Methods for Privacy Preserving Model Checking in LTL

December 12, 2019

Contents

1 Acknowledgement 2

2 Introduction 2

3 Preliminaries 4

3.1 Definitions ................................................................................. 4
3.2 Equivalences in LTL ................................................................. 6
3.3 Differences between CTL and LTL ............................................. 6

4 Privacy preserving LTL model checking 8

4.1 Overview .................................................................................. 8
4.2 Secure computation of the joint automaton $A_{M \otimes \phi}$ ...... 9
4.3 Method #1: Data-oblivious graph search ................................. 10
  4.3.1 Algorithms ......................................................................... 11
  4.3.2 Discussion of space and time complexity ......................... 14
  4.3.3 Proof of Security ............................................................... 14
4.4 Method # 2: Reduction to CTL Model checking with Fairness Conditions . 14
  4.4.1 Construction of $A_{\phi}$ ....................................................... 15
  4.4.2 Combining $A_{\neg \phi}$ and $M$ .............................................. 17
  4.4.3 Path finding in the combined automaton ......................... 17
  4.4.4 Proof of Security ............................................................... 17

5 Next Steps 18
1 Acknowledgement

This report would not have been possible without all the help and encouragement from Prof Piskac, Timos, Ning, and Sam. Besides being great research mentors, they have been so kind and patient, always willing to take the time to discuss various things with me and to provide guidance when I needed. The opportunity to think about interesting problems and to explore the fascinating world of formal methods has been truly fun, and working with people I admire and feel inspired by just made everything so much better. Attending the weekly group meetings and learning about what others are working on have always been a great time. I also thank Prof Spielman for giving me my first CS research experience at Yale and providing great guidance along the way, and for his wonderful algorithms course which has impacted how I approach various problems in a fundamental way.

I would also like to thank all the professors of computer science and mathematics at Yale whom I’ve had the privilege to take classes with, for all the knowledge and challenge they have given me and more importantly, for the tremendous influence they have had on me as a person.

All the mistakes and typos in the following text are mine.

2 Introduction

In the following write-up, we present two designs of Linear Temporal Logic (LTL)-based privacy-preserving model checking, in which the input is a program represented as a transition system $M = (S, I, \delta, L)$ and an LTL formula $\phi$. It is given that $M$ and $\phi$ share the set of atomic propositions (also called labels in the context of the transition system), denoted by $Prop$, and it is beyond our concern how the two parties agreed on the set $Prop$. We first define and clarify some concepts and definitions below.

In the definition of $M$, $S$ is the set of states, $I \subseteq S$ is the set of initial states, $\delta \subseteq S \times S$ is the transition relation that describes all the possible transitions between states, i.e., $(s_i, s_j) \in \delta$ if and only if state $s_j$ is a successor of state $s_i$, and $L : S \to 2^{Prop}$ is a labeling function that assigns truth values to each of the atoms in $Prop$ for each state in $S$. We denote $|S| = n, |Prop| = q$.

Informally, in designing protocols for PPMC, we try to solve the problem of enabling one party to verify that the program developed by the other party satisfies certain desired properties without either party obtaining more information beyond the verification results. For example, consider a financial trading firm that is required to not exceed certain limits on
their leverage or to maintain certain capital requirements. A regulator could theoretically demand not just retroactive verification that the firm has met their requirements, but a proactive guarantee that the control software of the firm will not allow traders to exceed these limits. Such a guarantee could be discharged through model checking, yet however, the firm may not want to expose their trading software in plain text at the risk of potentially letting others steal their trading algorithms.

**Model Checking** holds an eminent place in the suite of techniques for formally verifying the correctness of programs, where programs are formulated as finite transition systems and the various desired properties are expressed as logical formulas (which are called formal specifications). These desired properties are verified through explicitly checking all possible executions of a system model and verify that each of them satisfies the corresponding logical formulas.

**Linear temporal logic (LTL)** and computation tree logic (CTL) are both examples of temporal logic often used in model checking, capable of expressing linear time properties. *Formally*, a LTL formula \( \phi \) is recursively defined as

\[
\phi := \text{true} \mid \text{AF} \mid \phi_1 \land \phi_2 \mid \neg \phi \mid \text{X} \phi \mid \phi_1 \text{U} \phi_2 \mid \text{F} \phi \mid \text{G} \phi \mid \phi_1 \text{R} \phi_2 \mid \phi_1 \text{W} \phi_2 \mid \phi_1 \text{M} \phi_2
\]

where \( \text{AF} \) is atomic formula; \( \{ \text{G}, \text{F}, \text{X}\} \) are unary operators such that \( \text{X} \phi \) (next) means \( \phi \) holds true at the next execution step, \( \text{F} \phi \) (finally) means \( \phi \) eventually has to hold true somewhere on the execution path, \( \text{G} \phi \) (globally) means that \( \phi \) has to hold at all subsequent steps on this execution path; \( \{ \text{U}, \text{R}, \text{W}, \text{M}\} \) are binary operators such that \( \phi_1 \text{U} \phi_2 \) (until) means \( \phi_2 \) will hold true at some point along the execution path, and until then, \( \phi_1 \) has to hold true, \( \phi_1 \text{R} \phi_2 \) (release) means \( \phi_1 \) has to hold true until and including the point at which \( \phi_2 \) first becomes true, and if \( \phi_2 \) never becomes true, \( \phi_1 \) must remain true forever, \( \phi_1 \text{W} \phi_2 \) (weak until) means \( \phi_1 \) has to hold true until \( \phi_2 \) holds true, and if \( \phi_2 \) never becomes true, \( \phi_1 \) must remain true forever, \( \phi_1 \text{M} \phi_2 \) (strong release) means that \( \phi_2 \) will hold true at some point along the execution path, and \( \phi_1 \) has to hold true until and including the point at which \( \phi_2 \) first becomes true.

*Formally*, a CTL formula \( \phi \) is recursively defined as

\[
\phi := \text{true} \mid \text{AF} \mid \neg \phi \mid \phi_1 \land \phi_2 \mid \text{AX} \phi \mid \text{EX} \phi \mid \text{A}[\psi \text{U} \phi] \mid \text{E}[\psi \text{U} \phi] \mid \text{AG} \phi \mid \text{EG} \phi \mid \text{AF} \phi \mid \text{EF} \phi
\]

where compared to LTL defined above, the added \( \text{A} \) and \( \text{E} \) are the universal and existential quantifier, respectively. \( \text{A} \) means ‘along All paths’ (inevitably), and \( \text{E} \) means ‘there Exists one path’ (possibly).
Formally, Privacy Preserving Model Checking concerns the following problem: One party, the programmer $P$, has a finite transition system $\mathcal{M}$ with $n$ states, which implicitly induces infinite traces viewable as either linear runs (LTL) or as a computation tree (CTL). A second party, the verifier $V$, has some properties they wish for this software to satisfy which are expressed as a formula $\phi$ in a temporal logic $L$ of length $m$ logical operators and $m'$ atoms, the latter referring to whether a label is present on the states of the transition system. These labels, of which there are $q$, have an interpretation agreed upon by the programmer and verifier in a manner beyond our concern. Finally, we have a public checking algorithm $\text{check}_L(\mathcal{M}, \phi)$ which accepts if and only if the formula $\phi$ is satisfied by model $\mathcal{M}$. Our goal is to provide a cryptographic protocol such that $\text{check}_L(\mathcal{M}, \phi)$ may be computed correctly and efficiently — the latter by the standards of the checking algorithm without such privacy concerns — but with strict limits on the knowledge of $\phi$ learned by the programmer and the knowledge of $\mathcal{M}$ learned by the verifier.

Secure multi-party computation (MPC) is a subfield of cryptography with the goal of designing protocols that allow computation of functions that take input from multiple parties yet without letting any party obtain knowledge about other participating parties besides the result of the computation. In this project, we leverage MPC protocols for PPMC as we can formulate model checking as a function that takes as input the program (transition system $\mathcal{M}$) and property specifications (LTL-formula $\phi$), with the programmer and the verifier being the two participating parties. This way, the proof of security for any established MPC protocol would serve as a proof of security for our PPMC protocol that uses the MPC protocol.

Privacy in the context of cryptography is formally defined and verifiable, the proofs usually involve constructions of a simulator. Informally, a given protocol satisfies the privacy-preserving requirement of PPMC if after the two parties participated in the protocol, the programmer when given the verifier’s input $\phi$ can’t distinguish it from a randomly generated LTL formula of the same length using the same alphabet, and the verifier when given the programmer’s input $\mathcal{M}$ can’t distinguish it from a randomly generated transition system with the same number of states and using the same alphabet.

3 Preliminaries

3.1 Definitions

Definition 1. A non-deterministic finite automaton $A$ is a tuple $(\Sigma, S, S^0, \rho, F)$, where $S$ is the set of all states, $S^0$ the set of initial states, $\rho : S \times \Sigma \to 2^S$ such that $s' \in \rho(s, a)$ means
there exists an edge labeled with $a$ from state $s$ to state $s'$, $F$ is the set of accepting states. Note that the finite automaton is *deterministic* only when $|S^0| = 1$ and $|\rho(s,a)| \leq 1$ for all $s \in S, a \in \Sigma$, where $\Sigma$ is the alphabet.

**Definition 2.** A *run* $r$ of an automaton $A$ over a *finite word* $v$ of is a mapping $\rho : \{0, 1, \ldots, |v|\} \rightarrow Q$ such that $\rho(0) \in Q^0$ and $\forall 0 \leq i \leq |v| - 1, (\rho(i), v(i), \rho(i + 1)) \in \Delta$.

A run is *accepting* if it ends in an accepting state: $\rho(|v|) \in F$.

We say that automaton $A$ *accepts* word $v$ iff there exists an accepting run of $A$ on $v$.

In dealing with LTL model checking, the traces of program execution often correspond to infinite words, hence we introduce the following definition on automata on infinite words:

**Definition 3.** Given a run $r$, define $\lim(r) = \{s | s = s_i$ for infinitely many $i$'s$\}$.

An infinite word is accepted by a Büchi automaton if and only if there exists a run $r$ of the automaton on which some accepting state is visited *infinitely often*, i.e., $\lim(r) \cap F \neq \emptyset$.

**Definition 4.** A *trace* is a sequence of valuations of the propositional atoms.

**Definition 5.** The *(infinitary) language* $L_\omega(\phi)$ of formula $\phi$ is defined as

$$L_\omega(\phi) = \{\pi \in S^\omega | \pi \models^0 \phi\}$$

The *(infinitary) language* $L_\omega(A)$ of Büchi automaton $A$ is the set of infinite words over $\Sigma^\omega$ accepted by $A$.

We present a few useful propositions and sketches of their proofs, which will be useful in the later construction of our methods:

**Proposition 1** ([Cho74]). Let $A_1, A_2$ be Büchi automata. Then there is a Büchi automaton $A$ such that $L_\omega(A) = L_\omega(A_1) \cap L_\omega(A_2)$.

**Proof.** The construction of $A$ from $A_1$ and $A_2$ is much easier if we were concerned with finitary languages [RS59]. The complication arises because simply taking the product of the corresponding elements in the tuple $A_1$ and $A_2$ like in the finite case imposes a stronger condition than is needed: in order for $A$ to accept a word, we *don’t* require that $A_1$ and $A_2$ *simultaneously* run through their individual accepting states infinitely many times. The following construction fixes the issue:

Let $A_1 = (\Sigma, S_1, S_1^0, \rho_1, F_1)$, $A_2 = (\Sigma, S_2, S_2^0, \rho_2, F_2)$, and $A = (\Sigma, S_1 \times S_2 \times \{1, 2\}, S_1^0 \times S_2^0 \times \{1\}, \rho, F_1 \times S_2 \times \{1\})$, where $(s', t', j) \in \rho((s, t, i), a)$ if $s' \in \rho_1(s, a)$, $t' \in \rho_2(t, a)$ and $i = j$, unless $i = 1$ and $s \in F_1$, in which case $j = 2$ or $i = 2$ and $t \in F_2$, in which case $j = 1$.

With starting state $S_1^0 \times S_2^0 \times \{1\}$, if accepting state $F_1 \times S_2 \times \{1\}$ is reached infinitely many
times, then we first have that the accepting state \( F_1 \) of automaton \( A_1 \) is reached infinitely many times during the run, and secondly, by the definition of \( \rho \), this means that the accepting state \( F_2 \) of automaton \( A_2 \) is reached infinitely many times during the run, for between any two consecutive times when state \( F = F_1 \times S_2 \times \{1\} \) is reached, state \( S_1 \times F_2 \times \{2\} \) is reached by the definition of \( \rho \).

Here, \( |S| = O(|S_1| \times |S_2|) \).

**Proposition 2** ([Bü90]). Let \( A \) be a Büchi automaton over an alphabet \( \Sigma \). Then there is a (possibly nondeterministic) Büchi automaton \( \overline{A} \) such that \( L_\omega(\overline{A}) = \Sigma^\omega - L_\omega(A) \).

**Proof.** We omit the discussion of a constructive proof for now, and just note that because of \( A \)'s non-determinism and the fact that it’s Büchi automaton, the cost of the complementation of \( A \) is singly exponential with an almost linear exponent (i.e., the number of states in the constructed complement automaton is exponential w.r.t. the number of states in the original Büchi automaton). In contrast, the complementation of automaton on finite words is also exponential, but with a linear exponent.

### 3.2 Equivalences in LTL

We say that for two formulas, \( \phi \) and \( \psi \), \( \phi \equiv \psi \) if and only if \( L(\phi) = L(\psi) \). Here below are some important equivalences and dualities in LTL.

\[
\begin{align*}
\neg G \phi &\equiv F \neg \phi \\
\neg F \phi &\equiv G \neg \phi \\
\neg X \phi &\equiv X \neg \phi \\
\neg (\phi U \psi) &\equiv \neg \phi R \neg \psi \\
\neg (\phi R \psi) &\equiv \neg \phi U \neg \psi \\
F(\phi \lor \psi) &\equiv F \phi \lor F \psi \\
G(\phi \land \psi) &\equiv G \phi \land G \psi \\
F \phi &\equiv \top U \phi \\
G \phi &\equiv \bot R \phi \\
\phi U \psi &\equiv \phi W \psi \land F \psi \\
\phi W \psi &\equiv \psi R (\phi \lor \psi) \\
\phi R \psi &\equiv \psi W (\phi \land \psi)
\end{align*}
\]

From the above, we can see that each of the sets \( \{U, X\} \), \( \{R, X\} \), \( \{W, X\} \) is an adequate set of temporal connectives that together with usual logical connectives can express any LTL formula.

### 3.3 Differences between CTL and LTL

We offer a discussion on the differences between CTL and LTL here, not only because it is an important topic in formal methods literature, but also because one of the LTL model
checking algorithms we will examine in this paper is CTL model checking algorithms with 
extended (fairness) conditions.

CTL and LTL differ fundamentally in their models of time, such that LTL formulas treat 
each time point as if it leads to a unique possible future and thus are interpreted over 
individual computation sequences of programs, whereasCTL formulas treat each time point 
as if it can branch out into many possible futures and thus are interpreted over infinite 
computation trees. Hence, it is not surprising that CTL and LTL are not compatible and 
some formulas in CTL can’t be expressed equivalently in LTL and vice versa. We can see 
this difference from the fact that CTL has quantifiers $A,E$, making it possible for a CTL 
formula to describe what is possible along all possible execution of the program, i.e., during 
the checking, when faced with a branch either $A$ or $E$ can happen. Note also that in a CTL 
formula, what holds in all future states also holds in the present state. An LTL formula, on 
the other hand, is not equipped with such quantifiers and describes what is possible along 
a possible execution, i.e., it is checked on each linear run with no possibility of switching to 
another run during the checking. Implicitly, there is always an $A$ in front of a LTL formula 
so that the formula is checked against all possible paths from a given point, but importantly 
for LTL, the checking happens for a given path that doesn’t change/branch during the check. 

This subtlety is encapsulated in the following example LTL formula which can’t be expressed 
equivalently in CTL.

Example. $FGp$.

This LTL formula says along every possible execution path of the program, predicate $p$ will 
eventually hold true for until the program ends or forevermore if the program doesn’t end. 
This isn’t equivalently expressible in CTL: the most similar are $AFAGp$ and $AFEGp$, but 
the second one with $EGp$ clearly is too permissive and doesn’t enforce the requirement on all 
paths, whereas the first one is too strict for we can consider the following transition system 

$$
\begin{array}{c}
\text{start} \rightarrow \overrightarrow{q_0} \rightarrow \overrightarrow{q_1}
\end{array}
$$

where $p$ holds at $q_0$ and $\neg p$ holds at $q_1$, then we have that the property $FGp$ holds at given 
state $q_0$, and yet, $AFAGp$ does not hold, for there always exists a branch that would violate 
$AG$ at any given point in time (i.e., in the infinite computation tree, any $q_0$ node won’t 
satisfy $AGp$ for one of its branches will go to $q_1$, and certainly, any $q_1$ node won’t satisfy 
$AGp$ either). This also proves that $EFAGp \neq FGp$ as well.
Example. $AFAGp$.

The most similar LTL formula would be $FGp$, and yet from the example given above, we see that they are not equivalent. Hence, this is an example of a CTL formula not expressible in LTL.

Despite the incompatibility of these two logics in theory, in literature, LTL is generally viewed as more expressive over CTL and possessing many desired properties for use in verification, such as the uniform treatment of model checking and property-specific abstractions (the meaning of which will become clear in the later discussions) [Var01].

4 Privacy preserving LTL model checking

As explained in the introduction, the two parties want to jointly figure out whether $\mathcal{M} \models \phi$ holds, (i.e., $\forall \pi \in \text{Paths}(\mathcal{M}). \pi \models^0 \phi$ holds) in a privacy-preserving way.

Simply put, privacy-preserving in a multi-party computation protocol requires that for each party, its interactions with the other parties can be simulated by a simulator who does not possess the information the other parties possess, and hence in order to prove that such a protocol is privacy-preserving, we need to construct a simulator and prove that from the point of view of each party, its interactions with the simulator is indistinguishable from its interactions with the real other parties.

In our model checking protocols, we assume access to suitable secure multi-party computation techniques, so that the computations carried out are on protected data, reducing the problem to designing protocols that have data-independent memory accesses. That is, given access to subroutine calls that carry out secure multi-party computations for functions such as arithmetic or logic gates, our task is to decompose our model checking procedure into these basic functions and then design procedures for carrying them out in order such that the memory accesses to the abstract machine that computes the overall model checking function are data-independent. This point will be made more clear in our main procedure below.

4.1 Overview

First, given $\mathcal{M} = (S, I, \delta, L)$, we can construct a corresponding Büchi automaton $A_M = (\Sigma, S, I, \rho, W)$ where $\Sigma = 2^{\text{Prop}}$ and $s' = \rho(s, a)$ if and only if $(s, s') \in \delta$ and $a \in L(s)$. In other words, because $A_M$ has its accepting states equal to its total states, $L_\omega(A_M)$ is the set of all possible computations of program $\mathcal{M}$. By [VW94], we also have that given an LTL formula $\phi$, we can construct a Büchi automaton $A_\phi$ such that $A_\phi = (\Sigma, S, S^0, \rho, F)$ with
\[ \Sigma = 2^{ \text{Prop} } \text{ and } |S| \text{ in } 2^{O(|\phi|)} \], and \( L_\omega(A_\phi) \) is the set of computations satisfying the formula \( \phi \).

Now, the LTL-model checking problem is reduced to checking whether \( L_\omega(A_M) \subseteq L_\omega(A_\phi) \implies L_\omega(A_M) \cap L_\omega(A_{\neg \phi}) = \emptyset \implies L_\omega(A_M) \cap L_\omega(A_{\neg \phi}) = \emptyset \). Where note that \( L_\omega(A_{\neg \phi}) = \Sigma^\omega - L_\omega(A_\phi) \) because given a word in \( \Sigma^\omega \), it is either accepted by \( A_\phi \) or not, and it is accepted by \( A_{\phi} \) if and only if it is a computation satisfying \( \phi \), if and only if it does not satisfy \( \neg \phi \).

By Proposition 1 and Proposition 2, we can construct an automaton \( A_{M \otimes \neg \phi} \) such that \( L_\omega(A_{M \otimes \neg \phi}) = L_\omega(A_M) \cap L_\omega(A_{\neg \phi}) \) which has \( O(|M| \cdot 2^{|\phi|}) \) number of states.

### 4.2 Secure computation of the joint automaton \( A_{M \otimes \neg \phi} \)

We represent \( A_{M \otimes \neg \phi} \) by its adjacency matrix, and in this section, we describe how the two parties can together compute any entry of the matrix privately.

In the PPMC setting, the programmer and the verifier can each locally compute \( A_{\neg \phi} = (\Sigma, S_1, S_0^1, \rho_1, F_1) \) and \( A_M = (\Sigma, S_2, S_0^2, \rho_2, F_2) \), and since \( \phi \) and \( M \) share the same set of atomic propositions, \( \text{Prop} \), the two automata share the same alphabet. Because we allow the programmer and the verifier to know each other’s secret’s length, the programmer and the verifier can share with each other the number of states \( A_{\neg \phi} \) and \( A_M \) each has and they can then agree on a fixed labeling of the states (note here we crucially assume semi-honesty from the two parties, such that they will follow through with the protocol), such that for both parties state \( (s, t, j) \) unambiguously refers to the same unique state in the joint automaton, with \( s \in S_1, t \in S_2 \), and yet from the view of the verifier \( s \) is indistinguishable, and same holds for the view of the programmer. This way, because the joint automaton \( A_{M \otimes \neg \phi} \) is represented as a (permuted) adjacency matrix in the data-oblivious graph algorithms, the two parties essentially compute the entries of the adjacency matrix “on-the-fly”, where each entry at row \( (s, t, j) \) and column \( (s', t', j') \) (which we denote by \( M[(s, t, j)][(s', t', j')] \)) is 1 if and only if

1. \( j = j' \) and there exists \( a \in \Sigma \) such that \( s' \in \rho_1(s, a), t' \in \rho_2(t, a), \) and \( s' \not\in F_1, t' \not\in F_2 \) or
2. \( j = 1 \) and \( j' = 2 \) and \( s \in F_1 \) and there exists \( a \in \Sigma \) such that \( s' \in \rho_1(s, a) \) and \( t' \in \rho_2(t, a) \), or
3. \( j = 2 \) and \( j' = 1 \) and \( t \in F_2 \) and there exists \( a \in \Sigma \) such that \( s' \in \rho_1(s, a) \) and \( t' \in \rho_2(t, a) \).

by the proof of Proposition 1. The predicate “there exists \( a \in \Sigma \) such that \( s' \in \rho_1(s, a), t' \in \rho_2(t, a) \)” can be equivalently expressed as \( \{ a \in \Sigma | s' \in \rho_1(s, a) \} \cap \{ a \in \Sigma | t' \in \rho_2(t, a) \} \neq \emptyset \).
∅, where each set can be computed locally by the individual parties. Thus the problem becomes private set intersection problem, which has been solved using cryptographic tools such as homomorphic encryption in \cite{CLR17} with runtime $O(N_x \log N_y)$ with $N_x$ being the smaller set and $N_y$ being the larger set. Then, we would have, with \(\left[\right]\) enclosing predicates whose truth value can be computed at one party locally and \(p\) representing \(\{a \in \Sigma | s' \in \rho_1(s,a)\} \cap \{a \in \Sigma | t' \in \rho_2(t,a)\} \neq \emptyset\) which can be computed using secure MPC protocol described above, that

\[
M[(s,t,j)][(s',t',j')] = [p] \cdot ([j = j'] \cdot [s' \not\in F_1] \cdot [t' \not\in F_2] + [j = 1] \cdot [j' = 2] \cdot [s \in F_1] + [j = 2] \cdot [j' = 1] \cdot [t \in F_2])
\]

where the value of each predicate can be viewed as encrypted input and the overall computation is the same for any entry of the matrix \(M\).

Therefore, in the following sections, we simply treat each entry of the adjacency matrix as encrypted input without repeating the computations of which it is an encrypted output. Further, by assuming the cryptographic tools such as information-theoretically secure linear sharing schemes and homomorphic encryption (\cite{CDM00}, \cite{Sha79}), we can treat results of simple computations (e.g., linear combination, multiplication) on secret shared values which are locally computed at the two parties the same way.

### 4.3 Method #1: Data-oblivious graph search

Given $L_\omega(A_{M \otimes \phi})$, the model checking problem becomes

\[
\text{check}_{LTL}(\mathcal{M}, \phi) = \begin{cases} 
\text{True} & \text{if } L_\omega(A_{M \otimes \phi}) = \emptyset \\
\text{False} & \text{if } L_\omega(A_{M \otimes \phi}) \neq \emptyset 
\end{cases}
\]

$L_\omega(A_{M \otimes \phi})$ is non-empty if and only if there exists a path from a starting state $s \in S^0$ to a finish state $t \in F$, and that there is also a path from $t$ to itself. The problem hence reduces to graph reachability problems, for which we present data-oblivious graph search algorithms in this section. Note that while we model the given program and its specification both in a temporal logic and consider the resulting automata in terms of a special graph, this is a high level abstraction which assumes away many of the practical concerns a program normally has to deal with.

Formally, we have the definition
Definition 6 (**[BSA13]**). Let $d$ denote input to a graph algorithm. Also, let $A(d)$ denote the sequence of memory accesses that the algorithm makes. The algorithm is considered data-oblivious if for two inputs $d$ and $d'$ of equal length, the algorithm executes the same sequence of instructions and access patterns $A(d)$ and $A(d')$ are indistinguishable to each party carrying out the computation.

4.3.1 Algorithms

Kosaraju’s algorithm finds the strongly connected components (SCC) of a graph, where each SCC is identified with one of its nodes. In this section, we present a data-oblivious algorithm to compute $\text{check}_{\mathit{LTL}}(M, \phi)$, using Kosaraju’s algorithm for strongly connected component finding. By 4.2, in this step our input is the automaton graph of $\mathcal{A}_{M \otimes \neg \phi}$ represented by the adjacency matrix $M$, where state nodes are randomly and consistently permuted and entries are encrypted. In the algorithm below, we use $[\cdot]$ to refer to the encrypted value $\cdot$, and we use $\text{open}([\cdot])$ to refer to the decrypted value.

Before we describe in detail the full algorithm, we acknowledge that the design is inspired by the data-oblivious BFS proposed in **[BSA13]**.

**Algorithm 1:** First, we have input adjacency matrix $[M]$ of the joint automaton graph $G(V, E)$, whose nodes are randomly permuted. We can assume that $|S^0| = 1$, for if $|S^0| > 1$, we can “merge” these starting nodes into one by creating a new node that has outgoing directed edges to all nodes in $S^0$. Hence, let $s_0$ be the starting node.

1. Create array $[C]$ of size $|V|$ by the same node ordering as in $[M]$. Each element of $[C]$ contains two fields: color, and distance from $s_0$. Create and initialize empty array $[K]$ of size $|V|$, and initialize variable $\text{nextEmpty} = 1$. Create array $[L]$ of size $|V|$, where entry $i$ corresponds to node $i$ by the same node permutation as in $[M]$, and initialize each entry so that entry $i$ is 1 if and only if node $i$ is an accepting state, 0 otherwise. Note that this can be easily done through some secure MPC protocol, as each party holds the secret of which node in each automaton is accepting.

2. Initialize $[C]$ so that the distance is 0 for $s_0$ and $|V|$ for all other nodes, and color is gray for $s_0$ and white for all other nodes. Set current node $v$ to be $s_0$.

3. Append $v$ to $[K]$ by setting $[K[\text{nextEmpty}]] = v$, and then increment $\text{nextEmpty} \leftarrow \text{nextEmpty} + 1$.

4. Retrieve row $[M_v]$. Update $[C]$ and $[L]$ using $[M_v]$ as follows:

   (a) for $i = 1$ to $|V|$ do:
(b) \[ \text{cond} = ([M_{v,i}] \geq 1) \cdot ([C_i.color] = \text{white}), \text{cond}' = (L_i > 0) \]

(c) \[ [C_i.color] = \text{cond} \cdot \text{gray} + (1 - \text{cond}) \cdot [C_i.color] \]

(d) \[ [C_i.dist] = [\text{cond}](|C_v.dist| + 1) + (1 - [\text{cond}])[C_i.dist] \]

(e) \[ [L_i] = [\text{cond}'] \cdot 2 + (1 - [\text{cond}']) \cdot [L_i] \]

5. Then obliviously choose one of the gray nodes in \([C]\) with the largest distance from the starting node at random. We do so by first creating and initializing a new array \([C']\) of length \(|V|\) that has two fields, \textit{value} and \textit{key}, so that the \textit{key} holds the random permutation result \(\pi(i)\) for node \(i\), and \textit{value} is \(i\) if and only if the node is one of the nodes of the largest distance from \(s_0\), and then carrying out the following procedure:

(1) \([\text{min}] = |V|\)

(2) for \(i = 1\) to \(|V|\) do

(3) \([\text{cond}_i] = ([C_i.color] = \text{gray})\)

(4) \([\text{cond}'_i] = ([C_i.dist] < [\text{min}])\)

(5) \([\text{min}] = [\text{cond}_i][\text{cond}'_i][C_i.dist] + (1 - [\text{cond}_i][\text{cond}'_i])|\text{min}|\)

(6) for \(i = 1\) to \(|V|\) do

(7) \([\text{cond}''_i] = ([C_i.dist] \geq [\text{min}])\)

(8) \([C'_i.value] = [\text{cond}_i][\text{cond}''_i] \cdot i\)

(9) \([C'_i.key] = [\text{cond}_i][\text{cond}''_i] \cdot [\pi(i)]\)

(10) \([\text{max}] = 0, [i_{\text{max}}] = 0\)

(11) for \(i = 1\) to \(|V|\) do

(12) \([\text{cond}] = (C'_i.key) > \text{max}\)

(13) \([\text{max}] = [\text{cond}][C'_i.key] + (1 - [\text{cond}])|\text{max}|\)

(14) \([i_{\text{max}}] = [\text{cond}][C'_i.value] + (1 - [\text{cond}])[i_{\text{max}}]\)

(15) \(i_{\text{max}} = \text{open}([C'_{i_{\text{max}}.value}])\)

(16) Update \(v\) to be \(i_{\text{max}}\).

6. Repeat step 3-5 above for a total of \(|V|\) times.

7. Now that we have obtained array \([L]\) and \([K]\), we do a second DFS pass to determine if any one of the accepting state loops back to itself. Reinitialize vector \([C]\).
8. for $i = |V|$ to 1 do
9. $[\text{cond}] = (i \leq \text{nextEmpty}) \cdot ([C_i.\text{color}] = \text{white})$.
10. $[v] = [\text{cond}][K_i] + (1 - [\text{cond}]) \cdot 1$.

11. Retrieve row $[M^T_{[v]}]$. Update $[C]$ and $[L]$ by the procedure below:
   (a) for $j = 1$ to $|V|$ do:
   (b) $[\text{cond}] = (M^T_{v,j} = 1) \cdot ([C_j.\text{color}] = \text{white}), [\text{cond}'] = (L_j = 2)$.
   (c) $[C_j.\text{color}] = [\text{cond}] \cdot \text{gray} + (1 - [\text{cond}]) \cdot [C_j.\text{color}]$
   (d) $[C_j.\text{dist}] = [\text{cond}][C_v.\text{dist}] + 1 + (1 - [\text{cond}])[C_j.\text{dist}]$
   (e) $[L_j] = [\text{cond}'] \cdot 3 + (1 - [\text{cond}']) \cdot [L_j]$

12. Then obliviously choose one of the gray nodes in $[C]$ with the largest distance from node $[v]$ at random, following the same procedure as in Step 5. above.

13. Repeat step 11-12 for a total $|V|$ times.

14. $\text{ret.val} = ([L_1] = 3) \lor \cdots \lor ([L_{|V|}] = 3)$, with $\lor$ the OR operator.

15. Finally, output $\text{ret.val}$.

**Algorithm 2:** Instead of doing two DFS passes, we can instead do one BFS/DFS pass on a doubled graph, constructed by first making a copy $G'$ of $G$, and then duplicating each outgoing edge from an accepting state node in $G$ and making the duplicate edge point to the corresponding duplicate node in $G'$. In this new graph, the starting states are the same ones in $G$, and the accepting states are the duplicate accepting states in $G'$. Note that the adjacency matrix for this doubled graph is computable via secure MPC. Then using either oblivious BFS in [BSA13] or the oblivious DFS as above, together with the additional array $L$ and $K$ used as above in **Algorithm 1**, we obtain an alternative algorithm.

**Algorithm 3:** An alternative to Kosaraju’s algorithm is the Tarjan’s algorithm, which has a single DFS traversal instead of two. Here again, we have input graph $G = (V, E)$ represented by adjacency matrix $[M]$, where the nodes are randomly and consistently permuted, and source node $s_0$. Tarjan’s algorithm has a single DFS traversal via a recursive function. To make it data-oblivious, we would need to emulate the recursion stack, by keeping a stack that keeps not only the vertices we pushed onto the stack, but also the state of the program when the each node was pushed onto the stack. Then popping a node off the stack would be equivalent to the program returning from the recursive call. While we can hide the varying size of the stack by the same technique as used in algorithm 1, this would erase the difference
between Kosaraju’s and Tarjan’s algorithm, for then we would have to do two passes of DFS for Tarjan’s algorithm as well. Hence we omit the construction here.

4.3.2 Discussion of space and time complexity

We first have the result that the non-emptiness problem for Büchi automata is decidable in linear time and is NLOGSPACE-complete, meaning that checking whether $M$ satisfies $\phi$ can be done in $O(|M| \times 2^{O(|\phi|)})$ time. By making the algorithm oblivious, we have made the runtime $O(|M|^2 \times 2^{O(|\phi|)})$. Space is $O(|M| \times 2^{O(|\phi|)})$. This exponential complexity in regards to the size of the formula we wish to verify is known as the state explosion problem, which we also saw in the algorithm proposed in [LP85]. Note that in many cases, we are interested in program properties which can be expressed very succinctly in LTL, such as liveness, safety, and mutual exclusion, and hence, with $|\phi| << |M|$ in these cases, the complexity may not be so forbidding as to render it useless in practice.

4.3.3 Proof of Security

With the underlying assumption of secure multi-party computation protocols, we only need to check that our algorithms are data-oblivious. This is clear from the fact that all memory regions we used had sizes only dependent on the size of the input, and that the patterns of access to these regions are either in a uniform fashion such as by ascending order of memory address or in a pseudo-random fashion where the security reduces to the security guarantee of pseudo-random functions.

4.4 Method # 2: Reduction to CTL Model checking with Fairness Conditions

Given $M, \phi$ and starting state $s^0$ (note that in case there are multiple starting states, we can simply create an artificial starting state to unify all these starting states into one), we want to know whether $M, s \models \phi$. The general plan of attack is three-fold:

1. Construct $A_{\neg \phi}$ for input specification formula $\phi$, such that $A_{\neg \phi}$ encodes precisely those traces that satisfy $\neg \phi$.

2. Combine $A_{\neg \phi}$ and $M$ by the procedure we outlined above in [1] so that the traces accepted by the resulting automaton are precisely those accepted by both $A_{\neg \phi}$ and $M$ individually.

3. Find out whether there exists a path in this combined automaton from a state derived (logically) from $s^0$. The absence of such a path proves $M, s \models \phi$, and the presence of
it serves as a counterexample.

It is at step 3 that we can reduce the model checking problem into CTL model checking with fairness conditions, for which we have privacy-preserving model checking protocols developed by Sam and Ning.

### 4.4.1 Construction of $A_\phi$

We first outline the general steps for constructing $A_\phi$, where $\phi$ is an LTL formula in its initial form possibly expressed with any logical connectives, and by the equivalences we outlined above, we may assume that the only temporal connectives are $U$ and $X$.

1. Get the set of states $S$ for the automaton. Each state $q \in S$ is a maximal subset of $C(\phi)$, the closure of formula $\phi$ defined as the set of subformulas of $\phi$ and their complements, identifying $\neg \neg \psi$ and $\psi$. For instance, we have that $C(\neg aUb) = \{a, b, \neg a, \neg b, \neg aUb, \neg(\neg aUb)\}$. The states of $S$ satisfy the following:
   - For all (non-negated) $\psi \in C(\phi)$, either $\psi \in q$ or $\neg \psi \in q$, but not both.
   - $\psi_1 \lor \psi_2 \in q$ holds iff $\psi_1 \in q$ or $\psi_2 \in q$, whenever $\psi_1 \lor \psi_2 \in C(\phi)$.
   - $\psi_1 \land \psi_2 \in q$ holds iff $\psi_1 \in q$ and $\psi_2 \in q$, whenever $\psi_1 \land \psi_2 \in C(\phi)$.
   - $\psi_1 \rightarrow \psi_2 \in q$ holds iff $\psi_1 \in q$ implies $\psi_2 \in q$, whenever $\psi_1 \rightarrow \psi_2 \in C(\phi)$.
   - If $\psi_1 U \psi_2 \in q$, then we have $\psi_1 \in q$ or $\psi_2 \in q$.
   - If $\neg(\psi_1 U \psi_2) \in q$, then we have that $\neg \psi_2 \in q$.

Thus, the states of automaton $A_\phi$ encodes information for the truthfulness of the subformulas of $\phi$, by representing the different states of truthfulness with different states in the automaton. (i.e., each state is an element in the power set of $C(\phi)$ such that that a subformula belonging to the state represents the subformula being true in this state.)

2. The initial states $S^0$ are those states that contain $\phi$.

3. Define transition relation $\delta$ (in set notation) such that $(q, q') \in \delta$ holds iff all of the following conditions hold:
   - if $X \psi \in q$ then $\psi \in q'$;
   - if $\neg X \psi \in q$ then $\neg \psi \in q'$;
   - if $\psi_1 U \psi_2 \in q$ and $\psi_2 \notin q$, then we have $\psi U \psi_2 \in q'$;
   - if $\neg(\psi_1 U \psi_2) \in q$ and $\psi_1 \in q$ then $\neg(\psi_1 U \psi_2) \in q'$. 

15
Thus, the transition relation $\delta$ of $A_\phi$ encodes the logical relations between these subformulas. For instance, the last two conditions correspond to the following recursive relation:

$$\psi_1 U \psi_2 = \psi_2 \lor (\psi_1 \land X(\psi_1 U \psi_2))$$

$$\neg(\psi_1 U \psi_2) = \neg \psi_2 \land (\neg \psi_1 \lor X \neg(\psi_1 U \psi_2))$$

4. Define additional accepting conditions for dealing with Until subformulas in the closure: Let $a_1Ub_1, \ldots, a_kUb_k$ be all subformulas (less their negation) in $C(\phi)$ that contain connective $U$. A run is accepted if, for every $i$ such that $1 \leq i \leq k$, the run has infinitely many states satisfying $\neg(a_iUb_i) \lor b_i$, i.e., $a_iUb_i \rightarrow b_i$. The reason for this is simply that we want to ensure that the eventualities (in the definition of $U$) are satisfied. Also note here the correspondence with the accepting condition of the Büchi automaton.

By the equivalences we outlined above, just knowing the construction of automaton $A_{aUb}$ and $A_{Xa}$ and their negations is sufficient for constructing the corresponding automaton $A_\phi$ for any $\phi$.

Here we present the more complex construction for $A_\phi$, where $\phi = aUb$.

**Until $(\phi := aUb)$**

Given the accepting state, Figure 1 is an automaton that encodes $\phi = aUb$, and Figure 2 is the automaton for $aUb$ by following the general steps outlined above. Notice the difference here is that each state in Figure 2 also evaluates the truth value of $\phi$. The edges in Figure 2 are unlabeled, and that is because the labels have been “internalized” into the states, while the transition relation $\delta$ defines all possible transitions. This is important, as we want to know whether under all possible circumstances, $M, s^0$ implies $\phi$.

The construction for $\phi = Xa$ and $\phi = \neg a$ are much more straightforward, which only concern
point 1 in Step 3. and point 1 in Step 1. above.

4.4.2 Combining $A_{\neg \phi}$ and $M$

By the same procedure as we outlined in Proposition 2 to obtain $M \times A_{\neg \phi}$.

4.4.3 Path finding in the combined automaton

First, we explain fairness constraints in CTL model checking. The motivation for fairness constraints added to CTL model checking is that sometimes, we have some pre-knowledge about our program $M$ such that we know some theoretically possible behaviors will never happen and that we can safely assume they will never happen when reasoning about $M$. Thus, instead of checking $M, s^0 \models \phi$, we often want to instead check whether $M, s^0 \models \psi \rightarrow \phi$, where $\psi$ serves as a filter condition that filters out those unrealistic behaviors which we know in advance are irrelevant to our verification purposes. Below is a formal definition.

Definition 7. Let $C = \{\psi_1, \ldots, \psi_n\}$ be a set of fairness constraints. Then a computation path starting at $s^0$ is fair with respect to these constraints if and only if for each $\psi_i$, there are infinitely many states $s$ along this path such that $s \models \psi_i$.

That is, a computation path found by CTL is fair under the given constraints if and only if each constraint is true infinitely often along the path.

Then, in this last step, we can simply reduce the problem to CTL model checking with fairness constraints, where the model is $M \times A_{\neg \phi}$, the formula to check is $\text{EG} \top$, and the fairness conditions are $\neg (\chi_i U \psi_i) \lor \psi_i$ for each subformula of the form $\chi_i U \psi_i \in C(\phi)$. The formula $\text{EG} \top$ effectively checks for the existence of a path, while the fairness condition checks, for each $i$, that $\neg (\chi_i U \psi_i) \lor \psi_i$ holds infinitely often along the path found. Notice again the correspondence between this added fairness constraint and the accepting condition of Büchi automaton. Then, finally, the problem reduces to Ning and Sam’s work on privacy-preserving model checking for CTL.

4.4.4 Proof of Security

The first step can be done locally, the second step is part of Method #1 whose security proofs we’ve given, and the third step is reduced to the security proof of the privacy-preserving model checking protocol for CTL.
5 Next Steps

Going into the details of the program $\mathcal{M}$ which for now is modeled at a very high level, as well as the specifics of the property that we care about. For instance, reason about mutex and liveness of concurrent programs. Also explore more efficient model checking methods such as on-the-fly LTL model checking.

References


[Var01] Moshe Y Vardi. Branching vs. linear time: Final showdown. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems,