Privacy Preserving CTL Model Checking through Oblivious Graph Algorithms

Samuel Judson
Yale University
samuel.judson@yale.edu

Ning Luo
Yale University
ning.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 $\mathcal{M}$ of a computational system meets a specification of behavior ϕ. We apply the cryptographic theory of *secure multiparty computation* (MPC) to model checking. With our construction, adversarial parties D and A holding $\mathcal{M}$ and ϕ respectively may check satisfaction — notationally, whether $\mathcal{M} \models \phi$ — 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:
Samuel Judson, Ning Luo, Timos Antonopoulos, and Ruzica Piskac. 2020. Privacy Preserving CTL Model Checking through Oblivious Graph Algorithms. In *19th Workshop on Privacy in the Electronic Society (WPES'20), November 9, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3411497.3420212

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 directly into their operating system [35].

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
WPES'20, November 9, 2020, Virtual Event, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8086-7/20/11…$15.00
https://doi.org/10.1145/3411497.3420212

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 $\mathcal{M}$ 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 $\mathcal{M} \models \phi$, 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 $O(|\phi| \cdot |\mathcal{M}|)$ 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 $\mathcal{M}$ 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 $\mathcal{M}$?' to the incontrovertible 'did the developer misrepresent $\mathcal{M}$?', 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 ne**x**t), **F** (**f**inally, i.e. eventually), **G** (**g**lobally, i.e. always), and **U** (**u**ntil), while CTL and CTL* add quantifiers **E** (**e**xists a branch) and **A** (for **a**ll branches) over the tree. CTL allows expression of statements such as **AG** ($userdata \rightarrow$ **AG** $\neg network$) 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 $\mathcal{M} = (S, I, \delta, L)$, where S is a set of states with $n = |S|$, $I \subseteq S$ a set of initial states, $\delta \subseteq S \times S$ a transition relation — with $(s_i, s_j) \in \delta$ for $i, j \in [n]$ denoting that s_j is a successor of s_i — and $L : S \rightarrow 2^q$ a labeling function mapping states to subsets of the q available labels. We note that $O(n^2) = |\mathcal{M}|$.

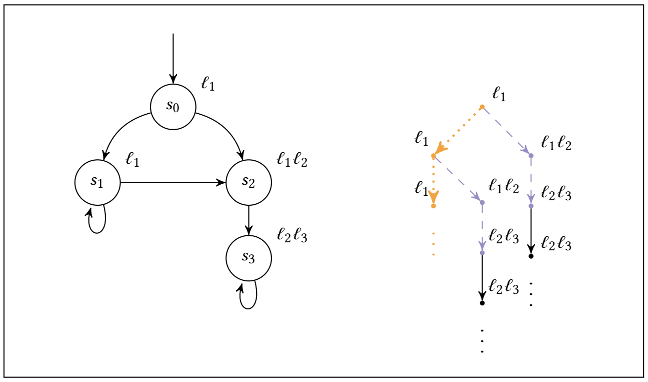

Figure 1: A Kripke structure (left) modeling the program given in Example 2.1, and its corresponding computation tree (right). The vertices and edges in the computation tree show a failed checking for A ℓ_1 U ℓ_3, which holds at s_2 and s_3 (dashed blue) but not s_0 or s_1 (dotted orange).

Example 2.1. Consider a toy authentication program implementing session management. When an unknown user arrives they are prompted for credentials, and reprompted if those credentials are invalid. Once valid credentials are provided, a session is then enabled. Abstracting away implementation details we can model the execution of the program through the following three predicates: NoSession denoted by ℓ_1, ValidCredentials denoted by ℓ_2, and SessionEstablished denoted by ℓ_3. A corresponding Kripke structure is given in Figure 1.

If we want to establish that (i) SessionEstablished occurs; and (ii) until then along all preceding paths NoSession holds, we can express this property with the CTL formula **A** ℓ_1 **U** ℓ_3. Our example structure does not meet this specification as the user need not ever successfully authenticate, in which case a session is never established.

As is common we treat the number of labels q as a constant, due to it being a systemic parameter rather than an instance-specific input. We assume that δ is left-total, so that every state has at least one successor (possibly itself). We let ℓ_k for $k \in [q]$ denote an arbitrary label, and define the Boolean function $\overline{\ell}_k(s)$ to indicate whether label ℓ_k is assigned to state s.

Each ℓ_k label corresponds to some predicate, and $\overline{\ell}_k(s)$ 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 $\mathcal{M}$ 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 $\mathcal{M}$ is the directed graph induced by δ 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 $\mathcal{M}$. These traces must capture all possible behaviors of the program represented by $\mathcal{M}$ with respect to the label predicates. We concern ourselves with discrete timing, so that the ith layer of the computation tree corresponds to time $t = i$ indexed from zero (so that the root occurs at $t = 0$).

CTL, introduced in [14], is a suitable modal logic for expressing properties regarding the labeling of states in the computation tree, and so implicitly for expressing properties of the computation the tree represents. With it, we may write specifications for how the program must behave. The full grammar of CTL is given by

$$\phi := false \mid true \mid \ell_k \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \phi \to \phi \mid \phi \leftrightarrow \phi \mid \\ \mathbf{EX}\,\phi \mid \mathbf{AX}\,\phi \mid \mathbf{EF}\,\phi \mid \mathbf{AF}\,\phi \mid \mathbf{EG}\,\phi \mid \mathbf{AG}\,\phi \mid \mathbf{E}\,\phi\,\mathbf{U}\,\phi \mid \mathbf{A}\,\phi\,\mathbf{U}\,\phi.$$

The standard propositional operators are as expected, the (informal) meanings of the various temporal operators were given in the preceding paragraphs, and an atom ℓ_k is an atomic predicate represented by that label. Note as well that throughout the discussion we will use $|\phi| = m$ to denote the *operator length* of ϕ — we do not count the atomic predicates.

We say that '$\mathcal{M}$ satisfies ϕ', denoted $\mathcal{M} \models \phi$, if and only if ϕ holds at the root of all computation trees, i.e. at $t = 0$ for all traces. A model checking algorithm is a decision procedure such that $\mathsf{check}_{\mathrm{CTL}}(\mathcal{M}, \phi) = 1$ if and only if $\mathcal{M} \models \phi$. Our example formula $\phi = \mathbf{AG}\,(userdata \to \mathbf{AG}\,\neg network)$ may be read as 'for all traces starting at $t = 0$, if a state labeled *userdata* is reached then the trace must never again reach a state labeled *network*'. We overload the notation $s \models \phi$ to denote that ϕ holds at a specific state s. That $\mathcal{M} \models \phi$ then becomes expressible as 'for all $i \in I$, $s_i \models \phi$'. Multiple minimal grammars — from which the remaining operators may be constructed — are known, of which we will consider model checking over the restriction

$$\phi := true \mid \ell_k \mid \phi \wedge \phi \mid \neg\phi \mid \mathbf{EX}\,\phi \mid \mathbf{E}\,\phi\,\mathbf{U}\,\phi \mid \mathbf{A}\,\phi\,\mathbf{U}\,\phi$$

due to the simplicity of the resultant algorithm. We may refer to the full grammar when convenient, and also note that in our algorithmic discussion we will freely use substitutions $false = 0$ and $true = 1$.

For brevity we will not provide the full semantics of CTL, nor will we provide for the model checking algorithm a proof of its substantiation of those neglected semantics. We consider an informal understanding of the computation tree and temporal operators to be more than satisfactory to understand the nature of our privacy preserving construction, and refer the interested reader to [16, 17] for a far more comprehensive discussion of these concerns.

Model Checking CTL. We give a global explicit model checking algorithm for our chosen minimal CTL grammar as Algorithm 1, up to the 'quantified until' operators of **EU** and **AU** which are given in Algorithm 2 and Algorithm 4 respectively. That every temporal operator in CTL is quantified has the crucial quality that all CTL formulae are *state formulae* — their truth at a given state is independent of when in a trace the state is reached, as opposed to a *path formula* which is trace-dependent. This allows model checking in time $O(mn^2)$ for $|\mathcal{M}| = O(n^2)$ and $|\phi| = m$ as we may recursively walk through the formula tree and use the per-state truth values for each subformula as the inputs to its parent, with each operator checkable in time $O(n^2)$. We once again refer the reader to [16, 17] for discussion of state and path formulae.

The checking subroutines for $\neg\phi$ and $\psi \wedge \phi$ are both immediate: $s \models \neg\phi$ iff $\neg(s \models \phi)$ and $s \models \psi \wedge \phi$ iff $(s \models \psi) \wedge (s \models \phi)$. So we just take the output of the recursive calls and apply the relevant

```
function check_CTL(M, φ):
    o^φ ← rec(M, φ)
    sat ← 1
    for all i ∈ [n] do
        sat ← sat ∧ (o^φ[i] ∨ ¬M.S[i].inI)
    return sat

function checkAND(M, l^ψ, r^φ):
    for all i ∈ [M.n] do
        o[i] ← l^ψ[i] ∧ r^φ[i]
    return o

function checkNOT(M, r^φ):
    for all i ∈ [M.n] do
        o[i] ← ¬r^φ[i]
    return o

function checkEX(M, r^φ):
    for all i ∈ [M.n] do
        for all j ∈ [M.n] do
            o[i] ← o[i] ∨ (M.δ[i][j] ∧ r^φ[j])
    return o

function rec(M, φ):
    (op, ψ, φ) ← parse(φ)

    if op = ∧ then
        return checkAND(M, rec(M, ψ), rec(M, φ))
    else if op = ¬ then
        return checkNOT(M, rec(M, φ))
    else if op = EX then
        return checkEX(M, rec(M, φ))
    else if op = EU then
        return checkEU(M, rec(M, ψ), rec(M, φ))
    else if op = AU then
        return checkAU(M, rec(M, ψ), rec(M, φ))
    else                                                  ▷ op is an atom ℓ_k.
        k ← label(op)
        for all i ∈ [M.n] do
            o[i] ← M.ℓ̄_k(M.S[i])
        return o
```

Algorithm 1: The $\mathsf{check}_{\mathrm{CTL}}$ algorithm up to the quantified until operator subroutines and various helper functions.

Boolean operator. Moreover, $s \models \mathbf{EX}\,\phi$ 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 $\phi' \in \{\psi \wedge \phi, \neg\phi, \mathbf{EX}\,\phi\}$ the relevant subroutine can determine whether $s_i \models \phi'$ 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 $\mathbf{E}\ \psi\ \mathbf{U}\ \phi$ and $\mathbf{A}\ \psi\ \mathbf{U}\ \phi$. 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 R to those states for which ϕ 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 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 ψ 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 R, and we use another set K to track 'known' states so that we do not add a state to 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 ϕ*

(1) $\mathsf{check}_{\mathrm{CTL}}(\mathcal{M}, \phi) = 1$ *if and only if* $\mathcal{M} \models \phi$*; and*
(2) $\mathsf{check}_{\mathrm{CTL}}(\mathcal{M}, \phi)$ *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 $f(x_1, \ldots, x_u) = y$ 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 $x_1, \ldots x_{i-1}, x_{i+1}, \ldots, x_u$ 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 *semihonest* 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 semihonest model falls under the *simulation* paradigm. Suppose we wish to design a protocol Π 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 $\lambda \in \mathbb{N}$ be a security parameter. Define $\mathtt{view}_A(\Pi, x_0, B(1^\lambda, x_1))$ to be the *view* of A when interactively 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* $\textsc{Sim}_B(1^\lambda, x_0, y)$ such that A cannot distinguish between an interaction with B and $\textsc{Sim}_B$. Note that $\textsc{Sim}_B$ 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 $\mathtt{view}_A(\Pi, x_0, B(1^\lambda, x_1))$ and $\mathtt{view}_A(\Pi, x_0, \textsc{Sim}_B(1^\lambda, x_0, y))$ by outputting a bit $b \in \{0, 1\}$ identifying which counterparty they are interacting with. We will define secure computation of a function $f : X_0 \times X_1 \to Y$ by a protocol Π on behalf of B if for all PPT adversaries $\mathcal{A}$ and all $x_0 \in X_0$ and $x_1 \in X_1$, $\mathtt{view}_A(\Pi, x_0, B(1^\lambda, x_1))$ and $\mathtt{view}_A(\Pi, x_0, \textsc{Sim}_B(1^\lambda, x_0, y))$ are

[1] Also known as the *honest-but-curious* model.

computationally indistinguishable, or

$$|\Pr[\mathcal{A}(1^\lambda, \texttt{view}_A(\Pi, x_0, \textsc{Sim}_B(1^\lambda, x_0, y))) = 1] \\ - \Pr[\mathcal{A}(1^\lambda, \texttt{view}_A(\Pi, x_0, B(1^\lambda, x_1))) = 1]| \leq \texttt{negl}(\lambda)$$

where $\texttt{negl}(\lambda)$ is a function eventually bounded above by the inverse of every polynomial function of λ.

We notate computational indistinguishability of views by

$$\texttt{view}_A(\Pi, x_0, B(1^\lambda, x_1)) \approx \texttt{view}_A(\Pi, x_0, \textsc{Sim}_B(1^\lambda, x_0, y)).$$

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 $\textsc{Sim}_A(1^\lambda, x_1, y)$ 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 $\texttt{check}_{\mathrm{CTL}}(\mathcal{M}, \phi)$ 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 $(y \parallel y) \leftarrow \mathrm{GC}(\texttt{c}; x_0 \parallel x_1)$, such that if $\texttt{c}$ 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 Π 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 $\mathrm{GC}(\texttt{c}; \cdot \parallel \cdot)$ maintains the asymptotic complexity of $\texttt{c}$.

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, $RAM = (CPU, MEM)$, 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 $(\texttt{write}, a, v)$ command *MEM* copies v into the word addressed by a, upon a $(\texttt{read}, a, \cdot)$ command returns the current value in the word addressed by a, and upon $(\texttt{halt}, \cdot, \cdot)$ outputs 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 $\mathcal{AP}(s, y) = \langle a_1, \ldots, a_i, \ldots \rangle$ such that for every i, the ith message sent by $CPU(s)$ when interacting with $MEM(y)$ is of the form $(\cdot, a_i, \cdot)$.

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 $y = \langle P, x \rangle$. 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 $|\mathcal{AP}(\langle P, x_1 \rangle)|$ and $|\mathcal{AP}(\langle P, x_2 \rangle)|$ be identically distributed, then so are $\mathcal{AP}(\langle P, x_1 \rangle)$ and $\mathcal{AP}(\langle P, x_2 \rangle)$. 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 $\mathcal{M}, \mathcal{M}'$ such that $\mathcal{M}.n = \mathcal{M}'.n$, and pairs ϕ, ϕ' such that $\phi.m = \phi'.m$. So we will further assume a standardized input format so that $|\langle \mathcal{M}, \phi \rangle| = |\langle \mathcal{M}', \phi \rangle|$ for all ϕ, and $|\langle \mathcal{M}, \phi \rangle| = |\langle \mathcal{M}, \phi' \rangle|$ for all $\mathcal{M}$.

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 ϕ, 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, $\Pi_S^{otp} = (\texttt{Share}^{otp}, \texttt{Reconstruct}^{otp})$, operates as follows. To share a value $x \in \mathbb{Z}_2$, denoted $[x]$, $\texttt{Share}^{otp}(x)$ uniformly samples $a \xleftarrow{\$} \mathbb{Z}_2$ and computes $b \leftarrow x - a$ (equiv. $x \oplus a$). One party holds a as a share, the other party b. $\mathtt{Reconstruct}^{otp}(a, b)$ computes $x \leftarrow a + b$ (equiv. $a \oplus b$). We may secret share arbitrarily long bitstrings by sharing each bit separately with Π_S^{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 $\Pi_S^{prf} = (\mathtt{Gen}^{prf}, \mathtt{Share}^{prf}, \mathtt{Reconstruct}^{prf})$ requires the existence of a *pseudorandom function* (PRF). This is a keyed function $\mathrm{PRF} : \{0, 1\}^\lambda \times \{0, 1\}^y \rightarrow \{0, 1\}^z$ for $\lambda, y, z \in \mathbb{N}$ for which the distribution of $\mathrm{PRF}(sk, x)$ is computationally indistinguishable from uniformly random for an adversary which does not know sk. We let $\mathtt{Gen}^{prf}(1^\lambda)$ be the key generation algorithm for the PRF which is run at setup. To share a value $x \in \{0, 1\}^z$, denoted $[[x]]$, $\mathtt{Share}_{sk}^{prf}(x)$ uniformly samples $r \overset{\$}{\leftarrow} \{0, 1\}^y$ and computes $c \leftarrow \mathrm{PRF}(sk, r) \oplus x$. One share is then sk, and the other is $\langle c, r\rangle$. To reconstruct the value, $\mathtt{Reconstruct}_{sk}^{prf}(\langle c, r\rangle)$ computes $x \leftarrow \mathrm{PRF}(sk, r) \oplus c$.

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 $[[x_i]]$, with one party having the same share — sk — for all of them. This allows the other party to vary which $\langle c, r\rangle_i$ they input into a given intermediary computation [49]. The cost for this flexibility is that Π_S^{prf} is far more computationally expensive than Π_S^{otp}, particularly as we will need to execute these secret sharing schemes — and so our PRF — within garbled circuits. We use AES-128 for Π_S^{prf} 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 Π_S^{otp} requires no such gates due to Free-XOR techniques [41, 58].

If ambiguous, we will notate that the key share for a Π_S^{prf} share is sk by $[[x]]_{sk}$. We will abuse notation by, given a vector $\hat{x} = \langle x_1, \ldots, x_v\rangle$, using $[\hat{x}]$ to represent $\langle [x_1], \ldots, [x_u]\rangle$ and similarly for $[[\hat{x}]]$ and $\langle [[x_1]], \ldots, [[x_v]]\rangle$. We will also write $[z] \leftarrow f([x], [y])$ as shorthand for

$$(\cdot \parallel b_3) \leftarrow \mathrm{GC}(\mathtt{f}'; a_1, a_2, a_3 \parallel b_1, b_2)$$

where $a_1 + b_1 = x$, $a_2 + b_2 = y$, $a_3 + b_3 = z$, and $\mathtt{f}' = (\mathtt{Share}^{otp} \circ f \circ \mathtt{Reconstruct}^{otp})$. It will be particularly common for us to write $[z] \leftarrow [x] \wedge [y]$ 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 Π_S^{otp} over Π_S^{prf} whenever possible. So, we define two algorithms, $[x] \leftarrow \mathtt{simplify}([[x]])$ and $[[x]] \leftarrow \mathtt{complicate}([x])$ which simply compose reconstruction from one secret sharing scheme and sharing from the other as necessary.[2] We let $[x] \leftarrow$ $\textsc{Simplify}(sk, [[x]])$ stand in for

$$(a \parallel b) \leftarrow \mathrm{GC}(\mathtt{simplify}; sk \parallel \langle c, r\rangle)$$

and analogously for $[[x]] \leftarrow \textsc{Complicate}(sk, [x])$. Finally, we let $(x \parallel x) \leftarrow \textsc{Reveal}([x])$ 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 $\mathcal{M} \models \phi$ when D holds $\mathcal{M}$ and A holds ϕ. We now show that — should D and A be willing to treat n and m as public inputs — the various operator subroutines of $\mathtt{check}_{\mathrm{CTL}}$ 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 Π_S^{otp} 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 δ. Branching on the former may leak information regarding ϕ to D, the latter information about $\mathcal{M}$ 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 δ. 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_i for which $s_i \models \mathbf{E}\ \psi\ \mathbf{U}\ \phi$. 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 $s_j \models \psi$ and $(s_j, s_i) \in \delta$, in which case $s_j \models \mathbf{E}\ \psi\ \mathbf{U}\ \phi$. 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 `simplify` from a Π_S^{prf} share to a Π_S^{otp} share, the reconstructed output is treated as a bitvector and each bit reshared using Π_S^{otp} 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 Π_S^{otp} shares can be padded out with 0s to length z for `complicate` into a Π_S^{prf} 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.

1: function checkEU($\mathcal{M},\ l^{\psi},\ r^{\phi}$):
2: $o \leftarrow r^{\phi}$
3: $R \leftarrow \{i \mid r^{\phi}[i] = 1\}$
4: $K \leftarrow \varnothing$
5: while $R \neq \varnothing$ do
6: $i \leftarrow \mathtt{draw}(R)$
7: for all $j \in \{j' \mid (s_{j'},\ s_i) \in \delta\}$ do
8: if $l^{\psi}[j] \wedge j \notin K$ then
9: $o[j] \leftarrow 1$
10: $R \leftarrow R \cup \{j\}$
11: $K \leftarrow K \cup \{j\}$
12: $R \leftarrow R \setminus \{i\}$
13: return o

Algorithm 2: **The checkEU algorithm.**

1: function obcheckEU($n,\ \mathcal{M}_{\pi_1},\ l^{\psi}_{\pi_1},\ r^{\phi}_{\pi_1},\ idxs_{\pi_2}$):
2: $\hat{R} \leftarrow r^{\phi}_{\pi_1};\ \hat{\psi} \leftarrow l^{\psi}_{\pi_1};\ \hat{K} \leftarrow 0^n$
3: for all $t \in [n]$ do
4: $i,\ i' \leftarrow \perp;\ m,\ m' \leftarrow 0$
5: for all $c \in [n]$ do
6: $b_1 \leftarrow \hat{R}[c] \wedge \neg\hat{K}[c];\ b_2 \leftarrow (idxs_{\pi_2}[c] > m)$
7: $i \leftarrow c b_1 b_2 + i(1 - b_1 b_2)$
8: $m \leftarrow idxs_{\pi_2}[c] \cdot b_1 b_2 + m(1 - b_1 b_2)$
9: for all $c' \in [n]$ do
10: $b'_1 \leftarrow \neg\hat{R}[c'] \wedge \neg\hat{K}[c'];\ b'_2 \leftarrow (idxs_{\pi_2}[c'] > m')$
11: $i' \leftarrow c' b'_1 b'_2 + i'(1 - b'_1 b'_2)$
12: $m' \leftarrow idxs_{\pi_2}[c'] \cdot b'_1 b'_2 + m'(1 - b'_1 b'_2)$
13: $b_3 \leftarrow (i = \perp)$
14: $i^* \leftarrow i' b_3 + i \cdot (1 - b_3)$
15: for all $j \in [n]$ do
16: $\hat{R}[j] \leftarrow \hat{R}[j] \vee (\hat{R}[i^*] \wedge \mathcal{M}_{\pi_1}.\delta[j][i^*] \wedge \hat{\psi}[j])$
17: $\hat{K}[i^*] \leftarrow 1$
18: return $\hat{R}$

Algorithm 3: **The oblivious obcheckEU algorithm.**

which are dependent on the chosen state index from leaking any information.

Our initial change for obcheckEU regards the inputs. We require that r^{ϕ}, l^{ψ}, and the rows and columns of $\mathcal{M}.\delta$ be permuted by π_1^{-1}, the inverse of a uniformly sampled $\pi_1 \xleftarrow{\$} S_n$ where S_n is the set of permutations of length n. Under this permutation, $r^{\phi}_{\pi_1}[i]$ indicates whether $s_{\pi_1(i)} \models \phi$, $l^{\psi}_{\pi_1}[i]$ indicates $s_{\pi_1(i)} \models \psi$, and $\mathcal{M}_{\pi_1}.\delta[i][j]$ indicates $(s_{\pi_1(i)},\ s_{\pi_1(j)}) \in \delta$. We also require an additional auxiliary input $[idxs_{\pi_2}]$, which is the permuted vector $\pi_2([n])$ for some $\pi_2 \xleftarrow{\$} S_n$ sampled independently of π_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 $\hat{R}$ and $\hat{\phi}$ using these inputs, and setting an empty bitvector $\hat{K}$.

For the inner loop iteration at Lines 5-8 of Algorithm 3, if $\hat{R}[c] \wedge \neg\hat{K}[c]$ then $s_{\pi_1(c)} \models \mathbf{E}\ \psi\ \mathbf{U}\ \phi$ 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 $\mathcal{M}_{\pi_1}.\delta$ (at Lines 15-16). Though the application of π_1 makes each c 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 an adversary would know that $\sum_{k=0}^{n} \hat{R}[k] \leq c$.

To make the choice of i random, we effectively map each candidate c to $idxs_{\pi_2}[c]$, and set to i whichever has the maximal mapping. Using m as a temporary variable storing the largest $idxs_{\pi_2}[c]$ yet seen for a candidate c, at the conclusion of the loop:

$$i = \underset{c \in [n]}{\operatorname{argmax}}\{\ idxs_{\pi_2}[c] \mid \hat{R}[c] \wedge \neg\hat{K}[c]\ \}.$$

As π_2 is uniformly random and independent of π_1, i is uniformly distributed across $[n]$. In effect, π_2 makes i a uniform choice of a ready state from $\hat{R}$, which ready states are themselves randomly distributed within $[n]$ by π_1. Altogether it is functionally equivalent to Line 6 in Algorithm 2, but leaks no information about $\mathcal{M}$.

For the inner loop at Lines 9-12 we do similarly, but this time pick an i' such that $s_{\pi_1(i')} \not\models \mathbf{E}\ \psi\ \mathbf{U}\ \phi$ and which has not yet been processed. When $i = \perp$ (because every c for which $\hat{R}[c] = 1$ has already been processed) we set $i^* = i'$ instead, which pads out the outer loop by uniform selection over all yet unvisited nodes. Whether $i^* = i$ or $i^* = i'$, we then iterate down the i^*th column of $\mathcal{M}_{\pi_1}.\delta = \pi_1^{-1}(\delta)$, and for all $j \in [n]$ set

$$\hat{R}[j] = \hat{R}[j] \vee (\hat{R}[i^*] \wedge \mathcal{M}_{\pi_1}.\delta[j][i^*] \wedge \hat{\psi}[j]).$$

When $\hat{R}[i^*] = 1$ this update follows the same logic as in the original algorithm. For any padding iterations as $\hat{R}[i^*] = 0$ the right hand clause of this predicate will never hold and $\hat{R}$ will not change. We note that as a minor optimization, any value c which has previously been selected as i^* may be ignored during all three inner loops — it will never be chosen again as i or i', and either $\hat{R}[c] = 1$ or $\hat{R}[i^*] = 0$ for the current iteration.

At the conclusion of these inner loops we set $\hat{K}[i^*] = 1$ and return to the outer loop, selecting a new active state. The algorithm requires exactly n iterations of the outer loop — after a state is selected it becomes 'known' as indicated by $\hat{K}$, and we never revisit a known state. Each iteration makes a constant number of passes over the n states, giving complexity $O(n^2)$. At conclusion the vector $\hat{R}$ contains the truth values for $\mathbf{E}\ \psi\ \mathbf{U}\ \phi$ in permuted form, and the caller may apply π_1 to return the nodes to their original indices.

For the **AU** operator we introduce an additional integer array of length n, each entry of which is initialized to the out-degree of the corresponding state. The permutation π^{-1} is applied to this array as well, and in the inner loop we first decrement the jth element of this permuted array when the relevant predicate holds, and only update $\hat{R}$ if that entry has reached zero.

Let obcheck$_{\mathrm{CTL}}(\mathcal{M},\ \phi)$ be check$_{\mathrm{CTL}}(\mathcal{M},\ \phi)$ where checkEU and checkAU are replaced with their oblivious variants. The following theorem is shown in Appendix A.

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

The full oblivious checking algorithm carries over the same correctness and complexity as the original algorithm – as also shown in the appendix.

```
function checkAU(𝓜, l^ψ, r^φ):
  o ← r^φ
  R ← {i | r^φ[i] = 1}
  K ← ∅
  d ← degrees(𝓜)
  while R ≠ ∅ do
    i ← draw(R)
    for all j ∈ {j' | (s_j', s_i) ∈ δ} do
      if l^ψ[j] ∧ j ∉ K then
        d[j] ← d[j] − 1
        if d[j] = 0 then
          o[j] ← 1
          R ← R ∪ {j}
          K ← K ∪ {j}
    R ← R \ {i}
  return o
```

Algorithm 4: **The checkAU algorithm.**

```
function obcheckAU(n, 𝓜_π1, l^ψ_π1, r^φ_π1, idxs_π2):
  R̂ ← r^φ_π1; ψ̂ ← l^ψ_π1; K̂ ← 0^n
  for all t ∈ [n] do
    i, i' ← ⊥; m, m' ← 0
    for all c ∈ [n] do
      b_1 ← R̂[c] ∧ ¬K̂[c]; b_2 ← (idxs_π2[c] > m)
        i ← c b_1 b_2 + i(1 − b_1 b_2)
      m ← idxs_π2[c] · b_1 b_2 + m(1 − b_1 b_2)
    for all c' ∈ [n] do
      b'_1 ← ¬R̂[c'] ∧ ¬K̂[c']; b'_2 ← (idxs_π2[c'] > m')
      i' ← c' b'_1 b'_2 + i'(1 − b'_1 b'_2)
      m' ← idxs_π2[c'] · b'_1 b'_2 + m'(1 − b'_1 b'_2)
    b_3 ← (i = ⊥)
    i* ← i' b_3 + i · (1 − b_3)
    for all j ∈ [n] do
          b_4 ← R̂[i*] ∧ 𝓜_π1.δ_π[j][i*]
      𝓜_π1.d[j] ← 𝓜_π1.d[j] − b_4
          R̂[j] ← R̂[j] ∨ (ψ̂[j] ∧ 𝓜_π1.d[j] = 0)
    K̂[i*] ← 1
  return R̂
```

Algorithm 5: **The oblivious obcheckAU algorithm.**

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

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

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 guarantee that $\mathcal{M}$ will be sparse so checking is often closer to linear despite the quadratic worst-case. This gap is particularly acute given the 'state explosion phenomenon', whereby n is frequently exponential in the natural representation of the program as software code, a hardware design, or a protocol specification. We note however that our approach is compatible with many widely used optimizations to mitigate the state explosion phenomenon, such as partial order reduction and bitstate hashing [16, 17].

A potential direction for limiting this concrete overhead would be to employ oblivious data structures instead of requiring extraneous computation. This would however cost 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

$$(\cdot \parallel b) \leftarrow \mathrm{PPMC}_{\mathrm{CTL}}(\mathcal{M} \parallel \phi)$$

such that b correctly indicates whether $\mathcal{M} \models \phi$. 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.

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

```
protocol Init(𝓜 ‖ φ):
  D, A : sk_D ← Gen^prf(1^λ), sk_A ← Gen^prf(1^λ)
  for all i ∈ [n] do
    D :  [[deg[i]]] ← Share^prf_skD(𝓜.S[i].deg)
    D : [[idxs[i]]] ← Share^prf_skD(i)
    for all j ∈ [n] do
      D : [[δ[i][j]]] ← Share^prf_skD(𝓜.δ[i][j])
    for all k ∈ [q] do
      D : [[ℓ̂_k[i]]] ← Share^prf_skD(𝓜.ℓ̄_k(s_i))
  D :  send ⟨n⟩
  D :  send ciphertexts of ⟨[[δ]], [[ℓ̂_1]], . . . , [[ℓ̂_q]], [[deg]], [[idxs]]⟩
  A :  send ⟨m⟩
```

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

The 'and', 'not', and **EX** subprotocols take a straightforward form. A selects the appropriate $[[l^\psi]]$ and $[[r^\phi]]$ share vectors. If the true operator is unary, they pick $[[l^\psi]]$ arbitrarily for the subroutines with binary input. *It is this selection by A where our use of* Π_S^{prf} *is essential. Since D has the same share* (sk_D) *for all vectors, A may*

D $\qquad\qquad\qquad\qquad$ A

INPUT: $\mathcal{M}$ $\qquad\qquad\qquad\qquad$ INPUT: ϕ

$D \xrightarrow{(n,\ [[\delta]],\ [[\hat{\ell}_1]],\ \ldots,\ [[\hat{\ell}_k]],\ [[deg]],\ [[idxs]])} A$

$D \xleftarrow{(m)} A$

Repeat for: $h \in [m]$

For: op

$\xleftrightarrow{\textsc{And}(sk_D \parallel [[l^{\psi}]],\ [[r^{\phi}]])}$ $[[o_{\wedge}^{(h)}]]$

$\xleftrightarrow{\textsc{Not}(sk_D \parallel [[r^{\phi}]])}$ $[[o_{\neg}^{(h)}]]$

$\xleftrightarrow{\text{EX}(sk_D,\ \delta \parallel [[r^{\phi}]])}$ $[[o_{\mathbf{EX}}^{(h)}]]$

$\xleftrightarrow{\text{EU}(sk_D \parallel sk_A,\ [[\delta]],\ [[l^{\psi}]],\ [[r^{\phi}]])}$ $[[o_{\mathbf{EU}}^{(h)}]]$

$\xleftrightarrow{\text{AU}(sk_D \parallel sk_A,\ [[\delta]],\ [[l^{\psi}]],\ [[r^{\phi}]],\ [[deg]])}$ $[[o_{\mathbf{AU}}^{(h)}]]$

Keep: $[[o_{op}^{(h)}]]$

$\xleftrightarrow{\text{SAT}(sk_D,\ I \parallel [[o^{(m)}]])}$ b

OUTPUT: b

Protocol 1: PPMC($\mathcal{M} \parallel \phi$)

choose in a manner dependent on ϕ as necessary. After this selection, the chosen Π_S^{prf} shares are simplified to Π_S^{otp} 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 Π_S^{otp} shares are raised back into Π_S^{prf} shares.

```
1:  protocol And(sk_D ‖ [[l^ψ]], [[r^φ]]):
2:    for all i ∈ [n] do
3:      [l^ψ[i]] ← Simplify(sk_D, [[l^ψ[i]]])
4:      [r^φ[i]] ← Simplify(sk_D, [[r^ψ[i]]])
5:      [o[i]] ← [l^ψ[i]] ∧ [r^φ[i]]
6:      [[o[i]]] ← Complicate(sk_D, [o[i]])
7:
8:  protocol Not(sk_D ‖ [[r^φ]]):
9:    for all i ∈ [n] do
10:     [r^φ[i]] ← Simplify(sk_D, [[r^φ[i]]])
11:     [o[i]] ← ¬[r^ψ[i]]
12:     [[o[i]]] ← Complicate(sk_D, [o[i]])
13:
14: protocol EX(sk_D, δ ‖ [[r^φ]]):
15:   for all i ∈ [n] do
16:     [r^φ[i]] ← Simplify(sk_D, [[r^φ[i]]])
17:     [o[i]] ← 0
18:     for all j ∈ [n] do
19:       [o[i]] ← [o[i]] ∨ ([r^φ[j]] ∧ δ[i][j])
20:     [[o[i]]] ← Complicate(sk_D, [o[i]])
```

Note that any Π_S^{otp} shared constant may be set, such as for $o[i] \leftarrow 0$, 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 π_1 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 π_D. A holds sk_A, a vector of ciphertexts $[[\hat{x}]]_{sk_D}$, and a pair of permutations π_A and $\pi_{A'}$. Our protocol will output $\pi_{A'}\pi_D\pi_A([[\hat{x}]])$. Conventionally, either π_A or $\pi_{A'}$ will be the identity permutation $\mathbf{1}$. This allows us to employ the same protocol to compute both $\pi^{-1}\pi'^{-1}\mathbf{1}([[\hat{x}]])$ and its inverse $\mathbf{1}\pi'\pi([[\hat{x}]])$.

The protocol is relatively straightforward in formulation. A first applies π_A to $[[\hat{x}]]_{sk_D}$. 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 decrypt and then the key of A to re-encrypt. At the conclusion, D possesses $\pi_A([[\hat{x}]]_{sk_A})$ which we notate by $[[\hat{x}_{\pi_A}]]_{sk_A}$. D then applies π_D to derive $[[\hat{x}_{\pi_A\pi_D}]]_{sk_A}$, and the parties then repeat the transfer in the opposite direction. Finally, A applies $\pi_{A'}$ to arrive at $[[\hat{x}_{\pi_A\pi_D\pi_{A'}}]]_{sk_D}$ as required.

If (as in our construction) there is a use of the subprotocol where $\pi_{A'} = \mathbf{1}$ and the reshares will immediately be simplified, an alternative final transfer procedure may be used where the resharing is directly into Π_S^{otp}, to remove a few unnecessary (and expensive) $\texttt{Share}^{prf}$ and $\texttt{Reconstruct}^{prf}$ operations. For brevity, we give the permutation subprotocols over vectors. They may be adopted to permuting the rows and columns of a matrix, and we will overload our notation by invoking them on $[[\delta]]$.

```
1:  function transfer(sk_t, sk_f, [[x]]):
2:      x ← Reconstruct^prf_{sk_t}([[x]])            ▷ sk_t is key of 'to' party
3:    [[x']] ← Share^prf_{sk_f}(x)                   ▷ sk_f is key of 'from' party
4:    return [[x']]
5:
6:  protocol Perm(sk_D, π_D ‖ sk_A, π_A, π_A', [[x̂]]_{sk_D}):
7:    A : [[x̂_{π_A}]]_{sk_D} ← π_A([[x̂]]_{sk_D})
8:    for all i ∈ [n] do
9:      ([[x̂_{π_A}[i]]]_{sk_A}) ← GC(transfer; sk_D ‖ sk_A, [[x̂_{π_A}[i]]]_{sk_D})
10:   D : [[x̂_{π_A π_D}]]_{sk_A} ← π_D([[x̂_{π_A}]]_{sk_A})
11:   for all i ∈ [n] do
12:     ([[x̂_{π_A π_D}[i]]]_{sk_D}) ← GC(transfer; sk_D, [[x̂_{π_A π_D}[i]]]_{sk_A} ‖ sk_A)
13:   A : [[x̂_{π_A π_D π_A'}]]_{sk_D} ← π_A'([[x̂_{π_A π_D}]]_{sk_D})
14:
15: protocol AltPerm(sk_D, π_D ‖ sk_A, π_A, [[x̂]]_{sk_D}):
16:   ...same as Lines 7-10...
17:   for all i ∈ [n] do
18:     ([x̂_{π_A π_D}[i]]) ← Simplify(sk_A, [[x̂_{π_A π_D}[i]]]_{sk_A})
```

For $\textsc{AltPerm}$, which bits of the output must be retained is dependent on the object being permuted. Label vectors, intermediary outputs, and the transition matrix each have indicator entries, and so only a single bit need be kept. For the degree vector, however many bits are necessary to store the integer (e.g., likely 32 or 64) must be retained.

Intuitively, the permutation protocol is privacy preserving as the shares are pseudorandom due to Π_S^{prf} 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 conveys to the nested shares as well. With these permutation subprotocols, the subprotocols for **EU** and **AU** follow from our discussion in §3.

1: protocol EU($sk_D \parallel sk_A$, $[[\delta]]$, $[[l^{\psi}]]$, $[[r^{\phi}]]$):
2: $\quad D, A : \pi_{1D}, \pi_{2D} \xleftarrow{\$} S_n, \pi_{1A}, \pi_{2A} \xleftarrow{\$} S_n$
3: $\quad [idxs_{\pi_2}] \leftarrow \textsc{AltPerm}(sk_D, \pi_{2D}^{-1} \parallel sk_A, \pi_{2A}^{-1}, [[idxs]])$
4: $\quad\quad [l_{\pi_1}^{\psi}] \leftarrow \textsc{AltPerm}(sk_D, \pi_{1D}^{-1} \parallel sk_A, \pi_{1A}^{-1}, [[l^{\psi}]])$
5: $\quad\quad [r_{\pi_1}^{\phi}] \leftarrow \textsc{AltPerm}(sk_D, \pi_{1D}^{-1} \parallel sk_A, \pi_{1A}^{-1}, [[r^{\phi}]])$
6: $\quad\quad [\delta_{\pi_1}] \leftarrow \textsc{AltPerm}(sk_D, \pi_{1D}^{-1} \parallel sk_A, \pi_{1A}^{-1}, [[\delta]])$
7: $\quad$...same as Algorithm 3 with Π_S^{otp} shares and revealed i^*...
8: $\quad [[\hat{R}]] \leftarrow \textsc{Complicate}(sk_D, [\hat{R}])$
9: $\quad [[o]] \leftarrow \textsc{Perm}(sk_D, \pi_{1D} \parallel sk_A, \mathbf{1}, \pi_{1A}, [[\hat{R}]])$
10:
11: protocol AU($sk_D \parallel sk_A$, $[[\delta]]$, $[[l^{\psi}]]$, $[[r^{\phi}]]$, $[[deg]]$):
12: $\quad$...same as Lines 2-6...
13: $\quad [deg] \leftarrow \textsc{AltPerm}(sk_D, \pi_{1D}^{-1} \parallel sk_A, \pi_{1A}^{-1}, [[deg]])$
14: $\quad$...same as Algorithm 5 with Π_S^{otp} shares and revealed i^*...
15: $\quad$...same as Lines 8-9...

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 $\texttt{obcheck}_{\text{CTL}}$.

1: function SAT(sk_D, $\mathcal{M} \parallel [[o^{\phi}]]$):
2: $\quad [sat] \leftarrow 1$
3: $\quad$for all $i \in [n]$ do
4: $\quad\quad [o[i]] \leftarrow \textsc{Simplify}(sk_D, [[o^{\phi}[i]]])$
5: $\quad\quad [sat] \leftarrow [sat] \wedge ([o^{\phi}[i]] \vee \neg\mathcal{M}.S[i].inI)$
6: $\quad (\cdot \parallel b) \leftarrow \textsc{Reveal}([sat])$

At Line 5, $\mathcal{M}.S[i].inI$ 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 $\mathcal{M} = (S, I, \delta, L)$ and CTL formula ϕ, let $(\cdot \parallel b) \leftarrow PPMC(sk_D, \mathcal{M} \parallel sk_A, \phi)$. Then,*

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

Proof. Each of the component algorithms of Π_S^{otp}, Π_S^{prf}, 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 $\texttt{obcheck}_{\text{CTL}}$. So by Theorem 3.2, the protocol is correct and runs in local and communication complexities $O(mn^2)$. □

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

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

(1) there exists a PPT simulator $\textsc{Sim}_D(1^{\lambda}, \phi, b)$ such that

$$\texttt{view}_A(PPMC, \phi, D(1^{\lambda}, \mathcal{M})) \approx \texttt{view}_A(PPMC, \phi, \textsc{Sim}_D(1^{\lambda}, \phi, n, b)); \text{ and}$$

(2) there exists a PPT simulator $\textsc{Sim}_A(1^{\lambda}, \mathcal{M}, \cdot)$ such that

$$\texttt{view}_D(PPMC, \mathcal{M}, A(1^{\lambda}, \phi)) \approx \texttt{view}_D(PPMC, \mathcal{M}, \textsc{Sim}_A(1^{\lambda}, \mathcal{M}, m, \cdot)).$$

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 $\text{GC}(\texttt{c}; \cdot \parallel \cdot)$ for arbitrary $\texttt{c}$ 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 $\text{GC}\textsc{Sim}(1^{\lambda}, f, y)$. The second result is that given a PRF, an encryption scheme $\Pi_{enc} = (\texttt{Gen}, \texttt{Enc}, \texttt{Dec})$ for which $\texttt{Gen} = \texttt{Gen}^{prf}$, $\texttt{Enc} = \texttt{Share}^{prf}$, and $\texttt{Dec} = \texttt{Reconstruct}^{prf}$ 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 $\Pi = (\texttt{Gen}, \texttt{Enc}, \texttt{Dec})$ be an encryption scheme and $\lambda \in \mathbb{N}$ a security parameter. We define the experiment $\textsc{Ind-CPA}_{\mathcal{A},\Pi}$ between adversary $\mathcal{A}$ and a challenger C by*

(1) C runs $sk \leftarrow \texttt{Gen}(1^{\lambda})$ and samples $b \xleftarrow{\$} \{0, 1\}$. C then invokes $\mathcal{A}(1^{\lambda})$ and exposes an encryption oracle to it.
(2) $\mathcal{A}$ sends a pair (m_0, m_1) to C through the oracle, and receives $\texttt{Enc}(sk, m_b)$ in response.
(3) $\mathcal{A}$ repeats (2) up to n times, for $n = poly(\lambda)$.
(4) $\mathcal{A}$ outputs $b' \in \{0, 1\}$. The output of $\textsc{Ind-CPA}_{\mathcal{A},\Pi}$ is then the truth of the predicate $b \stackrel{?}{=} b'$.

Then Π provides indistinguishability for multiple encryptions under chosen ciphertext attack if for all PPT $\mathcal{A}$,

$$\Pr[\textsc{Ind-CPA}_{\mathcal{A},\Pi}(\lambda) = 1] \leq \frac{1}{2} + \texttt{negl}(\lambda).$$

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_h be the function, $x_{Dh}, x_{Ah} \in \{0, 1\}^*$ the inputs, and $y_{Dh}, y_{Ah} \in \{0, 1\}^*$ the outputs of the hth such circuit for $h \in [\tau]$.

$\textsc{Sim}_D(1^{\lambda}, \phi, n, b)$ first constructs a random Kripke structure $\mathcal{M}'$. To accomplish this, the simulator uniformly samples bits $b_1, \ldots, b_{n^2+(q+1)n} \xleftarrow{\$} \{0, 1\}$ and uses them to populate $\mathcal{M}'.S[i].inI$ for $i \in [n]$, $\mathcal{M}'.\bar{\ell}_k$ for $k \in [q]$, and $\mathcal{M}'.\delta$. It then sets the values of $\mathcal{M}'.S[i].deg$ as appropriate. The simulator then executes Init as specified over $\mathcal{M}'$.

In each of the following $m + 1$ segments of the protocol $\textsc{Sim}_D$ executes all local computations as prescribed. For the hth 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 Π_S^{prf} or a Π_S^{otp} 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 Perm and AltPerm, as Sim$_D$ cannot inspect the encrypted $[[\hat{x}_{\pi_A}]]_{sk_A}$ vectors and recover π_A as it does not know sk_A. For AltPerm, given $\mathcal{M}'$ and ϕ, D knows the underlying plaintexts $\hat{x} = x_1, \ldots, x_n$. So it may uniformly sample a permutation π'_D, and for $i \in [n]$ embed $[\hat{x}_{\pi'_D}[i]]$ as the output of the ith execution of Simplify. As there is a unique π_D such that $\pi_A\pi_D = \pi'_D$ these embeddings correctly simulate the protocol.

For Perm, Sim$_D$ again knows the underlying plaintexts. However, without knowledge of π'_A the simulator cannot embed the outputs of GCSim(`transfer`$; \cdot \parallel \cdot$) so that $[[\hat{x}_{\pi_A\pi_D\pi_{A'}}]]_{sk_D}$ will return to the original order. So, instead it embeds them in arbitrary order. If $[[\hat{x}_{\pi_A\pi_D\pi_{A'}}]]_{sk_D}$ will be the input to a later protocol segment, which Sim$_D$ knows as it has ϕ, then the simulator just embeds the outputs to any Simplify invocation as though they were correctly ordered.

Finally, as 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 Sim$_D$ evaluates each computation in the protocol at most twice, it runs in PPT $O(mn^2)$ as required.

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

We let $\mathscr{A}'$ be parameterized by $\mathcal{M}^*$ (and n), and it is given 1^λ on start by C. It begins by sampling an $\mathcal{M}'$ 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 $\mathcal{M}'$) and as m_1 the plaintext for $\mathcal{H}_1$ (from $\mathcal{M}^*$). 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 $\mathscr{A}'$ perfectly instantiates $\mathcal{H}_0$ as it

i. executes a fixed order of garbled circuit simulators;
ii. uniformly generates all Π_S^{otp} shares as required;
iii. generates all Π_S^{prf} 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 π_{1A}.

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

Returning to our sequence of hybrid distributions, for all $h \in [\tau]$, hybrid $\mathcal{H}_{h+1}$ converts the hth intermediary computation from using the garbled circuit simulator to using the real garbled circuit functionality. Then, $\mathcal{H}_{h+1} \approx \mathcal{H}_{h+2}$ 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

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

by the triangle inequality. □

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 $\mathcal{M}$ or ϕ, these experiments are demonstrative for natural problems of similar dimension. Our evaluations were made on a commodity laptop with a Intel i5-3320M 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.

n, *m*	1	2	4	7
4	4.802s	9.198s	19.333s	32.030s
	252x AES	500x AES	996x AES	1740x AES
8	15.705s	29.629s	59.407s	107.745s
	696x AES	1384x AES	2760x AES	4824x AES
16	55.482s	107.760s	210.392s	374.140s
	2160x AES	4304x AES	8592x AES	15024x AES
32	204.769s	401.424s	794.022s	1411.514s
	7392x AES	14752x AES	29472x AES	51552x AES
64	751.318s	1519.174s	3027.711s	5311.291s
	27072x AES	54080x AES	108096x AES	189120x AES

We observe a consistent cost of ≈ 20-30*ms* per AES execution, rising as it incorporates (amortized) both local computations and circuits over Π_S^{otp}. 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*.

Recent work on developing more efficient PRFs for use within MPC has produced primitives with an order of magnitude better performance than AES [1–3, 30]. In addition, some of these primitives are naturally computed in arithmetic circuits, which may

[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 work 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 $\mathcal{M}$ 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 A 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

[1] Martin R. Albrecht, Lorenzo Grassi, Léo Perrin, Sebastian Ramacher, Christian Rechberger, Dragos Rotaru, Arnab Roy, and Markus Schofnegger. 2019. Feistel Structures for MPC, and more. In *European Symposium on Research in Computer Security*. Springer, 151–171.

[2] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. 2015. Ciphers for MPC and FHE. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT '15)*. Springer, 430–454.

[3] Abdelrahaman Aly, Tomer Ashur, Eli Ben-Sasson, Siemen Dhooghe, and Alan Szepieniec. 2019. *Design of Symmetric-Key Primitives for Advanced Cryptographic Protocols.* Technical Report. Cryptology ePrint Archive, Report 2019/426.

[4] David W. Archer, Dan Bogdanov, Yehuda Lindell, Liina Kamm, Kurt Nielsen, Jakob Illeborg Pagter, Nigel P. Smart, and Rebecca N. Wright. 2018. From Keys to Databases — Real-World Applications of Secure Multi-Party Computation. *Comput. J.* 61, 12 (2018), 1749–1771.

[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. 259–269.

[6] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. 2013. More Efficient Oblivious Transfer and Extensions for Faster Secure Computation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. 535–548.

[7] Manuel Barbosa, Gilles Barthe, Karthikeyan Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. 2019. SoK: Computer-Aided Cryptography. *IACR Cryptol. ePrint Arch.* 2019 (2019), 1393.

[8] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness Theorems for non-Cryptographic Fault-Tolerant Distributed Computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing (STOC '88)*. 1–10.

[9] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. 2013. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In *Annual International Cryptology Conference (CRYPTO '13)*. Springer, 90–108.

[10] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. 2013. Data-Oblivious Graph Algorithms for Secure Computation and Outsourcing. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIA CCS '13)*. 207–218.

[11] Justin Brickell and Vitaly Shmatikov. 2005. Privacy-Preserving Graph Algorithms in the Semi-Honest Model. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT '05)*. Springer, 236–252.

[12] Ran Canetti. 2000. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology* 13, 1 (2000), 143–202.

[13] Benny Chor, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. 1985. Verifiable Secret Sharing and Achieving Simultaneity in the Presence of Faults. In *26th Annual Symposium on Foundations of Computer Science (FOCS '85)*. IEEE, 383–395.

[14] Edmund M. Clarke and E. Allen Emerson. 1981. Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In *Workshop on Logic of Programs*. Springer, 52–71.

[15] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. 1986. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8, 2 (1986), 244–263.

[16] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. 2018. *Handbook of Model Checking*. Vol. 10. Springer.

[17] Edmund M. Clarke Jr., Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. 2018. *Model Checking*. MIT Press.

[18] David Darais, Chang Liu, Ian Sweet, and Michael Hicks. 2017. A Language for Probabilistically Oblivious Computation. *arXiv preprint arXiv:1711.09305* (2017).

[19] Edmund M. Clark, Jeannette M. Wing, et. al. 1996. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys (CSUR)* 28, 4 (1996), 626–643.

[20] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2011. PiOS: Detecting Privacy Leaks in iOS Applications. In *Network and Distributed Systems Symposium (NDSS '11)*. 177–183.

[21] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2014. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 1–29.

[22] David Eppstein, Michael T. Goodrich, and Roberto Tamassia. 2010. Privacy-Preserving Data-Oblivious Geometric Algorithms for Geographic Data. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 13–22.

[23] European Parliament and Council. 2016. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the Protection of Natural Persons with Regard to the Processing of Personal Data and on the Free Movement of Such Data, and Repealing Directive 95/46 (General Data Protection Regulation). *Official Journal of the European Union (OJ)* (2016).

[24] Oded Goldreich. 2009. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press.

[25] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play Any Mental Game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing (STOC '87)*. ACM, 218–229.

[26] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM (JACM)* 43, 3 (1996), 431–473.

[27] Michael T. Goodrich, Olga Ohrimenko, and Roberto Tamassia. 2012. Data-Oblivious Graph Drawing Model and Algorithms. *arXiv preprint arXiv:1209.0756* (2012).

[28] Michael T. Goodrich and Joseph A. Simons. 2014. Data-Oblivious Graph Algorithms in Outsourced External Memory. In *International Conference on Combinatorial Optimization and Applications*. Springer, 241–257.

[29] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. 2012. Secure Two-Party Computation in Sublinear (Amortized) Time. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. ACM, 513–524.

[30] Lorenzo Grassi, Christian Rechberger, Dragos Rotaru, Peter Scholl, and Nigel P. Smart. 2016. MPC-Friendly Symmetric Key Primitives. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 430–443.

[31] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. 2019. SoK: General-Purpose Compilers for Secure Multi-Party Computation. In *2019 IEEE Symposium on Security and Privacy (S&P '19)*.

[32] David Heath and Vladimir Kolesnikov. 2020. Stacked Garbling for Disjunctive Zero-Knowledge Proofs. Cryptology ePrint Archive, Report 2020/136. https://eprint.iacr.org/2020/136.

[33] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. 2012. Secure Two-Party Computations in ANSI C. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. 772–783.

[34] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. 2003. Extending Oblivious Transfers Efficiently. In *Annual International Cryptology Conference (CRYPTO '03)*. Springer, 145–161.

[35] Jack Nicas. April 27th, 2019. Apple Cracks Down on Apps That Fight iPhone Addiction. *The New York Times* (April 27th, 2019). Accessed November 11th, 2019 at https://www.nytimes.com/2019/04/27/technology/apple-screen-time-trackers.html.

[36] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. 2013. Zero-Knowledge using Garbled Circuits: How to Prove Non-Algebraic Statements Efficiently. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security (CCS '13)*. 955–966.

[37] Jonathan Katz and Yehuda Lindell. 2014. *Introduction to Modern Cryptography*. Chapman and Hall/CRC.

[38] Jonathan Katz, Alex J. Malozemoff, and Xiao Wang. 2016. Efficiently Enforcing Input Validity in Secure Two-Party Computation. *IACR Cryptol. ePrint Arch.* 2016 (2016), 184.

[39] Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2015. Actively Secure OT Extension with Optimal Overhead. In *Annual International Cryptology Conference (CRYPTO '15)*. Springer, 724–741.

[40] Marcel Keller and Peter Scholl. 2014. Efficient, Oblivious Data Structures for MPC. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT '14)*. Springer, 506–525.

[41] Vladimir Kolesnikov and Thomas Schneider. 2008. Improved Garbled Circuit: Free XOR Gates and Applications. In *International Colloquium on Automata, Languages, and Programming*. Springer, 486–498.

[42] Yehuda Lindell and Benny Pinkas. 2007. An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT '07)*. Springer, 52–78.

[43] Yehuda Lindell and Benny Pinkas. 2009. A Proof of Security of Yao's Protocol for Two-Party Computation. *Journal of Cryptology* 22, 2 (2009), 161–188.

[44] Yehuda Lindell and Benny Pinkas. 2012. Secure Two-Party Computation via Cut-and-Choose Oblivious Transfer. *Journal of Cryptology* 25, 4 (2012), 680–722.

[45] Chang Liu, Michael Hicks, and Elaine Shi. 2013. Memory Trace Oblivious Program Execution. In *2013 IEEE 26th Computer Security Foundations Symposium (CSF '13)*. IEEE, 51–65.

[46] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. Oblivm: A Programming Framework for Secure Computation. In *2015 IEEE Symposium on Security and Privacy (S&P'15)*. IEEE, 359–376.

[47] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. Chex: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. 229–240.

[48] Silvio Micali and Phillip Rogaway. 1991. Secure Computation. In *Proceedings of the 11th Annual International Cryptology Conference (CRYPTO '91)*. Springer, 392–404.

[49] Pinkas, Benny and Schneider, Thomas and Smart, Nigel P. and Williams, Stephen C. 2009. Secure Two-Party Computation is Practical. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT*

'09). Springer, 250–267.
[50] Amir Pnueli. 1977. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science (FOCS '77)*. IEEE, 46–57.
[51] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. 2014. Wysteria: A Programming Language for Generic, Mixed-mode Multiparty Computations. In *2014 IEEE Symposium on Security and Privacy (S&P '14)*. IEEE, 655–670.
[52] A. Prasad Sistla and Edmund M. Clarke. 1985. The Complexity of Propositional Linear Temporal Logics. *Journal of the ACM (JACM)* 32, 3 (1985), 733–749.
[53] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. 2017. Global-Scale Secure Multiparty Computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. 39–56.
[54] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious Data Structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. 215–226.
[55] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. 2009. Formal methods: Practice and Experience. *ACM Computing Surveys (CSUR)* 41, 4 (2009), 1–36.
[56] David J. Wu, Joe Zimmerman, Jérémy Planul, and John C. Mitchell. 2016. Privacy-Preserving Shortest Path Computation. In *23rd Annual Network and Distributed System Security Symposium (NDSS '16)*.
[57] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets. In *27th Annual Symposium on Foundations of Computer Science (FOCS '86)*. IEEE, 162–167.
[58] Samee Zahur, Mike Rosulek, and David Evans. 2015. Two Halves Make a Whole. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT '15)*. Springer, 220–250.

A DATA-OBLIVIOUS MODEL CHECKING

Our first theorem establishing the correctness of the $\texttt{obcheck}_{\text{CTL}}$ 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 ith 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 $x_1, x_2 \in X$, should $|\mathcal{AP}(\langle P, x_1\rangle)|$ and $|\mathcal{AP}(\langle P, x_2\rangle)|$ be identically distributed, then so are $\mathcal{AP}(\langle P, x_1\rangle)$ and $\mathcal{AP}(\langle P, x_2\rangle)$.*

Throughout the following argument let $\boldsymbol{\mathcal{M}} \subseteq \{0, 1\}^*$ be the set of all binary strings interpretable as a Kripke structure $\mathcal{M}$, and $\mathbf{BitVec}_n \subseteq \{0, 1\}^*$ 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 = checkAND and for all $x \in X$ we may write $x = \langle \mathcal{M}, l^\psi, r^\phi\rangle$ for $\mathcal{M} \in \boldsymbol{\mathcal{M}}$ and $l^\psi, r^\phi \in \mathbf{BitVec}_n$ for some $n \in \mathbb{N}$ with $\mathcal{M}.n = n$, then P is data-oblivious with respect to X; and
(2) if $P \in$ {checkNOT, checkEX} and for all $x \in X$ we may write $x = \langle \mathcal{M}, r^\phi\rangle$ for $\mathcal{M} \in \boldsymbol{\mathcal{M}}$ and $r^\phi \in \mathbf{BitVec}_n$ for some $n \in \mathbb{N}$ with $\mathcal{M}.n = n$, then P is data-oblivious with respect to X.

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

$$|\mathcal{AP}(\langle P, x\rangle)| = n(c_2 + c_3) + c_4$$

and when P = checkEX

$$|\mathcal{AP}(\langle P, x\rangle)| = n(nc_1 + c_2 + c_3) + c_4.$$

In each case $|\mathcal{AP}(\langle P, x\rangle)|$ is a deterministic injective function of n. Therefore 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 if and only if x_1 and x_2 represent (in part) models $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively such that $\mathcal{M}_1.n = \mathcal{M}_2.n$. 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 $\mathcal{M}$ for which $\mathcal{M}.n = n$,

$$\Pr[\mathcal{AP}(\langle P, x\rangle) = \mathcal{AP}^*_n] = 1.$$

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

$$\Pr[\mathcal{AP}(\langle P, x_1\rangle) = \mathcal{AP}^*_n] = \Pr[\mathcal{AP}(\langle P, x_2\rangle) = \mathcal{AP}^*_n] = 1$$

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 $P \in$ {obcheckEU, obcheckAU}, $t \in [1..n]$ be an arbitrary iteration, and $a, a' \in [n]$ be arbitrary indices. Let $I^{*(t)} = \langle i^{*(1)}, \ldots, i^{*(t-1)}\rangle$ be the sequence of prior i^* values. Then,*

(1) if $a \in I^{*(t)}$ then $\Pr[i^{*(t)} = a] = 0$; and
(2) if $a, a' \notin I^{*(t)}$ then $\Pr[i^{*(t)} = a] = \Pr[i^{*(t)} = a']$.

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 ath 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 $i^* = i \neq a$ or $i^* = i' \neq a$ with certainty, $\Pr[i^{*(t)} = a] = 0$.

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

(i) $u > 0$, $\hat{R}[\alpha] = 1$, and for all $\alpha' \neq \alpha$ such that $\hat{R}[\alpha'] = 1$ and $\hat{K}[\alpha'] = 0$, $idxs_{\pi_2}[\alpha] > idxs_{\pi_2}[\alpha']$; or
(ii) $u \leq 0$ and for all $\alpha' \neq \alpha$ such that $\hat{K}[\alpha'] = 0$, $idxs_{\pi_2}[\alpha] > idxs_{\pi_2}[\alpha']$.

For (2.ii), the uniform choice of π_2 gives each unvisited index equal probability of having the greatest $idxs_{\pi_2}$ value. Moreover, since by (1) the probability that $i^{*(t)}$ will be a previously visited index is zero, it follows from the law of total probability that $\Pr[i^{*(t)} = i' = \alpha] = 1/(n-t-1)$.

For (2.i), a closed form equation for $\Pr[i^{*(t)} = i = \alpha]$ is complex, but we can observe that it depends only on $\Pr[\hat{R}[\alpha] = 1]$ and if so whether α has the greatest $idxs_{\pi_2}$ value of all unvisited 'made ready' indices. The former is uniformly distributed across all indices by π_1 regardless of model and formula structure, and the latter is uniformly distributed by the independent π_1 and π_2. As such, the probability must be the same for all indices. Then, as before by (1) and the law of total probability we have $\Pr[i^{*(t)} = i = \alpha] = 1/(n-t-1)$. □

The oblivious nature of the algorithms then follows quickly from the prior lemma.

Lemma A.5 (Obliviousness of obcheckEU and obcheckAU). *If $P \in \{\text{obcheckEU}, \text{obcheckAU}\}$ and for all $x \in X$ we may write $x = \langle \mathcal{M}, l^{\psi}, r^{\phi} \rangle$ for $\mathcal{M} \in \boldsymbol{\mathcal{M}}$ and $l^{\psi}, r^{\phi} \in \mathbf{BitVec}_n$ for some $n \in \mathbb{N}$ with $\mathcal{M}.n = n$, 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 $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively such that $\mathcal{M}_1.n = \mathcal{M}_2.n$.

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 $I^{*(n)}$ be as in Lemma A.4. By loop unrolling, it follows that for a given $I^{*(n)}$ the access pattern of P is fixed. So the argument may be reduced further to showing that $I^{*(n)}$ is identically distributed for x_1 and x_2. But, by Lemma A.4 the choice of $i^{*(t)}$ 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 ϕ*

(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* $|\mathcal{M}| = O(n^2)$ *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 checkEU and checkAU subroutines and their oblivious variants hold. This implies the functional equivalence of the $\text{obcheck}_{\text{CTL}}$ algorithm to $\text{check}_{\text{CTL}}$, at which point Theorem 2.1 and the additive $O(n)$ cost of permutations completes the argument.

Proof Sketch. As the differences between $\text{obcheck}_{\text{CTL}}$ and $\text{check}_{\text{CTL}}$ lie exclusively within obcheckEU and obcheckAU, we 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 $[\mathcal{M}.n]$.

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 o, $\hat{R} - \hat{K}$ 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 $\text{obcheck}_{\text{CTL}}$ runs in time $O(mn^2)$ and is functionally equivalent to $\text{check}_{\text{CTL}}$, Theorem 2.1 completes a proof. □