \in \mathcal{I}} (\mathcal{E}) \)
5. else
6. return \( \bigwedge_{i \in \mathcal{I}} (\mathcal{E}) \)
7. end if
8. else
9. return \( \bigwedge_{i \in \mathcal{I}} (\mathcal{E}) \)
10. end if
11. end function

sub-formulas, we use the progression laws discussed in Section 4 to construct the formula for the future computations.

Distributed Computation We represent a distributed computation \((\mathcal{E}, \rightarrow)\) by a function \(f : \mathcal{E} \to \{0, 1, \ldots, |\mathcal{E}| - 1\}\). To represent the happen-before relation, we define a \(\mathcal{E} \times \mathcal{E}\) matrix called hbSet where \(\text{hbSet}(e_i, e_j) = 1\) represents \(e_i \rightarrow e_j\), for \(e_i, e_j \in \mathcal{E}\). Also, if \(|\sigma - \sigma'| \geq c\) then \(\text{hbSet}(\sigma, \sigma') = 0\). This is done in the pre-processing phase of the algorithm and in the rest of the paper, we represent events by the set \(\mathcal{E}\) and a happen-before relation by \(\rightarrow\) for simplicity.

In order to represent the possible time of occurrence of an event, we define a function \(\delta : \mathcal{E} \to Z_{\geq 0}\), where

\[\forall e_i \in \mathcal{E}, \exists e'_i \in \{\max(0, \sigma - c + 1), \sigma + c - 1\}, \delta(e'_i) = \sigma^i\]

To connect events, \(\mathcal{E}\), and propositions, \(\mathcal{P}\), on which the MTL formula \(\varphi\) is constructed, we define a boolean function \(\mu : \mathcal{P} \times \mathcal{E} \to \{\text{true}, \text{false}\}\). For formulas involving non-boolean variables (e.g., \(x_1 + x_2 \leq 7\)), we can update the function \(\mu\) accordingly. We represent a sequence of consistent cuts that start from {} and end in \(\mathcal{E}\), we introduce an uninterpreted function \(\rho : Z_{\geq 0} \rightarrow 2^{\mathcal{E}}\) to reach a verdict, given it satisfies all the constrains explained in 5.2. Lastly, to represent the sequence of time associated with the sequence of consistent cuts, we introduce a function \(\tau : Z_{\geq 0} \rightarrow Z_{\geq 0}\).

5.2. SMT constraints

Once we have the necessary SMT entities, we move onto including the constraints for both generating a sequence of consecutive cuts and also representing the MTL formula as a SMT constraint.

**Consistent cut constraints over \(\rho\):** In order to make sure the sequence of cuts represented by the uninterpreted function \(\rho\) is a sequence of consistent cuts, i.e., they follow the happen-before relations between events in the distributed system:

\[\forall i \in [0, |\mathcal{E}|], \forall e_i, e'_i \in \mathcal{E}, (e'_i \rightarrow e) \land (e \in \rho(i)) \rightarrow (e'_i \in \rho(i))\]

Next, we make sure that in the sequence of consistent cuts, the number of events present in a consistent cut is one more than the number of events that were present in the consistent cut before it:

\[\forall i \in [0, |\mathcal{E}|], |\rho(i + 1)| = |\rho(i)| + 1\]

Next, we make sure that in the sequence of consistent cuts, each consistent cut includes all the events that were present in the consistent cut before it, i.e., it is a subset of the consistent cut prior in the sequence:

\[\forall i \in [0, |\mathcal{E}|], \rho(i) \subseteq \rho(i + 1)\]

The sequence of consistent cuts starts from {} and ends at \(\mathcal{E}\).

\(\rho(0) = \emptyset, \rho(|\mathcal{E}|) = \mathcal{E}\)

The sequence of time reflects the time of occurrence of the event that has just been added to the sequence of consistent cut:

\[\forall i \geq 1, \tau(i) = \delta(e'_i),\text{ such that }\rho(i) - \rho(i - 1) = \{e'_i\}\]

We make sure the monotonicity of time is maintained in the sequence of time

\[\forall i \in [0, |\mathcal{E}|], \tau(i + 1) \geq \tau(i)\]

**Constraints for MTL formulas over \(\rho\):** These constraints will make sure that \(\rho\) will not only represent a valid sequence of consistent cuts but also makes sure that the sequence of consistent cuts satisfy the MTL formula. As is evident, a distributed computation can often yield two contradicting evaluation. Thus, we need to check for both satisfaction and violation for all the sub-formulas in the MTL formula provided. Note that monitoring any MTL formula using our progression rules will result in monitoring sub-formulas which are atomic propositions, eventually and globally temporal operators. Below we mention the SMT constrain for each of the different sub-formula. Violation (resp. satisfaction) for atomic proposition and eventually (resp. globally) constrain will be the negation of the one mentioned.

\[\varphi = \bigvee_{e \in \mathcal{E}} \mu(e) = \text{true}, \text{for } e \in \mathcal{P} \text{ (satisfaction, i.e., T)}\]

\[\varphi = \bigcap_{e \in \mathcal{E}} \mu(e) = \text{true}, \text{for } e \in \mathcal{P} \text{ (violation, i.e., L)}\]

\[\varphi = \bigcap_{e \in \mathcal{E}} \mu(e) = \text{true}, \text{for } e \in \mathcal{P} \text{ (violation, i.e., T)}\]

A satisfiable SMT instance denotes that the uninterpreted function was not only able to generate a valid sequence of consistent cuts but also that the sequence satisfies the MTL formula given the computation. This result is then fed to the progression cases to generate the final verdict.

5.3. Segmentation of a distributed computation

We know that predicate detection, let alone runtime verification, is NP-complete [25] in the size of the system (number of processes). This complexity grows to higher classes when working with nested temporal operators. To make the problem computationally viable, we aim to chop the computation, \((\mathcal{E}, \rightarrow)\) into \(g\) segments, \((\text{seg}_1, \rightarrow), (\text{seg}_2, \rightarrow), \ldots, (\text{seg}_g, \rightarrow)\). This involves creating small SMT-instances for each of the segments which improves the runtime of the overall problem. In a computation of length \(l\), if we were to chop it into \(g\) segments, each segment would be of the length \(\frac{l}{g} + \epsilon\) and the set of events included in it can be given by:

\[\text{seg}_i = \{e'_{j}, | e \in \left[\max(0, \frac{(j-1) \times l}{g} - \epsilon), \frac{j \times l}{g}\right] \land i \in [1, \frac{l}{g}]\}\]

Note that monitoring of a segment should include the events that happened within \(\epsilon\) time of the segment actually starting since it might include events that are concurrent with some other events in the system not accounted for in the previous segment.

6. Calculating the probability of RV verdicts

We run some diagnostic tests on the distributed system and estimate the Cumulative Density Function (CDF) of the clock skew. Details of the estimation process can be found in Section Appendix B.
Overview of the CDF: Given an event, we map each possible time of occurrence of the event with the respective probability using a function \( pr : \mathcal{E} \times \mathbb{Z}_{\geq 0} \rightarrow [0,1] \), where \( pr(e_j^\epsilon, \delta(e_j^\epsilon)) \) is some real number in the range \([0,1]\) such that:

\[
\forall e_j^\epsilon \in \mathcal{E}, \forall \sigma_1, \sigma_2 \in [\max(0, \sigma - \epsilon + 1), \sigma + \epsilon - 1].
\]

\[
(\sigma_1 < \sigma_2) \rightarrow pr(e_j^\epsilon, \sigma_1) \leq pr(e_j^\epsilon, \sigma_2)
\]

and

\[
\forall e_j^\epsilon \in \mathcal{E}, pr(e_j^\epsilon, \sigma + \epsilon - 1) = 1; pr(e_j^\epsilon, -\epsilon) = 0
\]

Calculating probability over \( \rho \): To avoid an iterative process of generating all possible traces, we use a consolidated method which limits the number of traces to be verified. For all \( j \geq 0 \), if \( \rho(j + 1) - \rho(j) = \{e_j^\epsilon\} \), then we define two entities such that

\[
\sigma_{\text{start}} = \max\{\tau(j), \sigma - \epsilon + 1\}
\]

and

\[
\sigma_{\text{end}} = \delta(e_j^\epsilon)
\]

We define a function \( P : \mathcal{E} \times \mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0} \rightarrow [0,1] \) which calculates the probability for the range of time of occurrence of the event given by \( \sigma_{\text{start}}, \sigma_{\text{end}} \) as

\[
P(e_j^\epsilon, \sigma_{\text{start}}, \sigma_{\text{end}}) = pr(e_j^\epsilon, \sigma_{\text{end}}) - pr(e_j^\epsilon, \sigma_{\text{start}})
\]

The probability of generating the corresponding trace is given by

\[
P(\rho, \tau) = \prod_{j=0}^{\lceil \frac{\tau}{\epsilon} \rceil} P(e_j^\epsilon, \sigma_{\text{start}}, \sigma_{\text{end}})
\]

where \( \rho(j + 1) - \rho(j) = \{e_j^\epsilon\} \).

7. Case study and evaluation

In this section, we analyze our SMT-based solution. We note that we are not concerned about data collections, data transfer, etc., as given a distributed setting, the runtime of the actual SMT encoding will be the most dominating aspect of the monitoring process. We evaluate our proposed solution using traces collected from benchmarks of the tool UPPAAL [35] models (Section 7.1) and two case studies involving smart contracts over multiple blockchains (Section 7.2) and industrial control system (Section 7.3).

7.1. UPPAAL benchmarks

We base our synthetic experiments on 3 different UPPAAL benchmark models described in [11]. For more details about these models and the MTL properties we monitor, refer to the Appendix C.1.

Each experiment involves three steps: (1) offset calculation of the given distributed system, (2) distributed computation/trace generation and (3) trace verification. As stated earlier, the value of the offset ranges from \((-\epsilon, \epsilon)\) with 0 signifying that there is no skew between the two processes. To study as to how the offset distribution effects the probabilistic guarantee of a verdict, we make use of five different distributions.

- A truncated normal distribution, \( TX_1 : (\mu = 0, \sigma = \frac{\epsilon}{\sqrt{3}}) \).
- A truncated normal distribution \( TX_2 : (\mu = 0, \sigma = \frac{\epsilon}{\sqrt{2}}) \).
- A uniform distribution \( U_1 : U(-\epsilon, \epsilon) \).
- A uniform distribution \( U_2 : U(-\frac{\epsilon}{2}, \frac{\epsilon}{2}) \).
- A sum of two normal distribution, \( TX_3 \), with \((\mu = -\frac{\epsilon}{2}, \sigma = \frac{\epsilon}{\sqrt{2}})\) and \((\mu = \frac{\epsilon}{2}, \sigma = \frac{\epsilon}{\sqrt{2}})\).

The truncated normal distribution has limits of \((-\epsilon, \epsilon)\). For each UPPAAL model, we consider each pair of consecutive events are 0.1 s apart, i.e., there are 10 events per second per process. For our verification step, our monitoring algorithm executes on the generated computation and verifies it against an MTL specification. We consider the following parameters (1) primary which includes time synchronization constant \( \epsilon \), (2) MTL formula under monitoring, (3) number of segments \( g \), (4) computation length \( l \), (5) number of processes in the system \( P \), (6) event rate, and (7) offset distribution. We study the runtime of our monitoring algorithm against each of these parameters. We use a machine with 2x Intel Xeon Platinum 8180 (2.5 Ghz) processor, 768 GB of RAM, 112 cores with gcc version 9.3.1.

7.1.1. Analysis: runtime

We study each of the parameters individually and analyze how it affects the runtime of our monitoring approach. All results correspond to \( \epsilon = 15ms, |P| = 2, g = 15, l = 2sec, \) an event rate of 10events/sec, \( \varphi_5 \) as the MTL specification and \( U_1 \) as the offset distribution unless mentioned otherwise. We vary the number of processes in the system from 2 to 4, since in most cross-chain transactions the number of blockchains involved is small.

Impact of different formula. We take a range of different kinds of formulas listed in Appendix C.1 in accordance with [51], for this experiment and see how the runtime of our approach is effected by it. Fig. 4a shows that runtime of the monitor depends on two factors: the number of sub-formulas and the depth of nested temporal operators. Comparing \( \varphi_1 \) and \( \varphi_5 \), both of which consists of the same number of predicates but since \( \varphi_5 \) has recursive temporal operators, it takes more time to verify and the runtime is comparable to \( \varphi_1 \), which consists of two sub-formulas. This is because verification of the inner temporal formula often requires observing states in the next segment in order to come to the final verdict. This accounts for more runtime for the monitor.

Impact of epsilon. Increasing the value of time synchronization constant \( \epsilon \), increases the possible number of concurrent events that need to be considered. This increases the complexity of verifying the computation and there-by increasing the runtime of the algorithm. In addition to this, higher values of \( \epsilon \) also correspond to more number of possible traces that are possible and should be taken into consideration. We observe that the runtime increases exponentially with increasing the value of time synchronization constant in Fig. 4b. An interesting observation is that, with longer segment length, the runtime increases at a higher rate than with shorter segment length. This is because with longer segment length and higher \( \epsilon \), it equates to a larger number of possible traces that the monitoring algorithm needs to take into consideration. This increases the overall runtime of the verification algorithm by a considerable amount and at a higher pace.

Impact of segment frequency. Increasing the segment frequency makes the length of each segment lower and thus verifying each segment involves a lower number of events. We observe the effect of segment frequency on the runtime of our verification algorithm in Fig. 4c. With increasing the segment frequency, the runtime decreases unless it reaches a certain value (here it is \( \approx 0.6 \)) after which the benefit of working with a lower number of events is overcast by the time required to setup each SMT instances. Working with higher number of segments equates to solving more number of SMT problem for the same computation length. Setting up the SMT problem requires a considerable amount of time which is seen by the slight increase in runtime for higher values of segment frequency.

Impact of computation length. As it can be inferred from the previous results, the runtime of our verification algorithm is majorly dictated by the number of events in the computation. Thus, when working with a longer computation, keeping the maximum clock skew and the number of segments constant, we should see a longer verification time as well. Results in Fig. 4d supports the above claim.
Impact of number truth values per segment. In order to take into consideration all possible truth values of a computation, we execute the SMT problem multiple times, with the verdict of all previous executions being added to the SMT problem such that no two verdict is repeated. Here in Fig. 4e we see that the runtime is linearly effected by increasing number of distinct verdicts. This is because, the complexity of the problem that the SMT is trying to solve does not change when trying to evaluate to a different solution.

Impact of event-rate. Increasing the event rate involves more number of events that needs to be processed by our verification algorithm per segment and thereby increasing the runtime at an exponential rate as seen in Fig. 4f. We also observe that with higher number of processes, the rate at which the runtime of our algorithm increases is higher for the same increase in event rate.

7.1.2. Probabilistic analysis

Next, we study the effect of different parameters on the probabilistic guarantee of the verdict computed by the monitor. All results correspond to $\epsilon = 20ms$, $|P| = 2$, $g = 15$, $l = 2sec$, an event rate of 10events/sec and $\varphi_4$ as the MTL specification unless mentioned otherwise.

Impact of epsilon. As can be imagined, impact of larger clock skew has a negative impact on the verification result of the system. Larger clock skew leads to more number of events being considered as concurrent and that leads to more number of traces that is possible with the correct order of events being compromised. This leads to a lower probability associated with system with larger $\epsilon$. In our case, as seen in Fig. 5a, we receive perfect score when $\epsilon = 10ms$, since this makes all the event perfectly ordered. Moreover, the guarantee slides uniformly with increasing value of $\epsilon$.

The other observation from Fig. 5a is how the probability is effected by the different offset distribution. The less is the standard deviation of the distribution, the closer to the global clock is the time of occurrence of the event. This makes the probability of yielding a satisfiable result more as compared to when the time of occurrence is far from the global clock. Thus $T_X$ yields higher percentage of satisfiable result when compared to $T_X'$, which has a larger standard deviation.

Impact of type of logical operator. Here, we compare how the type of logical operator effects the probabilistic guarantee of a verdict. As can be seen in Fig. 5b, sub-formulas separated with a disjunction has a higher probabilistic percentage than the formulas separated by a conjunction. This can be explained by how a formula separated with conjunction is evaluated compared to the one with disjunction. In case
of disjunction, any one sub-formula evaluated to be true rewrites the entire formula to be true, where-as in case of conjunction, all the sub-formulas need to be evaluated to be true and only then we come to a verdict of true. This marks the satisfiability percentage difference between the formulas separated by conjunction and disjunction.

7.1.3. Analysis : scalability

We tested the performance of our approach with respect to large time bounds, similar to [38,36]. The formulas in Fig. 6a are: $\diamond_{10,10} a$, $\diamond_{10,20} a$, $\diamond_{10,00} a$, $\diamond_{100,200} a$, $\diamond_{100,000} a$, $\diamond_{10000,20000}a$, labelled as F1 to F8 respectively. Similarly, the formulas in Fig. 6b are $a U^\diamond_{10,10} b$, $a U^\diamond_{10,20} b$, $a U^\diamond_{100,00} b$, $a U^\diamond_{1000,2000} b$, $a U^\diamond_{10000,00} b$, $a U^\diamond_{100000,20000} b$, labelled as U1 to U8 respectively. The results follow the complexity analysis of the Progression function in Theorem 1, i.e., given a distributed computation, the end time bound of a formula is directly proportional to the time taken to verify the formula, given the number of sub-formulas are remained constant. Furthermore, we observe that for $|P| = 4$, we are able to verify the distributed computation in less time than the length of the computation.

Furthermore, when studying the throughput of our SMT-based verification algorithm, we found that it is independent on the time bound of the formula. However, due to the increasing complexity of verifying a distributed computation with increasing number of processes, the throughput decreases as well. As seen in Fig. 6c, with increasing number of processes, the number of interleavings increases exponentially and there-by reducing the throughput.

7.2. Case study 1: monitoring crosschain protocols

We implemented the following cross-chain protocols from [54]: two-party swap, multi-party swap, and auction. The protocols are written as smart contracts in Solidity and tested using Ganache, a tool that creates mocked Ethereum blockchains. Using a single mocked chain, we mimicked cross-chain protocols via several (discrete) tokens and smart contracts, which do not communicate with each other. For more details about the experiments including the MTL specifications, refer to Appendix C.2.

7.2.1. Analysis of results

We put our monitor to test the traces generated by the Truffle-Ganache framework. To monitor the 2-party swap protocol we do not divide the trace into multiple segments due to the low number of events that are involved in the protocol. On the other hand, both 3-party swap and auction protocol involve a higher number of events and thus we divide the trace into two segments ($p=2$). In Fig. 7a, we show how the runtime of the monitor is effected by the number of events in each transaction log.

Additionally, we generate transaction logs with different values for deadline ($\Delta$) and time synchronization constant ($\epsilon$) to put the safety of the protocol in jeopardy. We observe both true and false verdict when $\epsilon \gtrless \Delta$ as seen in Fig. 7b. This is due to the non deterministic time stamp owning to the assumption of a partially synchronous system. The observed time stamp of each event can at most be off by $\epsilon$. Thus, we recommend to use a value of $\Delta$ that is strictly greater than to the value of $\epsilon$ when designing the smart contract.

7.3. Case study 2: monitoring industrial control system (ICS)

We use an industrial control system dataset, Secure Water Treatment (SWaT) [26], of a fully operational scaled down water treatment plant with a small footprint, producing 5 gallons/minute of doubly filtered water. For more details about the dataset and the MTL properties we monitor, refer to the Appendix C.3.
### 7.3.1. Analysis of results

We put our monitor to the test with verifying the traces generated by the water treatment plant with respect to the above mentioned MTL specifications. Although the event rate in some industrial systems can go up to millisecond level, as part of the experimental dataset, it only reports event every second. As per as our monitoring approach, we divide the trace into multiple segments, each with a constant number of events and time-synchronization constant to be $\epsilon = 1.5$ sec as shown in Fig. 8a.

Additionally, we compare the verification result from our approach with the one reported by the dataset. Although our approach does not report satisfaction of the system property when there has been a violation (false-negatives), but it does report violation of the system specification even when there was actually no violation (false-positives). As seen in Fig. 8b, with increase in the time synchronization constant the %-age of false-positives increase as well. This is due to the fact that with higher values of $\epsilon$, the monitor considers more number of events to be possibly concurrent and there-by making the partial-ordering of events as observed by the monitor away from the actual total-ordering of the events taking place in the system.

### 8. Related work

Centralized and decentralized online predicate detection in an asynchronous distributed system have been extensively studied (e.g., [14, 40]). Extensions to include temporal operators appear in [45, 42]. The line of work in [14, 40, 45, 42] considers a fully asynchronous system. Monitoring of distributed system with network delay and imprecise clock skew but constant clock rate for MTL specifications was introduced in [9, 56]. A pattern query based verification methodology for imprecise event time was discussed in [55]. These methods lack both the generality as well as the ability to monitor large recurring specifications. Our method is more generalized and also able to calculate the probabilistic guarantee of the verification verdict. An SMT-based predicate detection solution has been introduced in [52]. On the other hand, runtime monitoring for synchronous distributed system has been studied in [19, 16, 12]. The major shortcoming of these approaches is the assumption of a common global clock, shared among all the processes. Compared to the benchmarks of verifying MTL specification on synchronous system presented in [38, 36], we perform poorly due to our more realistic assumption of a partially-synchronous distributed system and the exponential number of interleavings that needs to be taken into account for verifying such a distributed system. Finally, fault-tolerant monitoring, where monitors can crash, has been investigated in [13] for asynchronous and in [34] for synchronized distributed processes.

Runtime monitoring of time sensitive distributed system has been studied in [7, 8, 53, 50] and security vulnerabilities posed by blockchains have also been extensively studied [24, 4, 5, 15, 46]. However, these methods fall short to verify the correctness of cross-chain protocols, due to their assumption regarding synchronous systems and the presence of a global clock. On the contrary, we assume the presence of a clock synchronization algorithm which limits the maximum clock skew among processes (blockchain in this context) to a constant. This is a realistic assumption since different blockchains have their own local clock and it is certain to have a skew between them. A similar SMT-based solution was studied for LTL specifications in [22], which we extend to include a more expressive time bounded logic relevant to the usage we mention in this paper.

Probabilistic model checking [31] can be used to compute the probability that a Markov model satisfies a temporal formula. Runtime Ver-
ification with probabilistic guarantee has only been studied where a Discrete-Time Markov Chain (DTMC) was used to represent the complete system and a Hidden Markov Model (HMM) was used to present a partial system in [49,6]. To investigate the problem of monitoring a partially observable system with nondeterminism, a Markov Decision Process (MDP) based verification approach was introduced in [29]. A common limitation for all these approaches is the presence of a common global clock. When studying a partially synchronous distributed system, the absence of a common global clock makes for an extremely large state space and there by limiting the use of a Markov model. Moreover, the probabilistic guarantee is associated with the clock skew on the different components and not the state of those component.

9. Conclusion

In this paper, we study distributed runtime verification. We propose a technique which takes an MTL formula and a distributed computation as input. By assuming partial synchrony among all processes, first we chop the computation into several segments and then apply a progression-based formula rewriting monitoring algorithm implemented as an SMT decision problem in order to verify the correctness of the distributed system with respect to the formula. We conducted extensive synthetic experiments on traces generated by the tool UPPAAL, a set of blockchain smart contracts and an industrial water treatment plant.

For future work, we plan to study the trade off between accuracy and scalability of our approach. Another important extension of our work is decentralized runtime verification where the monitors are vulnerable to crash failures and Byzantine failures with respect to stream based specifications. Compared to temporal logic such as LTL and MTL, stream-based specification offers additional capabilities of calculating aggregate functions which enables utilities in programmable logic controller, used to control electro-mechanical processes for use in manufacturing, distribution and other automation environments. The monitor faults would enable applications in real-life systems where components are subject to failures and anomalies.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Borzoo Bonakdarpour reports financial support was provided by National Science Foundation. Borzoo Bonakdarpour reports a relationship with National Science Foundation that includes: funding grants. This work is sponsored by the United States National Science Foundation (NSF) awards FMIF 2102106 and SHF 2118356. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Progression example

Example In Fig. A.9, the time line shows propositions and their time of occurrence, for formula $\bigwedge_{0,6} r \rightarrow (\neg p U_{[0,6]} q)$. The entire computation has been divided into 3 segments, $(a, f)$, $(a', f')$, and $(a'', f'')$ and each state has been represented by $(x, t)$:

- We start with segment $(a, f)$. First we evaluate $\bigwedge_{0,6} r$, which requires evaluating $Pr(a, f, r)$ for $i \in [0, 1, 2]$, all of which returns the verdict $\text{false}$ and then by rewriting the sub-formula as $\bigwedge_{0,6} r$. Next, to evaluate the sub-formula $\neg p U_{[0,6]} q$, we need to evaluate (1) $Pr(a', f', \neg p)$ for $i \in [0, 1]$ since $t_i - t_0 < 2$ and both evaluates to true, (2) $Pr(a, f, \bigwedge_{0,6} r)$ which also evaluates to true and (3) $Pr(a'', f'', q)$ which evaluates as $\text{false}$. Thereby, the rewritten formula after observing $(a, f)$ is $\bigwedge_{0,3} r \rightarrow (\neg p U_{[0,6]} q)$.

- Similarly, we evaluate the formula now with respect to $(a', f')$, which makes the sub-formula $\bigwedge_{0,3} r$ evaluate to true at $r = 3$ and the sub-formula $\neg p U_{[0,3]} q$ (there is no such $i \in [0, 1, 2]$ where $t_i - t_0 < 0$ and for all $j \in [0, 1, 2]$, $Pr(a', f', q)$ is $\text{false}$) is rewritten as $\neg p U_{[0,4]} q$.

- In $(a'', f'')$, for $j = 1$, $Pr(a'', f'', \bigwedge_{[0,3]} \neg p) = \text{true}$ and $Pr(a'', f'', q) = \text{true}$, and thereby rewriting the entire formula as true.

Appendix B. Estimating cumulative density function (CDF)

We run a number of diagnostic tests on every pair of processes in the distributed computation. Since the distributed system considered allows message passing, tests include sending and receiving of messages. A client process sends a dummy message to a server process and once the server process receives a message, it replies the client. Using the timestamps of the messages, we calculate the offset

$$\Theta = \frac{(t_1 - t_0) + (t_2 - t_1)}{2}$$

and the round trip delay

$$\delta = (t_3 - t_2) - (t_2 - t_1)$$

where, $t_0$ is the client’s timestamp of the requested packet transmission, $t_1$ is the server’s timestamp of the requested packet reception, $t_2$ is the server’s timestamp of the response packet transmission and $t_3$ is the client’s timestamp of the response packet reception. We derive the expression for the offset, for the request packet (resp. response packet),

$$t_0 + \Theta + \delta/2 = t_1 \quad t_1 + \Theta - \delta/2 = t_2$$

Solving for $\Theta$ yields the time offset. This procedure is repeated for $n$ times for each pair of processes and a vector of offsets is collected that defines the system, $\{x_1, x_2, \ldots, x_n\}$. This set of independent, identical distributed bounded random numbers constitute our sample. We assume that it follows a common cumulative distribution function $F(x)$.

Then the empirical distribution function is defined by

$$\hat{F}(x) = \frac{\text{number of elements in the sample } \leq x}{n} = \frac{1}{n} \sum_{i=1}^{n} I_{x_i \leq x},$$

where $I_a$ is the indicator of event $a$. This makes, $\hat{F}(x)$ an unbiased estimator of $F(x)$. Since the data in our setting is bounded by $(-\infty, \infty)$ where $\hat{F}(-\infty) = 0$ and $\hat{F}(\infty) = 1$. We decide to break the entire range into $h$ steps where each step is of length $2\epsilon/h$. Thus, the estimated probability of a time offset, $t$ is given by $p(t) = \hat{F}(t) - \hat{F}(t - 2\epsilon/h)$.

For example, for a vector of observed offsets, $(-4, -3, -2, -2, -1, -1, -1, 0, 0, 0, 0, 0, 1, 1, 2, 2, 3, 4)$, and $\epsilon = 5$, the estimated distribution function can be graphically represented by Fig. B.10 where $h = 5$. Thus we calculate the estimated probability of a time offset, $p(\epsilon = 0) = \hat{F}(0) - \hat{F}(-2) = 0.9 - 0.65 = 0.25$.

For a better understanding as to how close our estimated distribution function, $\hat{F}(t)$, is to the real distribution, $F(t)$, we run a hypothesis test for the mean and standard deviation with a p-value of $\leq 0.05$. It is also to be noted that any other non-parametric density estimation method can be used, e.g. Kernel density estimator, spectral density estimator, etc.
Appendix C. Case study

Here, in Section Appendix C.1 we explain how the different UPPAAL models work and in Section Appendix C.2 we dive into the MTL specifications we use to verify 3-party swap and the auction protocol.

C.1. UPPAAL models

Below we explain in details how each of the UPPAAL\footnote{UPPAAL is a model checker for a network of timed automata. The tool-set is accompanied by a set of benchmarks for real-time systems. Here, we assume that the components of the network are partially synchronized.} models work. In respect to our monitoring algorithm, we consider multiple instances of each of the models as different processes. Each event consists of the action that was taken along with the time of occurrence of the event. In addition to this, we assume a unique clock for each instance, synchronized by the presence of a clock synchronization algorithm with a maximum clock skew of $\epsilon$.

\textit{The train-gate} It models a railway control system which controls access to a bridge for several trains. The bridge can be considered as a shared resource and can be accessed by one train at a time. Each train is identified by a unique id and whenever a new train appears in the system, it sends an appr message along with it’s id. The Gate controller has two options: (1) send a stop message and keep the train in waiting state or (2) let the train cross the bridge. Once the train crosses the bridge, it sends a leave message signifying the bridge is free for any other train waiting to cross. (See Fig. C.11.)

The gate keeps track of the state of the bridge, in other words the gate acts as the controller of the bridge for the trains. If the bridge is currently not being used, the gate immediately offers any train appearing to go ahead, otherwise it sends a stop message. Once the gate is free again from a train leaving the bridge, it sends out a go message to any train that had appeared in the mean time and was waiting in the queue. (See Fig. C.12.)

We monitor two properties:

\[
\varphi_1 = \bigwedge_{i \in P} \text{Train[i].Cross} \lor \text{Train[i].Cross}
\]

\[
\varphi_2 = \bigwedge_{i \in P} \left(\text{Train[i].Appr} \rightarrow \Diamond (\text{Gate.Occ} \lor \text{Train[i].Cross})\right)
\]

where $P$ is the set of trains.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\linewidth]{fig.png}
\caption{Estimated Cumulative Density Function.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.8\linewidth]{fig.png}
\caption{Train model.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.8\linewidth]{fig.png}
\caption{Gate model.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.8\linewidth]{fig.png}
\caption{Fischer model.}
\end{figure}

\textit{The Fischer’s protocol} It is a mutual exclusion protocol designed for $n$ processes. A process always sends in a request to enter the critical section (cs). On receiving the request, a unique pid is generated and the process moves to a wait state. A process can only enter into the critical section when it has the correct id. Upon exiting the critical section, the process resets the id which enables other processes to enter the cs. Below mentioned are the properties we monitor. (See Fig. C.13.)

\[
\varphi_3 = \bigwedge_{i \in P} P[i].cs \leq 1
\]

\[
\varphi_4 = \bigwedge_{i \in P} P[i].req \rightarrow \Diamond P[i].cs
\]

\textit{The gossiping people} The model consists of $n$ people, each having a private secret they wish to share with each other. Each person can Call
Fig. C.14. Gossiping people model.

another person and after a conversation, both person mutually knows about all their secrets. With respect to our monitoring problem, we make sure that each person generates a new secret that needs to be shared among others infinitely often. The properties we monitor are mentioned below. (See Fig. C.14.)

\[ \phi_i = \left\{ \bigwedge_{i \neq j} (i \neq j) \rightarrow \text{Person}[i].\text{secret}[j] \right\} \]

\[ \phi_p = \bigwedge_{i \in P} \left( \bigwedge_{j \in P} \bigwedge_{k \in P} \text{Person}[i].\text{secrets} \right) \]

C.2. Blockchain

The huge success of blockchain [37,43] 1.0, that includes cryptocurrencies, especially Bitcoin [43] and blockchain 2.0, known as smart contracts [17], is also promising in many scenarios. A smart contract is a program running on the blockchain. Its execution is triggered automatically and is enforced by conditions preset in the code. In this way, the transfer of assets can be automated by the rules in the smart contracts, and no human intervention can stop it. A typical smart contract implemented in Ethereum [20], uses Solidity [20], which is a Turing-complete language. However, automating the transactions by smart contracts also has its downsides. If the smart contract has bugs and does not do what is expected, then lack of human intervention may lead to massive financial losses. For example, as pointed out by [21], the Parity Multisig Wallet smart contract [1] version 1.5 included a vulnerability which led to the loss of 30 million US dollars. Thus, developing effective techniques to verify the correctness of smart contracts is both urgent and important to protect against possible losses. Furthermore, when a protocol is made up of multiple smart contracts across different blockchains, the correctness of protocols also need to be verified.

C.2.1. Setup

We implemented the following cross-chain protocols from [54]: two-party swap, multi-party swap, and auction. The protocols are written as smart contracts in Solidity and tested using Ganache, a tool that creates mocked Ethereum blockchains. Using a single mocked chain, we mimicked cross-chain protocols via several (discrete) tokens and smart contracts, which do not communicate with each other.

We use the hedged two-party swap example from [54] to describe our experiments. Suppose Alice would like to exchange her apricot tokens with Bob’s banana tokens, using the hedged two-party swap protocol shown in Fig. 1. This protocol provides protection for parties compared to a standard two-party swap protocol [44], in that if one party locks their assets to exchange which is refunded later, this party gets a premium as compensation for locking their assets. The protocol consists of six steps to be executed by Alice and Bob in turn. In our example, we let the amount of tokens they are exchanging be 10 ERC20 tokens and the premium \( p_b \) be 1 token and \( p_a + p_b \) be 2 tokens. We deploy two contracts on both apricot blockchain (the contract is denoted as ApricotSwap) and banana blockchain (denoted as BananaSwap) by mimicking the two blockchains on Ethereum. Denote the time that they reach an agreement of the swap as start\( \text{Time} \). \( \Delta \) is the maximum time for parties to observe the state change of contracts by others and take a step to make changes on contracts. In our experiment, \( \Delta = 500 \) milliseconds. By the definition of the protocol, the execution should be:

- Step 1. Alice deposits 2 tokens as premium in BananaSwap before \( \Delta \) elapses after start\( \text{Time} \).
- Step 2. Bob should deposit 1 token as premium in ApricotSwap before \( 2\Delta \) elapses after start\( \text{Time} \).
- Step 3. Alice escrows her 100 ERC20 tokens to ApricotSwap before \( 3\Delta \) elapses after start\( \text{Time} \).
- Step 4. Bob escrows her 100 ERC20 tokens to BananaSwap before \( 4\Delta \) elapses after start\( \text{Time} \).
- Step 5. Alice sends the preimage of the hashlock to BananaSwap to redeem Bob’s 100 tokens before \( 5\Delta \) elapses after start\( \text{Time} \). Premium is refunded.
- Step 6. Bob sends the preimage of the hashlock to ApricotSwap to redeem Alice’s 100 tokens before \( 6\Delta \) elapses after start\( \text{Time} \). Premium is refunded.

If all parties are conforming, the protocol is executed as above. Otherwise, some asset refund and premium refund events is triggered to resolve the case where some party deviates. To avoid distraction, we do not provide details here.

Each smart contract provides functions to let parties deposit premiums DepositPremium(), escrow an asset EscrowAsset(), send a secret to redeem assets RedeemAsset(), refund the asset if it is not redeemed after timeout, RefundAsset(), and counterparts for premiums RedeemPremium() and RefundPremium(). Whenever a function is called successfully (meaning the transaction sent to the blockchain is included in a block), the blockchain emits an event that we then capture and log. The event interface is provided by the Solidity language. For example, when a party successfully calls DepositPremium(), the PremiumDeposited event emits on the blockchain. We then capture and log this event, allowing us to view the values of PremiumDeposited’s declared fields: the time when it emits, the party that initiated DepositPremium(), and the amount of premium sent. These values are later used in the monitor to check against the specification.

C.2.2. Log generation and monitoring

Our tests simulated the protocols and generated 1024, 4096, and 3888 different sets of logs for the aforementioned protocols, respectively. We again use the hedged two-party swap as an example to show how we generate different logs to simulate different execution of the protocol. On each contract, we enforce the order of those steps to be executed. For example, step 3 EscrowAsset() on the ApricotSwap cannot be executed before Step 1 is taken, i.e. the premium is deposited. This enforcement in the contract restricts the number of possible different states in the contract. Assume we use a binary indicator to denote whether a step is attempted by the corresponding party. 1 denotes a step is attempted, and 0 denotes this step is skipped. If the previous step is skipped, then the later step does not need to be attempted since it will be rejected by the contract. We use an array to denote whether each step is taken for each contract. On each contract, the different executions of those steps can be [1,1] meaning all steps are attempted, or [1,0] meaning the last step is skipped, and so on. Each chain has 4 different executions. We take the Cartesian product of arrays of two contracts to simulate different combinations of executions on two contracts. Furthermore, if a step is attempted, we also simulate whether the
step is taken late, or in time. Thus we have $2^6$ possibilities of those 6 steps. In summary, we succeeded generating $4 \cdot 4 \cdot 2^6 = 1024$ different logs.

In our testing, after deploying the two contracts, we iterate over a 2D array of size 1024 $\times$ 12, and each time takes one possible execution denoted as an array length of 12 to simulate the behavior of participants. For example, [1, 0, 1, 1, 1, 1, 1, 1, 1] stands for the first step is attempted but it is late, and the steps after second step are all attempted in time. Indexed from 0, the even index denotes if a step is attempted or not and the odd index denotes the former step is attempted in time or late. By the indicator given by the array, we let parties attempt to call a function of the contract or just skip. In this way, we produce 1024 different logs containing the events emitted in each iteration.

We check the policies mentioned in [54]: liveness, safety, and ability to hedge against sore loser attacks. Liveness means that Alice should deposit her premium on the banana blockchain within Δ from when the swap started. (0,Δ)ban.premium_deposited(alice) and then Bob should deposit his premiums, and then they escrow their assets to exchange, redeem their assets (i.e., the assets are swapped), and the premiums are refunded. In our testing, we always call a function to settle all assets in the contract if the asset transfer is triggered by timeout. Thus, in the specification, we also check all assets are settled:

$$\varphi_{\text{liveness}} = (0,\Delta)\text{ban}.\text{premium}_{\text{deposited}}(\text{alice}) \land$$

$$\neg (0,\Delta)\text{apr}.\text{premium}_{\text{deposited}}(\text{bob}) \land$$

$$\neg (0,\Delta)\text{apr}.\text{asset}_{\text{escrowed}}(\text{alice}) \land$$

$$\neg (0,\Delta)\text{ban}.\text{asset}_{\text{escrowed}}(\text{bob}) \land$$

$$\neg (0,\Delta)\text{ban}.\text{asset}_{\text{redeemed}}(\text{alice}) \land$$

$$\neg (0,\Delta)\text{apr}.\text{asset}_{\text{redeemed}}(\text{bob}) \land$$

$$\neg (0,\Delta)\text{ban}.\text{premium}_{\text{refunded}}(\text{alice}) \land$$

$$\neg (0,\Delta)\text{apr}.\text{premium}_{\text{refunded}}(\text{bob}) \land$$

$$\neg (6,\Delta)\text{apr}.\text{all}.\text{asset}_{\text{settled}}(\text{any}) \land$$

$$(5,\Delta,\omega)\text{ban}.\text{all}.\text{asset}_{\text{settled}}(\text{any}) \land$$

Safety is provided only for conforming parties, since if one party is deviating and behaving unreasonably, it is out of the scope of the protocol to protect them. Alice should always deposit her premium first to start the execution of the protocol (0,Δ)ban.premium_deposited(alice) and proceed if Bob proceeds with the next step. For example, if Bob deposits his premium, then Alice should always go ahead and escrow her asset to exchange (0,2Δ)apr.premium_deposited(bob) and (0,3Δ)apr.asset_escrowed(alice). Alice should never release her secret if she does not redeem, which means Bob should not be able to redeem unless Alice redeems, which is expressed as $\neg \text{apr}.\text{asset}_{\text{redeemed}}(\text{bob}) U' \text{ban}.\text{asset}_{\text{redeemed}}(\text{alice})$.

$$\varphi_{\text{alice.conform}} = (0,\Delta)\text{ban}.\text{premium}_{\text{deposited}}(\text{alice}) \land$$

$$\neg (0,\Delta)\text{apr}.\text{premium}_{\text{deposited}}(\text{bob}) \land$$

$$\neg (0,\Delta)\text{apr}.\text{asset}_{\text{escrowed}}(\text{alice}) \land$$

$$\neg (0,\Delta)\text{ban}.\text{asset}_{\text{escrowed}}(\text{bob}) \land$$

$$\neg (0,\Delta)\text{ban}.\text{asset}_{\text{redeemed}}(\text{alice}) \land$$

$$\neg (\neg \text{apr}.\text{asset}_{\text{redeemed}}(\text{bob}) U' \text{ban}.\text{asset}_{\text{redeemed}}(\text{alice})$$

By definition, safety means a conforming party does not end up with a negative payoff. We track the assets transferred from parties and transferred to parties in our logs. Thus, a conforming party is safe. e.g. Alice, is specified as safe $\varphi_{\text{alice.safety}}$:

$$\varphi_{\text{alice.safety}} \Leftrightarrow \varphi_{\text{alice.conform}} \land \sum_{\text{TransTo} = \text{alice}} \text{amount} \geq \sum_{\text{TransFrom} = \text{alice}} \text{amount}$$

To enable a conforming party to hedge against the sore loser attack if they escrow assets to exchange which is refunded in the end, our protocol should guarantee the aforementioned party get a premium as compensation, which is expressed as $\varphi_{\text{alice.hedged}}$:

$$\varphi_{\text{alice.hedged}} = \Diamond(\varphi_{\text{alice.conform}} \land$$

$$\neg\text{apr}.\text{asset}_{\text{escrowed}}(\text{alice}) \land$$

$$\neg\text{apr}.\text{premium}_{\text{refunded}}(\text{any}) \rightarrow$$

$$\Diamond(\sum_{\text{TransTo} = \text{alice}} \text{amount} \geq$$

$$\sum_{\text{TransFrom} = \text{alice}} \text{amount} + \text{apr}.\text{premium}.\text{amount})$$

Below shows the specifications we used to verify the correctness of hedged three-party swap and auction protocols, as shown in [54]. The format of the specifications are similar to that of hedged two-party swap protocol.

C.2.3. Hedged 3-party swap protocol

The three-party swap example we implemented can be described as a digraph where there are directed edges between Alice, Bob and Carol. For simplicity, we consider each party transfers 100 assets. Transfer between Alice and Bob is called _ApricotSwap_, meaning Alice proposes to transfer 100 apricot tokens to Bob, transfer between Bob and Carol called _BananaSwap_, meaning Bob proposes to transfer 100 banana tokens to Carol, transfer between Carol and Alice, called _CherrySwap_, meaning Carol proposes to transfer 100 cherry tokens to Alice. Different tokens are managed by different blockchains (Apricot, Banana and Cherry respectively).

We denote the time they reach an agreement of the swap as $\text{start\_time}$. Δ is the maximum time for parties to observe the state change of contracts by others and take a step to make changes on contracts. According to the protocol, the execution should follow the following steps:

- Step 1. Alice deposits 3 tokens as $\text{escrow\_premium}$ in _ApricotSwap_ before $\text{start\_time}$.Δ elapses after $\text{start\_time}$.
- Step 2. Bob deposits 3 tokens as $\text{escrow\_premium}$ in _BananaSwap_ before 2Δ elapses after $\text{start\_time}$.Δ.
- Step 3. Carol deposits 3 tokens as $\text{escrow\_premium}$ in _CherrySwap_ before 3Δ elapses after $\text{start\_time}$.Δ.
- Step 4. Alice deposits 3 tokens as $\text{redemption\_premium}$ in _CherrySwap_ before 4Δ elapses after $\text{start\_time}$.Δ.
- Step 5. Carol deposits 2 tokens as $\text{redemption\_premium}$ in _BananaSwap_ before 5Δ elapses after $\text{start\_time}$.Δ.
- Step 6. Bob deposits 1 token as $\text{redemption\_premium}$ in _ApricotSwap_ before 6Δ elapses after $\text{start\_time}$.Δ.
- Step 7. Alice escrows 100 ERC20 tokens to _ApricotSwap_ before 7Δ elapses after $\text{start\_time}$.Δ.
- Step 8. Bob escrows 100 ERC20 tokens to _BananaSwap_ before 8Δ elapses after $\text{start\_time}$.Δ.
- Step 9. Carol escrows 100 ERC20 tokens to _CherrySwap_ before 9Δ elapses after $\text{start\_time}$.Δ.
- Step 10. Alice sends the preimage of the hashlock to _CherrySwap_ to redeem Carol’s 100 tokens before 10Δ elapses after $\text{start\_time}$.Δ.
- Step 11. Carol sends the preimage of the hashlock to _BananaSwap_ to redeem Bob’s 100 tokens before 11Δ elapses after $\text{start\_time}$.Δ.
- Step 12. Bob sends the preimage of the hashlock to _ApricotSwap_ to redeem Alice’s 100 tokens before 12Δ elapses after $\text{start\_time}$.Δ.
If all parties are conforming, the protocol is executed as above. Otherwise, some asset refund and premium redeem events will be triggered to resolve the case where some party deviates. To avoid distraction, we do not provide details here.

Liveness Below shows the specification to liveness, provided all the steps of the protocol has been taken:

\[
\varphi_{\text{liveness}} = \Box_{[0, \Delta]} \text{apr}.\text{depositEscrowPr}(alice) \\
\wedge \Box_{[0, \Delta]} \text{ban}.\text{depositEscrowPr}(bob) \\
\wedge \Box_{[0, \Delta]} \text{che}.\text{depositEscrowPr}(carol) \\
\wedge \Box_{[0, \Delta]} \text{che}.\text{depositRedemptionPr}(alice) \\
\wedge \Box_{[0, \Delta]} \text{ban}.\text{depositRedemptionPr}(carol) \\
\wedge \Box_{[0, \Delta]} \text{apr}.\text{depositRedemptionPr}(bob) \\
\wedge \Box_{[0, \Delta]} \text{apr}.\text{assetEscrowed}(alice) \\
\wedge \Box_{[0, \Delta]} \text{ban}.\text{assetEscrowed}(bob) \\
\wedge \Box_{[0, \Delta]} \text{che}.\text{assetEscrowed}(carol) \\
\wedge \Box_{[0, \Delta]} \text{che}.\text{hashlockUnlocked}(alice) \\
\wedge \Box_{[0, \Delta]} \text{ban}.\text{hashlockUnlocked}(carol) \\
\wedge \Box_{[0, \Delta]} \text{ap}.\text{hashlockUnlocked}(bob) \\
\wedge \Box \text{assetRedeemed}(alice) \\
\wedge \Box \text{assetRedeemed}(bob) \\
\wedge \Box \text{assetRedeemed}(carol) \\
\wedge \Box \text{EscrowPremiumRefunded}(alice) \\
\wedge \Box \text{EscrowPremiumRefunded}(bob) \\
\wedge \Box \text{EscrowPremiumRefunded}(carol) \\
\wedge \Box \text{RedemptionPremiumRefunded}(alice) \\
\wedge \Box \text{RedemptionPremiumRefunded}(bob) \\
\wedge \Box \text{RedemptionPremiumRefunded}(carol)
\]

Safety Below shows the specification to check if an individual party is conforming. If a party is found to be conforming we ensure that there is no negative payoff for the corresponding party. Specification to check Alice is conforming:

\[
\varphi_{\text{alice,conf}} = \Box_{[0, \Delta]} \text{apr}.\text{depositEscrowPr}(alice) \\
\wedge \Box_{[0, \Delta]} \text{che}.\text{depositEscrowPr}(carol) \\
\wedge \Box_{[0, \Delta]} \text{che}.\text{depositRedemptionPr}(alice) \\
\wedge \Box_{[0, \Delta]} \text{che}.\text{depositRedemptionPr}(carol) \\
\wedge \Box_{[0, \Delta]} \text{apr}.\text{depositRedemptionPr}(bob) \\
\wedge \Box_{[0, \Delta]} \text{apr}.\text{assetEscrowed}(alice) \\
\wedge \Box_{[0, \Delta]} \text{che}.\text{assetEscrowed}(carol) \\
\wedge \Box_{[0, \Delta]} \text{che}.\text{hashlockUnlocked}(alice) \\
\wedge \Box_{[0, \Delta]} \text{che}.\text{hashlockUnlocked}(carol) \\
\wedge \Box_{[0, \Delta]} \text{che}.\text{hashlockUnlocked}(bob) \\
\wedge \Box \text{che}.\text{assetEscrowed}(carol) \\
\wedge \Box \text{che}.\text{hashlockUnlocked}(alice) \\
\wedge \Box \text{che}.\text{hashlockUnlocked}(bob) \\
\wedge \Box \text{che}.\text{assetEscrowed}(carol) \\
\wedge \Box \text{che}.\text{hashlockUnlocked}(alice) \\
\wedge \Box \text{che}.\text{hashlockUnlocked}(bob)
\]

Specification to check conforming Alice does not have a negative payoff:

\[
\varphi_{\text{alice,safety}} = \varphi_{\text{alice,conf}} \\
\rightarrow \bigvee_{\text{TransTo} = \text{alice}} \sum_{\text{TransFrom} = \text{alice}} \text{amount} \geq \sum_{\text{TransFrom} = \text{alice}} \text{amount}
\]

Hedged Below shows the specification to check that, if a party is conforming and its escrowed asset is refunded, then it gets a premium as compensation.

\[
\varphi_{\text{alice,hedged}} = \Box_{\varphi_{\text{alice,conf}}} \\
\rightarrow \bigvee_{\text{TransTo} = \text{alice}} \sum_{\text{TransFrom} = \text{alice}} \text{amount} \geq \sum_{\text{TransFrom} = \text{alice}} \text{amount} \\
\rightarrow \Box_{\text{ap}.\text{assetEscrowed}(alice)} \\
\rightarrow \Box_{\text{ap}.\text{hashlockUnlocked}(alice)} \\
\rightarrow \Box_{\text{ap}.\text{redemptionPremium}.\text{amount}}
\]

C.2.4. Auction protocol

In the auction example, we consider Alice to be the auctioneer who would like to sell a ticket (worth 100 ERC20 tokens) on the ticket (ticker) blockchain, and Bob and Carol bid on the coin blockchain and the winner should get the ticket and pay for the auctioneer what they bid, and the loser will get refunded. We denote the time that they reach an agreement of the auction as startTime. Δ is the maximum time for parties to observe the state change of contracts by others and take a step to make changes on contracts. Let TicketAuction be a contract managing the “ticket” on the ticket blockchain, and CoinAuction be a contract managing the bids on the coin blockchain. The protocol is briefed as follows.

- Setup. Alice generates two hashes \( h(s_b) \) and \( h(s_c) \). \( h(s_b) \) is assigned to Bob and \( h(s_c) \) is assigned to Carol. If Bob is the winner, then Alice releases \( s_b \). If Carol is the winner, then Alice releases \( s_c \). If both \( s_b \) and \( s_c \) are released in TicketAuction, then the ticket is refunded. If both \( s_b \) and \( s_c \) are released in CoinAuction, then all coins are refunded. In addition, Alice escrows her ticket as 100 ERC20 tokens in TicketAuction and deposits 2 tokens as premiums in CoinAuction.
- Step 1 (Bidding). Bob and Carol bids before Δ elapses after startTime.
- Step 2 (Declaration). Alice sends the winner’s secret to both chains to declare a winner before 2Δ elapses after startTime.
- Step 3 (Challenge). Bob and Carol challenges if they see two secrets or one secret missing, i.e. Alice cheats, before 4Δ elapses after startTime. They challenge by forwarding the secret released by Alice using a path signature scheme [28].
- Step 4 (Settle). After 4Δ elapses after startTime, on the CoinAuction, if only the hashlock corresponding to the actual winner is unlocked, then the winner’s bid goes to Alice. Otherwise, the winner’s bid is refunded. Loser’s bid is always refunded. If the winner’s bid is refunded, then all bidders including the loser gets 1 token as premium to compensate them. On the TicketAuction, if only one secret is released, then the ticket is transferred to the corresponding party who is assigned the hash of the secret. Otherwise, the ticket is refunded.

Liveness Below shows the specification to check that, if all parties are conforming, the winner (Bob) gets the ticket and the auctioneer gets the winner’s bid.
\( \varphi_{\text{licens}} = \bigwedge_{(0, \Delta)} \text{coin}.\text{bid}(\text{bob}) \)
\( \bigwedge_{(0, \Delta)} \text{coin}.\text{declaration}(\text{alice}, s_i) \)
\( \bigwedge_{(0, \Delta)} \text{tckt}.\text{declaration}(\text{alice}, s_i) \)
\( \bigwedge_{(0, \Delta)} \text{coin}.\text{redeemBid}(\text{any}) \)
\( \bigwedge_{(0, \Delta)} \text{coin}.\text{refundPremium}(\text{any}) \)
\( \text{coin}.\text{bid}(\text{carol}) \rightarrow \bigwedge_{(0, \Delta)} \text{coin}.\text{refundBid}(\text{any}) \)
\( \text{tckt}.\text{redeemTicket}(\text{any}) \)
\( \neg \text{coin}.\text{challenge}(\text{any}) \)
\( \neg \text{tckt}.\text{challenge}(\text{any}) \)

**Safety** Below shows the specification to check that, if a party is conforming, this party does not end up worse off. Take Bob (the winner) for example.

Specification to define Bob is conforming:
\( \varphi_{\text{bob,conform}} = \bigwedge_{(0, \Delta)} \text{coin}.\text{bid}(\text{bob}) \)
\( \bigwedge_{(0, \Delta)} \text{coin}.\text{declaration}(\text{alice}, s_i) \)
\( \bigwedge_{(0, \Delta)} \text{tckt}.\text{declaration}(\text{alice}, s_i) \)
\( \bigwedge_{(0, \Delta)} \text{coin}.\text{redeemBid}(\text{any}) \)
\( \bigwedge_{(0, \Delta)} \text{coin}.\text{refundPremium}(\text{any}) \)
\( \text{coin}.\text{bid}(\text{carol}) \rightarrow \bigwedge_{(0, \Delta)} \text{coin}.\text{refundBid}(\text{any}) \)
\( \text{tckt}.\text{redeemTicket}(\text{any}) \)
\( \neg \text{coin}.\text{challenge}(\text{any}) \)
\( \neg \text{tckt}.\text{challenge}(\text{any}) \)

**Hedged** Below shows the specification to check that, if a party is conforming and its escrowed asset is refunded, then it gets a premium as compensation.
\( \varphi_{\text{bob,hedged}} = \Box \left( \varphi_{\text{bob,conform}} \right) \)
\( \bigwedge_{(0, \Delta)} \text{tckt}.\text{refundTicket}(\text{alice}) \)
\( \text{tckt}.\text{redeemTicket}(\text{carol}) \)
\( \Box \left( \text{coin}.\text{refundBid}(\text{any}) \right) \)
\( \text{coin}.\text{redeemPremium}(\text{any}) \)

**C.3. Industrial control system**

Industrial Control Systems (ICS) [32] are information systems to control industrial processes such as manufacturing, product handling, distribution, etc. It includes supervisory control and data acquisition systems used to control geographically dispersed assets and distributed control systems using a programmable logic controller for each of the localized processes. A typical programmable logic controller (PLC) receives data produced by a large number of sensors, fitted across the system. The data produced by these components are often the target of cyber and ransomware attack putting the security of the system in jeopardy. Since these systems are linked to essential services, any attack on these facilities puts the users’ life on the front line. The integrity of the data produced from these distributed components is very important as the PLC’s behavior is dictated by it. Recent attacks have shown that an attack on a company’s ICS costs the company around $5 million and 50 days of system down time. Additionally, according to a recent report [47], it takes the affected company around 191 days to fully recover from the attack and around 54% of all organizations are vulnerable to such attacks.

**C.3.1. Setup**

Secure Water Treatment (SWaT) [26] utilizes a fully operational scaled down water treatment plant with a small footprint, producing 5 gallons/minute of doubly filtered water. It comprises of six main processes corresponding to the physical and control components of the water treatment facility. It starts from process P1 where it takes raw water and stores it in a tank. It is then passed through the pre-treatment process, P2, where the quality of the water is assessed and maintained through chemical dosing. The water then reaches P3 where undesirable materials are removed using fine filtration membranes. Any remaining chlorine is destroyed in the dechlorination process in P4 and the water is then pumped into the Reverse Osmosis system (P5) to reduce inorganic impurities. Finally in P6, water from the RO system is stored ready for distribution.

The dataset classifies different attack on the system into four types, based on the point and stage of the attack: Single Stage-Single Point, Single Stage-Multi Point, Multi Stage-Single Point and Multi Stage-Multi Point. We for the scope of this paper are the most interested in the attacks either covering multiple stages or multiple points. Few of the MTL specifications used are listed below.

\( \varphi_{\text{flow}} = \Box (P-101 \rightarrow (\text{FIT-201} > 0)) \)
\( \varphi_{\text{flow}} = \Box (\neg P-101 \rightarrow (\text{FIT-201} == 0)) \)
\( \varphi_{\text{tankFlow}} = \Box ((\text{MV-101} \lor P-101) \rightarrow \Box (\text{LIT-101} = \text{LIT-101} + \text{FIT-101} - \text{FIT-201})) \)

where \( \varphi_{\text{flow}} \) (resp. \( \varphi_{\text{flow}} \)) checks positive flow of water (resp. no flow of water) depending upon the status of the valve, P-101. Additionally, \( \varphi_{\text{tankFlow}} \) checks if the level of water in the tank, LIT-101, reflects change with respect to the amount of water flowing-in and flowing-out of the tank, given by FIT-101 and FIT-201 respectively.
$$\varphi_{\text{valveOpen}} := \begin{cases} (\text{DPIT-301} > 0.4) \\ (MV-302 ∧ \text{P-602}) ∨ (\text{DPIT-301} < 0.4) \end{cases}$$

$$\varphi_{\text{valveOpen2}} := \begin{cases} \left( O_7 \left( (\text{UV-401 ∧ ATT-502} < 150 ∧ \text{P-501}) \right) \right) \end{cases}$$

where both $\varphi_{\text{valveOpen}}$ and $\varphi_{\text{valveOpen2}}$ makes sure individual parts of the system operates under safety guidelines and in case of violation, we have safety procedures in place to bring them back to the safe operation mode. For example, the pressure sensor, DPIT-301, should always operate with a pressure of less than 0.4 bars but in case the pressure shoots above 0.4 bars, the actuators MV-302 and P-602 should be used to bring the pressure back to below 0.4 bars level within some amount of time, to avoid any unforeseen situation. Similarly, to avoid any damage to the Reverse Osmosis (RO) unit, the RO unit is to be switched off alog with maintaining ATT-502 at 150 and P-501 kept on, for a finite amount of time, if the value of ATT-502 drops below 150.

References


Ritam Ganguly received his bachelor’s degree in Computer Science and Engineering from Maulana Abdul Kalam Azad University of Technology, India, in 2017 and is his Ph.D. in Computer Science from Michigan State University in 2023. He is current a Professor of Instruction of Computer Science with the Department of Computer Science and Engineering at Michigan State University. His research interests include distributed computing, formal methods, blockchains and runtime-verification.

Yingjie Xue received her bachelor’s degree in Information Security in 2015 and master’s degree in Electronics and Communication Engineering in 2018, both from University of Science and Technology of China. She is currently studying at Brown university, pursuing her Ph.D. in Computer Science. Her research interests include blockchain, distributed computing, security and privacy.

Aaron Jonckheere is currently pursuing a bachelor’s degree in Computer Engineering at Michigan State University. His technical interests include Distributed Systems, Embedded Systems, and Computer Graphics.

Parker Ljung received his bachelor’s degree in Computer Science and Economics in 2022 from Brown University. His research interests include blockchain protocols, distributed systems, and hyper-text and hyper-media.

Benjamin Schurstein is currently pursuing a bachelor’s degree in Neuroengineering at Brown University. His research interests include neural demodulation, neural dynamics, and soft robotics.

Borzoo Bonakdarpour received his bachelor’s degree in Computer Engineering – Software from The University of Esfahan, Iran, in 1999 and his Ph.D. in Computer Science from Michigan State University in 2009. He is currently an Associate Professor of Computer Science with the Department of Computer Science and Engineering at Michigan State University. His research interests include distributed computing, formal methods, cyber-physical systems, and computer security.

Maurice Herliy has an A.B. in Mathematics from Harvard University, and a Ph.D. in Computer Science from M.I.T. He has served on the faculty of Carnegie Mellon University and the staff of DEC Cambridge Research Lab. He is the recipient of the 2003 Dijkstra Prize in Distributed Computing, the 2004 Gödel Prize in theoretical computer science, the 2008 ISCA influential paper award, the 2012 Edsger W. Dijkstra Prize, and the 2013 Wallace McDowell award. He received a 2012 Fulbright Distinguished Chair in the Natural Sciences and Engineering Lecturing Fellowship, and he is fellow of the ACM, a fellow of the National Academy of Inventors, the National Academy of Engineering, and the National Academy of Arts and Sciences. In 2022, he won his third Dijkstra Prize.