Privacy Preserving CTL Model Checking through Oblivious Graph Algorithms

Samuel Judson  
Yale University  
samuel.judson@yale.edu

Ning Luo  
Yale University  
ing.luo@yale.edu

Timos Antonopoulos  
Yale University  
timos.antonopoulos@yale.edu

Ruzica Piskac  
Yale University  
ruzica.piskac@yale.edu

ABSTRACT

Model checking is the problem of verifying whether an abstract model $M$ of a computational system meets a specification of behavior $\phi$. We apply the cryptographic theory of secure multiparty computation (MPC) to model checking. With our construction, adversarial parties $D$ and $A$ holding $M$ and $\phi$ respectively may check satisfaction — notionally, whether $M \models \phi$ — while maintaining privacy of all other meaningful information. Our protocol adopts oblivious graph algorithms to provide for secure computation of global explicit state model checking with specifications in Computation Tree Logic (CTL), and its design ameliorates the asymptotic overhead required by generic MPC schemes. We therefore introduce the problem of privacy preserving model checking (PPMC) and provide an initial step towards applicable and efficient constructions.

CCS CONCEPTS

• Security and privacy → Privacy-preserving protocols; • Theory of computation → Cryptographic protocols; Verification by model checking: Logic and verification;

KEYWORDS

privacy; model checking; cryptography; multiparty computation; verification; temporal logic

ACM Reference Format:


1 INTRODUCTION

The techniques and theory of formal methods provide valuable confidence in the correctness of programs and protocols. However, these tools are often costly to employ in both computational effort and human effort. Their use is biased towards applications where failures bring substantial economic or social cost — and commensurate legal risk and attention. The verification of cryptographic libraries and protocols has become a recent focus of research [7] as the use of cryptography to secure user data has come under regulations such as the GDPR [23]. A classic domain for formal methods is cyberphysical systems in aerospace engineering, transportation, medicine, and industrial control systems [19, 55]. All are heavily regulated (in the United States) by various federal and state agencies such as the FAA and FDA. It is no coincidence that often those settings that have seen the greatest use of formal methods are those which are most closely governed, and receive the most intense regulatory and legal scrutiny.

The traditional story of formal verification does not consider conflicting purposes. Usually an engineer has a program, derives a mathematical formalism representing its behavior, and automatically checks that behavior meets a given specification of correctness — all through their own modeling and computational effort [19, 55]. But this may be insufficient when the requirement for that verification is imposed by an external party, such as a regulator. An analysis is only as good as the modeling of the computation, the quality of the tools, and the soundness of the assumptions. Rather than trust procedure, a regulator may reasonably prefer to execute the formal verification themselves or through a trusted, technically adept agent.

Such an effort may clash with concerns of privacy and propriety. A vehicle manufacturer or high-frequency trader might doubt that a government regulator will respect the privacy of code of immense economic value, or might doubt that employees of that regulator will not carry knowledge to their competitors through a revolving door. A concrete example arises in private governance, as Apple and Google regulate distribution of apps on their mobile device platforms, a role which gives them special access to the software of the competitors who create third-party alternatives to their own services. The anti-competitive uses of this power have come under scrutiny — such as in 2019 when Apple removed parental control apps they had long allowed just after integrating competing functionality through a device platform.

Nonetheless, Apple and Google have compelling economic and social justifications in requiring app review before allowing distribution. Static analysis tools have been developed to evaluate apps for malware and privacy invasion through tracking [5, 20, 21, 47]. The use of such tools during review may prevent proliferation.
of harmful software for the benefit of both users and platforms. Further, the example of app store maintainers raises that privacy concerns regarding verification may go in both directions. A tool such as PIOS [20] evaluates the potential for data exfiltration by iOS apps. But what information, and to what extent, is considered unnecessary or harmful may be nebulous, personal, or dependent on context. Any line drawn would be arbitrary, and a regulator may wish to keep their requirements private so as to not present a simple and static target.

Our work commences a study of the use of applied cryptography to mitigate this tension between privacy and assurance. To allow two parties to execute a formal verification — in this case, by way of model checking — while maintaining privacy over both the program or protocol being verified and the specification of behavior it is required to meet.

We consider a setting where an auditor A wishes to verify a program held by D, the developer. D possesses a model of program execution M rendered as a graph-theoretic Kripke structure while A has a specification of program behavior ϕ written in Computation Tree Logic (CTL) [14, 16, 17]. We construct an interactive protocol to decide whether M ⊨ ϕ, i.e., whether the specification holds over the model. By use of the cryptographic theory of secure multiparty computation (MPC) [8, 25, 26, 29, 43, 44, 48, 57], our protocol severely limits the information D and A learn about the input of the other under standard adversarial assumptions. Moreover, our protocol runs in local and communication complexities O(∥ϕ∥ + |M|) and therefore requires no asymptotic overhead. Our work adopts and combines recent advances in efficient MPC execution, secret sharing, and data-oblivious algorithms, particularly over graphs [10].

We note that the utility of our protocol requires that D inputs an M which accurately and adequately represents the program execution. Systemic factors must motivate honest inputs by the parties. In a regulatory setting, this may be because of substantial punitive powers or legal recourse available to A should they learn of dishonest behavior, or because they provide the tools necessary for model extraction. For example, Apple and Google provide tooling for application development on their platforms. As with all privacy engineering, our construction requires careful consideration of how it fits into the broader system to make sure its privacy and correctness goals are practically met. Even if not fully absolving the need for trust or binding agreement between developer and auditor, our protocol recasts harm from the potentially murky and indeterminate ‘did the auditor gain valuable information from M?’ to the incontrovertible ‘did the developer misrepresent M?’, which may make asymmetrical privacy and correctness concerns easier to negotiate. We discuss relevant related work and potential future directions in §7.

In summary, this paper contributes (i) recognizing that privacy concerns may arise in the use of modern program analysis and verification techniques; (ii) observing that the graph-theoretic nature of model checking renders it amenable to approach through oblivious graph algorithms; (iii) the full design and implementation of an MPC protocol for privacy preserving model checking; and (iv) an experimental evaluation of that construction.

We proceed as follows. In §2 we introduce both model checking of CTL and our necessary cryptographic primitives. Our contributions begin in §3, with data-oblivious model checking subroutines based on prior work for oblivious graph algorithms. We then give our full model checking construction in §4. We follow with discussion of our implementation and experimentation in §5, cover related work in §6, and consider potential future work and conclude in §7.

2 PRELIMINARIES

The best known temporal modal logics are Linear Temporal Logic (LTL) operating over program traces, Computation Tree Logic (CTL) operating over the computation tree of program traces, and their superset CTL* [14, 16, 17, 50]. Each are propositional logics extended with temporal operators X (at the next), F (Finally, i.e. eventually), G (globally, i.e. always), and U (until), while CTL and CTL* add quantifiers E (exists a branch) and A (for all branches) over the tree. CTL allows expression of statements such as AG (userdata → AG ~network) where userdata and network are atomic predicates over the program state. Verifying a program meets such a specification then assures that whenever it accesses user data it does not later invoke networking functionality. In this manner, temporal logics allow expressing liveness (something must always happen) and safety (something must never happen) properties of a computation.

CTL requires temporal operators be directly preceded by a quantifier. This requirement allows it to be model checked in polynomial time through a relatively straightforward and efficient recursive algorithm, whereas model checking LTL and CTL* have been shown to be PSPACE-complete [15, 52]. As such we limit our attention to CTL, and leave LTL and CTL* to future work. The interested reader may find far more comprehensive discussions of these logics, their similarities and differences, and their checking in [16, 17].

Secure multiparty computation (MPC) is the cryptographic problem of executing an algorithm where the inputs are held by different parties, such that no participant learns any information other than what is implied by their own input and the output. We will restrict our interest to secure two-party computation (2PC), as it fits our setting and simplifies analysis as parties need not be concerned with collusion — we will somewhat improperly use both terms interchangeably within this paper. Generic techniques for secure computation of circuits — potentially employed with oblivious RAM — may be used ‘off-the-shelf’ to provide 2PC for any computable function, but at cost of at least logarithmic overhead [8, 25, 26, 29, 43, 44, 48, 57]. Instead, we will present a tailored protocol for our problem with minimal leakage and no additional asymptotic cost.

We proceed with short introductions on both topics.

2.1 Model Checking

A Kripke structure [16, 17] is a standard formalism used to abstractly represent the possible executions of a program. It is defined as a tuple $M = (S, I, δ, L)$, where $S$ is a set of states with $n = |S|$, $I ⊆ S$ a set of initial states, $δ ⊆ S × S$ a transition relation — with $(s_i, s_j) ∈ δ$ for $i, j ∈ [n]$ denoting that $s_j$ is a successor of $s_i$ — and $L : S → 2^{M}$ a labeling function mapping states to subsets of the $q$ available labels. We note that $O(n^2) = |M|$. 
We assume that where each
as a constant, due to its being a systemic parameter rather than an instance-specific input. We assume that $\delta$ is left-total, so that every state has at least one successor (possibly itself). We let $\ell_k$ for $k \in \{q\}$ denote an arbitrary label, and define the Boolean function $\ell_k(s)$ to indicate whether label $\ell_k$ is assigned to state $s$.

Each $\ell_k$ label corresponds to some predicate, and $\ell_k(s)$ indicates whether that predicate is true at $s$. In Example 2.1, the labels capture the knowledge the system has of user session status at each state in its execution. For instance, at $s_2$ it is true that the user has provided valid credentials, but it is false that they have had a session established. We presume a given $M$ is a sound representation of a computation, but beyond that how it is derived from a specific program or protocol is beyond our concern.

The essential structure of $M$ is the directed graph induced by $\delta$ where each $s \in S$ is treated as a vertex. Originating from an initial (source) state, the set of infinite walks on this graph may be viewed as a computation tree of infinite depth. Every initial state in $I$ produces a different tree. Each infinite walk (or trace) through a tree corresponds to an infinite walk through the directed graph representation of $M$. These traces must capture all possible behaviors of the program represented by $M$ with respect to the label predicates. We concern ourselves with discrete timing, so that

Figure 1: A Kripke structure (left) modeling the program given in Example 2.1, and its corresponding computation tree (right). The vertices and edges in the computation tree show a failed checking for $A \ell_1 U \ell_3$, which holds at $s_2$ and $s_3$ (dashed blue) but not $s_0$ or $s_1$ (dotted orange).
made ready’ for a state being added to ‘ready’ states
analogous to a breadth-first search — although we don’t need to on the existence of a path or paths through similar states to an subroutine can determine whether to see if such a successor exists, using the output of the recursive
Algorithm 1: The checkCTL algorithm up to the quantified until operator subroutines and various helper functions.

Boolean operator. Moreover, s \models \text{EX } \phi \iff there exists an s' such that (s, s') \in \delta and s' \models \phi. So we may iterate over all state pairs to see if such a successor exists, using the output of the recursive call. That these algorithms are O(n), O(n), and O(n^2) respectively is straightforward. Notably, for all \phi' \in \{\psi \land \phi, \neg \psi, \text{EX } \psi\} the relevant subroutine can determine whether s_j \models \phi'' without consideration of s_j \models \phi' for any j \neq i. Each state may be processed in isolation, a trait which will shortly be of great convenience.

Such is not true for \text{E } \psi \text{ U } \phi and A \text{ E } \psi \text{ U } \phi. For any state s for which s \models \psi yet s \not\models \phi, the truth of the formula is dependent on the existence of a path or paths through similar states to an s' for which s' \models \phi. As such, we may not just look directly to the output of the recursive calls to determine satisfaction — rather we’ll have to build any such paths, which we may do efficiently by working backwards. The essential insight is that the algorithm is analogous to a breadth-first search — although we don’t need to be concerned with the depth of the vertex, and we do need to handle labels. Instead of emanating out from the source to the periphery, we start at the periphery of states where s \models \phi, and walk back towards the source.

To do so, in Algorithm 2 and Algorithm 4 we initialize a set of ‘ready’ states \mathcal{R} to those states for which \phi holds, and then pull an ‘active’ state s in each loop iteration. We will use the language ‘made ready’ for a state being added to \mathcal{R}, and ‘made active’ for a state being chosen as s. We then walk over the predecessors of s, and (for EU) add the predecessor if \psi holds at it or (for AU) add the predecessor if all the successors of it have been active, tracked using a decremented counter. The formula then holds at any state which is ever added to \mathcal{R}, and we use another set \mathcal{K} to track ‘known’ states so that we do not add a state to \mathcal{R} multiple times. Since a state may be active exactly once and when it is we review all n possible predecessors, both these algorithms are O(n^2).

The model checking of CTL and its optimizations are discussed at far greater length in [16, 17], and we refer to them the interested reader who finds our discussion overly terse. We conclude with the following theorem, and refer to [15] for a proof.

**Theorem 2.1.** For any Kripke structure \(\mathcal{M} = (S, I, \delta, L)\) and CTL formula \(\phi\)

(1) \(\text{check}_{\text{CTL}}(\mathcal{M}, \phi) = 1\) if and only if \(\mathcal{M} \models \phi\); and

(2) \(\text{check}_{\text{CTL}}(\mathcal{M}, \phi)\) runs in time \(O(mn^2)\) where \(|\mathcal{M}| = O(n^2)\) and \(|\phi| = m\).

**2.2 Privacy Preserving Computation**

Secure multiparty computation (MPC) provides for the joint computation of a function \(f(x_1, \ldots, x_k) = y\) when \(f\) is public and each \(x_i\) is the private input of mutually distrustful parties. We require that the computation of \(y\) be correct, but at the conclusion a party \(i\) should know nothing more about \(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_k\) than what is implied by \(x_i\) and \(y\). In our setting, we will concern ourselves solely with secure two-party computation (2PC).

Our construction will provide privacy in the semi-honest model\(^1\) — we assume that our parties follow the protocol as prescribed honestly, but still attempt to learn about the input of the other party to the extent possible. This is in contrast to malicious security, where the parties may violate the protocol to try and learn information. Proving privacy in the semi-honest model falls under the simulation paradigm. Suppose we wish to design a protocol \(\Pi\) to compute \(f(x_0, x_1) = y\) where \(x_0\) is the input of \(A\) and \(x_1\) is the input of \(B\). Let \(\lambda \in \Pi\) be a security parameter. Define \(\text{view}_A(\Pi, x_0, B(1^\lambda, x_1))\) to be the view of \(A\) when interactingly computing \(f\) with \(B\): an object containing \(x_0\), every random coin flip \(A\) samples, every message \(A\) receives from \(B\), every intermediary value \(A\) computes, and \(y\). The view captures all information known to \(A\) at the conclusion of the joint computation of \(f\).

We prove privacy by showing that we can replace \(B\) with a probabilistic polynomial-time (PPT) simulator \(\text{Sim}_B(1^\lambda, x_0, y)\) such that \(A\) cannot distinguish between an interaction with \(B\) and \(\text{Sim}_B\). Note that \(\text{Sim}_B\) takes only public information and the information of \(A\) — by definition \(A\) cannot learn anything from interacting with it. Formally, we model \(A\) as a PPT adversary \(\mathcal{A}\) who must attempt to distinguish between \(\text{view}_A(\Pi, x_0, B(1^\lambda, x_1))\) and \(\text{view}_A(\Pi, x_0, \text{Sim}_B(1^\lambda, x_0, y))\) by outputting a bit \(b \in \{0, 1\}\) identifying which counterparty they are interacting with. We will define secure computation of a function \(f : X_0 \times X_1 \rightarrow Y\) by a protocol \(\Pi\) on behalf of \(B\) if for all PPT adversaries \(\mathcal{A}\) and all \(x_0 \in X_0\) and \(x_1 \in X_1\), \(\text{view}_A(\Pi, x_0, B(1^\lambda, x_1))\) and \(\text{view}_A(\Pi, x_0, \text{Sim}_B(1^\lambda, x_0, y))\) are

\(^1\)Also known as the honest-but-curious model.
We call the left hand the real world and the right hand the ideal world. We refer the interested reader to [24, 43, 44] for more verse of every polynomial function of the form
\[ a \text{ bitstring of length } k \] be written, operated upon, and read for copying. The \( k \) the passing of messages. The \( a \) pair of Turing machines with access to shared tapes facilitating to communicate with each other. This object may be formalized as RAM.

We notate computational indistinguishability of views by
\[ \text{view}_{\Pi}(\Pi, x_0, B(1^k, x_1)) \approx \text{view}_{\Pi}(\Pi, x_0, \text{Sim}_{B}(1^k, x_0, y)). \]

We call the left hand the real world and the right hand the ideal world, as privacy follows by definition within the latter. Since Sim_{B} is constructed based only on the knowledge of A and the output of the computation, were information leaked by B in the real world then A would be able to use that information to distinguish between the interactions. We may prove privacy for the inputs of A identically, by constructing a Sim_{A}(1^k, x_1, y) such that the view of B in the resultant ideal world is also indistinguishable from the real world. We refer the interested reader to [24, 43, 44] for more formal treatments and further discussion of the theory of multiparty computation.

**Multiparty Computation Primitives.** Generic techniques are known which provide for secure computation of all computable functions with logarithmic asymptotic overhead and computational (or better) security [8, 25, 26, 29, 43, 44, 48, 57]. Of these primitives, our work will make use of garbled circuits with oblivious transfer in the semihonest model due to Yao [43, 57]. However, we will not simply be rendering the decision procedure check_{CTL}(M, \phi) as a circuit. Rather, we will employ constant-sized binary circuits for certain intermediary computations, and then use data-oblivious computation [26] and compositionality [12] to combine these intermediary results to execute the full checking.

For concision we will not delve into the details of garbled circuits or their underlying cryptographic components. Rather, for the remainder of the discussion we will assume access to a protocol \( y \parallel y \leftarrow \text{GC}(c; x_0 \parallel x_1) \), such that if \( c \) is a circuit description of a function \( f \), then GC securely and correctly computes \( y = f(x_0, x_1) \). Here, our notation \Pi(x_0 \parallel x_1) indicates that protocol \Pi is interactively executed between two parties with inputs \( x_0 \) and \( x_1 \) respectively, while \( y \parallel y \) indicates which parties receive the output. It is possible to execute a garbled circuit computation so that either party — or both — receive it. We will make use of this flexibility throughout our construction. We also note that \( \text{GC}(c; \cdot \parallel \cdot) \) maintains the asymptotic complexity of \( c \).

**Data-Oblivious Computation.** Our treatment of data-oblivious computation follows that of Goldreich and Ostrovsky in [26] in the random access machine (RAM) model of computation. We define a RAM as composed of two components, \( \text{RAM} = (CPU, MEM) \), able to communicate with each other. This object may be formalized as a pair of Turing machines with access to shared tapes facilitating the passing of messages. The CPU machine contains some constant number of registers each of size \( k \) bits, into which information may be written, operated upon, and read for copying. The MEM machine contains \( 2^k \) words, each of constant size \( w \), and each addressed by a bitstring of length \( k \). The CPU sends messages to MEM of the form \( (i, a, v) \) where \( i \in \{0, 1\}^2 \) represents one of write, read, or halt, \( a \in [2^k] \) is an address, and \( v \in [2^w] \) a value. Upon receipt of a \( \text{write} \) command \( \text{MEM} \) copies \( v \) into the word addressed by \( a \), upon a \( \text{read} \) command returns the current value in the word addressed by \( a \), and upon \( \text{halt} \) output some delineated segment of memory, such as the segment of lowest addressed words until that containing a special value is reached.

A RAM is initialized with input \( (s, y) \), where \( s \) is a special start value for the CPU, and \( y \) an initial input configuration to MEM which writes both program commands and data values into various addresses of MEM. We denote by MEM(y) the memory when initialized with \( y \), and CPU(s) analogously. The RAM then executes by reading commands and data to registers of the CPU, computing on them while there, and writing back to MEM, before finally issuing a halt command. We denote the output of this computation by RAM(s, y), and can define a corresponding access pattern. The access pattern of a RAM on input \( (s, y) \) is a sequence \( \text{AP}(s, y) = (a_1, \ldots, a_i, \ldots) \) such that for every \( i \), the \( i \)th message sent by CPU(s) when interacting with MEM(y) is of the form \( ('a_i', \cdot) \).

To formulate a definition of a data-oblivious program, we first split the input \( y \) into two substrings, a program \( P \) and data \( x \), so that \( y = (P, x) \). Then, we say a program \( P \) is data-oblivious with respect to an input class \( X \), if for any two strings \( x_1, x_2 \in X \), should \( \text{AP}(P, x_1) \) and \( \text{AP}(P, x_2) \) be identically distributed, then so are \( \text{AP}(P, x_1) \) and \( \text{AP}(P, x_2) \). Intuitively, an observer learns nothing more than the length of the inputs from the access patterns of a data-oblivious program.

We restrict our inputs to a class \( X \) as a form of `promise’ that the inputs are interpretable as the objects of the correct structure, which we may reasonably assume in the semihonest model. Our analysis of data-oblivious computation will be natural for inputs of the same structural length — pairs \( M, M’ \) such that \( M.n = M’.n \), and pairs \( \phi, \phi’ \) such that \( M.n = \phi’.m \). So we will further assume a standardized input format so that \( |\text{AP}(P, x_1)| = |\text{AP}(P, x_2)| \) for all \phi, and \( |\text{AP}(M, \phi)| = |\text{AP}(M, \phi’)| \) for all \( M \).

Given a data-oblivious computation — either (i) a data-oblivious algorithm, or (ii) any program which has been made oblivious by an application of Oblivious RAM (ORAM) — an MPC protocol follows [29]. As the control flow of the program is fixed and known publicly, both parties may follow it in lockstep. All intermediary computation over variables is done using a suitable protocol for secure computation of binary or arithmetic circuits [8, 25, 43, 44, 57]. The one final component is a scheme for secret sharing, which allows intermediary values for each variable to remain private during the execution of the program. In our protocol we will also take advantage of a particular secret sharing scheme which allows some additional flexibility to the computation — \( A \) will be able to vary their inputs to certain intermediary computations based on \( \phi \), at some additional concrete cost.

**Secret Sharing.** A secret sharing scheme allows a value \( x \) to be stored communally by two parties. The collaboration of both are required to reconstruct \( x \). We will employ two secret sharing schemes. The first, \( \Pi_{\text{ot}} = (\text{Share}^{\text{ot}}, \text{Reconstruct}^{\text{ot}}) \), operates as follows. To share a value \( x \in \mathbb{Z}_2 \), denoted \( [x] \), \text{Share}^{\text{ot}}(x) \) uniformly samples \( a \leftarrow \mathbb{Z}_2 \) and computes \( b \leftarrow x - a \text{ (equiv. } x \oplus a \). One party
holds a as a share, the other party b. \(\text{Reconstruct}^{\text{otp}}(a, b)\) computes \(x \leftarrow a + b\) (equiv. \(a \oplus b\)). We may secret share arbitrarily long bitstrings by sharing each bit separately with \(\Pi^{\text{otp}}\) using independent randomness. Although for brevity we omit the formal security definition of secret sharing, it is straightforward to see that given just one of \(a\) or \(b\), the value of \(x\) is uniformly distributed and so the scheme hides it with information-theoretic security.

The scheme \(\Pi^{\text{ref}}_S = (\text{Gen}^{\text{prf}}, \text{Share}^{\text{prf}}, \text{Reconstruct}^{\text{prf}})\) requires the existence of a pseudorandom function (PRF). This is a keyed function \(\text{PRF} : \{0, 1\}^k \times \{0, 1\}^* \rightarrow \{0, 1\}^*\) for \(\lambda, y, z \in \mathbb{N}\) for which the distribution of \(\text{PRF}(sk, x)\) is computationally indistinguishable from uniformly random for an adversary which does not know \(sk\). We let \(\text{Gen}^{\text{prf}}(1^n)\) be the key generation algorithm for the PRF which is run at setup. To share a value \(x \in \{0, 1\}^*\), denoted \([\left[ x \right]]\), \(\text{Share}^{\text{prf}}(x)\) uniformly samples \(r \overset{\$}{\leftarrow} \{0, 1\}^y\) and computes \(c \leftarrow \text{PRF}(sk, r) \oplus x\). One share is then \(sk\), and the other is \((c, r)\). To reconstruct the value, \(\text{Reconstruct}^{\text{prf}}((c, r))\) computes \(x \leftarrow \text{PRF}(sk, r) \oplus c\).

The sharing and reconstruction algorithms given are identical to a standard construction for producing a semantically secure symmetric key scheme for multiple encryptions out of a PRF [37]. By using an encryption scheme for secret sharing, we have the benefit that we can have multiple shared values \([\left[ x_1 \right]]\), with one party having the same share — \(sk\) — for all of them. This allows the other party to vary which \((c, r)\) they input into a given intermediary computation [49]. The cost for this flexibility is that \(\Pi^{\text{ref}}_S\) is far more computationally expensive than \(\Pi^{\text{otp}}\), particularly as we will need to execute these secret sharing schemes — and so our PRF — within garbled circuits. We use AES-128 for \(\Pi^{\text{ref}}_S\) as it is commonly modeled as a PRF, which in our implementation requires 5440 encrypted gates within a garbled circuit to share or reconstruct, while \(\Pi^{\text{otp}}_S\) requires no such gates due to Free-XOR techniques [41, 58].

If ambiguous, we will note that the key share for a \(\Pi^{\text{ref}}_S\) share is \(sk\) by \([\left[ x \right]]\)\(_S\). We will abuse notation by, given a vector \(x = (x_1, \ldots, x_N)\), using \(\hat{x}\) to represent \([\left[ x_1 \right]], \ldots, [\left[ x_N \right]]\) and similarly for \([\hat{s}]\) and \([\left[ x_1 \right]], \ldots, [\left[ x_N \right]]\)\(_S\). We will also write \([z] \leftarrow f([x]), [y])\) as shorthand for

\[(\| b_3) \leftarrow \text{GC}(f'; a_1, a_2, a_3 \| b_1, b_2)\]

where \(a_1 + b_1 = x, a_2 + b_2 = y, a_3 + b_3 = z\), and \(f' = (\text{Share}^{\text{otp}} \circ f \circ \text{Reconstruct}^{\text{otp}})\). It will be particularly common for us to write \([z] \leftarrow [x] \land [y]\) or similar for various binary operations. To take advantage of the opportunity for increased efficiency where our protocol adapts truly data-oblivious processing, we will prefer to use \(\Pi^{\text{otp}}_S\) over \(\Pi^{\text{ref}}_S\) whenever possible. So, we define two algorithms, \([x] \leftarrow \text{simplify}([\left[ x \right]])\) and \([x] \leftarrow \text{complicate}(x)\) which simply compose reconstruction from one secret sharing scheme and sharing from the other as necessary.\(^2\) We let \([x] \leftarrow \text{simplify}([\left[ x \right]])\) stand in for

\[(a \| b) \leftarrow \text{GC}([\text{simplify}\; sk \| (c, r)])\]

and analogously for \([\left[ x \right]] \leftarrow \text{complicate}(sk, [\left[ x \right]])\). Finally, we let \((x \| x) \leftarrow \text{Reveal}(x)\) refer to a subprotocol which just interactively executes share reconstruction.

## 3 OBLIVIOUS MODEL CHECKING

Our goal is to construct a secure computation protocol for computing the predicate \(M \models \phi\) when \(D\) holds \(M\) and \(A\) holds \(\phi\). We now show that — should \(D\) and \(A\) be willing to treat \(m\) and \(n\) as public inputs — the various operator subroutines of check\(_{\text{CTL}}\) are either data-oblivious or may be rewritten to be so. This allows us direct adaption of these subroutines into (a part of) an MPC protocol using the preferred \(\Pi^{\text{otp}}_S\) secret sharing scheme.

As shown in Algorithm 2 and Algorithm 4, the checkEU and checkAU subroutines branch in a manner dependent on the truth values of both their subformulae and on \(\delta\). Branching on the former may leak information regarding \(\phi\) to \(D\), the latter information about \(M\) to \(A\). Moreover, both algorithms draw an ‘active’ state \(s\) from a set \(R\) in each outer loop iteration, and may add another state \(s'\) to \(R\) for later drawing only if \((s', s) \in \delta\). The resultant order in which states are accessed reveals information about \(\delta\). Our modified algorithms obscure these data access patterns through padding of branches and randomization.

We must also provide data-oblivious variants for the other operators, but this will require no effort. All of checkAND, checkNOT, and checkEX as given in Algorithm 1 are data-oblivious.\(^3\)

As noted in §2, there is a conceptual parallel between the checkEU and checkAU subroutines and breadth-first search. As such, our oblivious variants are derived from the oblivious BFS algorithm due to Blanton et. al. [10]. However, that work only considers a single source and does not support any label structure, so it does not directly fit our setting. For clarity, we will describe the simpler obcheckEU algorithm in full, and briefly discuss the straightforward addition required for obcheckAU at the end. We refer the reader to Algorithm 3 to follow the discussion as it formally presents the oblivious algorithm.

### 3.1 The Until Operators

The high-level description of obcheckEU is as follows. As within checkEU, we progress through a loop where each iteration we draw a yet unvisited state. In the original algorithm, we only ever draw states \(s_j\) for which \(s_j \models \psi \wedge \check{\phi}\). In the oblivious variant we draw all states, but give priority to those for which the subformulae holds. Only after these have been exhausted do we pad out the loop with the remainder. Then, for each drawn state we walk over all states \(s_j\), and update a status bitvector with whether \(s_j \models \psi \wedge (s_j, s_i) \in \delta\), in which \(s_i \models E \psi \wedge \phi\). In addition to this padding, where we differ most substantially from the non-oblivious algorithm is that the order of the states, and the mechanism by which we draw them, are both uniformly distributed. This prevents the operations

\(^2\)For \text{simplify} from a \(\Pi^{\text{ref}}_S\) share to a \(\Pi^{\text{otp}}_S\) share, the reconstructed output is treated as a bitvector and each bit reshared using \(\Pi^{\text{otp}}_S\) separately. The parties can then retain the necessary number of bits for the type, e.g., just the most significant bit if the object is an indicator. In the other direction \(\Pi^{\text{otp}}_S\) shares can be padded out with 0s to length \(z\) for \text{complicate} into a \(\Pi^{\text{ref}}_S\) share.

\(^3\)We take as assumptions that reading, writing, and incrementing/decrementing elements of \(\mathbb{N}\), array lookups, and evaluation of any specific arithmetic or propositional formula all take a constant number of instructions — assumptions valid under careful cryptographic engineering.
which are dependent on the chosen state index from leaking any information.

Our initial change for obcheckEU regards the inputs. We require that $r^\phi$, $l^\psi$, and the rows and columns of $M, \delta$ be permuted by $\pi^{-1}_1$, the inverse of a uniformly sampled $\pi_1 \leftarrow S_n$ where $S_n$ is the set of permutations of length $n$. Under this permutation, $r_{\pi_1}[i] \leftarrow \psi$, $l^\psi_{\pi_1}[i]$ indicates whether $s_{\pi_1(i)} \leftarrow \psi$, $M, \delta\{\pi_1[i]\}$ indicates $s_{\pi_1(i)}, \delta_{\pi_1(i)} \in \delta$. We also require an additional auxiliary input $[idxs_{\pi_1}]$, which is the permuted vector $\pi_1([n])$ for some $\pi_2 \leftarrow S_n$ sampled independently of $\pi_1$. Looking ahead, this vector will be used to select from a set of elements with uniformly distributed priority. The obcheckEU algorithm begins by initializing two bitvectors $\hat{K}$ and $\bar{\phi}$ using these inputs, and setting an empty bitvector $\hat{R}$.

For the inner loop iteration at Lines 5-8 of Algorithm 3, if $\hat{R}[c] \land \neg\hat{K}[c]$ then $s_{\pi_1(c)} \leftarrow E \psi U \phi$ and, as per §2, $s_{\pi_1(c)}$ has been ‘made ready’ but has not yet been ‘made active’. To match Line 6 of Algorithm 2, we want to pick just such a state to process in each loop iteration by setting $i = c$. Moreover, to avoid overhead we will want our access pattern to be able to depend on $i$ so that we only need to process its column of $M, \delta$ (at Lines 15-16). Though the application of $\pi_1$ makes each $i$ independent of the original state identifier, a deterministic rule for choosing $i$ might leak information. For example, if we were to take the maximal $c$ then this adversary would know that $\sum_{k=0}^{n} \bar{R}[k] \leq c$.

To make the choice of $i$, we effectively map each candidate $c$ to $[idxs_{\pi_1}[c]]$, and set to $i$ whichever has the maximal mapping. Using $m$ as a temporary variable storing the largest $[idxs_{\pi_1}[c]]$ yet seen for a candidate $c$, at the conclusion of the loop:

$$i = \arg\max_{c \in [n]} \{[idxs_{\pi_1}[c]] \land \neg\hat{K}[c] \land \neg\hat{R}[c].\}$$

As $\pi_2$ is uniformly random and independent of $\pi_1$, $i$ is uniformly distributed across $[n]$. In effect, $\pi_2$ makes $i$ a uniform choice of a ready state from $\hat{R}$, which ready states are themselves randomly distributed within $[n]$ by $\pi_1$. Altogether it is functionally equivalent to Line 6 in Algorithm 2, but leaks no information about $M$.

Algorithm 2: The checkEU algorithm.

Algorithm 3: The oblivious obcheckEU algorithm.

For the inner loop at Lines 9-12 we do similarly, but this time pick an $i'$ such that $s_{\pi_1(i')} \land \neg\hat{R}[c] = \hat{R}[i] \land \hat{K}[i']$. Moreover, to avoid overhead we will want our access pattern to be able to depend on $i$ so that we only need to process its column of $M, \delta$ (at Lines 15-16). Though the application of $\pi_1$ makes each $i$ independent of the original state identifier, a deterministic rule for choosing $i$ might leak information. For example, if we were to take the maximal $c$ then this adversary would know that $\sum_{k=0}^{n} \bar{R}[k] \leq c$.

To make the choice of $i$, we effectively map each candidate $c$ to $[idxs_{\pi_1}[c]]$, and set to $i$ whichever has the maximal mapping. Using $m$ as a temporary variable storing the largest $[idxs_{\pi_1}[c]]$ yet seen for a candidate $c$, at the conclusion of the loop:

$$i = \arg\max_{c \in [n]} \{[idxs_{\pi_1}[c]] \land \neg\hat{K}[c] \land \neg\hat{R}[c].\}$$

As $\pi_2$ is uniformly random and independent of $\pi_1$, $i$ is uniformly distributed across $[n]$. In effect, $\pi_2$ makes $i$ a uniform choice of a ready state from $\hat{R}$, which ready states are themselves randomly distributed within $[n]$ by $\pi_1$. Altogether it is functionally equivalent to Line 6 in Algorithm 2, but leaks no information about $M$.  

Algorithm 3: The oblivious obcheckEU algorithm.

For the inner loop at Lines 9-12 we do similarly, but this time pick an $i'$ such that $s_{\pi_1(i')} \land \neg\hat{R}[c] = \hat{R}[i] \land \hat{K}[i']$. Moreover, to avoid overhead we will want our access pattern to be able to depend on $i$ so that we only need to process its column of $M, \delta$ (at Lines 15-16). Though the application of $\pi_1$ makes each $i$ independent of the original state identifier, a deterministic rule for choosing $i$ might leak information. For example, if we were to take the maximal $c$ then this adversary would know that $\sum_{k=0}^{n} \bar{R}[k] \leq c$.

To make the choice of $i$, we effectively map each candidate $c$ to $[idxs_{\pi_1}[c]]$, and set to $i$ whichever has the maximal mapping. Using $m$ as a temporary variable storing the largest $[idxs_{\pi_1}[c]]$ yet seen for a candidate $c$, at the conclusion of the loop:

$$i = \arg\max_{c \in [n]} \{[idxs_{\pi_1}[c]] \land \neg\hat{K}[c] \land \neg\hat{R}[c].\}$$

As $\pi_2$ is uniformly random and independent of $\pi_1$, $i$ is uniformly distributed across $[n]$. In effect, $\pi_2$ makes $i$ a uniform choice of a ready state from $\hat{R}$, which ready states are themselves randomly distributed within $[n]$ by $\pi_1$. Altogether it is functionally equivalent to Line 6 in Algorithm 2, but leaks no information about $M$.  

Algorithm 3: The oblivious obcheckEU algorithm.
With these oblivious subroutines we are now able to construct our
we dictate that the auditor receives the output, though the protocol
ware code, a hardware design, or a protocol specification. We note
Protocol 1.

A potential direction for limiting this concrete overhead would be
For any Kripke structure $K = (S, I, \delta, L)$ and CTL

Although asymptotically Theorem 3.2 is equivalent to Theorem 2.1
our oblivious checking algorithm incurs substantial concrete costs.
We require a scalar multiple of $n^2$ steps always. In many model
checking problems, the semantics of the computational system

A potential direction for limiting this concrete overhead would be
to employ oblivious data structures instead of requiring extraneous
computation. This would however logarithmic overhead both
asymptotically and concretely [40, 54]. An ideal solution would
be to design an oblivious algorithm for local or symbolic model
checking without requiring extraneous computation dependent on
n. We leave further exploration to future work.

4 AN MPC PROTOCOL FOR CTL MODEL CHECKING

With these oblivious subroutines we are now able to construct our
privacy preserving checking protocol

\[ \phi \iff \text{PPMC}_{\text{CTL}}(M \parallel \phi) \]

such that $b$ correctly indicates whether $M \models \phi$. Given our setting
we dictate that the auditor receives the output, though the protocol
may be trivially extended to provide $b$ to $D$ by having $A$ send it
publicly should they wish to. The high level design is shown as
Protocol 1.

<table>
<thead>
<tr>
<th>Algorithm 4: The checkAU algorithm.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Theorem 3.2.</strong> For any Kripke structure $M = (S, I, \delta, L)$ and CTL</td>
</tr>
<tr>
<td><em>(1)</em> obcheck$\text{CTL}(M, \phi) = 1$ if and only if $M \models \phi$; and</td>
</tr>
<tr>
<td><em>(2)</em> obcheck$\text{CTL}(M, \phi)$ runs in time $O(mn^2)$ where $</td>
</tr>
<tr>
<td>and $</td>
</tr>
</tbody>
</table>

Algorithm 5: The oblivious obcheck\text{AU} algorithm.

There are $m + 2$ separate ‘segments’ of the protocol. In the initial
segment $D$ and $A$ each generate PRF keys, while $D$ locally constructs
$\Pi_S^{\phi^A}$ shares of the transition matrix $[|\delta|]$, degree vector $[|\delta deg|]$, and
vector representations of the labelings $[|\delta \pi|]$ for all $k \in [q]$, i.e.,
$\hat{\delta}_k[i] = (\delta_k[i], s_k)$. $D$ then sends the ciphertext components of
these shares to $A$ while keeping $sk_D$ private. Note that we abuse
notation here by using $[[x]]$ to indicate just the vectors of $(c, r)$
pairs. Additionally, $D$ discloses $n$ and $A$ discloses $m$.

<table>
<thead>
<tr>
<th>Algorithm 5: The oblivious obcheck\text{AU} algorithm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. function obcheck\text{AU}(n, M_{x1}, 1_{x1}, \pi_{x1}, id_{x1}, n_{x1})</td>
</tr>
<tr>
<td>2. $\hat{K} = r_{x1}$, $\hat{\psi} = r_{x1}$, $\hat{K} = 0_{x1}$</td>
</tr>
<tr>
<td>3. for all $r \in [n]$ do</td>
</tr>
<tr>
<td>4. $i, f \leftarrow 1, m$, $m' \leftarrow 0$</td>
</tr>
<tr>
<td>5. for all $e \in [n]$ do</td>
</tr>
<tr>
<td>6. $b_2 \leftarrow [c], \neg[\hat{c}], b_2 \leftarrow (ids_{x2}[c] &gt; m)$</td>
</tr>
<tr>
<td>7. $i = b_2 + 1 + (1 - b_2)$</td>
</tr>
<tr>
<td>8. $m = ids_{x2}[c] + 1 + m(1 - b_2)$</td>
</tr>
<tr>
<td>9. for all $e' \in [n]$ do</td>
</tr>
<tr>
<td>10. $b_2' \leftarrow [c'], \neg[\hat{c'}], b_2' \leftarrow (ids_{x2}[c'] &gt; m')$</td>
</tr>
<tr>
<td>11. $i' = b_2' + 1 + (1 - b_2')$</td>
</tr>
<tr>
<td>12. $m' = ids_{x2}[c'] + 1 + m(1 - b_2')$</td>
</tr>
<tr>
<td>13. $b_3 \leftarrow (i = \lambda)$</td>
</tr>
<tr>
<td>14. $i' \leftarrow i + 1$</td>
</tr>
<tr>
<td>15. for all $j \in [n]$ do</td>
</tr>
<tr>
<td>16. $b_4 \leftarrow R[i] \land M_{x1}[\delta_j[i]]$</td>
</tr>
<tr>
<td>17. $R[j] \leftarrow R[j] \lor (\hat{\psi}[i] \land M_{x1}[\delta_j[i]] = 0)$</td>
</tr>
<tr>
<td>18. $\hat{K}[i'] \leftarrow 1$</td>
</tr>
<tr>
<td>19. return $\hat{K}$</td>
</tr>
</tbody>
</table>

Each of the following $m$ segments will check a single operator
appearing in $\phi$. Before commencing the checking protocol, $A$ must
produce some linear ordering $\hat{\phi}$ of the parse tree of $\phi$. For any pair
of subformulæ $\phi_a, \phi_b \in \hat{\phi}$, if $\phi_a$ depends on $\phi_b$, then we require
$b < a$. A suitable ordering may be found by reversing a topological
sort. In the $j$th segment for $j \in [m]$, subprotocols for each of the
five possible operators are executed. We note that this allows a
degree of parallelism in our checking protocol, as each operator
may be checked concurrently. $A$ will keep the output for whichever
operator actually appears at $\hat{\phi}_j$.

The ‘and’, ‘not’, and EX subprotocols take a straightforward form.
$A$ selects the appropriate $[|\hat{\phi}^A|]$ and $[|\hat{\phi}^R|]$ shares vectors. If the true
operator is unary, they pick $[|\hat{\phi}^A|]$ arbitrarily for the subroutines
with binary input. It is this selection by $A$ where our use of $\Pi_S^{\phi^A}$ is
essential. Since $D$ has the same share $(sk_D)$ for all vectors, $A$ may
Protocol 1: PPMC($M$ $\parallel$ $\phi$)

choose in a manner dependent on $\phi$ as necessary: After this selection, the chosen $\Pi^{opr}_S$ shares are simplified to $\Pi^{ otp}_S$ shares. Then, the oblivious checking subroutine is executed using garbled circuits for all intermediary computations. For EX this includes $D$ providing some transition matrix information as a private input. Finally, the output $\Pi^{ otp}_S$ shares are raised back into $\Pi^{opr}_S$ shares.

Note that any $\Pi^{otp}_S$ shared constant may be set, such as for $o[i] = 0$, by having both parties set their share in a manner dictated by the protocol — e.g., at Line 17 both parties just set their share to 0. At Line 19, $\delta[i][j]$ is a private input of $D$ into the garbled circuit.

For EU and AU we want to use a similar approach of adapting our oblivious algorithms. However, we have a difficulty in that those algorithms require uniformly permuted inputs. We cannot simply have $A$ choose and execute a permutation over their shares, as they will then be able to follow the access patterns — in the most trivial case, $A$ may just choose $\pi_i$ to be the identity permutation.

For similar reasons, the choice of permutation cannot be entrusted simply to $D$. Rather, we need both $D$ and $A$ to permute the vectors so that each may be assured no information leaks to the other. As permutations compose, we can accomplish this by having $D$ and $A$ each choose and apply a random permutation, while using encryption to keep $D$ from learning the shares of $A$, and either party from learning the permutation of the other.

Joint Permutations. Our subprotocol for jointly computing permutations proceeds as follows. At commencement, $D$ holds $sk_D$ and some permutation $\pi_D$. $A$ holds $sk_A$, a vector of ciphertexts $[[x]]_{sk_D}$, and a pair of permutations $\pi_A$ and $\pi_{A'}$. Our protocol will output $\pi_A \cdot \pi_{D} \cdot \pi_A[[[x]]]$. Conventionally, either the same $\pi_A$ or $\pi_{A'}$ will be the identity permutation 1. This allows us to employ the same protocol to compute both $\pi^{-1} \pi^{-1} ([[x]])$ and its inverse $1' \pi ([[x]])$.

The protocol is relatively straightforward in formulation. $A$ first applies $\pi_{A'}$ to $[[x]]$, $sk_D$. The parties then execute a sequence of garbled circuit executions to transfer these ciphertexts from $D$ to $A$. The transfer subroutine uses the key of $D$ to encrypt and then the key of $A$ to re-encrypt. At the conclusion, $D$ possesses $\pi_{A}([[x]]), sk_A$, which we note by $[[\chi_D^x]], sk_A$. $D$ then applies $\pi_A$ to derive $[[\chi_D^x \pi_D]], sk_A$, and the parties then repeat the transfer in the opposite direction. Finally, $A$ applies $\pi_{A'}$ to arrive at $[[\chi_D^x \pi_{D} \pi_{A'}]], sk_A$, as required.

Intuitively, the permutation protocol is privacy preserving as the shares are pseudorandom due to $\Pi^{opr}_S$ being an encryption
scheme. Given the inability of either D or A to distinguish the encryption of one plaintext from another, they are unable to learn anything about the permutation that has been placed on those plaintexts. This privacy converts to the nested shares as well. With these permutation subprotocols, the subprotocols for EU and AU follow from our discussion in §3.

\begin{align*}
1. \text{protocol EU}(sk_D) \| sk_A([\delta]), ([r^0]), ([\nu^0])); \\
2. D, A \oplus \pi_D, \pi_{2D} \rightarrow S_n, \pi_A, \pi_{2A} \rightarrow S_p; \\
3. \pi_{D2x} \rightarrow \text{AltPerm}(sk_D, \pi_{1D}^{\perp} \| sk_A, \pi_{1A}^{\perp} ([r^0]x)); \\
4. \pi_{D1x} \rightarrow \text{AltPerm}(sk_D, \pi_{1D}^{\perp} \| sk_A, \pi_{1A}^{\perp} ([\nu^0])); \\
5. [\nu^0] \rightarrow \text{AltPerm}(sk_D, \pi_{1D}^{\perp} \| sk_A, \pi_{1A}^{\perp} ([\nu^0])); \\
6. [\pi_{1D}] \rightarrow \text{AltPerm}(sk_D, \pi_{1D}^{\perp} \| sk_A, \pi_{1A}^{\perp} ([\nu^0])); \\
7. \ldots \text{same as Algorithm 3 with } \Pi^0_{\text{net}} \text{ shares and revealed } r^n. \\
8. ([R]) \leftarrow \text{Complicate}(sk_D, [R]). \\
9. ([O]) \leftarrow \text{Permute}(sk_D, \pi_{1D}^{\perp} \| sk_A, \pi_{1A}^{\perp} ([R])); \\
10. \text{protocol AUS}[k] \| [\delta], ([r^0]), ([\nu^0]), ([\nu^0]). \\
11. \ldots \text{same as Lines 2-6.} \\
12. [\deg] \leftarrow \text{AltPerm}(sk_D, \pi_{1D}^{\perp} \| sk_A, \pi_{1A}^{\perp} ([\deg])); \\
13. \ldots \text{same as Algorithm 5 with } \Pi^0_{\text{net}} \text{ shares and revealed } r^n. \\
14. \ldots \text{same as Lines 6-9.}
\end{align*}

The final segment of the protocol is to determine whether all initial states satisfy the specification. This may be done with a straightforward adaption of the same functionality from obcheckCTL.

\begin{align*}
1. \text{function SAT} (sk_D, M \| ([\nu^0])); \\
2. [\sigma] \leftarrow 1; \\
3. \text{for all } i \in [n] \text{ do} \\
4. \left[ [\sigma[i]] \right] \leftarrow \text{Sample}(sk_D, [\nu^0][i])); \\
5. \left[ [\sigma] \right] \leftarrow \left[ [\nu^0][i)] \right) \wedge M[S][i], \text{ init}. \\
6. (\| [\delta] \leftarrow \text{Reveal}([\sigma])
\end{align*}

At Line 5, M[S][i].init is a private input of D. The output of Line 6 completes the model checking protocol.

4.1 Correctness, Complexity, and Security

Our result with respect to correctness and complexity is an analogue of Theorem 3.2.

**Theorem 4.1.** For any Kripke structure M = (S, I, δ, L) and CTL formula φ, let \(\cdot \| \cdot \leftrightarrow \text{PPMC}(sk_D, M \| sk_A, \phi)\). Then,

1. \[ b \equiv 1 \text{ if and only if } M \models \phi; \text{ and} \]
2. \(\text{PPMC}(M \| \phi) \text{ runs in local and communication complexities } O(mn^2) \text{ where } |M| = O(n^2) \text{ and } |\phi| = m. \]

**Proof.** Each of the component algorithms of \(\Pi^{opt}_{\text{S}}, \Pi^{prf}_{\text{S}}\), and the GC subprotocol run in constant-time with respect to n and m and so require no asymptotic overhead. The first segment of our protocol costs local and communication complexities \(O(n^2)\). For the remaining \(m + 1\) segments our protocol faithfully adapts obcheckCTL. So by Theorem 3.2, the protocol is correct and runs in local and communication complexities \(O(mn^2)\). \(\square\)

Our second result establishes the privacy preserving nature of the protocol.

**Theorem 4.2.** For any Kripke structure M = (S, I, δ, L) and CTL formula φ, let bit b indicate whether M \models \phi. Then,

1. \(\text{there exists a PPT simulator } Sim_D(1^\lambda, \phi, b) \text{ such that } \)
\(\text{view}_A(PPMC, \phi, D(1^\lambda, M)) = \text{view}_A(PPMC, \phi, \text{Sim}_D(1^\lambda, \phi, n, b)); \text{ and} \)
2. \(\text{there exists a PPT simulator } Sim_M(1^\lambda, M, \cdot) \text{ such that } \)
\(\text{view}_D(PPMC, M, \text{Sim}_M(1^\lambda, M, m, \cdot)). \)

Note that although in our protocol n and m are private inputs which the parties agree to leak, here we treat them as public inputs available to the simulators. This may be formalized through leakage oracles, but we use this informal approach for simplicity.

We require a few preliminaries towards this proof. First, that for the protocol GC(c; \cdot \| \cdot) for arbitrary c there exist simulators for both participants [43]. Since we are agnostic to the roles in the GC protocol, we just refer to the appropriate simulator as GCSim(1^\lambda, f, y). The second result is that given a PRF, an encryption scheme \(\Pi_{\text{enc}} = (\text{Gen}, \text{Enc}, \text{Dec})\) for which \(\text{Gen} = \text{Gen}_{prf}\), \(\text{Enc} = \text{Share}_{prf}\), and \(\text{Dec} = \text{Reconstruct}_{prf}\) provides indistinguishability for multiple encryptions under chosen ciphertext attack, or IND-CPA security [37]. Finally, we need the experiment used to formalize this security notion. The specific experiment we use is often referred to as the left-right oracle formulation.

**Definition 4.3.** Let \(\Pi = (\text{Gen}, \text{Enc}, \text{Dec})\) be an encryption scheme and \(\lambda \in \mathbb{N}\) a security parameter. We define the experiment IND-CPA, \(\Pi\) between adversary A and a challenger C by

1. C runs \(sk \leftarrow \text{Gen}(1^\lambda)\) and samples b \(\triangleleft \{0, 1\}\).

2. C sends a pair \((m_0, m_1)\) to C through the oracle, and receives Enc(sk, m_b) in response.

3. A repeats (2) up to n times, for \(n = \text{poly}(\lambda)\).

4. A outputs \(b' \in \{0, 1\}\). The output of IND-CPA, \(\Pi\) is then the truth of the predicate \(b \equiv b'\).

Then \(\Pi\) provides indistinguishability for multiple encryptions under chosen ciphertext attack if for all PPT A,

\[\text{Pr}[\text{IND-CPA}, \Pi(1^\lambda) = 1] = \frac{1}{2} + \text{negl}(\lambda).\]

We now have the necessary machinery to prove Theorem 4.2. For brevity we mostly sketch part (1) of the argument. The argument for (2) follows along similar lines.

**Proof Sketch.** Let \(\tau = O(mn^2)\) be the number of circuits computed over the course of the protocol. We let \(f_k\) be the function, \(x_{2Dh}, x_{2Ah} \in \{0, 1\}^n\) the inputs, and \(y_{2Dh}, y_{2Ah} \in \{0, 1\}^n\) the outputs of the hth circuit for \(h \in [\tau]\).

\(\text{Sim}_D(1^\lambda, \phi, n, b)\) first constructs a random Kripke structure \(M'\). To accomplish this, the simulator uniformly samples bits \(b_1, \ldots, b_{n^2 + (q + 1)n} \triangleleft \{0, 1\}\) and uses them to populate \(M'[S][i].\text{init}\) for \(i \in [n]\), \(M'[S][i].\text{deg}\) as appropriate. The simulator then executes \(\text{INIT}\) as specified over \(M'\).

In each of the following \(m + 1\) segments of the protocol \(\text{Sim}_D\) executes all local computations as prescribed. For the \(h\)th garbled
circuit the simulator locally computes \( f(x_{Dh}, x_{Ah}) = (y_{Dh}, y_{Ah}) \). It then invokes GCSim(\( 1^\lambda \), \( f, y_{Ah} \)) and so embeds the correct output to be received by \( A \), which will be either a \( \Pi^{prf}_S \) or a \( \Pi^{otp}_S \) share. When the time comes to reveal an \( i^* \) value, as \( D \) knows both the value and the input share of \( A \) they may design their share to produce the correct reveal.

The only complications are for \( \text{Perm} \) and \( \text{AltPerm} \), as \( \text{Sim}_D \) cannot inspect the encrypted \([\hat{x}_{\pi_D}],[x_{\pi_D},\pi_D]\) vectors and recover \( \pi_A \) as it does not know \( s_{k_A} \). For \( \text{AltPerm} \), given \( M' \) and \( \phi \), \( D \) knows the underlying plaintexts \( x_1, \ldots, x_n \). So it may uniformly sample a permutation \( \pi_D \), and for \( i \in [n] \) embed \([\hat{x}_{\pi_D}[i]]\) as the output of the \( i \)th execution of \( \text{Simplify} \). As there is a unique \( \pi_D \) such that \( \pi_A \pi_D = \pi_D' \) these embeddings correctly simulate the protocol.

For \( \text{Perm} \), \( \text{Sim}_D \) again knows the underlying plaintexts. However, without knowledge of \( \pi_A \) the simulator cannot embed the outputs of \( \text{GCSim} \) into its \( \pi_D' \) so that \([\hat{x}_{\pi_A\pi_D}[x_{\pi_D},\pi_D]]\) will return to the original order. So, instead it embeds them in arbitrary order. If \([\hat{x}_{\pi_A\pi_D}[x_{\pi_D},\pi_D]]\) will be the input to a later protocol segment, which \( \text{Sim}_D \) knows as it has \( \phi \), then the simulator just embeds the outputs to any \( \text{Simplify} \) invocation as though they were correctly ordered.

Finally, as \( \text{Sim}_D \) is able to locally compute the share of \( [sat] \) held by \( A \) it may correctly embed \( b\) into the final reveal by negating its share if necessary. As \( \text{Sim}_A \) makes each computation in the protocol at most twice, it runs in \( \text{PPT} O(mn^3) \) as required.

We next construct a sequence of hybrid distributions, starting from \( \mathcal{H}_0 = \text{view}_A(\text{PPMC}, \phi, \text{Sim}_D(1^\lambda, \phi, n, b)) \) and ending at \( \mathcal{H}_{\tau+1} = \text{view}_A(\text{PPMC}, \phi, D(1^\lambda, M)) \). The first hybrid \( \mathcal{H}_0 \) captures an interaction with identical functionality to \( \text{Sim}_D \), except using \( M \) instead of sampling \( M' \). We show that \( \mathcal{H}_0 = \mathcal{H}_1 \) by reduction to the assumption that we have a secure PRF, through the functional equivalence between \( \Pi^{prf}_S \) and a IND-CPA secure encryption scheme. Let \( \mathcal{A} \) be an adversary with non-negligible advantage in distinguishing \( \mathcal{H}_0 \) and \( \mathcal{H}_1 \) for some \( M' \). We show this implies an adversary \( \mathcal{A}' \) with non-negligible advantage in the IND-CPA security experiment, violating our assumption.

We let \( \mathcal{A}' \) be parameterized by \( M' \) (and \( n \)), and it is given \( 1^\lambda \) on start by \( C \). It begins by sampling an \( M' \) as per \( \mathcal{H}_0 \). It then executes the remaining functionality of both \( \mathcal{H}_0 \) and \( \mathcal{H}_1 \) (which are consistent with each other). But, for all encryptions that it would usually carry out locally with \( sk_D \) it instead uses its oracle access from \( C \), sending as \( m_0 \) the plaintext for \( \mathcal{H}_0 \) (from \( M' \)) and as \( m_1 \) the plaintext for \( \mathcal{H}_1 \) (from \( M' \)). It then embeds these encryptions into the garbled circuit simulator outputs as appropriate.

Let \( b' \) be the coin flipped by \( C \). If \( b' = 0 \), then \( \mathcal{A}' \) perfectly instantiates \( \mathcal{H}_0 \) as it

i. executes a fixed order of garbled circuit simulators;
ii. uniformly generates all \( \Pi^{otp}_S \) shares as required;
iii. generates all \( \Pi^{prf}_S \) shares appropriately under the challenge \( sk \) using the oracle access from \( C \); and
iv. reveals each sequence of \( i' \) values in a uniformly distributed order which is consistent with any (possible but unknown to it) choice of \( \pi_{IA} \).

If \( b' = 1 \), \( \mathcal{A}' \) perfectly instantiates \( \mathcal{H}_1 \) by an identical argument. So, upon receipt of the distinguishing bit \( b'' \) from \( A \), \( \mathcal{A}' \) sets its own output bit \( b'' = b'' \). It therefore retains the non-negligible advantage of \( A \), and so has a non-negligible advantage in the IND-CPA experiment. We conclude that \( \mathcal{A}' \) may not exist as it derives a contradiction, and so neither may \( A \).

Returning to our sequence of hybrid distributions, for all \( h \in [\tau] \), hybrid \( \mathcal{H}_{h+1} \) converts the \( h \)th intermediary computation from using the garbled circuit simulator to using the real garbled circuit functionality. Then, \( \mathcal{H}_{h+1} = \mathcal{H}_{h+2} \) follows by the compositionality of secure computation protocols, as proven in detail in [12]. As no distinguisher exists for any two adjacent hybrids in our sequence, we may conclude that

\[
\text{view}_A(\text{PPMC}, \phi, \text{Sim}_D(1^\lambda, \phi, n, b)) = \mathcal{H}_0 \approx \mathcal{H}_{\tau+1} = \text{view}_A(\text{PPMC}, \phi, D(1^\lambda, M))
\]

by the triangle inequality. \( \square \)

5 IMPLEMENTATION

We implemented our protocol using the semihonest 2PC functionality within the EMP-Toolkit [53]. For AES, we used the key-expanded Bristol Format circuit,\(^4\) which requires 5440 AND gates per execution — none of the approx. 22500 combined XOR and INV gates require encrypted gates, due to Free-XOR techniques [41, 58].

In the following table we report both the time elapsed and number of in-circuit AES executions to check random models of size \( n \) and \( m \) respectively, with \( q = 4 \). Given that our construction is completely agnostic to the structure of either \( M \) or \( \phi \), these experiments are demonstrative for natural problems of similar dimension. Our evaluations were made on a commodity laptop with an Intel i5-3520M CPU running at 2.60GHz and 8GB of RAM, and no parallelism was employed. Since the cost is dominated by EU and AU, we predict a parallel implementation will cut running times roughly in half.

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
n & m & 1 & 2 & 3 & 4 & 7 \\
\hline
4 & 4.802s & 5.197s & 5.533s & 5.026s & 3.010s \\
16 & 15.706s & 26.629s & 29.407s & 107.746s & 54.462s \\
64 & 15.706s & 26.629s & 29.407s & 107.746s & 54.462s \\
\hline
8 & 6.968 AES & 13.846 AES & 27.606 AES & 5026 AES & 48.246 AES \\
32 & 35.462s & 107.760s & 210.392s & 374.140s & 19526 AES \\
64 & 2160 AES & 4504 AES & 8592 AES & 15926 AES & 4504 AES \\
\hline
\end{array}
\]

We observe a consistent cost of \( \approx 20-30 \)ms per AES execution, rising as it incorporates (amortized) both local computations and circuits over \( \Pi^{otp}_S \). Latency is minimal, due to both processes running on the same physical hardware. As expected the number of AES executions grows linearly in \( m \). For each increment of \( n \) we observe the number of executions growing quadratically due to the domination of the \( n^2 \) term for EU and AU. All of these observations are consistent with the relatively static nature of our algorithm — the number of circuits executed is a relatively simple and deterministic function of \( n \) and \( m \).

\(^4\)https://homes.esat.kuleuven.be/~nsmart/MPC/old-circuits.html
provide a more efficient setting for some of the other intermediary computations we require. However, the growth rates borne out by our experimentation lead us to conclude that although these primitives may noticeably reduce concrete cost, practical PPMC on non-trivial problems will likely require further algorithmic developments. The orders of magnitude of $n$ for which our current protocol projects as viable may suffice for some small protocol and hardware designs, but not likely any software verification task of meaningful complexity. We hope to develop significantly more efficient constructions, especially by adopting the local or symbolic techniques necessary for combating the state explosion phenomenon.

6 RELATED WORK

Recent years have seen a proliferation of work applying MPC to real-world problems, with [4] an excellent overview. This work has been enabled by developments in the efficiency of primitives [6, 34, 39, 41, 58], by the creation of usable compilers and software libraries — see [31] for a comprehensive SoK — and by increased research interest in the definition of tailored protocols. Our work fits into this narrative that MPC is practical and valuable to privacy-conscious settings [49]. At the specific intersection of privacy preserving computation and program analysis and verification, recent work has employed zero-knowledge proofs to prove the presence of software bugs [9, 32].

In addition to generic MPC tools and techniques, our protocol is particularly dependent on both the in-circuit PRF and oblivious graph algorithms. Constructing PRFs specifically for multiparty computation is an active area of research, providing promising schemes which may dramatically reduce the concrete overhead of our protocol [1–3, 30]. Data-oblivious graph algorithms have also received attention both generically and within a variety of problem domains [10, 11, 22, 27, 28, 56]. Also relevant is work on generic oblivious data structures [40, 54]. Although these usually come with asymptotic overhead, they allow for straightforward adoption of many graph algorithms into 2PC.

Finally, we note that while our work applies cryptography to formal methods, the opposite direction — applying formal methods to cryptography — has also seen substantial recent development. Computer-aided cryptography attempts to provide formal proofs of cryptographic security — see [7] for a comprehensive SoK. Work from the programming languages community has developed languages and compilers tailored to oblivious computation and MPC [18, 45, 46, 51]. Of particular note is [33], where a model checker is used in the compilation of C programs for MPC execution.

7 CONCLUSION

We have presented an oblivious algorithm for global explicit state model checking of CTL formulae, and shown how it may be extended with the use of cryptographic primitives into an MPC protocol secure against semihonest adversaries. The result requires no asymptotic local overhead and communication complexity consistent with the local complexity, while the concrete cost and feasibility remain a focus of future effort. Although our work is so limited, we have shown the potential application of privacy preserving techniques to modern techniques for program analysis and verification.

7.1 Future Work

We consider there to be substantial opportunity for further work on privacy preserving formal methods — and privacy preserving model checking in particular — in the following directions:

- Our proof of security is in the semihonest model only. Though generic techniques allow us to execute our garbled circuits with security against malicious adversaries [42], verifiable secret sharing (VSS) is also required for composition [13]. Elevating PRF-based secret sharing scheme to VSS may be necessary if the complexity and structure of formal methods frequently requires partially data-dependent processing. The use of a PRF for a message authentication code (MAC) to accompany the encryption scheme may be a starting point, but further investigation is needed.

- As noted in our introduction, a substantial limitation of our construction is the inability to guarantee that $M$ accurately and adequately represents the program execution. There has been active work — perhaps most prominently [9] — in providing for zero-knowledge proofs (ZKP) of program executions. A potential direction would be to integrate these schemes with our privacy preserving construction, so that $M$ gains assurance the model they checked does represent a program with certain functionality, while otherwise maintaining the privacy of it and the specification. Such approaches would need to be mediated through techniques for input validity for MPC [36, 38].

- Our protocol only applies for specifications written in CTL. Whether similar protocols may be developed for LTL, CTL*, and (temporal) epistemic logics is an open question. Additionally, our scheme suffers from being global and for requiring the worst-case always. Protocols adapting local explicit state or symbolic checking algorithms would dramatically increase the practicality of PPMC.

- Finally, development of a privacy preserving model checking tool for use with real software would confirm the utility of our construction.

ACKNOWLEDGMENTS

The authors thank Mariana Raykova, Mark Santolucito, and the anonymous reviewers for their helpful comments, and Michael Chen for his assistance in the implementation. This work was supported in part by the Office of Naval Research through Grant No. N00014-17-1-2787 and a National Defense Science and Engineering Graduate (NDSEG) Fellowship, by the National Science Foundation through Grants No. CNS-1562888, CNS-1565208, and CCF-1553168, and by DARPA through Grant No. W911NF-16-1-0389.

REFERENCES

A DATA-OBLIVIOUS MODEL CHECKING

Our first theorem establishing the correctness of the obcheck−CTL algorithm is that the checking subroutines it employs are data-oblivious, restated here.

**Theorem 3.1.** The checkAND, checkNOT, checkEX, obcheckEU, and obcheckAU algorithms are data-oblivious.

For our proofs, we give our most relevant definitions with more formality than in §2.

**Definition A.1 (Access Pattern).** The access pattern of a RAM = (CPU, MEM) on input (s, y) is a sequence

\[ \mathcal{AP}(s, y) = (a_1, \ldots, a_i, \ldots) \]

such that for every \( i \), the \( i \)-th message sent by CPU(s) when interacting with MEM(y) is of the form \((\cdot, a_i, \cdot)\).

**Definition A.2 (Data-Oblivious Program).** A program \( P \) is data-oblivious with respect to an input class \( X \), if for any two strings \( x_1, x_2 \in X \), such that \( \{\mathcal{AP}(P, x_1)\} \) and \( \{\mathcal{AP}(P, x_2)\} \) are identically distributed, then so are \( \mathcal{AP}(P(x_1)) \) and \( \mathcal{AP}(P(x_2)) \).

Throughout the following argument let \( M \subseteq \{0, 1\}^* \) be the set of all binary strings interpretable as a Kripke structure, and \( \text{BitVec}_n \subseteq \{0, 1\}^n \) be the set of all binary strings interpretable as a bitvector of length \( n \). We also proceed (as previously noted) with the assumptions that reading, writing, and incrementing/decrementing elements of \( \mathbb{N} \), array lookups, and evaluation of any specific arithmetic or propositional formula all take a constant number of instructions — assumptions valid under careful cryptographic engineering.

**Lemma A.3 (Obliviousness of checkAND, checkNOT, and checkEX).**

(1) If \( P = \text{checkAND} \) and for all \( x \in X \) we may write \( x = (M, r^\phi) \) for \( M \in M \) and \( r^\phi \in \text{BitVec}_n \) for some \( n \in \mathbb{N} \) with \( M.n = n \), then \( P \) is data-oblivious with respect to \( X \); and

(2) if \( P \in \{\text{checkNOT}, \text{checkEX}\} \) and for all \( x \in X \) we may write

\[ x = (M, r^\phi) \text{ for } M \in M \text{ and } r^\phi \in \text{BitVec}_n \text{ for some } n \in \mathbb{N} \]

with \( M.n = n \), then \( P \) is data-oblivious with respect to \( X \).

**Proof.** Let \( P \in \{\text{checkAND}, \text{checkNOT}, \text{checkEX}\} \), and define four constants \( c_1, c_2, c_3, c_4 \in \mathbb{N} \). If \( P = \text{checkEX} \) let \( c_1 \) denote the number of instructions issued by CPU to MEM in the inner loop of \( P \). For any \( P = \text{checkNOT} \) let \( c_2 \) denote the number of instructions issued in the outer loop, \( c_3 \) the number of instructions issued managing the loop, and \( c_4 \) the number of instructions issued outside the loop context. Then for \( P \in \{\text{checkAND}, \text{checkNOT}\} \) and all \( x \in X \) (for corresponding \( X \)) we may write

\[ |\mathcal{AP}(P(x), y) = n_1 \cdot c_1 + n_2 \cdot c_2 + n_3 \cdot c_3 + n_4 \cdot c_4 \]

and when \( P = \text{checkEX} \)

\[ |\mathcal{AP}(P(x)) = n_1 \cdot c_1 + n_2 \cdot c_2 + n_3 \cdot c_3 + n_4 \cdot c_4 \]

In each case \( |\mathcal{AP}(P(x)) \) is a deterministic injective function of \( n \). Therefore for \( x_1, x_2 \in X \), \( |\mathcal{AP}(P(x_1)) \) and \( |\mathcal{AP}(P(x_2)) \) are identically distributed if and only if \( x_1 \) and \( x_2 \) represent (in part) models \( M_1 \) and \( M_2 \) respectively such that \( M_1.n = M_2.n \). Moreover, for a fixed \( n \) by unrolling the loop(s) in \( P \) we may easily see that the program executes a fixed access pattern. And so, for all \( n \in \mathbb{N} \) there exists a sequence \( \mathcal{AP}_n \) such that for any \( x \in X \) representing (in part) a model \( M \) for which \( M.n = n \),

\[ \mathbb{P} [\mathcal{AP}(P(x)) = \mathcal{AP}_n^*] = 1. \]

As such, if \( x_1, x_2 \in X \) are such that \( |\mathcal{AP}(P(x_1)) \) and \( |\mathcal{AP}(P(x_2)) \) are identically distributed, then

\[ \mathbb{P} [\mathcal{AP}(P(x_1)) = \mathcal{AP}_n^*] = \mathbb{P} [\mathcal{AP}(P(x_2)) = \mathcal{AP}_n^*] = 1 \]

for some \( n \in \mathbb{N} \).

Next we argue that obcheckEU and obcheckAU are both data-oblivious. We first prove a helpful lemma.

**Lemma A.4.** Let \( P \in \{\text{obcheckEU, obcheckAU}\} \), \( t \in \{1, \ldots, n\} \) be an arbitrary iteration, and \( a', a'' \in \{0, 1\} \) be arbitrary indices. Let \( i^{(t)} = (i^{(t)}_1, \ldots, i^{(t)}_{n-1}) \) be the sequence of prior \( i^{(t)} \) values. Then,

(1) if \( a \in i^{(t)} \) then \( \mathbb{P}[r^{(t)} = a] = 0 \); and

(2) if \( a, a' \notin i^{(t)} \) then \( \mathbb{P}[r^{(t)} = a] = \mathbb{P}[r^{(t)} = a'] \).

**Proof.** For (1), since there exists \( t' < t \) such that \( i^{(t')} = a \), during iteration \( t' \) the algorithm set \( \hat{K}[a] = 1 \). For the \( t \)-th iteration of the first (resp. second) inner loop of Algorithm 3 and Algorithm 5, \( b_1 = 0 \) (resp. \( b_1' = 0 \)), and so \( i \neq a \) (resp. \( i' \neq a \)). Since \( i^t \neq i \neq a \), with certainty, \( \mathbb{P}[r^{(t)} = a] = 0 \).

For (2), let \( u \in \{0, 1\} \) be such that at the beginning of the \( t \)-th iteration of the outer loop, \( \sum_{k=0}^{n} \hat{K}[u] - \hat{K}[u] = u \). The index \( a \in \{a, a'\} \) will be chosen as \( i^{(t)} \) if

(i) \( u > 0 \), \( \hat{K}[a] = 1 \), and for all \( a'' \neq a \) such that \( \hat{K}[a''] = 1 \) and \( \hat{K}[a''] = 0 \), \( \mathbb{P}[r^{(t)} = a] > \mathbb{P}[r^{(t)} = a'] \); or

(ii) \( u \leq 0 \) and for all \( a'' \neq a \) such that \( \hat{K}[a''] = 0 \), \( \mathbb{P}[r^{(t)} = a] > \mathbb{P}[r^{(t)} = a'] \).

For (2,ii), the uniform choice of \( \pi_2 \) gives each unvisited index equal probability of having the greatest \( idx_{\pi_2} \). Moreover, since by (1) the probability that \( i^{(t)} \) will be a previously visited index is
We will not provide a detailed proof of this theorem, but rather
argue these subroutines are functionally equivalent to their non-
oblivious variants and retain complexity $O(n^2)$. The complexity
follows immediately for both subroutines due to their nested loop
structure with both inner and outer iterating over $[M,n]$. 

As for functional equivalence, the core of the argument is that
(i) we process all states 'made ready' before any others; that (ii)
we process those states in an order consistent with the use of $R$
in the original algorithms; that (iii) while processing 'made ready'
states under identical selection we, as compared to the original
algorithms, update $\hat{R}$ to be identical to $R$, $\hat{K}$ to be an exact
representation of inclusion into $R$, and $\hat{K}$ to be an exact
representation of inclusion into $K$. That all processing done on
states not 'made ready' in the oblivious algorithm does not modify
$\hat{R}$ then establishes the equivalency. Once we have shown that
obcheck\textsubscript{CTL} runs in time $O(mn^2)$ and is functionally
equivalent to check\textsubscript{CTL}, Theorem 2.1 completes a proof.

---

**Lemma A.5 (Obliviousness of obcheck\textsubscript{EU} and obcheck\textsubscript{AU}).** If
$P \in \{\text{obcheck\textsubscript{EU}}, \text{obcheck\textsubscript{AU}}\}$ and for all $x \in X$ we may write
$x = (M_l, l^2, r^\delta)$ for $M \in M$ and $l^2, r^\delta \in \text{BitVec}_n$ for some
$n \in \mathbb{N}$ with $M,n = n$ then $P$ is data-oblivious with respect to $X$.

**Proof.** By an identical argument to Lemma A.3 the number of
instructions issued in an execution of $P$ is a deterministic injective
function of $n$. So, for $x_1, x_2 \in X$, $|\mathcal{AP}(\langle P, x_1 \rangle)|$ and $|\mathcal{AP}(\langle P, x_2 \rangle)|$
are identically distributed only if $x_1$ and $x_2$ represent models $M_1$
and $M_2$ respectively such that $M_1.n = M_2.n$.

Let $x_1, x_2 \in X$ be an arbitrary pair of such inputs. The argument
reduces to showing that for such $x_1$ and $x_2$ their access patterns
are identically distributed. Further, let $I^t(n)$ be as in Lemma
A.4. By loop unrolling, it follows that for a given $I^t(n)$ the access pattern
of $P$ is fixed. So the argument may be reduced further to showing
that $I^t(n)$ is identically distributed for $x_1$ and $x_2$. But, by Lemma
A.4 the choice of $I^t(n)$ for all $t \in [1..n]$ is always uniformly distributed
over all unvisited indices regardless of model structure and prior
choices. It follows that whole sequences are also uniformly — and
so identically — distributed for $x_1$ and $x_2$. 

The proof of Theorem 3.1 now follows immediately from the
conclusions of these lemmas.

**Proof.** Apply Lemma A.3 and Lemma A.5.

Our second theorem establishing the functional correctness and
efficiency of the oblivious checking algorithm is also restated here.

**Theorem 3.2.** For any Kripke structure $\mathcal{M} = (S, I, \delta, L)$ and CTL
formula $\phi$

1. $\text{obcheck}_{\text{CTL}}(\mathcal{M}, \phi) = 1$ if and only if $\mathcal{M} \models \phi$; and
2. $\text{obcheck}_{\text{CTL}}(\mathcal{M}, \phi)$ runs in time $O(mn^2)$ where $|\mathcal{M}| = O(n^3)$
and $|\phi| = m$.

We will not provide a detailed proof of this theorem, but rather
sketch the proof by arguing (somewhat informally) that certain
invariants between the original check\textsubscript{EU} and check\textsubscript{AU} subroutines
and their oblivious variants hold. This implies the functional
equivalence of the obcheck\textsubscript{CTL} algorithm to check\textsubscript{CTL} at which point
Theorem 2.1 and the additive $O(n)$ cost of permutations completes
the argument.

**Proof Sketch.** As the differences between obcheck\textsubscript{CTL} and
check\textsubscript{CTL} lie exclusively within obcheck\textsubscript{EU} and obcheck\textsubscript{AU}, we