Abstract
Privacy-Preserving Formal Methods
Ning Luo
2023

Software and hardware systems can exhibit undesirable behaviors that can have catastrophic consequences. To address this, formal methods techniques provide a wide range of mathematical tools to analyze and verify the correctness, robustness, and safety of these systems against a given specification. However, while formal methods have had significant success, they do not typically consider privacy requirements during their design. This can be problematic since formal methods are often used to verify critical system components that are valuable intellectual property. Moreover, in certain contexts, outside parties such as regulatory agencies or software marketplaces impose verification requirements that system owners may not fully trust. To address these concerns, my study focuses on designing efficient formal methods that preserve the program's privacy and specification. As such, this thesis presents a suite of privacy-preserving formal method techniques.

- Privacy-preserving SAT solving. This thesis introduces ppSAT, a privacy-preserving SAT solver that addresses the need for confidentiality when solving Boolean formulas. The SAT problem involves determining whether a Boolean formula can be satisfied by assigning values to its variables. It is a core component of many verification techniques, including SMT solvers. However, existing SAT solvers do not consider privacy concerns, which can be problematic when multiple parties need to collaborate to solve a problem.

With ppSAT, two parties can determine whether the conjunction of their secret formulas is satisfiable while preserving the confidentiality of each party's input. This is achieved through the use of secure multi-party computation (MPC). As a result, ppSAT can facilitate secure collaboration between multiple parties without compromising the privacy of their inputs.

- Refutation proofs in zero knowledge. The unsatisfiability of a Boolean formula is a crucial element in formal methods for demonstrating the safety and robustness of a system. Current state-of-the-art techniques for proving unsatisfiability rely on resolution proofs. However, these methods do not consider privacy concerns.

  To address this limitation, this thesis proposes a zero-knowledge protocol for proving the unsatisfiability of Boolean formulas in propositional logic. Our protocol first encodes the validation of a resolution proof using polynomial equivalence checking. This encoding enables us to leverage fast zero-knowledge protocols for verifying relations between polynomials, ensuring that the proof remains confidential. Our zero-knowledge protocol allows for fast and efficient verification of proofs while preserving the privacy of the proof and input formulas.

- Privacy-preserving model checking for CTL formulas. Model checking is a fundamental problem in formal methods that involves verifying whether an abstract model of a computational system satisfies a given behavior specification. However, privacy concerns can arise when conducting model checking since both the model and the specification may contain confidential information.

  To address this issue, this thesis proposes a privacy-preserving model checking ap-

proach that employs MPC to maintain the privacy of both the model and the specification. Our approach uses oblivious graph algorithms to enable secure computation of global explicit state model checking with specifications written in computation tree logic (CTL).

After converting the oblivious graph algorithms to circuits consumed by MPC, our proposed approach guarantees that the private inputs of each party remain confidential during the computation. This ensures that the model and the specification remain private, and the resulting model checking is performed in a privacy-preserving manner. Overall, this thesis introduces a novel approach to privacy-preserving model checking for CTL formulas, addressing the privacy concerns associated with traditional model-checking techniques.

- Private regular expression pattern matching based on non-deterministic automata. Regular expression matching is an essential task in computer science that supports critical applications like network intrusion detection and DNS policy checking. However, with increasing privacy concerns, this thesis explores privacy-preserving regular expression matching in two different scenarios.

  The first scenario is two-party computation, where a string holder and a pattern holder privately match their inputs without revealing them to each other. This thesis proposes efficient protocols tailored to this scenario that ensure the privacy of both the string and the regular expression. Two versions of the protocol are provided: one optimized for low-bandwidth networks and short regular expressions, and the other suited for longer patterns and high-latency networks.

The second scenario is zero-knowledge proofs, where a prover generates a proof that a public regular expression matches her input string without revealing the string itself. This thesis designs and implements the first zero-knowledge proof of regular expression pattern matching, achieving concrete efficiency.

Privacy-Preserving Formal Methods

A Dissertation
Presented to the Faculty of the Graduate School
Of
Yale University
In Candidacy for the Degree of
Doctor of Philosophy

By
Ning Luo

Dissertation Directors: Ruzica Piskac

May 2023

This dissertation is dedicated to my dear grandfather, Shuncheng Luo, who passed away in 2012. He played a significant role in shaping me into the person I am today, instilling in me the values of kindness, integrity, and hard work. I believe that he is always with me and watches me in a way beyond life and death.

We will meet again.

Acknowledgment

First, I am eternally grateful to my academic advisor Ruzica Piskac. I joined her lab at the end of my second year without knowing much about formal methods. She could have me do whatever was in her comfort zone. But instead, she started learning cryptography herself and allowed me to explore where my passion was. Ruzica has created a nurturing and inspiring environment that has allowed me to focus on my research and be true to myself. Our project on developing privacy-preserving formal methods was a new area of research for our group, and I often worry about its prospects. However, Ruzica's guidance and expertise helped me overcome these doubts and achieve seemingly impossible goals. I want to express my gratitude to Ruzica for not only being an excellent academic advisor but also a supportive anchor who has empowered me to be confident and take the initiative. Without her unwavering trust, support, and encouragement, I would not have been able to achieve my research goals and personal growth

I am grateful to Timos Antonopoulos, Clark Barrett, Dan Boneh, Mariana Raykova, and Zhong Shao for agreeing to serve on my thesis committee. I thank them for their time on reviewing this thesis despite their busy schedule. I appreciate their feedback on my thesis.

I would like to express my sincere gratitude to all of my wonderful collaborators who have accompanied me on my winding journey to obtaining my Ph.D. Their guidance and teachings have been invaluable. I especially want to thank Mariana Raykova and Puwen Wei for introducing me to the world of research and cryptography. Additionally, I want to extend my greatest appreciation to Timos Antonopoulos and Xiao Wang, who I consider to be my Ph.D. mentors. The concept of privacy-preserving formal methods was initially proposed and explored by Timos, and this thesis is entirely devoted to this topic. He always encourages me and values my ideas and efforts, even if they are still in their infancy. When I thought of quitting my Ph.D. program at the end of the third year, Xiao provided me with support and began a fruitful long-term collaboration with us on privacy-preserving formal methods. Without the support and assistance of Timos and Xiao, the work in this thesis would have been impossible.

Additionally, I benefited greatly from two internships at Galois during the summer and appreciated the mentorship provided by Bill Harris, Alex Malozemoff, and James Parker. I also had enlightening and productive conversations with Eran Tromer on the

ZKUNSAT project, for which I want to express my heartfelt gratitude. Furthermore, I would like to thank Ang Chen, Yichao Chen, Gefei Tan, Jaspal Singh, Yuyang Sang, Chenkai Weng, Qiao Xiang, and Ennan Zhai for their collaboration.

I would like to express my deep appreciation for the support and kindness extended to me by my colleagues throughout my journey. I am particularly grateful to Mark Santolucito, who, during CAV 2019, took me for a walk in New York City and lifted my spirits when I was feeling down and anxious about my future. Mark has been an incredible mentor and role model, showing me how to take the initiative and pursue my goals. Additionally, I am grateful for the productive collaboration I had with Samuel Judson, who always had insightful and inspiring ideas. Samuel was the first person to welcome me warmly when I joined the ROSE lab. I also want to thank Ben Chaimberg and John Kolesar for their invaluable contributions and for making my work experience more enjoyable. Finally, I would like to express my gratitude to Ferhat Erata, Matt Elacqua, Bill Hallahan, and Jialu Zhang for their unwavering support and for tolerating all my occasional screaming and drama, which was annoying, I know, especially before deadlines. A special thanks to John Kolesar for proofreading this thesis and providing valuable feedback.

I am grateful to my parents, Jirong Luo and Hongyu Kuang, for their unwavering love and trust. I would also like to extend my gratitude to my twin sister, Jing Luo, for serving as my reflection and motivating me to acknowledge and pursue my aspirations. A special thank you goes to Yuyang Sang for his unwavering presence and comforting hugs throughout my Ph.D. Friends are a crucial part of my graduate school experience. I want to thank my non-academic friends, especially Yishu Zhou and Gongmou Wang, for continually reminding me of my worth. I have shared some wild times and may have vented too much when drinking with Yishu Zhou, yet she still perceives me as a kind person. During my most trying moments in graduate school, Gongmou Wang called me every morning and offered words of encouragement. Although I doubt they will peruse a single page of this thesis, their steadfast support has been a valuable gift.

Finally, I would like to extend my thanks to *Koffee?*, a coffee shop near the computer science department. Over the past five years, I have enjoyed a cup of their coffee almost every day. Their coffee helped me stay awake and fueled my productivity.

Funding I gratefully acknowledge the funding that supported the research that appears in this thesis. In particular, I acknowledge the generous support from the National Science Foundation (NSF) and Defense Advanced Research Projects Agency (DAPAR).

Contents

List of Figures

Chapter 1

Introduction

Software and hardware have the potential to exhibit unwanted behavior, which can lead to disastrous consequences. For instance, the maiden flight of the Ariane 5 launcher on 4 June 1996 ended due to a software failure, resulting in a public inquiry and a loss of approximately $370m [91]. In 2012, Knight Capital Group, a financial services company, lost $440 million in less than an hour due to a software bug [182]. Recently, Tesla had to recall 130,000 vehicles in the United States due to a software bug in its infotainment system, which caused overheating and resulted in the processor lagging or restarting.

To prevent software from exhibiting unexpected behavior, the field of formal methods offers a vast collection of techniques for mathematically analyzing and verifying software and hardware systems, ensuring their correctness, robustness, and safety. Many of these techniques, such as Z3 [179], CVC4 [28], BLAST [147], TLA$^+$ [229], Kudzu [196], and KLEE [53], are not only of research interest but are also widely used and implemented in industry.

Although verification tools have brought about significant social and economic benefits, their initial design and development do not consider privacy requirements. This lack of awareness has led to pressing issues, particularly because these tools usually focus on critical system components that may contain sensitive intellectual property. Moreover, the process of specifying and verifying these components may involve third-party entities that system owners do not fully trust. A concrete example of this is the challenges faced by centralized software distribution platforms that use a curated central marketplace to formally analyze and verify the software they distribute, which can identify and prevent malicious applications.

However, centralized curated markets require trust from both software developers and users. This can be especially difficult for developers of closed-source software, who need to trust that the market will not disclose their implementation to others. Meanwhile, users must trust that the market will properly analyze the software and verify its properties. The centralized nature of these markets can also create monopolies in the distribution process, giving companies such as Apple and Google special access to competitors' software and hindering third-party service alternatives. This

power has come under scrutiny, such as when Apple removed parental control apps it had long allowed after integrating competing functionality directly into its system in 2019 [130].

Centralized security monitoring also creates a conflict between the propriety of a developer's intellectual property and the need for verification. This issue has recently gained significant attention and has prompted multiple enforcement actions and legislative proposals throughout the EU and the US [5, 7, 8, 69, 130, 131, 140].

In order to address these concerns, privacy-preserving verification tools can be developed to ensure that the software being distributed is secure and does not compromise user privacy. This would reduce the need for centralized curation, as the software can be verified locally by the user's device rather than relying on a third-party marketplace. It would also alleviate concerns about the proprietary nature of close-source software, as the verification process can be performed without revealing the underlying implementation. Additionally, developing open-source verification tools can help address issues of monopolies in the distribution process and anti-competitive practices by centralized marketplaces.

To achieve privacy-preserving verification, it is possible to implement formal methods under existing privacy-preserving frameworks, such as multiparty computation (MPC) [95, 174, 227] and zero-knowledge proof (ZKP) [94, 34]. ZKPs enable one party, the *prover*, to convince a second party, the *verifier*, of the validity of a claim without revealing the information about the evidence of the claim. MPC allows multiple parties to compute a function that takes their secret inputs jointly. A secure MPC protocol guarantees the privacy of these inputs. By applying these privacy-preserving techniques to existing formal methods, the privacy of the programs can be maintained. However, it should be noted that while modern cryptographic protocols have made significant progress in terms of efficiency and scalability, generic applications may still be impractical and inefficient.

This thesis aims to resolve the tensions between verifiability and privacy by introducing efficient verification algorithms that retain the privacy of a program or even the property to be checked. This thesis focuses on four fundamental verification techniques widely used in practice.

- SAT solving: The SAT problem [65, 71, 70, 134] is the task of finding an assignment of variables to make a Boolean formula evaluate to true. SAT problem is NP-complete, which means that there is no general polynomial-time algorithm to solve it. Recent decades have witnessed the development of SAT solvers that scale to formulas containing millions of variables and clauses. As one of the most fundamental problems in computer science, SAT solving is the core of many verification techniques. This thesis offers a privacy-preserving SAT solver that enables two parties to decide if the conjunction of their input formulas is satisfiable without revealing their input formulas.

- Resolution proof: Proving that a Boolean formula is unsatisfiable (UNSAT) is a critical task for illustrating the safety or robustness of a system. The existing state-

of-the-art techniques utilize resolution proofs for this task. Resolution proofs are certificates of unsatisfiability for formulas in first-order logic. This thesis introduces an efficient protocol that enables a prover to demonstrate the unsatisfiability of a private formula to other parties, say verifiers, without revealing the formula or the resolution proof.

- Regular expression pattern matching: The inputs of regular expression pattern matching are a string and a pattern described by a regular expression. The goal is to determine whether the given pattern appears in the string [209]. As one of the most fundamental query operations in computer science and formal languages, regular expressions are widely used in networks, bioinformatics, databases, and text processing. This thesis provides private regular expression pattern-matching protocols that ensure the privacy of the strings and the regular expressions in two different settings. The first setting is two-party computation, where a string holder and a pattern holder privately match their inputs without revealing them to each other. In the second setting, this thesis explores zero-knowledge proofs, where a prover generates a proof that a public regular expression matches her input string without revealing the string itself. This thesis also demonstrates private regular expression pattern matching designs for both settings based on the oblivious stack [232] and oblivious transfer protocols [142] and ZKP [94, 34].

- Model checking for computational tree logic (CTL) formulas: model checking [60] plays an important role in the suite of techniques available for formally verifying the correctness of programs. Given a finite transition system model of a program along with the specification of the safety or liveness property, a model checker will directly confirm that that program will fulfill that specification. These specifications for model checking are usually written as formulas over the temporal-logic formula. In this thesis, we focus on the computation tree logic [63, 62], one of the best-known and most commonly used temporal logics in model checking. This thesis presents an efficient protocol for privacy-preserving model checking, which enables a verifier or auditor to check if a programmer's private finite transition system satisfies its secret specification. The protocol we show in this thesis ensures the privacy of the finite-state transition system and the specification.

1.1 Contributions

In this thesis, a set of tools for formal methods that maintain privacy has been created, including SAT solving with privacy, proving unsatisfiability in zero knowledge, CTL model checking with privacy, and private regular expression pattern matching. We summarize the contributions of this thesis for the development of these privacy-preserving tools for formal methods in this section

1.1.1 Privacy-Preserving SAT Solver

Secure 2-party computation (2PC) [227] enables two parties who do not trust each other to compute a public function on their private inputs. The security of 2PC ensures that only limited information about private inputs will be revealed. Furthermore, all 2PC constructions use a data-oblivious execution pattern to prevent information leakage.

On the contrary, existing SAT decision procedures, such as DPLL [71], use highly input-dependent data structures to achieve high efficiency. It is non-trivial to develop an oblivious variant of the SAT decision procedure. Moreover, these existing SAT decision procedures rely heavily on heuristics that guide the unforced choices and unwind the choices to backtrack when necessary. These smart heuristics are the core of highly effective modern SAT solvers. For example, one of the simplest heuristics randomly assigns a randomly chosen variable that does not yet have a valuation. However, even the simplest heuristics can be unsuitable in 2PC as they involve expensive data accesses or arithmetic operations. Lastly, the classical security definition for 2PC requires runtime independent of the input formula. That is because the adversary can infer information about the input without the runtime independence. However, the classical definition will be too strong to be practical: the runtime is exponential to the size of input formulas in the worst case due to the NP-Complete nature of the SAT problem.

This thesis presents a privacy-preserving SAT solver, ppSAT, based on 2PC. We approach the above challenges to achieve a privacy-preserving 2-party SAT solver by the following:

1. This thesis introduces a suitable data structure to represent formulas, which is a pair of binary matrices. The implementation of the DPLL procedure is presented as linear scans over the matrix.

2. Developing heuristics that fit the 2PC setting. This thesis also addresses the challenge of developing heuristics that are suitable for the 2PC setting. To accomplish this, we present three heuristics for SAT solving: DLIS, RAND, and Weighted-RAND. DLIS is a deterministic heuristic that selects the most frequently appearing literal and assigns it a value to make it true. On the other hand, RAND and Weighted-RAND are probabilistic heuristics. RAND randomly selects an unassigned variable and assigns it a random value, while Weighted-RAND chooses a literal based on its frequency in the formula.

3. Formalizing a variant of a simulation-based security definition for ppSAT solving. This thesis formalizes a simulation-based security definition for ppSAT solving, which accounts for some leakage. However, the amount of leakage required depends on whether the runtime is leaked to enable early termination. The thesis also investigates the trade-off between the efficiency of solving SAT in the privacy-preserving setting and the information leakage resulting from revealing the run-

time. Additionally, the thesis explores the possibility of reducing the amount of leakage using techniques such as differential privacy (DP) [77].

4. A prototype of a privacy-preserving 2-party SAT solver was deployed and used to benchmark instances arising from the haplotype inference problem in bioinformatics. The benchmarks showed that ppSAT takes more than two days to solve some instances, whereas these same instances can be solved within 0.02 seconds if privacy is not a concern. Nonetheless, the results demonstrate that there is potential for further improving SAT solving in the privacy-preserving setting.

1.1.2 Proving UNSAT in Zero Knowledge

Zero-knowledge proofs enable one party, prover, to prove a claim to the second party, verifier, while retaining the confidentiality of the evidence for the claimed secret. Although existing efficient protocols mainly focus on NP problems, there are many intriguing problems in program verification and formal methods outside NP. UNSAT, which captures the program's safety proof, is one such problem of interest.

For any unsatisfiable propositional Boolean formula, a resolution proof exists as the certificate of unsatisfiability. The resolution proof is a sequence of clauses that can be derived from the input formula. If the input formula is unsatisfiable, a false can be derived at the end of the resolution, proving the formula's unsatisfiability by contradiction.

This thesis presents a novel and efficient protocol, ZKUNSAT, for proving the unsatisfiability of a propositional Boolean formula in zero knowledge. We leverage the low width of resolution proofs, which means that clauses in the proof typically contain a small number of literals. Based on this observation, we introduce a method for encoding resolution proofs using low-degree polynomials over a finite field. The validation of resolution proofs is reduced to verifying the relationship between polynomials. This thesis makes the following contributions:

1. This thesis is the first to investigate the practicality of proving the unsatisfiability of propositional Boolean formulas with privacy, and how proving the unsatisfiability in zero knowledge can assist in proving the properties of private programs.

2. We describe a method for encoding Boolean formulas and resolution proofs of their unsatisfiability using polynomials over a finite field in this thesis. Our ZK-friendly algebraic encodings provide significant scalability, making ZKUNSAT efficient enough to handle real-program verification formulas and proofs.

3. Additionally, this thesis presents a prototype of ZKUNSAT and benchmarks it against large formulas. The prototype can verify the unsatisfiability of formulas corresponding to the verification tasks of Linux/Windows drivers.

1.1.3 Private Regular Expression Pattern Matching

Regular expression pattern matching determines whether a string satisfies the pattern specified by a regular expression. As one of the fundamental query operations, regular expression pattern matching is used widely in many privacy-sensitive settings, such as private DNS queries, DNA tandem repeat detection, and network package intrusion detection. To decide whether a string matches the pattern, a pattern-matching algorithm usually starts by converting the regular expression to a deterministic finite automaton (DFA) or a nondeterministic finite automaton (NFA). Many existing approaches to matching regular expression patterns that preserve privacy choose the former and scale relative to the size of DFAs [231, 176, 151, 89, 211]. Unfortunately, these approaches can incur significant overhead since the size of a DFA can be exponential relative to the size of the original regular expression. On the other hand, NFA-based approaches in the literature achieve obliviousness by representing NFAs as matrices [151, 195]. Their overhead is linear to the size of input regular expressions.

In this dissertation, we propose efficient protocols for regular expression matching based on this linear scan-based algorithm for TNFA simulation. Our contribution is summarized as follows.

1. We take advantage of the TNFA's structure [209] and find that it is a sparse graph, with nodes accessible through a path that includes only one backward edge if reachable. Using this discovery, we have designed an algorithm that uses two linear scans to simulate TNFA with $O(mn)$ complexity. This algorithm is oblivious except during the retrieval of each state's preceding states for epsilon transitions.

2. We provide a protocol for secure Thompson NFA simulation based on the oblivious stack [218]. This protocol achieves the constant round complexity, while its communication complexity is $O(mn\kappa \log n)$. This protocol can offer security under both semi-honest and malicious models. Our evaluation shows that for regular expressions of length 2^6 and strings of length 2^{12} bytes, our tool takes about 1 minute for secure pattern matching.

3. Using the fact that the pattern holder knows the state's predecessors, we further improve the performance of secure-regex in practice. We propose a protocol for secure Thompson NFA simulation based on oblivious transfer (OT) [189], which enables accessing the predecessors without revealing the interconnection of the states. This protocol achieves $O(n \log n + mn\kappa)$ communication and computation complexity when the pattern is small (where κ is the security parameter) and $O(mn^2)$ round complexity and security under the semi-honest model.

4. Based on this oblivious algorithm, we develop a zero-knowledge proof protocol for proving the secret string's membership of a public regular expression. Our protocol produces a proof of size $O(mn)$. We evaluate our approach on a dataset containing a set of regular expressions used for a blacklisting check for DNS queries [6]. Our algorithm scales well: with a DNS query size of 100 bytes, matching against the

longest regular expression from the benchmarks results in a circuit containing around 14k gates.

1.1.4 Privacy-Preserving Model Checking for CTL formulas

Finite-state transition systems serve as program execution models and can be extended further as a computation tree. Computation tree logic (CTL) is a temporal modal logic that operates on the computation tree of program execution. It can express the properties of the program's execution indirectly as properties of the computation tree. This thesis considers the privacy-sensitive setting where an *auditor* A wishes to verify a program held by D, the *developer*. D possesses a model $\mathcal{M}$ of program execution represented as the finite-state transition system, while A has a specification of program behavior ϕ written in CTL. Both the finite-state transition system and the specification are private. We note that the validity of this setting depends on the assumption D input of an $\mathcal{M}$ that accurately and adequately represents the execution of the program. A must rely on some other means to require D to provide honest input, such as substantial legal recourse and additional privacy-preserving tools for model extraction. This thesis solves the problem of privacy-preserving model checking and makes the following contribution.

1. This thesis recognizes the demand for privacy-preserving model checking. It discusses the necessary considerations for fitting it into the broader system to ensure that its privacy and correctness goals are practically met. Although it does not fully eliminate the need for trust or binding agreement between the developer and the auditor, our protocol reframes disputes between the developer and the auditor in terms of the easily formalized question of whether the developer misrepresented M, as opposed to the murky indeterminate question of whether the auditor gained valuable information from M. Having increased precision may make asymmetric privacy and correctness concerns easier to negotiate.

2. This thesis takes advantage of the cryptographic theory of MPC to propose an efficient protocol that heavily limits the information that D and A learn about each other's inputs under standard adversarial assumptions. The protocol achieves constant overhead through oblivious graph algorithms [45].

3. This thesis provides a prototype of privacy-preserving model checking and evaluates transition systems and specifications of various sizes. The bottleneck of the protocol is symmetric key encryption under MPC. Replacing symmetric key encryption with PRFs that are more efficient under MPC may be an avenue for future improvement and optimization of the protocol. [10, 9, 11, 106].

1.2 Thesis Outline

The rest of this thesis is organized as follows:
In **Chapter 2**, we provide a comprehensive explanation of ppSAT, a privacy-preserving SAT solver, based on [164]. This chapter begins by discussing the motivation for privacy-preserving SAT solving and its potential applications. We then delve into the detailed construction of an efficient privacy-preserving SAT solver and the heuristics the solver employs. Our evaluation results of the solver against real-world benchmarks are also presented in this chapter, along with a comparison with state-of-the-art SAT solvers that do not preserve privacy. Finally, we explore advanced heuristics that may improve the performance of ppSAT.
Chapter 3 explains ZKUNSAT, a protocol designed to verify unsatisfiability in zero knowledge, as presented in the paper [162]. The chapter begins by highlighting how verifying the safety of a program can be achieved by proving the unsatisfiability of a Boolean formula. We then explore the zero-knowledge proof techniques and polynomial-based representations of resolution proofs and Boolean formulas that ZKUNSAT implements. Moreover, the chapter delves into ZKUNSAT's novel relaxed resolution rule, which can be efficiently verified in zero knowledge without compromising soundness when polynomials encode the resolution proofs and formulas. Lastly, we provide our results from evaluating ZKUNSAT against formulas and proofs derived from real-world verification tasks. By the end of the chapter, readers will gain a comprehensive understanding of ZKUNSAT's functioning, including its applications and the advantages it offers over other protocols.
Chapter 4 proposes efficient protocols tailored for private regular expression pattern matching in different scenarios. In the first scenario, we propose two protocols that ensure the privacy of both the string and regular expression and come in two flavors: one is more efficient for low-bandwidth networks and short regular expressions, while the other is better suited for longer patterns and high-latency networks. In the second scenario, we design and implement the first zero-knowledge proof of regular expression pattern matching and achieve concrete efficiency. Our experiments in encrypted DNS policy checking and intrusion detection demonstrate the practicality of our protocols in real-world applications.
Chapter 5 explains our oblivious algorithm for global explicit state model checking for CTL formulas. It is based on [133]. In this chapter, we show how cryptographic primitives allow the algorithm to be converted into a secure MPC protocol against semi-honest adversaries. The protocol has constant asymptotic overhead. At the end of the chapter, we discuss potential improvements in concrete cost and feasibility for future work.
Chapter 6 concludes the thesis and discusses future work.

Limitation of this Thesis. Chapters 2 and 3 delve into the problem of solving and certifying the satisfiability or unsatisfiability of formulas in a privacy-preserving setting. These chapters only consider propositional Boolean formulas. The ppSAT

solver described in Chapter 2 is based on the original DPLL algorithm, which does not learn from failed branches in its search. CDCL, the central enhancement of modern SAT solving, is the dominant approach for addressing this shortcoming [90]. However, since ppSAT lacks CDCL, it only scales to small-to-medium-sized SAT instances and exhibits underperformance on UNSAT instances. Chapter 3 is motivated by the verification of private programs. This thesis focuses on verifying the unsatisfiability of private formulas and resolution proofs but leaves the verification of the refinement relation between private formulas and programs for future work.

Both Chapter 4 and 5 guarantee privacy only under the semi-honest model. Chapter 4 offers secure regular expression pattern matching, where the regular expressions considered only allow concatenation, union, and the Kleene star. Chapter 5 presents privacy-preserving model checking, where the algorithm is based on explicit state model checking. The implementation is used as a proof of concept, and the evaluation results are limited to models and formulas of relatively small size.

Chapter 2

Privacy-preserving SAT solving

We design and implement a privacy-preserving Boolean satisfiability (ppSAT) solver, which allows mutually distrustful parties to evaluate the conjunction of their input formulas while maintaining privacy. We first define a family of security guarantees reconcilable with the (known) exponential complexity of SAT solving, and then construct an oblivious variant of the classic DPLL algorithm which can be integrated with existing secure two-party computation (2PC) techniques. We further observe that most known SAT solving heuristics are unsuitable for 2PC, as they are highly data-dependent in order to minimize the number of exploration steps. Faced with how best to trade off between the number of steps and the cost of obliviously executing each one, we design three efficient oblivious heuristics, one deterministic and two randomized. As a result of this effort we are able to evaluate our ppSAT solver on small but practical instances arising from the haplotype inference problem in bioinformatics. We conclude by looking towards future directions for making ppSAT solving more practical, most especially the integration of conflict-driven clause learning (CDCL).

2.1 Introduction

Boolean satisfiability (SAT) is a foundational problem in computer science [65, 71, 70, 134]. SAT asks whether there is a variable assignment (or *model*, $\mathcal{M}$) that makes a Boolean propositional formula ϕ evaluate to true. A SAT solver is a tool that takes an instance ϕ as input and checks its satisfiability; solvers can also output a model when one exists. The SAT problem is NP-complete and widely believed to require at least superpolynomial, if not exponential, time [127, 197]. Most state-of-the-art SAT solvers are in fact enhancements of the worst-case exponential branch and backtrack algorithm of Davis-Putnam-Logemann-Loveland (DPLL) [71, 70]. Nonetheless, well-engineered modern solvers such as Kissat, the basis for the winner of the 2021 SAT competition [25], can efficiently resolve large and complex SAT instances containing tens of millions of variables and clauses [25, 44, 42, 114, 178, 201, 205, 204, 90], arising from within program verification [111, 152, 173], networks [32, 160, 165], and numerous other domains [166, 167, 204].

All existing SAT solvers are designed for execution by a single party possessing complete information on ϕ.[1] However, in certain settings SAT instances arise as the conjunction of inputs from two or more distinct parties, *i.e.*, $\phi \equiv \bigwedge_{i \in [k]} \phi_i$ where party P_i formulates ϕ_i independently of ϕ_j for all $j \neq i$. If the P_i are mutually distrustful and their inputs are valuable, privileged, or legally encumbered – and so must be kept private from the other parties – then those solvers are no longer applicable without a trusted intermediary or secure execution environment to run them. In settings without recourse to such trust assumptions, secure computation techniques are instead required to privately resolve SAT instances. My work work combines recent advancements in oblivious algorithm design [218] with classic techniques for SAT solving [71, 70] and secure two-party/multiparty computation (2PC) [227] to develop a solver for *privacy-preserving Boolean satisfiability*, or ppSAT. We also consider the promise and challenge of augmenting this secure computation with differential privacy (DP) [77, 78] to trade off privacy and efficiency, following a strand of recent research [107, 118].

SAT solvers take as input Boolean formulas in conjunctive (also known as clausal) normal form (CNF). We focus on the setting with two parties P_0 and P_1 and a Boolean formula $\phi \equiv \phi_0 \wedge \phi_1 \wedge \phi_{pub}$ such that ϕ_b is the private input of P_b for $b \in \{0, 1\}$ and ϕ_{pub} is an optional public input. This allows us to use the most concretely efficient designs and software for secure computation available. Also, the general architecture and optimizations of my construction should be extendable to support more than two parties given suitable secure computation primitives. I assume that P_0 and P_1 have agreed on the meaning of a set of n variables $v_1, \ldots, v_n$. A two-party ppSAT solver takes private inputs ϕ_0 and ϕ_1 from P_0 and P_1 respectively, where ϕ_b is over the v_i with $|\phi_b| = m_b$ clauses. The solver should correctly output a bit $s = (\exists \mathcal{M}.\ \mathcal{M} \models \phi)$, and optionally output a satisfying $\mathcal{M}$ when possible. Intuitively, we desire a security guarantee that P_b learns nothing more about ϕ_{1-b} than is implied by s, ϕ_b, $|\phi_{1-b}|$, (when input) ϕ_{pub}, and (when output) $\mathcal{M}$.[2]. We design and implement a sound ppSAT solver that meets a slightly relaxed security guarantee, necessary due to the exponential worst-case runtime of my SAT decision procedure.

A Motivating Example. In bioinformatics, inference of haplotypes can uncover genetic information valuable to biological research and medical treatment. Haplotypes are DNA sequences which originate from a single parent, and their information can aid studies on, *e.g.*, genetic risk factors for cardiovascular disease [191] or the effective use of medications [105]. However, current genetic sequencing technology usually only resolves genotypes, which are mixtures of haplotypes from both parents. Given a set of genotypes G drawn from a population, the process of inferring a set of haplotypes H whose elements explain every $g \in G$ (*i.e.*, that are plausibly the biologically-

[1] Prior work does consider parallel/distributed SAT solving, but only where that single party coordinates networked computing resources [171].

[2] As is common in 2PC it is difficult for P_{1-b} to hide the length $|\phi_{1-b}|$. When necessary SAT instances can be padded out with tautological clauses.

realized haplotypes in that population) is called haplotype inference [105, 112]. One computational approach is haplotype inference by pure parsimony (HIPP) [105, 112]. HIPP finds a minimally-sized H to explain an input G, and is known to be APX-hard [150].

Formally, (sections of) both haplotypes and genotypes may be expressed as strings of length ℓ, with a haplotype $h \in \{0, 1\}^\ell$ and a genotype $g \in \{0, 1, 2\}^\ell$. Given a pair of haplotypes $hs = (h^0, h^1)$ and a genotype g, the predicate

$$\begin{aligned} &\mathit{explainI}(hs, g) \iff \\ &\qquad \forall i \in [\ell].\ (h^0_i = h^1_i = g_i) \vee (h^0_i \neq h^1_i \wedge g_i = 2) \end{aligned}$$

is true iff g can be explained as a mixture of h^0 and h^1. A set of haplotypes H explains a set of genotypes G if every $g \in G$ can be obtained by pairing two h^0, $h^1 \in H$:

$$\mathit{explain}(H, G) \iff \forall g \in G.\ \exists\, hs \in H \times H.\ \mathit{explainI}(hs, g).$$

For example, $H = \{010, 110, 001\}$ explains $G = \{210, 022\}$, as (010, 110) explains 210 while (010, 001) explains 022. It is straightforward to see that a minimally-sized H has $2 \leq |H| \leq 2|G|$.

SHIPs [166, 167] solves the HIPP problem for genotype set G by invoking a SAT solver to find an H which makes $\mathit{explain}(H, G)$ true. Specifically, for a conjectured size of the haplotype set $r \in [2, 2|G|]$, the SHIPs algorithm converts $\mathit{explain}(H, G) \wedge |H| = r$ into a CNF formula ϕ where the elements of H are represented by $r \cdot \ell$ variables. The satisfiability of ϕ is then checked by a SAT solver, and if true the resultant model $\mathcal{M}$ encodes H. To find a minimal H, SHIPs starts with $r = 2$ and increments it until ϕ is satisfiable (as a model necessarily exists when $r = 2|G|$).

Genetic information such as genotypes can be expensive to obtain, can carry significant privacy risk, and/or can be encumbered with legal and regulatory protections. Commercial and academic researchers who collect and analyze genotype data often have strong incentives to control access to it [207]. Anonymization and summarization of genetic data is not a panacea, to the extent those notions are technically and legally meaningful at all [181]. Homer *et al.*'s attack [124, 216] shows auxillary information paired with the genotype of a target may be used to identify their participation in a genome-wide association study (GWAS). If two parties respectively hold databases of genotypes G_0 and G_1 and want to run haplotype inference over $G = G_0 \cup G_1$ without exposing their data to the other participant, current technologies for HIPP require trading off the privacy of that data against the economic and social value of the research. Even when legal infrastructure can provide and enforce privacy guarantees to allow otherwise reticent parties to share data, the time and cost of lawyers and negotiations and contracts may very well outpace even the most expensive secure computations.

The application of SHIPs through a ppSAT solver could help mitigate all of this tension. Up to a minor optimization not required for correctness, the CNF formula ϕ

encoding $explain(H, G) \wedge |H| = r$ is naturally composed of (i) a public ϕ_{pub} encoding $|H| = r$; and (ii) two independent ϕ_b each derived only from G_b, such that $\mathcal{M} \models \phi$ iff $\mathcal{M} \models \phi_0 \wedge \phi_1 \wedge \phi_{pub}$. As such, each party P_b can construct ϕ_b locally, at which point they can jointly execute the ppSAT solver over ϕ. Solving this formula infers an H that explains G while keeping G_0 and G_1 private, up to their cardinalities. This approach may not completely mitigate privacy concerns, since H is itself (inferred) genetic data which may be, *e.g.* correlated with observable medical conditions and the community of origin of the individuals who provided G. However, as haplotypes are far less diverse within a population, their exposure may carry less risk than underlying genotypes [66, 185].

Naive Approaches. While to the best of our knowledge, no specific study of privacy-preserving SAT solving exists in the literature, both basic secure computation primitives and general completeness results can be used to naively instantiate ppSAT solvers. One immediate approach is to use the private set disjointness (PSD) test [88, 143]. Each P_b could enumerate the set $M_b = \{\mathcal{M}_i \mid \mathcal{M}_i \models \phi_b\}$ of assignments that satisfy their formula, and then the parties could jointly execute a PSD protocol to check whether $|M_0 \cap M_1| > 0$. A model $\mathcal{M}$ could be recovered by replacing PSD with private set intersection (PSI) [187]. However, $|M_b|$ can be worst-case exponential in $|\phi_b|$, is often very large in concrete terms [31, 30], and is leaked by this approach, as is $|M_0 \cap M_1|$ when using PSI. In general, this enumeration is #P-hard [44], so straightforward application of PSD/PSI will likely be impractical.

Another naive approach is to raise a preexisting SAT solver to a ppSAT solver through a generic 2PC compiler. However, current such compilers (often bluntly) apply techniques such as the padding out of loops and linear scan array lookup to create data-oblivious execution paths [115], which will likely be impractical given the amount of state management and data-dependent processing of known SAT decision procedures. Adoption of RAM-based 2PC methods [103] is more promising, but generic use of their compilers is for the moment unreasonably expensive.

2.2 Notation

We denote the two parties as P_0 and P_1, whose inputs are CNF formulas ϕ_0 and ϕ_1 respectively. When applicable we represent a public component of the formula, known to both parties, by ϕ_{pub}. The set of variables in ϕ is $V = \{v_1, \ldots, v_n\}$, so $|V| = n$. The number of clauses of an input subformula is $|\phi_b| = m_b$, so that $|\phi| = m = m_0 + m_1 + m_{pub}$. Each clause is composed of the logical disjunction of literals, each of which is either v_i or $\neg v_i$ for $v_i \in V$. We compute satisfiability over $\phi \equiv \phi_0 \wedge \phi_1 \wedge \phi_{pub}$ where the last term appears only as appropriate. A model $\mathcal{M} \in \{0, 1\}^n$ is a function $\mathcal{M} : V \to \{0, 1\}$ mapping variables to truth values. When referring to the satisfiablity value s output by the ppSAT solver, we will often represent $s = 0$ (resp. $s = 1$) by UNSAT (resp. SAT), *i.e.*, UNSAT indicates that there is no satisfying model for ϕ, while SAT indicates the opposite. In practice s

will not necessarily be a bit, as we let $s = -1$ indicate that the satisfiability of ϕ is unknown. We notate access to the ith element in x by $x[i]$, and use e_i to represent the unit vector such that $e_i[i] = 1$ and $e_i[j] = 0$ for all $j \neq i$. Finally, the notation $\Pi(a \parallel b; c)$ denotes the execution of a two-party protocol Π with private inputs a and b and public input c.

2.3 Preliminaries

2.3.1 Overview of DPLL

To illustrate how the DPLL procedure works we walk through its execution on an example. Consider the following Boolean formula with four variables $\phi^{(0)} \equiv (v_1 \vee v_2) \wedge (v_2 \vee \neg v_3 \vee v_4) \wedge (\neg v_1 \vee \neg v_2) \wedge (\neg v_1 \vee \neg v_3 \vee \neg v_4) \wedge (v_1)$. The procedure iteratively builds a model for the formula by repeating a sequence of steps. As the input formula is in CNF its model must be a model for (*i.e.*, satisfy) each of its clauses.

We start with the empty model, $\mathcal{M} = \varnothing$. The first step, UNITSEARCH, searches for a unit clause: a clause consisting of a single literal. Assigning a truth value which makes that literal true is the only way to find a model for the formula. In our example the only unit clause is (v_1), the last clause. We add its satisfying assignment to the model: $\mathcal{M} = \{v_1 = 1\}$. This model is not only a model for that last clause, but for the first clause as well; it is a model for every clause where v_1 appears as a positive literal. This observation is the basis for the PROPAGATION step, which is run every time a new element is added to the model. The PROPAGATION step removes that element from the formula: all clauses where v_1 appears positively are removed, while in the remaining clauses we can safely remove the negated v_1 variable as $\neg v_1$ evaluates to false under $\mathcal{M}$, and false is a neutral element in disjunctions. In our particular example it means that we are left with the formula $\phi^{(1)} \equiv (v_2 \vee \neg v_3 \vee v_4) \wedge (\neg v_2) \wedge (\neg v_3 \vee \neg v_4)$.

The procedure now again executes the UNITSEARCH step, finding the unit clause $(\neg v_2)$. In general, after every UNITSEARCH step which finds a unit clause ℓ, the procedure executes the CHECK step, which checks that $\neg\ell$ is not also a unit clause. The CHECK step does not find any such conflict here, so we add $\neg v_2$ to the model: $\mathcal{M} = \{v_1 = 1, v_2 = 0\}$.

After the PROPAGATION step we are left with two clauses: $\phi^{(2)} \equiv (\neg v_3 \vee v_4) \wedge (\neg v_3 \vee \neg v_4)$, and running UNITSEARCH does not find a unit clause. When this occurs we pick a variable, guess its value, and then add that to the model. This is a DECISION step. For example, we can add $v_3 = 1$ to the model: $\mathcal{M} = \{v_1 = 1, v_2 = 0, v_3^d = 1\}$. Note that v_3 is annotated with d, indicating that it is a decision variable. Its value was guessed and not inferred. We then run the PROPAGATION step, which results in a new formula $\phi^{(3)} \equiv (v_4) \wedge (\neg v_4)$.

We again run the UNITSEARCH step on $\phi^{(3)}$: now (v_4) is a unit clause. However, running the CHECK step will find that $(\neg v_4)$ is also a unit clause. Therefore we need to backtrack. Backtracking is possible only when there is a decision literal in the

model. The BACKTRACK step retreats to the point in the procedure just before the last decision variable was added. Instead, we add its negation to the model and remove the d annotation. It is now inferred that the negated value has to be in the model, otherwise we would derive a contradiction.

Running the BACKTRACK step results in the model $\mathcal{M} = \{v_1 = 1, v_2 = 0, v_3 = 0\}$. We apply PROPAGATION on $\phi^{(2)}$ and the resulting formula is empty, *i.e.*, $\mathcal{M}$ is a model for all clauses in the original formula, which means that $\phi^{(0)}$ is satisfiable and $\mathcal{M}$ is its model. Note that v_4 can have any value. When the CHECK step finds a contradiction and no prior guesses can be undone, DPLL terminates and reports that the original input formula is unsatisfiable.

2.3.2 Cryptographic Preliminaries

Basic Primitives. We use standard techniques for 2PC, wherein P_0 and P_1 employ binary garbled circuits (GC) built from oblivious transfer (OT) and encryption primitives to jointly compute the sequence of functionalities which make up our ppSAT solver. Our design guarantees the order of these functionalities is fixed (up to the length of the execution), and that the access patterns over all intermediary values are data-oblivious, *i.e.*, either fixed or randomized independently of the private protocol inputs (up to their length). Those intermediary values are then stored at rest distributed between the P_b with information-theoretic security. At the end of the protocol, the final outputs are revealed to both parties.

Oblivious Stack. An oblivious stack data structure allows for conditional operations, which take a secret Boolean value that dictates whether the operation is actually performed or simulated through a dummy execution [218]. We will rely on the following operations, where $\perp$ and $\perp'$ are arbitrary but distinguishable special symbols:

- $\mathsf{ObStack} \leftarrow \mathsf{stack}()$: initialize an oblivious stack;
- $(\cdot) \leftarrow \mathsf{ObStack.CondPush}(b, x)$: (conditionally) push element x to the oblivious stack if $b = 1$, else skip.
- $(x) \leftarrow \mathsf{ObStack.CondPop}(b)$: (conditionally) pop and return the top element x if $b = 1$, else return $\perp$. If $b = 1$ and the stack is empty, fail and output $\perp'$.

2.4 Overview

As noted, a complete ppSAT solver can run for exponential time, which violates standard 2PC security definitions. In this section we formulate a definition for which meaningful security is practically achievable. We then give a high-level overview of our solver, before presenting it in detail in §2.6.

2.4.1 Formalizing ppSAT Security

A ppSAT solver is a two-party secure computation (2PC) protocol executed by P_0 and P_1. We operate in the *semi-honest* model, *i.e.*, we consider an adversarial P_b which attempts to learn private information about ϕ_{1-b}, but does not otherwise deviate from the protocol. We formalize security under the simulation paradigm [55]. The *view* of party P_b is an object containing all information known to it at the conclusion of a protocol Π: its private and the public inputs, every random coin flip it samples, every message it receives from P_{1-b}, every intermediary value it computes, and the output. We consider the protocol secure if there exists a *simulator* Sim_{1-b} such that no efficient algorithm $\mathcal{A}$ can distinguish between the view of P_b when interacting with Sim_{1-b} in an *ideal world* vs. with P_{1-b} in the *real world*. The simulator is given only (i) the private inputs of P_b, (ii) the public inputs, and (iii) the output of the protocol. Since the view of P_b in the ideal world cannot directly contain any information about ϕ_{1-b} by definition, this indistinguishability implies an adversarial P_b cannot learn more about it than what is implied by the output in the real world either.

However, standard computational security definitions for simulation are not blindly applicable to ppSAT solving as they require all parties run in probabilistic-polynomial time (PPT), while a complete solver may require exponential time with non-negligible probability. As such, to retain these definitions we choose to yield completeness for our solver and force polynomial runtimes, at the further cost of information leakage. In § 2.5 we rigorously formalize four different variants of a simulation-based security definition for ppSAT solving. Each requires some leakage, but how much depends on (i) whether the running time is leaked to allow early termination; and (ii) whether a model is output. We will primarily focus on a two-party-exact-time-revealing solver (2*p-etr*-solver), which reveals only (i) and so is the most efficient and concise formulation. We also discuss a further modification of the definition permitting the use of differential privacy to hide some of the information leakage in § 2.9.

2.4.2 Oblivious DPLL

Our ppSAT solver consists of a sequence of *giant steps* implementing an oblivious variant of the DPLL procedure. For concision we walk through solving without $\mathcal{M}$, and briefly discuss its addition at the end. At initialization the solver sets $\phi^{(0)} \leftarrow \phi$ and $a^{(0)} \leftarrow \perp$, where the latter is a "dummy value" encoded as a pair $(0^n, 0^n)$. As shown in Figure 2.1, the t-th giant step takes as input a formula $\phi^{(t-1)}$ and an assignment $a^{(t-1)}$. Each giant step either returns SAT/UNSAT or outputs an updated formula $\phi^{(t)}$ and a single assignment $a^{(t)}$ for the next giant step to consume.

A giant step sequentially executes oblivious variants of the five core small step algorithms of the DPLL procedure: UNITSEARCH, DECISION, CHECK, BACKTRACK, and PROPAGATION. First, UNITSEARCH and then DECISION output an assignment, $a^{(t)}_{\mathsf{unit}}$ and $a^{(t)}_{\mathsf{dec}}$ respectively. The UNITSEARCH routine scans $\phi^{(t-1)}$ and (when one exists) finds a unit clause. If such a clause is found then $a^{(t)}_{\mathsf{unit}}$ encodes its single

literal as an assignment, otherwise it encodes the dummy value $\perp$. The DECISION routine invokes a chosen heuristic (usually, but not necessarily, fixed for the entire execution) and obtains $a^{(t)}_{\mathsf{dec}}$ as a guess. If $a^{(t-1)} = a^{(t)}_{\mathsf{unit}} = \perp$ then $(\phi^{(t-1)}, a^{(t)}_{\mathsf{dec}})$ is pushed onto the oblivious stack. Otherwise a dummy push operation is performed. The multiplexer Mux_0 then selects a non-dummy assignment according to the priority $a^{(t-1)} > a^{(t)}_{\mathsf{unit}} > a^{(t)}_{\mathsf{dec}}$. Note that $a^{(t)}_{\mathsf{dec}} \neq \perp$ always. The selected assignment, denoted $a^{(t)}_{\mathsf{sel}}$, and $\phi^{(t-1)}$ are then taken by the CHECK routine as input.

This routine is the only possible point when the procedure can terminate and return SAT/UNSAT. For the moment we assume the procedure terminates immediately when possible, and discuss alternative behaviors later. A CNF formula conflicts with an assignment if it leads to an unsatisfiable clause. For input $\phi^{(t-1)}$ and assignment $a^{(t)}_{\mathsf{sel}}$ there are four possible cases for CHECK, PROPAGATION, and BACKTRACK:

1. The CHECK subroutine finds that $\phi^{(t-1)}$ is satisfied and returns SAT.

2. The CHECK subroutine finds that $\phi^{(t-1)}$ conflicts with $a^{(t)}_{\mathsf{sel}}$ and the stack is empty, and so returns UNSAT.

3. No conflict occurs, so CHECK passes $\phi^{(t-1)}$ and $a^{(t)}_{\mathsf{sel}}$ to the PROPAGATION and BACKTRACK routines. The PROPAGATION routine simplifies $\phi^{(t-1)}$ to ϕ_{prop} by eliminating both clauses which have been satisfied by $a^{(t)}_{\mathsf{sel}}$ and literals $\neg a^{(t)}_{\mathsf{sel}}$. The Mux_1 multiplexer will then set $\phi^{(t)} \leftarrow \phi_{\mathsf{prop}}$ and $a^{(t)} \leftarrow \perp$ as the output of the giant step. The BACKTRACK routine will execute a dummy pop operation.

4. A conflict occurs and the stack is not empty. The BACKTRACK routine pops a formula $\phi_{\mathsf{back}} = \phi^{(t')}$ for some $t' < t - 1$ and its associated a_{back} from the stack. It then sets $\overline{a}_{\mathsf{back}}$ by swapping ind^- and ind^+ from a_{back}. The Mux_1 multiplexer will then select $\phi^{(t)} \leftarrow \phi_{\mathsf{back}}$ and $a^{(t)} \leftarrow \overline{a}_{\mathsf{back}}$ as the output of the giant step. The output of the PROPAGATION routine will be ignored.

When the model $\mathcal{M}$ is desired as output this design is easily modified to continually update its state during the PROPAGATION routine, as well as save that state within and retrieve it from the stack during backtracking.

2.5 Security Definition

Our formal definition of a secure ppSAT solver has four variants depending on when it halts and whether a model is output. For brevity, we only explicitly give the most comprehensive.

Definition 2.5.1 (Two-Party Exact-Time-and-Model-Revealing ppSAT Solver)**.** *Let λ be a security parameter, ϕ_0, ϕ_1, ϕ_{pub} be Boolean propositional formulas over variables $v_1, \ldots, v_n$ such that $m = |\phi_0| + |\phi_1| + |\phi_{pub}|$, and $T(\lambda, n, m) = \tau_{\lambda,n,m}$ be a*

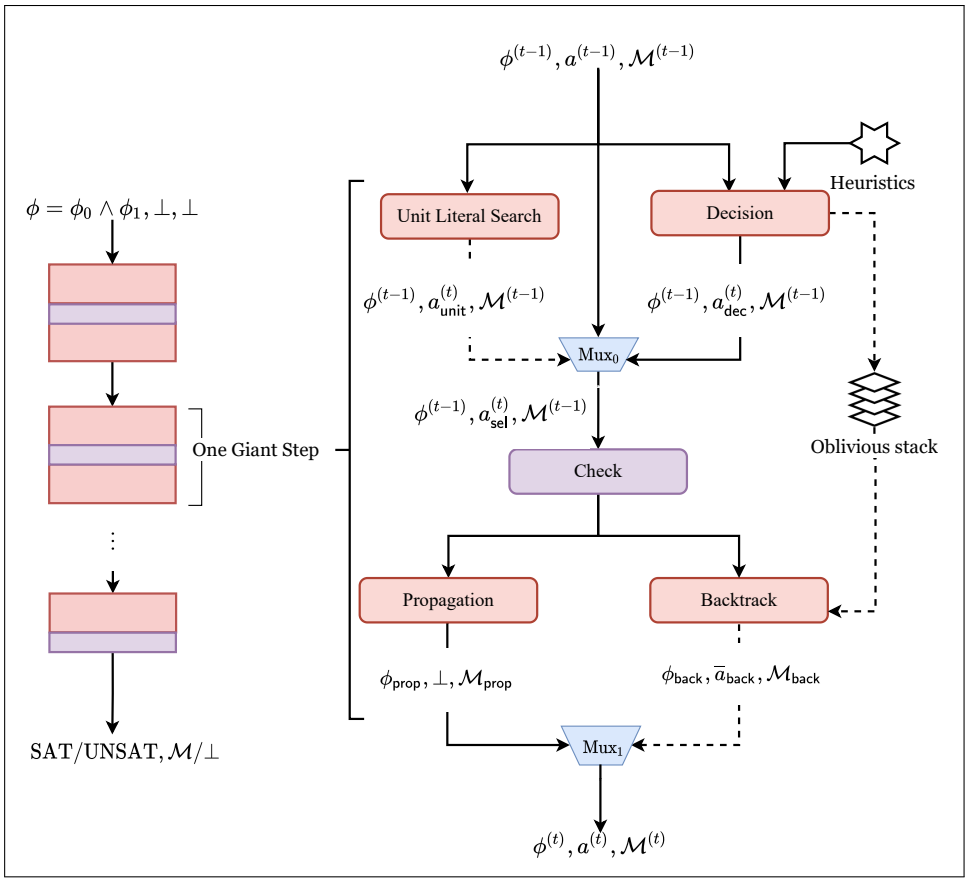


Figure 2.1: The structure of a giant step and its role in our ppSAT solver, demonstrating the high-level design given in §2.4. Dashed arrows indicate potential dummy return values.
polynomial. For $b \in \{0, 1\}$, *let*

$$v_b^{real} = \text{view}_b^{\Pi}(\phi_b,\ n,\ m,\ \phi_{pub}, \tau_{\lambda,n,m}, \\ P_{1-b}(\phi_{1-b},\ n,\ m,\ \phi_{pub},\ \tau_{\lambda,n,m})),$$

and

$$v_b^{ideal} = \text{view}_b^{\Pi}(\phi_b,\ n,\ m,\ \phi_{pub}, \tau_{\lambda,n,m}, \\ \text{Sim}_{1-b}(\phi_b,\ n,\ m,\ \phi_{pub},\ \tau_{\lambda,n,m},\ s,\ \mathcal{M},\ \tau)).$$

be the real and ideal views of P_b *after executing protocol*

$$(s,\ \mathcal{M}) \leftarrow \Pi(\phi_0 \parallel \phi_1;\ n,\ m,\ \phi_{pub},\ \tau_{\lambda,n,m})$$

terminating in $\tau \leq \tau_{\lambda,n,m}$ *steps. Then* Π *is a two-party exact-time-and-model-revealing privacy-preserving SAT solver (2p-etmr-solver) if*

1. $s = 1$ *iff* $\exists \mathcal{M}'.(\mathcal{M} = \mathcal{M}') \wedge \mathcal{M}' \models \phi$;
2. $s = 0$ *iff* $\nexists \mathcal{M}'.\mathcal{M}' \models \phi$; *and*
3. *there exists* Sim_{1-b} *such that for any probabilistic polynomial-time (PPT) decision*

algorithm $\mathcal{A}$*:*

$$|\Pr[\mathcal{A}(1^\lambda, v_b^{real}) = 1] - \Pr[\mathcal{A}(1^\lambda, v_b^{ideal}) = 1]| \leq \text{negl}(\lambda)$$

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

The three other variants of this definition are (i) a time-bound-and-model-revealing solver (*2p-tbmr*-solver), where the simulator is not given τ; (ii) an exact-time-revealing solver (*2p-etr*-solver), where $\mathcal{M}$ is removed from the definition; and (iii) a time-bound-revealing solver (*2p-tbr*-solver) which combines the changes from (i) and (ii). Intuitively, the time-bound-revealing definitions require the protocol to run for $\tau_{\lambda,n,m}$ steps always, while the exact-time-revealing definitions allow halting immediately upon resolution, and require aborting at $\tau_{\lambda,n,m}$ if necessary.

2.6 A ppSAT Solver

In this section we formalize the algorithm sketched in §2.4 as the basis for our ppSAT solver protocol. We begin by defining a pair of abstract data structures for a formula ϕ and its constituent clauses, and then describe an instantiation of these objects and their operations using bit-vectors. These operations are all data-oblivious and include most of the functionalities that will be computed using garbled circuits when the overall design is raised into a secure computation protocol. Finally, we describe how our ppSAT solver is structured as a data-oblivious sequence of these operations. For concision and clarity we sometimes describe the solver with data-dependent branching, but all conditions are simple checks of Boolean variables which can be merged into the operations they guard.

2.6.1 Data Structures for CNF formulas

Let β be an integer and $L = \{\ell_1, \ldots, \ell_{2n}\}$ be a set of literals. A clause is a subset $C \subseteq L$ for which $|C| \leq \beta$ and no two distinct literals reference the same underlying variable – implying that $\beta \leq n$ is the maximum clause length in ϕ. This parameter allows us to design different instantiations for when it is public knowledge that $\beta \approx n$, as opposed to when, *e.g.*, $\beta \ll n$. We require three operations over clauses:

- $C.\mathsf{unit}()$: returns a bit indicating whether $|C| = 1$.
- $C.\mathsf{contain}(\ell)$: returns a bit indicating whether $\ell \in C$.
- $C.\mathsf{remove}(a)$: updates C to $C \setminus \{a\}$ in place.

A formula ϕ is composed of a set of clauses $\{C_1, \cdots, C_m\}$, for which we also need two operations:

- $\phi.\mathsf{empty}()$: returns a bit indicating whether $\phi = \varnothing$.
- $\phi.\mathsf{remove}(C_j)$: updates ϕ to $\phi \setminus \{C_j\}$ in place.

Instantiating these abstract data structures for a given β requires both state encodings and supporting these methods.

Instantiating the ADS for $\beta \approx n$. Recall that e_i is the unit vector where $e_i[i] = 1$ and $e_i[j] = 0$ for all $j \neq i$. A literal ℓ is represented by the pairing of a unit vector and zero vector $(\mathsf{ind}^+, \mathsf{ind}^-)$. A positive variable v_i is encoded as $(e_i, 0^n)$, while its negation $\neg v_i$ is encoded as $(0^n, e_i)$. A dummy assignment $\perp$ is represented by $(0^n, 0^n)$. The representation of variables cooperates with the encoding of clauses (see next paragraph) so that operations over them can be implemented using only linear scans.

A clause $C_j \in \phi$ is represented by an integer n_L and the pair of vectors (P_j, N_j) such that

$$(P_j[i], N_j[i]) = \begin{cases} (1,0) & \text{if } v_i \text{ appears in } C_j \\ (0,1) & \text{if } \neg v_i \text{ appears in } C_j \\ (0,0) & \text{o.w.} \end{cases}$$

The integer n_L is used to track the number of literals in the clause. The implementation of the three clausal operations are given in Algorithm 1. Determining whether C_j is a unit clause can be implemented by checking whether $n_L = 1$. To implement contain we use that the structure of the clause and literal vectors provides that $\ell = v \in C_j$ iff $\bigvee_{i=1}^{n} (P_j[i] \wedge \mathsf{ind}^+[i]) = 1$ and similarly for $\ell = \neg v$ using N_j and ind^-. To remove a literal v_i (resp. $\neg v_i$) from C_j due to an assignment requires setting $P_j[i] = 0$ (resp. $N_j[i] = 0$) and deducting from n_L. Given the indicating assignment a, the former is the same as updating $P_j[i] \leftarrow P_j[i] \wedge (P_j[i] \oplus \mathsf{ind}^+[i])$ for each $i \in [n]$, and similarly over N_j and ind^-.

A formula ϕ is encoded as matrices (P, N) where the jth column of P is P_j and of N is N_j, as well as a vector $\mathsf{isAlive} \in \{0, 1\}^m$ whose jth entry indicates whether C_j has been removed from the formula. The $\phi.\mathsf{remove}(C)$ functionality can be implemented by setting $\mathsf{isAlive}[j] = 0$ during a linear scan, while $\phi.\mathsf{empty}()$ by checking whether $\mathsf{isAlive} = 0^n$. We omit their formal descriptions due to their simplicity.

Alternate Approaches. I primarily focus on the above approach, as it is a generic solution applicable to every possible instance of ppSAT. We also consider an alternate instantiation for when the maximum number of literals in a clause is much smaller than n (*i.e.*, $\beta \ll n$) in § 2.6.3.

Operations on clauses can in theory be instantiated via RAM-model secure computation [103], which requires running an ORAM client algorithm in MPC. This could potentially reduce the asymptotic cost of the ADS operations to $O(\log^2 n)$ from $O(n)$.[3]

[3]Note that although the best ORAM [19] can incur only a $O(\log n)$ overhead, that requires an

Algorithm 1: Clausal Algorithms when $\beta \approx n$

Function C_j.unit():
  return $(n_L = 1)$
Function C_j.contain($\ell = (ind^+, ind^-)$):
  $b \leftarrow 0$
  for $i \leftarrow 1$ **to** n **do**
    $b \leftarrow b \vee (P_j[i] \wedge \mathsf{ind}^+[i]) \vee (N_j[i] \wedge \mathsf{ind}^-[i])$
  return b
Function C_j.remove($a = (ind^+, ind^-)$):
  if C_j.contain *(a)* **then**
    $n_L \leftarrow n_L - 1$
  for $i \in [n]$ **do**
    $P_j[i] \leftarrow P_j[i] \wedge (P_j[i] \oplus \mathsf{ind}^+[i])$
    $N_j[i] \leftarrow N_j[i] \wedge (N_j[i] \oplus \mathsf{ind}^-[i])$

We can empirically compare our bit-vector based solution with the most practically efficient ORAM secure computation [76]. To read or write a bit from a n-bit vector the linear scan circuit contains exactly $2n - 1$ AND gates. As a result, the crossover point (conservatively) occurs when $n \approx 2^{17}$, where our circuit takes about 0.13s and the ORAM-based solution takes about 0.2s.

2.6.2 Data-Oblivious ppSAT Solving

Algorithm 2 formally presents the algorithmic structure of our ppSAT solver. We only provide a full description of our *2p-etr*-solver for brevity, which can be extended to support $\mathcal{M}$ as described in §2.4, and raised into a secure 2PC protocol using the standard techniques referenced in §2.3. Finally, we abuse notation by considering τ and $\tau_{\lambda,n,m}$ to track and upper bound respectively the number of giant steps. Their true values are some constant factor (representing the number of computational steps within a giant step) of how we use them algorithmically.

Every giant step (Lines 13-24 and 3-12 across two loop iterations) starts with a formula ϕ and an assignment a, and either passes a new formula and assignment to the next giant step or terminates. The flag b_{conflict} indicates a conflict; when one (and therefore backtracking) occurred in the prior giant step then $b_{\mathsf{conflict}} = 1$.

The solver first executes UNITSEARCH, sets b_{unit} to indicate its success, and if successful returns the unit literal as an assignment a_{unit}. Then the solver invokes the heuristic in DECISION and receives a branching assignment a_{dec}. If $b_{\mathsf{conflict}} = 1$ the negation of a_{back} and ϕ_{dec} from the previous giant step are taken as the input

underlying data block of at least $\log n$ bits. In our case, each data block is a single bit and thus the best available requires $O(\log^2 n)$. We are not aware of any ORAM designed specifically for bit accesses, which may be a promising line of future work to increase its efficacy in this and other secure computations over bit-vectors.

of CHECK (Lines 21-23 and 4). Otherwise, either the output of UNITSEARCH or the output of DECISION will be used depending on b_{unit} (Lines 16-19 and 4).

The CHECK routine resolves the application of assignment a to ϕ. There are three possibilities, each corresponding to a value of σ:

1. $\sigma = 0$: ϕ is satisfied after applying a. Then CHECK terminates the procedure and outputs SAT (Line 6);

2. $\sigma = 1$: ϕ contains a unit clause with the negation of a (Line 7). The solver then pops the top element (if any) off the stack (Line 8). If the stack is empty the solver will terminate and output UNSAT (Lines 9-10). Otherwise, b_{conflict} is set to 1 and the solver backtracks, ultimately recovering the assignment a_{back} and formula ϕ_{back} for the next giant step. The result of PROPAGATION will be ignored; or

3. $\sigma = 2$: the formula is neither SAT nor in conflict after applying a. The PROPAGATION routine simplifies ϕ to ϕ_{prop} using a and passes the simplified formula to the next giant step (Line 12).

The behavior of BACKTRACK is directly integrated into Algorithm 2, while DECISION is the focus of §2.6.4. Next, we describe the remaining UNITSEARCH, CHECK, and PROPAGATION routes in detail.

UNITSEARCH (Algorithm 3): The UNITSEARCH routine finds a unit clause in the current formula ϕ when one exists, and outputs a bit b and an assignment a. If no unit clause exists in ϕ then $b = 0$, else $b = 1$ and a corresponds to its single literal. To find the encoding of that literal to set a, we need to locate the clause C_j such that $C_j.\mathsf{unit}() = 1$. We achieve this through a linear scan of all the clauses, setting $b \leftarrow 1$ and $a \leftarrow C_j$ once find a suitable output (Lines 2-5).[4]

CHECK (Algorithm 4): The CHECK routine determines whether formula ϕ is SAT, and if not whether the assignment a causes a conflict or whether the resultant ϕ remains viable but unproven. It returns $\sigma \in \{0, 1, 2\}$. The three cases are (i) $\sigma = 0$, indicating that ϕ is satisfied; (ii) $\sigma = 1$, indicating that ϕ conflicts with a; and (iii) $\sigma = 2$, otherwise. The routine uses Boolean variables b_0 and b_1 to track if ϕ is SAT or conflicts with a respectively. The $\phi.\mathsf{empty}$ operation resolves b_0 (Line 1). A clause conflicts with a if it only contains $\neg a$. The routine scans over all clauses, and sets $b_1 \leftarrow 1$ if any is unit and conflicts with the assignment (Lines 3-5). If neither b_0 nor b_1 is set to 1, the routine returns 2 to indicate that ϕ is still viable under a.

PROPAGATION (Algorithm 5): The PROPAGATION routine simplifies ϕ by eliminating clauses containing a literal ℓ with identical indicators to the assignment a. Additionally, $\neg\ell$ is removed from any clause containing it. So, during propagation there

[4] I slightly abuse notation here for readability, as not all instantiations may have unit clause representations which can be used immediately as an assignment. The procedure can be generalized in practice by extending $C_j.\mathsf{unit}()$ to return the literal when it is true.

Algorithm 2: *2p-etr* ppSAT Solver

Input: ϕ, n, m, $\tau_{\lambda,n,m}$
Output: SAT/UNSAT

```
1  ObStack ← stack(); τ ← 0; a ← ⊥; b_conflict ← 0;
2  while τ ≤ τ_{λ,n,m} do
3      if τ ≠ 0 then
4          σ ← Check(φ, a);
5          if σ = 0 then
6              return SAT
7          b_conflict ← (σ = 1);
8          e ← ObStack.pop(b_conflict);
9          if e ← ⊥′ then
10             return UNSAT
11         a_back, φ_back ← e;
12         φ_prop ← Propagation(φ, a);
13     (b_unit, a_unit) ← UnitSearch(φ_prop) ;
14     a_dec ← Decision(φ_prop);
15     ObStack.CondPush(¬b_unit ∧¬b_conflict,(a_dec, φ_prop));
16     if b_unit = 0 then
17         a ← a_dec;
18     else
19         a ← a_unit;
20     φ ← φ_prop;
21     if b_conflict = 1 then
22         a ← ¬a_back;
23         φ ← φ_back;
24     τ = τ + 1;
```

Algorithm 3: Unit Search

Input: ϕ
Output: $b \in \{0, 1\}$, $a = (\mathsf{ind}^+, \mathsf{ind}^-)$

```
1  a ← ⊥; b ← 0 ;
2  for j ← 1 to m do
3      u_j ← C_j.unit();
4      if u_j = 1 then
5          a ← C_j; b ← 1;
6  return b, a
```

are three types of clauses $C \in \phi$, those for which: (i) $\neg a = \ell \in C$, in which case $C.\mathsf{remove}(\neg a)$ is executed (Line 7); (ii) $a = \ell \in C$, so C is satisfied and $\phi.\mathsf{remove}(C)$ is therefore invoked (Line 5); or (iii) clause C contains neither $\ell = a$ nor $\ell = \neg a$, and so is left unchanged.

Algorithm 4: Check

Input: ϕ, $\ell = (\mathsf{ind}^+, \mathsf{ind}^-)$
Output: $b \in \{0, 1, 2\}$

```
b_0 ← φ.empty();
b_1 ← 0;
for j ← 1 to m do
    if C_j.unit() ∧ C_j.contain(¬ℓ) then
        b_1 ← 1;
if b_0 = 1 then
    return 0;
else if b_1 = 1 then
    return 1;
else
    return 2;
```

Algorithm 5: Propagation

Input: ϕ, $a = (\mathsf{ind}^+, \mathsf{ind}^-)$
Output: ϕ'

```
for j ← 1 to m do
    b_0 ← C_j.contain(a);
    b_1 ← C_j.contain(¬a);
    if b_0 = 1 then
        φ.remove(C_j);
    if b_1 = 1 then
        C_j.remove(¬a);
return φ
```

2.6.3 Instantiating the ADS for $\beta \ll n$

Recall that $\beta \leq n$ is the maximum clause length in an instance ϕ. When it is publicly known that the maximum number of literals appearing in any given clause is small, an alternative approach to binary clausal vectors is to use signed integers to represent both assignments and literals; *i.e.*, $\neg v_j$ and the assignment $v_j = 0$ are each represented by $-j$. A clause $C \in \phi$ is encoded by two vectors, $\mathsf{literals} \in \mathbb{Z}^\beta$ and $\mathsf{active} \in \{0, 1\}^\beta$. The $\mathsf{literals}$ vector is composed of β signed integers which encode the literals and their sign, padded out with zeros as necessary. The active vector encodes whether the i-th literal has been removed from C. As an example, at initialization $(v_1 \vee v_3 \vee \neg v_5)$ with $\beta = 4$ will be encoded as $\mathsf{literals} = [1, 3, -5, 0]$ and $\mathsf{active} = [1, 1, 1, 0]$. A literal or assignment is negated by flipping the sign.

Determining if a clause is unit can be implemented by scanning active and checking if it has a unique non-zero entry. To check membership of an $\ell = j \in C$ the algorithm checks if there exists an i such that $\mathsf{active}[i] = 1$ and $\mathsf{literals}[i] = j$, while removing it

is accomplished by setting $\mathsf{active}[i] = 0$ when $\mathsf{literals}[i] = j$.

Algorithm 6: Clausal Algorithms when $\beta \ll n$

```
1  Function C_j.unit():
2      b_1 ← 0; b_2 ← 0
3      for i ← 1 to β do
4          b_2 ← (active[i] ∧ b_1) ∨ b_2
5          b_1 ← active[i] ∨ b_1
6      return b_1 ∧ ¬b_2
7  Function C_j.contain(ℓ ∈ ℤ):
8      b ← 0
9      for i ← 1 to β do
10         b ← b ∨ (active[i] ∧ literals[i] = ℓ)
11     return b
12 Function C_j.remove(a ∈ ℤ):
13     for i ← 1 to β do
14         b ← (literals[i] = a)
15         active[i] = ¬b ∧ active[i]
```

At rest this instantiation provides more efficient clausal representations so long as $\beta \cdot k < n$ where $k > \log n$ is the bit-length of the integer encoding. In practice, our evaluation of this approach suffered in comparison to that for $\beta \approx n$ due to the reduced efficacy of an implementation optimization for the oblivious stack. Specifically, to reduce the cost of the stack operations our code only stores the current set of assignments (including isAlive) within it, and then reconstructs the formula during a backtrack. The signed integer encoding requires spending a few hundred gates to compare every assignment to every literal, of which there are $\beta \cdot nm$ such comparisons. Resolving this gap is a potential optimization path.

2.6.4 ppSAT Decision Heuristics

The final component of our solver is the DECISION routine, which is not a single functionality but rather a family of possible procedures for guessing a variable assignment. Designing such techniques is historically a very rich area of SAT research [44, 170, 90], though many of these constructions are not naturally implementable through oblivious computation and 2PC primitives. For this initial work we focus on three heuristics: (i) the deterministic dynamic largest independent sum (DLIS) heuristic, where we choose the most common literal as the assignment, (ii) the randomized (RAND) heuristic where we make a uniform choice over both variable and assignment, and (iii) a weighted randomized heuristic (Weighted-RAND), where the choice of literal is weighted by frequency. These are all relatively simple but still useful, and implementing their many variants along with more complex heuristics is a promising avenue for future work which we discuss at the end of the section.

Algorithm 7: DLIS Heuristic

Input: $L = \{\ell_1, \ldots, \ell_{2n}\}$, $C = \{C_1, \ldots, C_m\}$
Output: $d \in [1..n] \times \{0, 1\}$

$max \leftarrow 0$; $d \leftarrow (\bot, \bot)$;
for $i \leftarrow 1$ **to** n **do**
  $s_P \leftarrow 0$, $s_N \leftarrow 0$;
  for $j \leftarrow 1$ **to** m **do**
    $s_P = s_P + C_j.\mathsf{contain}(\ell_i)$;
    $s_N = s_N + C_j.\mathsf{contain}(\neg\ell_i)$;
  if $s_N \geq max$ **then**
    $d \leftarrow (i, 0)$; $max \leftarrow s_N$;
  if $s_P \geq max$ **then**
    $d \leftarrow (i, 1)$; $max \leftarrow s_P$;
return d;

Note that for brevity we do not provide the full DECISION routine. Each heuristic either returns the assignment a itself or a tuple $d = (i, b)$ encoding that the DECISION routine should construct an assignment by setting $v_i = b$ in the form needed for the given ADS instantiation.

DLIS. The DLIS heuristic selects the most commonly appearing literal and returns the assignment that makes it true. Our formulation of it as Algorithm 7 undertakes a linear scan over every $C_j \in \phi$ for every $v_i \in V$. The heuristic calculates the frequency of v_i and $\neg v_i$ as $\sum_{j=1}^{m} C_j.\mathsf{contain}(v_i)$ and $\sum_{j=1}^{m} C_j.\mathsf{contain}(\neg v_i)$ respectively. It then determines whether either of v_i or $\neg v_i$ is the most frequent literal seen so far, and if so sets d as necessary to encode it. After iterating over all the variables d encodes the most frequent literal and is returned.

RAND. Let the binary vector $\hat{U} \in \{0, 1\}^n$ indicate whether the i-th variable in V has been assigned. The RAND heuristic guesses a variable assignment by uniformly selecting a random unassigned variable and setting it to a bit also chosen uniformly at random. Since $\hat{U}$ is derived from ϕ as well as prior assignments, the computation in this procedure must be data-oblivious and amenable to efficient realization by secure computation primitives. At the core of our design is Algorithm 8, which with probability $p_1 \geq 1/2$ obliviously selects a secret $r \in [Q]$ for private $Q \in \mathbb{N}$.

We assume that Q is encoded as a binary string of length $l \in \mathbb{N}$. This is the natural encoding for binary garbled circuits, but may not be for other 2PC primitives. We let $h \in \mathbb{N}$ be one greater than the index (from zero) of the most significant non-zero bit in Q, *e.g.*, $h = 4$ for $l = 6$ and $Q = 9 = 001001$. The construction builds a multiplexer which maps an integer $x' \in [2^l]$ to $x \in [2^h]$ by keeping the lower h bits unchanged while setting the upper $l - h$ bits to zero. It then applies this multiplexer to a random binary string $r' \in \{0, 1\}^l$, generating $r \in \{0, 1\}^l$ to be interpreted as an integer upon return. To sample r' within 2PC we define it as $r' = r'_0 \oplus r'_1$, where r'_b is privately sampled by P_b.

Algorithm 8: Random Value Sampler internal (RVSi)

Input: $Q \in \mathbb{N}$, $r'_0 \in \{0,1\}^l$, $r'_1 \in \{0,1\}^l$
Output: $r \in \{0,1\}^l$

```
b ← 0; Q' ← 0^l;
for i ∈ {l − 1, . . . , 0} do
    if Q[i] ≠ 0 ∨ b = 1 then
        Q'[i] ← 1; b ← 1;
r' ← r'_0 ⊕ r'_1;
r ← Σ_{i∈[l]} r'[i] · (Q'[i] · 2^i);
return r;
```

Notice that $r < Q$ with some probability $p_1 \geq 1/2$, as it is guaranteed when $r'[l-h] = 0$. The parties can repeat Algorithm 8 for sufficient $\kappa \in \mathbb{N}$ so that the probability every returned r lies outside $[Q]$, or $p_2 \leq 2^{-\kappa}$, is suitably negligible. A reasonably small constant such as $\kappa = 32$ suffices. A wrapper function RVS(), which we otherwise omit, can make these repeated invocations of RVSi() and then undertake a linear scan over the outputs to finally return, *e.g.*, the last which lies within the desired range.

Our construction to uniformly select an unassigned variable is Algorithm 9. It linearly scans $\hat{U}$ and counts the number of unassigned variables, before invoking Algorithm 8 to get a random index k. It then selects the k-th unassigned variable through another scan of $\hat{U}$, and assigns to it a random $b = b_0 \oplus b_1$, where b_b is sampled and provided by P_b.

Weighted-RAND. Let $L = \{\ell_1, \ldots, \ell_{2n}\}$ be a set of literals and $\mathcal{D}$ a distribution over L expressed as set of positive integers $W = \{w_1, \ldots, w_{2n}\}$ such that for all $i \in [1..2n]$

$$\Pr(\ell_i) = \frac{w_i}{S_W}, \text{ for } S_W = \sum_{j=1}^{2n} w_j.$$

For our decision heuristic w_i will be the frequency count of the i-th literal. We design an oblivious algorithm that randomly samples an element of L according to W in Algorithm 10. First we compute $c = w_1 + \cdots + w_{2n}$. Then, using Algorithm 8 an integer k is sampled from $[c]$. Let $F(\ell_i) = \sum_{j=1}^{i} w_j$. The algorithm finds ℓ_i such that $F(\ell_{i-1}) < k \leq F(\ell_i)$ through a linear scan, which is then returned as the assignment. The intuition behind the correctness of the algorithm is that for any random variable Y, a sample value can be generated by drawing a random $r \in [0, 1]$ and then finding its preimage on the cdf of Y.

CDCL. Perhaps the most important development in SAT solving since DPLL itself is conflict-driven clause learning (CDCL) [201, 178, 90]. The essential idea of CDCL is that whenever a conflict is found it is possible to resolve a self-contained subset of the assignments which triggered the conflict. For example, the CDCL learning procedure might learn that $x_1 = 1$, $x_{12} = 0$, $x_{27} = 1$ is impossible for any model. So, by creating

Algorithm 9: Uniform Random Selection

Input: $\hat{U} \in \{0, 1\}^n$, r'_0, $r'_1 \in \{0, 1\}^l$, b_0, $b_1 \in \{0, 1\}$
Output: $d \in [1..n] \times \{0, 1\}$

$c \leftarrow 0$; $d \leftarrow (\bot, \bot)$;
for $i \leftarrow 1$ **to** n **do**
  $c \leftarrow c + \hat{U}[i]$;
$k \leftarrow$RVS(c, r'_0, r'_1);
for $i \leftarrow 1$ **to** n **do**
  $k \leftarrow k - \hat{U}[i]$;
  if $k = 0$ **then**
    $d \leftarrow (i, b_0 \oplus b_1)$;
return d;

Algorithm 10: Weighted Random Selection

Input: $W \in \mathbb{Z}^{2n}_{\geq 0}$, $L = \{\ell_1, \ldots, \ell_{2n}\}$, r'_0, $r'_1 \in \{0, 1\}^l$
Output: $a = (\mathsf{ind}^+, \mathsf{ind}^-)$

$c \leftarrow 0$;
for $i \leftarrow 1$ **to** $2n$ **do**
  $c = c + w_i$;
$k \leftarrow$RVS(c, r'_0, r'_1);
for $i \leftarrow 1$ **to** $2n$ **do**
  if $0 < k < w_i$ **then**
    $a \leftarrow \ell_i$;
  $k \leftarrow k - w_i$;
return a;

a new clause made out of their negation, *i.e.*, $(\neg x_1 \vee x_{12} \vee \neg x_{27})$ and adding it to ϕ, any branches of the search tree which would try that impossible combination are cut off immediately. This dramatically reduces the size of the space which the solver explores while retaining soundness and completeness. CDCL is particularly essential for efficiently resolving UNSAT instances, which require establishing a universal (*all models do not satisfy*), rather than existential (*there exists a satisfying model*), proposition over the search tree.

For ppSAT solving to reach its potential will almost certainly require supporting CDCL. However, this is a challenging task. As an immediate issue, CDCL only learns clauses when backtracking happens. Continuing to hide those occurrences would require a deterministic schedule for adding clauses. Though a simple approach is to add one clause per giant step while padding out with tautologies, every increase to the size of the formula makes every ensuing giant step more expensive. An alternative approach might be to add clauses less frequently, perhaps keeping the largest learned in the interim and discarding the rest. Exploring this frontier will be critical to use

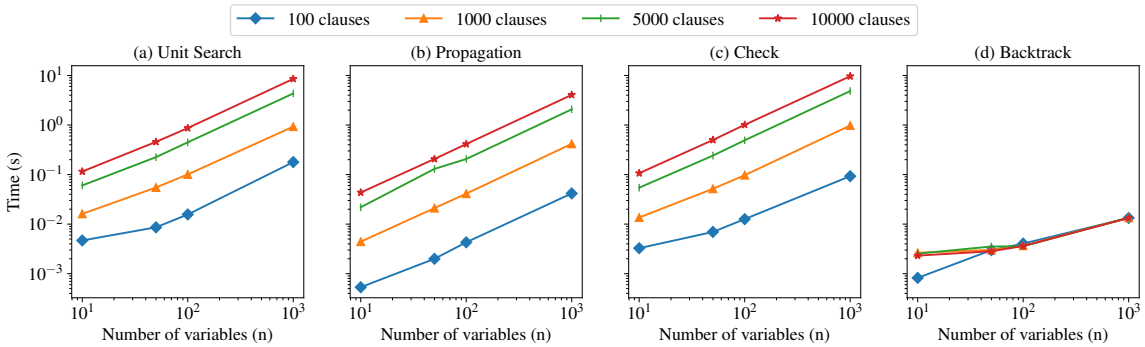


Figure 2.2: Subroutine time for our ADS instantiation when $\beta \approx n$. The runtimes of all four subroutines show linear growth as the number of variables rises. The runtimes of UNITSEARCH, PROPAGATION and CHECK also increase with the number of clauses. Due to our optimized implementation, the runtime of DECISION is independent of the number clauses.

of CDCL within a ppSAT solver.

Additionally, the CDCL learning process itself would need to be made oblivious. Usually it is understood as the building of an implication graph for which a suitable (and not necessarily minimal) cut produces the assignments to negate. Though this process may be rendered as a sequence of resolution operations potentially amenable to oblivious formulation [44], doing so without undue overhead may require care.

2.7 Complexity

The overall circuit complexity of our protocol is $O(S \times C)$, where S is the total number of giant steps and C is the cost of each one. The round complexity of our protocol is $O(S)$, as every giant step is a constant-round 2PC of a single circuit. The value of S depends on $\tau_{\lambda,n,m}$, on the number of steps necessary to resolve ϕ, and on whether an exact-time, time-bound, or noisy-time (§ 2.9) solver is used.

Our value for C is $O(mn + n \log n)$, though it may vary based on the details of the ADS instantiation and the heuristics. The circuit complexities of UNITSEARCH, CHECK, and PROPAGATION are each $O(mn)$, while that of BACKTRACK is $O(mn + n \log n)$ with the logarithmic term arising from the oblivious stack [218]. UNITSEARCH, CHECK, and PROPAGATION each require $O(1)$ linear scans over the m clauses; during each scan the procedures apply the various ADS operations, each of which require $O(1)$ or $O(n)$ time in our $\beta \approx n$ instantiation. BACKTRACK consists of a pop operation from the oblivious stack of a (partial) model represented in $O(n)$ bits, taking $O(n \log n)$ time, followed by the application of that model to recover the formula state in time $O(mn)$. Finally, DECISION has complexity $O(H + n \log n)$, where H is the complexity of the chosen heuristic and $n \log n$ is the complexity of the oblivious stack push operation. The DLIS and RAND heuristics have $H = O(mn)$. The complexity of Weighted-RAND also requires $H = O(mn)$ so long as the weight of a literal is its frequency in the formula. Other heuristics could have worse (or better)

asymptotic cost.

As discussed in §2.6.1, a RAM-based secure computation solution would reduce the cost of accessing n bits from $O(n)$ to $O(\log^2 n)$. This would, *e.g.*, reduce the cost of ADS operations where we must touch a given literal at every clause, such as $C.\mathsf{contain}(\ell)$, from $O(mn)$ to $O(m \log^2 n)$. With sufficient refining of the data structures, heuristics, and composition of the subroutines these improvements may improve the asymptotic runtime of C in total. However, as noted we can project the protocol efficiency would not be concretely superior at present until at least $n \geq 2^{17}$ [76], with the true crossover depending in part on network conditions as ORAM also requires logarithmic, instead of constant, rounds. We leave the potential of ORAM to future work once algorithm enhancements, like CDCL, make it relevant.

2.7.1 Obliviousness and Security

At its core our solver raises DPLL into a secure computation using oblivious algorithms and supporting data structures. For the general purpose ADS instantiation (when $\beta \approx n$), our fundamental design choices were to represent both literals and clauses as binary vectors and to manage the decision tree with an oblivious stack to permit backtracking [218]. Using these data structures, the standard DPLL subroutines (such as PROPAGATION or BACKTRACK) can be implemented with linear scans over vectors of fixed public size, oblivious multiplexing of vectors, and push and pop of vectors to and from the oblivious stack. Hence, each of them is data-oblivious.

These individual subroutines must be combined in a manner that maintains data-obliviousness, which is why we adapt DPLL to enforce so-called giant steps (cf. § 2.6.2). A giant step executes these several small steps in a deterministic order, which are then consolidated through multiplexing. The use of giant steps may result in redundant and ultimately discarded operations, but are necessary to hide the (usually data-dependent) DPLL step being applied. Obliviousness must also be enforced for the decision heuristics. Hence our careful choice of three standard DPLL heuristics amenable to formulation using the same linear scanning and multiplexing techniques: DLIS, RAND, and Weighted-RAND (cf. § 2.6.4).

A security argument follows from the data-oblivious nature of our solver, the security of two-party computation, and standard composition results [55]. A simulator Sim_{1-b} invokes the simulators for the fixed sequence of circuits, and halts according to τ or $\tau_{\lambda,n,m}$ by injecting s and (optionally) $\mathcal{M}$ into the final output. We refer to [156, 154] for discussion of the proof techniques underlying this sketched argument.

2.8 Evaluation

Testbed. We implemented our solver using the semi-honest 2PC library of the EMP-toolkit [217]. For managing the oblivious stack we adopted an existing reimplementation of the circuits of Zahur *et al.* [232]. All evaluations were run on a machine with

8GB of RAM and an Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz * 6 processor, with network bandwidth up to 10 Gbps.

Experiment Design. We first measured (§2.8.1) the performance of a single giant step – in terms of both the number of gates and the runtime of each subroutine – over formulas of various sizes. These evaluations verified our analysis of the asymptotic complexity of ppSAT and its most critical bottlenecks. We then benchmarked (§2.8.2) our ppSAT solver by testing what proportion of instances it was able to solve (up to a timeout). Finally, we compared (§2.8.3) the performance of our ppSAT solver against two plaintext solvers: our oblivious ppSAT algorithm executed without cryptographic primitives or communication, and the state-of-the-art Kissat solver [25].

Instance Generation. For measuring the cost of a single giant step the instances were generated randomly. Due to the oblivious nature of the algorithm this has no bearing on the evaluation. For the full evaluation benchmarks, we projected the cost of our solver on small haplotype satifiability instances drawn from the dataset of [168] which was used in the original SHIPs papers [166, 167]. Our instances have parameters of either (i) $|G| \in [1..8]$ and $r = 2|G|$, intended to evaluate the effect of instance size; or (ii) $|G| = 3$ and $r \in [3..6]$, so as to evaluate the effect of instance hardness and (un)satisfiability. We list the resultant formula sizes in terms of n and m in Table 2.1. Although these databases are smaller than modern HIPP benchmarks, important medical research that motivated early work in computational haplotype inference occurred over datasets where $|G| \approx 10$ [153, 191].

$\|G\|$	#var × #clause (≈)		$\|G\|$	#var × #clause (≈)	
1	60×170		2	150×700	
3	$r = 3$	150×750	3	$r = 5$	200×1200
	$r = 4$	180×1000		$r = 6$	250×1400
4	350×2600		5	400×4000	
6	600×6000		7	800×8000	
8	900×10000		—	—	

Table 2.1: The size of the formulas for our benchmarks.

2.8.1 Micro Benchmarks for Single Giant Steps

We measured the time consumption and number of gates of UNITSEARCH, DECISION, CHECK, and PROPAGATION for each combination of when $n \in \{10, 50, 100, 1000\}$ and when $m \in \{100, 1000, 5000, 10000\}$, which covers the typical size of instances in older benchmarks such as [125].

Unit Search, Propagation, and Check: The first three rows of Table 2.2 show the number of gates in UNITSEARCH, PROPAGATION and CHECK for formulas of different sizes. The number of gates increases linearly with both n and m, which is consistent with our asymptotic analysis.

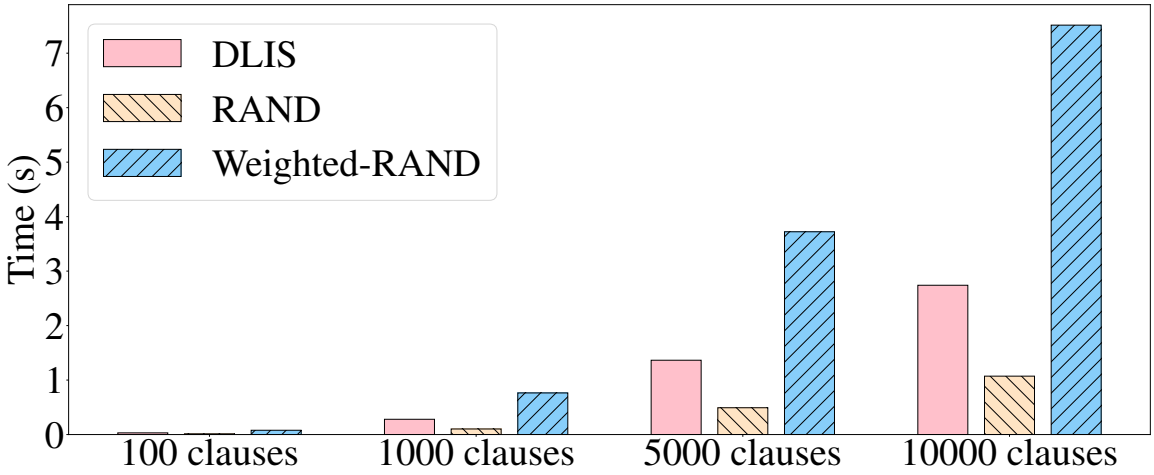


Figure 2.3: Time for heuristics when $n = 100$. The runtime of each heuristic grows with an increase in the number of clauses.

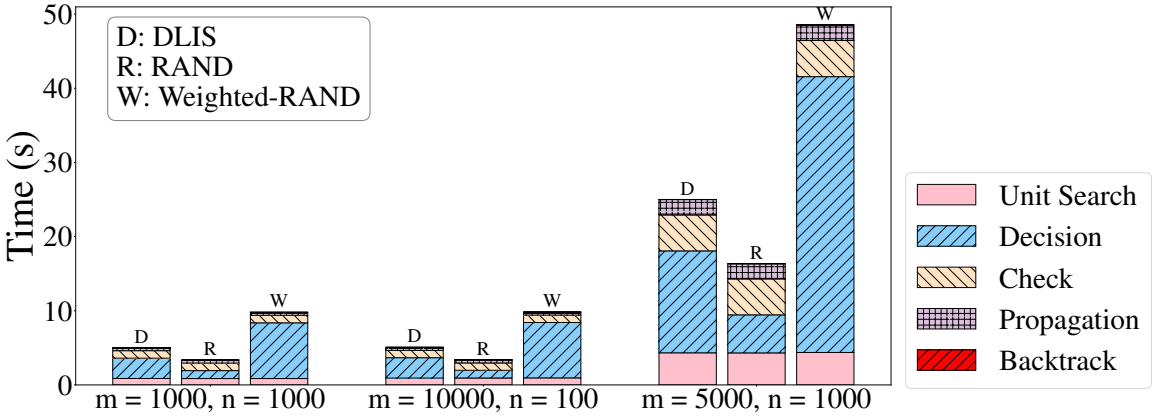


Figure 2.4: Time for one giant step varying n, m, and the heuristic. The DECISION routine dominates the performance of each giant step. The runtime of BACKTRACK is almost negligible in comparison to the other subroutines, and therefore is not visible in the figure.

Figures 2.2(a-c) graph the execution time of UNITSEARCH, CHECK and PROPAGATION. The observed time appears linear in the number of variables and clauses, though the growth in the latter decreases, likely due to amortization of general overhead. This is as expected, since UNITSEARCH, CHECK and PROPAGATION all run in $O(mn)$ time. The evaluation shows that even for larger instances with $n \approx 1000$ and $m \approx 10000$ these routines cost less than 10 seconds each.

Backtrack: Figure 2.2(d) shows the time per BACKTRACK execution, which reflects the fourth row of Table 2.2 in showing that the number of gates increases linearly with n, but is independent of m. Due to an optimization in our implementation the cost of backtracking only depends on the number of variables: we store just the current model (including isAlive) in the oblivious stack, and then recover the formula state within the next multiplexer. As models are just $O(n)$ bits this is independent of the number of clauses. Due to this efficient oblivious stack design the BACKTRACK time for an instance where $n = 1000$ and $m = 10000$ takes only ≈ 0.01s.

#var × #clause		$100 \times 5K$	$100 \times 10K$	$1K \times 10K$
Unit Search		10	20	200
Propagation		6	12	120
Check		8	16	160
Backtrack		0.02	0.02	0.22
Decision	D	30	60	600
	R	12	24	240
	W	88	175	1740

Table 2.2: The approximate number of gates for each subroutine. The units are in **millions**. D, R, and W refer to the DLIS, RAND, and Weighted-RAND heuristics respectively.

Decision: The last row of Table 2.2 presents the number of gates for DECISION, when using our different heuristics from §2.6.4. The figure shows DECISION is the most expensive component of ppSAT. While each heuristic linearly scales up in $O(mn)$ time, RAND takes the fewest concrete gates. Figure 2.3 compares the experimental runtimes when $n = 100$ and with various clause sizes. Again, the observed growth for each heuristic is as expected linear in the number of clauses. The Weighted-RAND heuristic is the most expensive at almost twice the cost of DLIS – likely as it combines RVS with frequency counting. The simpler RAND is cheapest at about only half the time of DLIS.

Giant Step: Figure 2.4 displays the observed time for a full giant step across a variety of choices for n, m, and heuristic. For instances of the same size the fraction that each component takes remains stable, as expected. For instances of size typical for old benchmarks ($n \approx 100, m \approx 10000$) the time cost is roughly 3s, 5s, and 10s with RAND, DLIS and Weighted-RAND respectively.

2.8.2 Solving Benchmarks

To evaluate the performance of our solver we first measured the total number of giant steps S for our instances. Although our solver implementation is complete, as cryptographic operations do not affect S to save time we gathered this data with all cryptographic operations removed. We then used the methodology of the micro benchmarks to get a timing C for a single 2PC giant step for those instances, from which the total runtime can be projected as $S \times C$. We also ran the complete ppSAT solver over an UNSAT formula of 1000 variables and 1000 clauses, which took 3019.7 seconds and 532 communication rounds. The projected time was 2993.8 seconds, differing from the real run time by only 0.8%.

We benchmarked our solver using instances we reduced in size from the haplotype inference dataset of [168], specifically the $100kb$ genotype data. We used 232 instances in total to benchmark our solver, varying over $|G|$ and r as previously described. When $r = 2|G|$ the formulas are necessarily satisfiable, while the remaining are mostly unsatisfiable. Both Figure 2.5 and Figure 2.6 depict the proportion of the instances

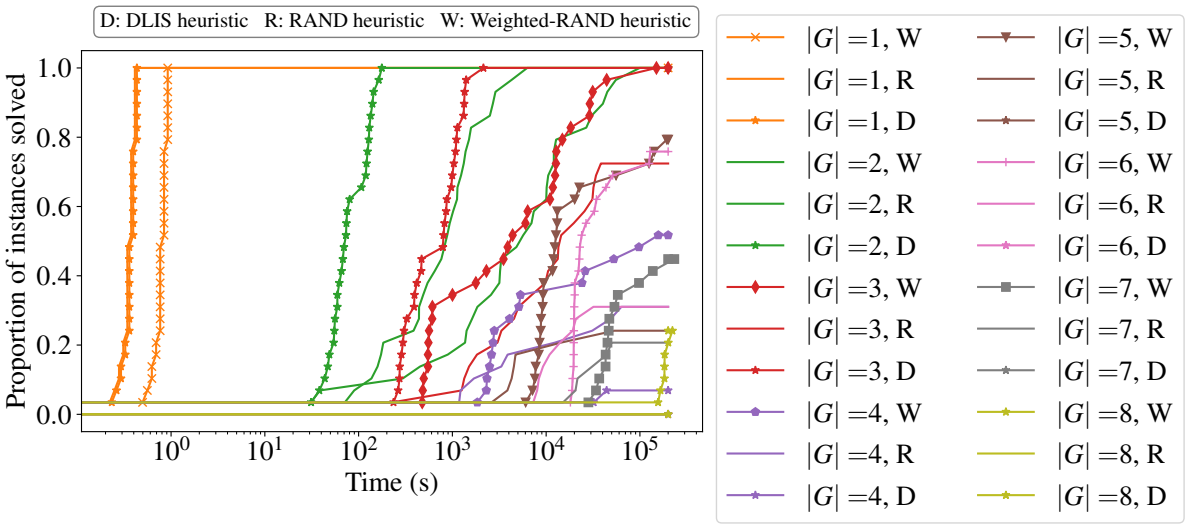


Figure 2.5: Haplotype benchmarks when $|G| \in [1..8]$ and $r = 2|G|$ with timeout of 200k seconds. With all heuristics, the solver is successful on small formulas, *e.g.*, $|G| \leq 2$, for which DLIS outperforms the randomized heuristics. Weighted-RAND becomes the most effective for larger databases.

solved within a particular time, *i.e.*, a point (x, y) indicates that y proportion of the instances are projected to be solved within x seconds. We set the timeout to 200k seconds (≈ 2.3 days).

For the first trial (Figure 2.5) the instances vary in $|G|$ while r is fixed to be $2|G|$. All three heuristics can solve most formulas before the timeout for $|G| \leq 3$, but vary in performance when the formulas get larger. For those smaller formulas DLIS outperforms RAND and Weighted-RAND. However, when $|G| > 3$, Weighted-RAND outperforms the other two, and when $|G| = 8$ it is the only heuristic with which the solver can successfully solve any benchmarks. This result is expected and reasonable: though expensive per giant step, it is the only one of the three to combine randomness with insight into the formula structure (through the weighting).

In the second trial (Figure 2.6) we evaluated the performance of our solver for various r with fixed $|G| = 3$. When $r < 2|G|$ the formula can (i) be unsatisfiable; or (ii) remain satisfiable but potentially be more difficult, as it requires some haplotypes to explain more than one genotype. The solver can handle over 70% instances before the timeout for all heuristics and almost all r, though the RAND heuristic leads to only 30% success when $r = 5$. Despite the larger formulas the solver is more successful for $r = 6$ than for $r = 5$, likely due to the greater solution freedom explained by (i) and (ii).

2.8.3 Comparison to Plaintext Solvers

Finally, we compared our ppSAT solver against itself when run in the plaintext. We wrote our plaintext solver in Python by implementing our pseudocode in the natural way, *i.e.*, every garbled circuit was replaced with standard RAM model operations, and a non-oblivious stack was used. Table 2.3 shows the results and so the overhead

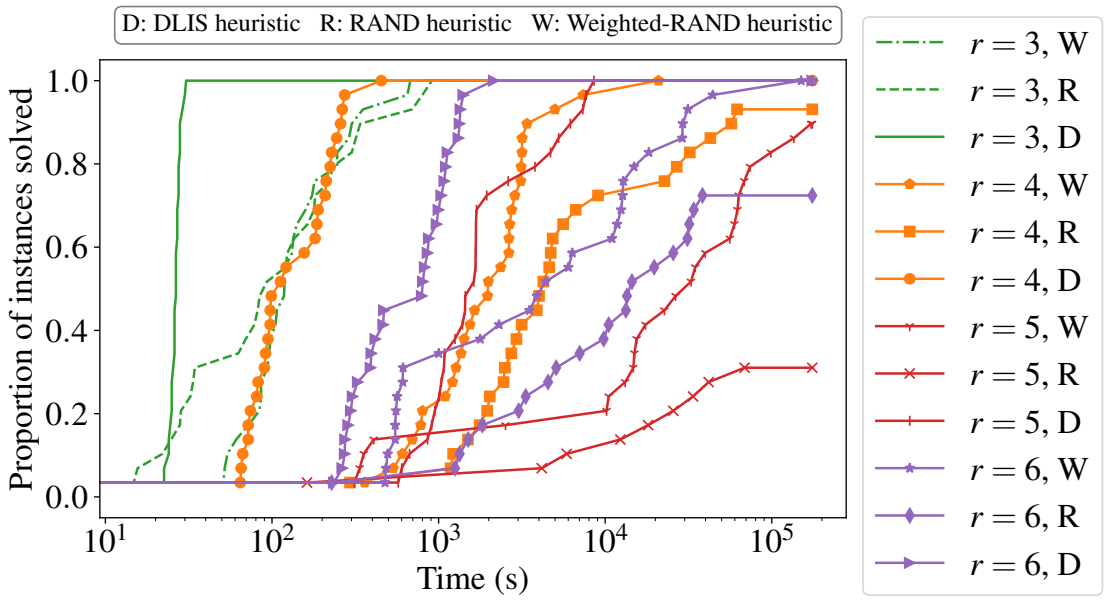


Figure 2.6: Haplotype benchmarks for $r \in \{3, 4, 5, 6\}$ when $|G| = 3$ with timeout of 200k seconds. Our solver resolves over 80% of benchmarks before timing out, except for RAND when $r = 5$.

#var × #clause	50 × 10K	100 × 10K	1K × 10K
RAND	3.4×	5.1×	47.0×
Weighted-RAND	8.3×	11.0×	165×
DLIS	4.6×	6.4×	136.8×

Table 2.3: Slowdown of ppSAT compared with it in the plaintext. In the plaintext means all data and operations are public during the computation.

brought by communication and 2PC. We also compared ppSAT with the state-of-the-art Kissat SAT solver [25]. For our 232 benchmarks Kissat can solve 231 of the instances within 0.02s, and the last within 1s. For brevity we omit the raw data from this experiment.

2.9 Additional Considerations

We raise a few additional considerations worthy of expansion, which also point towards potential future research directions.

Noisy Termination. There are inherent compromises to both exact-time-revealing and time-bound-revealing solvers. For the former, it is not immediate how much information leakage occurs when halting at resolution. In some circumstances it may be significant. For example, if ϕ_0 and ϕ_1 each contain unit clauses (v_i) and $(\neg v_i)$ respectively then the solver will always halt on the first giant step. If the inclusion of these clauses carries privacy implications then this leakage may be unacceptable. Nonetheless, it seems plausible that for many natural instances such runtimes are too coarse a measure to contain information compromising to privacy in practice –

especially when using randomized heuristics. As for time-bound-revealing solvers, running for $\tau_{\lambda,n,m}$ steps may be expensive and undesirable when not required for correctness. Always requiring such a high cost could very well limit the economic or social value of the solver.

A potential third way is to not terminate exactly upon resolution, but instead to add calibrated noise to extend the runtime for a manageable but privacy-enhancing number of steps. The theory of differential privacy (DP) [77, 78] would seem to provide an applicable toolkit, and has in fact been integrated with 2PC for the closely related purpose of "noisy load overestimation" in a line of recent work [107, 118]. The intuitive idea, following He *et al.* [118], is to relax Definition 2.5.1 to allow a bounded difference in the output of the adversary for any two formulas of the same length. However, we cannot directly use their formulation of *output-constrained DP*, since in our case only some of the output (τ) will have added noise on release – both s and (when applicable) $\mathcal{M}$ will be released exactly.

Instead, we formulate a (ϵ_0, ϵ_1, δ_0, δ_1)-noisy-time-and-model-revealing ppSAT solver (or (ϵ_0, ϵ_1, δ_0, δ_1)-*2p-ntmr-solver*) by altering Definition 2.5.1 so that (i) instead

$$v^{ideal}_{b,\,\phi'_{1-b}} = \text{view}^{\Pi}_b(\phi_b,\, n,\, m,\, \phi_{pub}, \tau_{\lambda,n,m}, \epsilon_b,\, \delta_b, \\ \text{Sim}_{1-b}(\phi_b,\, \phi'_{1-b},\, n,\, m,\, \phi_{pub},\, \tau_{\lambda,n,m}, \epsilon_{1-b}, \delta_{1-b},\, s,\, \mathcal{M}))$$

for ϵ_b, $\epsilon_{1-b} > 0$ and $0 \leq \delta_b$, $\delta_{1-b} < 1$, and (ii) requiring that there exist a Sim_{1-b} such that:

$$\Pr[\mathcal{A}(1^\lambda,\, v^{real}_b) = 1] \leq e^{\epsilon_{1-b}} \cdot \Pr[\mathcal{A}(1^\lambda, v^{ideal}_{b,\,\phi'_{1-b}}) = 1] + \delta_{1-b}$$

for all ϕ'_{1-b} such that $|\phi_{1-b}| = |\phi'_{1-b}|$. Intuitively the simulator no longer has τ, and so instead must internally execute the ppSAT solver over $\phi' = \phi_b \wedge \phi'_{1-b} \wedge \phi_{pub}$ to determine a resolution time ρ_S, before adding noise to determine a τ_S to halt at. As the real world stopping time τ_R is also a noisy version of the true resolution time ρ_R, suitable noise will allow the simulator to meet the requirement for (ϵ_{1-b}, δ_{1-b})-indistinguishability.

Unfortunately the instability of SAT instances makes this guarantee difficult to practically realize. With this definition we are viewing the ppSAT solver as a mechanism which on input a "database" in the form of ϕ noisily outputs the resultant runtime $\tau_\phi \in [1..\tau_{\lambda,n,m}]$. The amount of noise required depends on the sensitivity $\Delta\tau = \max_{\phi,\,\phi'} \max_{\tau_\phi,\tau_{\phi'}} |\tau_\phi - \tau_{\phi'}|$ for all pairs ϕ, ϕ' where $|\phi_{1-b}| = |\phi'_{1-b}|$ and $\phi_{pub} = \phi'_{pub}$. However, $\Delta\tau$ can often be a significant fraction of $\tau_{\lambda,n,m}$ as it is taken over all ϕ' of a certain length, including, *e.g.*, cryptographic instances [90]. This is far larger of a sensitivity than DP mechanisms naturally work well with. For example, applying the load overestimating techniques from [107, 118] would lead to increasing the runtime by $\approx 40 \cdot \Delta\tau/\epsilon$ giant steps on average [107], while we ideally want $\epsilon < 1$ and certainly desire it to be small [78].

When applying DP outside its origin in private statistical data analysis, a common technique is to reduce the anonymity set by restricting the sensitivity to some other

definition of pairs of "adjacent" instances [74]. However, it is unclear how to do this for SAT instances in a reasonable way. Characterizing what makes SAT instances natural is a deeply rich and complex question with numerous mathematical and empirical notions in the literature – clause density, treewidth, backdoors, modularity, etc. [44, 90]. There is no immediately apparent way to decide which formulas to include in a definition of adjacency based on these metrics, nor how to prove a usefully reduced sensitivity from them. Alternatively, in some settings empirical analysis might show that distributions of runtimes are nicely clustered for both SAT and UNSAT instances, allowing noise calibrated to smaller sensitivities justified on those statistics. But even then the sensitivity may very well require impractical noise.

we mostly leave these questions open. One potential direction might be to make a leap in logic and characterize the output of the mechanism as the proportion $\tau/\tau_{\lambda,n,m}$. The exponential mechanism [78] could then be used with bucketing of runtimes and an MPC outcome selection similar to the approach used in Algorithm 10. Though this is not justified on first principles it may be defensible. In private data analysis the sensitivity of a uniquely identifying query (*e.g.*, "count 'John Doe with UID: 1234' entries in the database") exactly captures the worst-case information leakage. Since the leakage from SAT runtimes is arguably much coarser, the proportion of the available time used may be a more natural interpretation of leakage than the actual number of giant steps. But such an approach would compromise the firm foundations of the DP guarantee.

Preprocessing Optimizations. Continuing along the lines of trading off efficiency for leakage, we briefly raise a few potential preprocessing optimizations. In general, developing a suite of similar such techniques, as well as an understanding of the tradeoffs they bring, may be a rich avenue for future work.

1. Though private set disjointedness is likely unreasonable over the (often massive) set of valid models, it could be used to find unit clause conflicts before initializing the full SAT solver. For example, suppose ϕ_1 has forced variables $\text{pos}_0 = \{v_1, v_3\}$, and $\text{neg}_0 = \{v_{17}\}$, *i.e.*, $v_1 = 1$, $v_3 = 1$, $v_{17} = 0$ must all necessarily be assigned. If ϕ_1 has forced $\text{pos}_1 = \{v_1, v_9\}$ and $\text{neg}_1 = \{v_3\}$, then the P_i could determine that $|\text{pos}_0 \cap \text{neg}_1| + |\text{neg}_0 \cap \text{pos}_1| > 0$ and output UNSAT immediately. The tradeoff would be to leak information about the composition of these sets.

2. Another potential technique would be to allow parties to provide "hints", *i.e.*, partial models which satisfy (part of) their formula. In particular, each party could provide a set of assignments which resolve especially tricky structures within their input. By loading all of these hints into another oblivious stack they could each be explored from, ultimately falling back on an empty model if none are successful.

3. A final idea applies when ϕ can be split into subsets of clauses all of which are independent from each other in terms of the variables they reference. For example, $\phi = (v_1) \wedge (v_1 \vee v_2) \wedge (v_3 \vee v_4)$ may be split into $\phi_a = (v_1) \wedge (v_1 \vee v_2)$ and $\phi_b = (v_3 \vee v_4)$.

If the P_i are willing to leak the variable inclusions in these subinstances it would allow running them independently, potentially reducing costs as

$$(2^{n_1} \log n_1) \cdots (2^{n_k} \log n_k) \leq (2^{n_1} + \cdots + 2^{n_k}) \log(n_1 + \cdots + n_k).$$

for k subinstances each of n_i variables for $i \in [k]$. Privately finding these subinstances could be done through an oblivious breadth-first search for strongly connected components over the adjacency graph of ϕ, adapting from [45].

2.10 Conclusion

The field of SAT solving has seen a superb (and continuing) developmental arc since the publication of DPLL almost 60 years ago. Given its centrality to computing and the importance of data privacy to modern technology and society, efficient privacy-preserving SAT solving would likely be a versatile and powerful tool for research and practice. In this chapter we established the core security definitions, oblivious DPLL design, and private decision heuristics necessary to implement a ppSAT solver capable of resolving small but practical instances. Of perhaps greater importance than our empirical results is the basis this lays for future work towards more efficient and effective ppSAT solvers, which we might hope will retrace the developmental arc of SAT solving itself.

Limitations & Future Directions. The centrality of CDCL to modern SAT solving makes its reformulation for ppSAT, as discussed in §2.6.4, the most important direction for future work. The greatest limitation of the original DPLL algorithm – and so also of our oblivious adaption of it – is its inability to learn from failed branches of its search effectively. CDCL is the dominant enhancement of DPLL for rectifing this shortcoming [90], and it is hard to imagine a path to general practicality for ppSAT solving that does not also rely upon it, especially for UNSAT instances.

Pruning the search tree is not the only tactic, however, for beating back the combinatorial explosion of DPLL. Modern SAT solvers rely heavily on "making their own luck" for searching what remains through intelligent decision heuristics. Our DLIS, RAND, and Weighted-RAND heuristics are limited by the standards of modern solvers [170], but have the benefit for us of being naturally implementable within Boolean circuits. Adapting or developing a fresh suite of effective heuristics suitable for oblivious computation – perhaps even using mixed-mode MPC (*i.e.*, with both Boolean and arithmetic circuit primitives) – is another major future direction. Decision heuristics also provide a particularly fertile ground for further collaboration between the cryptography and formal methods communities, as they will require reconciling the algorithmic techniques of each. Together, adapting CDCL and developing suitable decision heuristics are the foremost steps to generally practical ppSAT solving.

The last two directions for future work we emphasize are discussed more comprehensively in § 2.9. The first is to begin to understand the practical meaning of any privacy loss permitted by our security definitions. There is limited prior work on characterizing information leakage from MPC for general computations, with that of Mardziel *et al.* [169] the only known to the authors. At present we cannot in any meaningful sense explain what a party loses in privacy by, *e.g.*, setting a specific $\tau_{\lambda,n,m}$ based on an economic analysis of projected runtime using our micro benchmarks methodology, or choosing an exact-time-revealing solver over its time-bound-revealing cousin. Given the rich and complex encoding of information within the structure of SAT formulas, exploring how ppSAT solvers should leak information (which may be especially important to integrating CDCL) is likely necessary for widespread confidence in their future practical deployment. As the core algorithms of ppSAT solving develop and mature, expanding the suite of such techniques may further encourage the development of practical tooling.

Chapter 3

Proving unsatisfiability in zero knowledge

Zero-knowledge (ZK) protocols enable one party to prove to others that it knows a fact without revealing any information about the evidence for such knowledge. There exist ZK protocols for all problems in NP, and recent works developed highly efficient protocols for proving knowledge of satisfying assignments to Boolean formulas, circuits and other NP formalisms. This work shows an efficient protocol for the the converse: proving formula *unsatisfiability* in ZK (when the prover posses a non-ZK proof). An immediate practical application is efficiently proving safety of secret programs.

The key insight is to prove, in ZK, the validity of *resolution proofs* of unsatisfiability. This is efficiently realized using an algebraic representation that exploits resolution proofs' structure to represent formula clauses as low-degree polynomials, combined with ZK random-access arguments. Only the proof's dimensions are revealed.

I implemented our protocol based on recent interactive ZK protocols and used it to prove unsatisfiability of formulas that encode combinatoric problems and program correctness conditions in standard verification benchmarks, including Linux kernel drivers and Intel cryptography modules. The results demonstrate both that our protocol has practical utility, and that its aggressive optimizations, based on non-trivial encodings, significantly improve practical performance.

3.1 Introduction

Zero-knowledge proofs enable one party, the *prover*, to convince a second party, the *verifier*, that they know the validity of a claim, without revealing information about their evidence for the claim. There exist zero-knowledge protocols for proving knowledge of solutions to all problems in NP [94] and perhaps beyond [34]. In recent years, numerous efficient protocols and optimized implementations have been developed for ZK proofs of NP problems such as circuit satisfiability, correct execution of programs

(e.g., [108, 92, 184, 37, 38, 215, 126, 177, 12, 49, 46, 119, 37, 121, 87]). These found a rapidly-expanding set of applications, including: blockchain privacy [36, 37] and scalability [48, 214], legal systems [86] and anonymous networks [15].

However, there are plenty of hard problems of practical interest outside of NP, and in particular, instances of the UNSAT problem. UNSAT is the decision problem of determining if a given Boolean formula does *not* have any satisfying assignment. Beside its theoretical interest as the quintessential coNP-complete problem, UNSAT also naturally captures the task of proving that program is *secure* (under various desirable definitions of security). Indeed, various approaches to program and system verification essentially reduce program verification (specifically, proving that a program does not reach an undesired state, e.g. in which the program accesses memory incorrectly or performs an arithmetic operation that results in overflow) to proving that a given SAT formula is unsatisfiable [173].

Thus, proving UNSAT in *zero knowledge* would enable applications where a code analyst wishes to prove to another party that a public program is correct. A number of existing firms, including Coverity, ShiftLeft, and SonarQube [1, 3, 2], provide value to their users via code-analysis-as-a-service. While not all of these services attempt to provide formal guarantees about the states that a program may reach, such guarantees are of immediate value to developers, have been produced by various in-house analyses in the recent past, and could realistically be produced by analysis services in the near future [22, 129].

Even when the code of a bounded program is public, determining the states that the program can reach is computationally hard, and is achieved in practice only through the use of subtle heuristics and carefully tuned implementations. Thus, even when a service that determines reachable states are applied to public programs, the service's results may constitute sensitive IP. Clauses of a resolution proof of program safety are intermediate deductions about the program's reachable states: thus, if the analyst's IP is to be protected, such clauses must be kept secret.

In principle, a party who knows that a formula is unsatisfiable and has a certificate for this fact, can prove knowledge of this certificate using generic ZK for NP [97] applied to the certificate-checker. However, such approaches would be too inefficient to be used in practice because reducing UNSAT to these problems that are provable in ZK directly incurs a high, albeit polynomial, overhead. An approach that would compile programs (of bounded runtime) to Boolean circuits [123] would also need to include a proof of the circuit's unsatisfiability. Similarly, an approach that would perform static analysis of general programs in zero knowledge based on abstract interpretation [84] would critically rely on efficient implementations of operations over SAT formulas, including the validation of proofs of their logical entailment or equivalence.

In this work, we designed and implemented a novel, efficient protocol for proving UNSAT in zero-knowledge. In general, our protocol can be used directly to efficiently prove knowledge of solutions to any problem in coNP, once the problem has been

reduced to proving UNSAT. In particular, our protocol can be used as *highly efficient backend* for proving safety of potentially-secret programs in zero knowledge, either by validating proofs of SAT formulas generated by model checkers, or by efficiently implementing primitives required by analyses based on abstract interpretation.

The key insight behind our approach is to efficiently validate an additional argument for UNSAT in the form of a *resolution proof*, a sequence of clauses that can be derived from the given formula and which concludes in a contradiction. Such proofs are both well-understood in principle and efficiently supported in practice. In principle, they are a sound and complete proof system for proving UNSAT. Although short resolution proofs may not always exist for UNSAT formulas in general, they are often found efficiently by state-of-the-art SAT solvers applied to encodings of practical problems in planning and program verification. Thus, we can develop ZK protocol for instances of UNSAT by requiring the resolution proof as advice, revealing its *length* (the number of clauses in the derivation), and validating the resolution proof by executing a RAM program in ZK [50, 38, 215, 126, 177, 49, 46, 119, 37, 121, 87].

A second insight, critical for efficiency, is that in practice resolution proofs usually have low *width* in addition to short length: i.e., each clause in the derivation contains only a small number of literals. By revealing the proof's width along with its length, we can implement a significantly optimized protocol that represents clauses in the derivation as *low-degree polynomials* and validates the derivation itself by checking a small number of polynomial equalities. The resulting protocol's performance is essentially independent of the number of literals, and depends only on the width and length of the proof. It outperforms the previous one (which hides the width) when clauses are sparse, e.g., when there are more than 1000 variables but each clause contains at most 100 literals.

I evaluated our protocol empirically by implementing it via the EMP framework [217] and using it to prove unsatisfiability of formulas that encode problems in combinatorial optimization, planning, and the verification of safety-critical programs drawn from the SV-COMP [40] benchmark set. This includes verification of Linux device drivers, Windows NT device drivers, and C implementations of floating-point computation.

Contribution

- We initiate the study of the practicality of proving the unsatisfiability of Boolean formulas in zero knowledge, and its applications to proving properties of programs in zero knowledge.

- Bringing together formal methods and cryptography, we propose ZK-friendly algebraic encodings of Boolean formulas and of (relaxed) resolution proof of formula unsatisfiability.

- Using these, we design and optimize concrete ZK proof schemes for UNSAT that are efficient enough to support useful program-verification formula sizes.

- we present a prototype implementation, which can be found at `https://github.com/zkunsat/zkunsat`, and benchmark this implementation on large formulas, including ones representing the safety of Linux kernel drivers and Intel cryptography modules.

Non-goals Our ZK protocol can also be directly applied to prove unsatisfiable of secret formulas, which can in turn be committed. However, more efforts on top of our protocol are needed to enable ZK proof of program correctness for private (and possibly committed) programs. To build a complete tool that verifies the safety of a secret program in ZK, it is also necessary to verify that an formula models the secret program's semantics. This means that any unsafe executions of the secret program corresponds an interpretation of a secret formula. Proving that an formula models the secret program's semantics, and thus verifying secret programs in ZK, is beyond the scope of the presented in this chapter. we provide more discussion at the end of the chapter.

Organization The remainder of this chapter is organized as follows: Section 3.2 presents an overview of our protocol by example; Section 3.3 reviews foundational definitions and results on which this chapter is based; Section 3.4 presents our protocol in technical detail; Section 3.8 describes our implementation and empirical evaluation of the protocol; Section 3.9 compares our contribution to related work, and Section 3.10 concludes.

3.2 ZK program safety by example

This section describes how our protocol proves UNSAT efficiently and how it can be applied to prove safety of a public program. To contextualize, we start with a brief tutorial to the standard techniques of proving program properties using resolution proofs I then give an overview of the zero-knowledge protocol and an optimization that significantly improves its performance.

Building a formula To illustrate how program verification can be encoded as the satisfiability problem of Boolean formulas, we use the small C program `sum3` given in Figure 3.1a. `sum3` returns the sum of three integers, while avoiding integer overflows past the maximum representable integer `MAX`. For simplicity, we consider the case of single-bit integers and `MAX`=1 (in which case `sum3` is simply the OR of 3 bits).

In this case the operators `+` and `-` over `int1` both correspond to XOR, and `<=` corresponds to implication. We can thus write a Boolean formula φ, in Figure 3.1b, that describes the program execution. Within φ, propositional variable $\texttt{acc}_i$ denotes the value of C variable `acc` after the i-th update. Propositional variables b_i are used to denote the branching condition; *ret* corresponds to the value returned by the program; o_1 and o_2 are Boolean values denoting that overflow occurs, and the other propositional variables correspond to program parameters and local variables.

```
1 int1 sum3(int1 a0, int1 a1, int1 a2) {
2     int1 acc = a0;
3     if (acc <= MAX - a1)
4         acc = acc + a1;
5
6     if (acc <= MAX - a2)
7         acc = acc + a2;
8     return acc;
9 }
```

(a) `sum3`: program that sums three 1-bit numbers without overflow.

$$
\begin{array}{llll}
acc_0 \leftrightarrow a_0 & \wedge & b_0 \leftrightarrow & (acc_0 \rightarrow (\mathsf{True} \oplus a_1)) \wedge \\
acc_1 \leftrightarrow acc_0 \oplus a_1 & \wedge & o_1 \leftrightarrow & b_0 \wedge acc_0 \wedge a_1 \wedge \\
acc_2 \leftrightarrow b_0 \ ?\ acc_1 \ :\ acc_0 & \wedge & b_1 \leftrightarrow & (acc_2 \rightarrow (\mathsf{True} \oplus a_2)) \wedge \\
acc_3 \leftrightarrow acc_2 \oplus a_2 & \wedge & o_2 \leftrightarrow & b_1 \wedge acc_2 \wedge a_2 \wedge \\
acc_4 \leftrightarrow b_1 \ ?\ acc_3 \ :\ acc_2 & \wedge & & \\
ret \leftrightarrow acc_4 & & &
\end{array}
$$

(b) A Boolean formula φ that models the semantics of `sum3`.

Figure 3.1: An example program and Boolean formula that characterizes its executions.

Every satisfying assignments of formula φ correspond to a valid execution of program `sum3`. A program overflow happens if and only if any of o_i are true, i.e., if the formula $\varphi_o \equiv o_1 \vee o_2$ is also satisfied. Thus, verifying that `sum3` never overflows `MAX` can be done by proving unsatisfiability of the formula $\varphi \wedge \varphi_o$, which asserts that in a correct execution (asserted by φ) an overflow occurred (asserted by φ_o). In general, translating verification tasks for C programs into Boolean formulas can be done with existing tools such as CBMC [59].

Having a relatively low number of variables, we could simply enumerate all possible variable assignments, evaluate $\varphi \wedge \varphi_o$ on each assignment, and confirm that no assignments satisfies the formula. However, this obviously does not scale, since the number of assignments grows exponentially in the number of variables.

Resolution refutation A better method of showing that a formula is unsatisfiable is a *resolution refutation* [192]. A formula is unsatisfiable if and only if we can derive $\bot$ (false) by applying *resolution steps*, according to the fundamental theorem about refutational completeness of first-order logic [21] (which applies also to the propositional logic we employ here). Resolution proofs are reviewed in formal detail in Section 3.3.2, but we give here the details needed to follow the example:

Resolution is performed on formulas in the *clausal normal form*, i.e., a conjunction of disjunctions. Each conjunct is called a *clause*. For example, $(x_1 \vee x_2 \vee \neg x_3) \wedge (x_3 \vee x_1) \wedge \neg x_4$ is in the clausal normal form and it consists of three clauses. Negations can be applied only to variables. Every propositional formula can be converted into an equivalent conjunctive normal form.

The resolution step is given by the following schema:

$$\frac{A \vee p \qquad \neg p \vee B}{A \vee B}$$

This reads as follows: the resolution step takes as input two clauses $A \vee p$ and $\neg p \vee B$, and derives a new clause, $A \vee B$, which is a logical consequence of two input clauses. The derived clause is called the *resolvent*, and variable p is called the *pivot*. In the context of refutational completeness theorem, on the given set of clauses, the resolution rule can be applied as many time as needed until it is either no longer possible to derive new clauses, or the $\bot$ formula has been derived.

Although simple, the resolution rule is the basis of modern automated first-order reasoners [193], and their applications to program verification. Indeed, we proceed to show its use to prove that `sum3` does not overflow.

I show that $\varphi \wedge \varphi_o$ is unsatisfiable through several steps. First, we convert $\varphi \wedge \varphi_o$ into the clausal normal form, denoting the resulting formula with φ_{CNF}. This results in a large formula. For readability, we list here only four of its clauses, which suffice to derive $\neg o_1$. These clauses are: $\neg b_0 \vee \neg acc_0 \vee \neg a_1$, $b_0 \vee \neg o_1$, $acc_0 \vee \neg o_1$ and $a_1 \vee \neg o_1$. From these we can derive $\neg o_1$ by applying the resolution rule 3 times, as follows:

$$\dfrac{\dfrac{\dfrac{\neg b_0 \vee \neg acc_0 \vee \neg a_1 \qquad acc_0 \vee \neg o_1}{\neg b_0 \vee \neg a_1 \vee \neg o_1} \qquad a_1 \vee \neg o_1}{\neg b_0 \vee \neg o_1} \qquad b_0 \vee \neg o_1}{\neg o_1}$$

Similarly, we can derive $\neg o_2$. Finally, we can derive $\bot$ by using the resolution rule twice more, applied to $\neg o_1$ and $\neg o_2$ (whose derivations, above, are denoted by ... below) and to the clause $o_1 \vee o_2$ that is also in φ_{CNF}:

$$\dfrac{\dfrac{o_1 \vee o_2 \quad \dfrac{\ldots}{\neg o_1}}{o_2} \quad \neg \dfrac{\ldots}{o_2}}{\bot}$$

We managed to derive $\bot$, establishing that the original formula $\varphi \wedge \varphi_o$ was unsatisfiable, hence `sum3` does not have integer overflows.[1]

Resolution proofs as non-ZK proofs of UNSAT The derivation of $\bot$ (called the *resolution proof*) is a certificate of unsatisfiability. Indeed, given an alleged resolution proof, it can be efficiently checked by a *resolution-proof checker* that follows a claimed derivation tree and verifies that: in every invocation of the resolution rules, all inputs have appeared in the original formula or prior derivations, and the resolvent is correctly derived with respect to some pivot; and the last resolvent is $\bot$.

Thus, a trivial proof protocol for UNSAT is for the prover to hand over a resolution

[1] Had the formula been satisfiable, applying the resolution rules could never have derived $\bot$, and moreover (for propositional logic), the process would have eventually terminated and let us read a satisfying assignment out of the derived clauses [21], revealing inputs to `sum3` that cause an overflow.

proof to the verifier. However, this is far from zero knowledge. A resolution proof, constructed and derived as above, reveals information about the program (which is encoded in the formula) and the analysis technique (which created the derivations).

In general, resolution refutations can be hard artifacts to construct from a program: there is no efficient algorithm to generate them and in fact no polynomial bound on the length that such derivations may have. In the domain of Boolean formulas that correspond to program verification conditions, the structure of a resolution proof may reflect the insights of a manual or automatic program analyzer. In particular, a valid refutation of $\varphi \wedge \varphi_o$ could include derived properties of the variables acc_3 and acc_4 or relating variables a_1 and a_3 (e.g., it could derive the clause

$$\neg b_0 \vee \neg acc_0 \vee \neg a_1.$$

Indeed, one of the main technical challenges for first-order automated reasoners is to make sure that they are deriving (mainly) goal-oriented clauses. Often it is the the case that a reasoner will derive more and more clauses that are indeed consequences of previous clauses but are not used in the proof of deriving the $\bot$ clause.

In our example, we produced a proof derivation that only derived clauses needed to derive $\bot$. Our clause selection was guided by insights about the structure of `sums3` and selecting only clauses relevant to refuting the overflow clause $o_1 \vee o_2$.

ZK proofs of UNSAT Our first ZK protocol for UNSAT mitigates the above information leakage, by proving that a public formula is unsatisfiable while only revealing the number of clauses in one of its refutations.

Essentially, the prover uses a ZK proof system to prove that it *locally* executed the computation "run the resolution-proof checker on the given formula and a secret resolution proof", and the checker accepted. The resulting ZK proof, presented to the ZK verifier, is as convincing as the original resolution proof (by the soundness property of the ZK proof system), but effectively redacts all details of the checker's input and execution trace.

Technically, this works by representing the resolution-proof checker as an algebraic constraint system, and applying a suitable zero-knowledge proof scheme to this constraint system. Efficiency hinges on suitable choice of ZK proof system, and careful encoding of the resolution-proof checker as algebraic constraints. Details are given in Section 3.4.

Optimization by revealing resolution width Implementing a resolution-proof checker requires a representation of formulas and clauses. The natural one is encoding clauses as vectors, whose length is the number of propositional variables in the formula. For example: one binary vector specifying which variables appear in the clause, and another specifying their polarity. Validating the proof then is reduced to Boolean operations over the binary vectors that represent clauses.

Applying the aforementioned ZK transformation to this representation yields a scheme that is already efficient enough to prove knowledge of resolution proofs for

interesting formulas on a practical machine: it takes about 80 seconds to verify a proof of 2^{15} literals and 3000 resolvents. However, its limitations are revealed in plenty of cases that arise in practice: according to our evaluation, it fails to prove that driver benchmarks are safe up to 2000 steps as there are over 150K variables in the resulting formula.

A possible optimization is apparently already in the verification condition of `sums3`: $\varphi \wedge \varphi'$ are defined over eleven propositional variables modeling all parameters, return values, local variables, and overflow conditions, but each individual clause contains literals over at most three variables; i.e., the proof's *width* is three. Intuitively, this is because the two additions can be proved not to overflow by independently analyzing them and the conditions that guard them. As discussed in Section 3.8, this is typical, and reputations of verification conditions collected from practical programs indeed tend to width much lower than their total number of variables.

Resolution proofs of low width w can be validated more efficiently than the general case by representing each clause of the proof as a degree-w univariate polynomial, in a formal variable X, over a large-enough finite field. For each literal a in a clause C, the polynomial representation of C, denoted p_C, contains a term $X - \phi(a)$, where $\phi(a)$ denotes a distinct field element that identifies a; identifiers of literals and their negations satisfy a simple arithmetic relation that ensures that the laws of Boolean arithmetic are embedded faithfully.

E.g., Clause (3.2) is represented as the degree-3 polynomial

$$(X - \phi(b_0))(X - \phi(acc_0))(X - \phi(a_1))$$

Under this representation, checking that some clause C_0 *logically implies* some clause C_1 amounts to checking that the associated polynomial p_{C_0} *divides* polynomial p_{C_1} or equivalently, that there is some polynomial q such that $q \cdot p_{C_0} = p_{C_1}$. This correspondence can be applied to validate steps of resolution by checking polynomial equalities: instead of checking polynomial division, we ask the prover to provide q and then proving the equality between a given polynomial and the multiplication of polynomials. The equality can be checked efficiently via the Schwartz-Zippel lemma, while polynomial multiplication can be done based on any compatible ZK protocol. I describe this encoding in detail in Section 3.4.1.

3.3 Technical Preliminaries

3.3.1 Fields and polynomials

A *field* $\mathbb{F}$ is a set equipped with two binary operations, referred to as addition and multiplication, that forms a commutative group under addition (with additive identity denoted $0_{\mathbb{F}}$), has a multiplicative identity (denoted $1_{\mathbb{F}}$), contains a multiplicative

inverse for each non-zero element, and in which multiplication distributes over addition. For field elements $a, b \in \mathbb{F}$, the sum and product of a and b are denoted $a + b$ and $a \cdot b$, respectively.

We will define protocols that use univariate polynomials over a given field $\mathbb{F}$, which will be referred to for the rest of the chapter simply as "polynomials" and denoted $\mathbb{F}[X]$. A *root* of polynomial p is a field element $a \in \mathbb{F}$ for which $p(a) = 0_{\mathbb{F}}$. For polynomials p and q, the sum and product of p and q are denoted $p + q$ and $p \cdot q$, respectively. If there is some polynomial r such that $r \cdot p = q$, then p *divides* q, denoted $p \mid q$. A polynomial that can be expressed as a product of d ($d \geq 0$ linear polynomials is *completely reducible*. Constant polynomials are always completely reducible polynomials. For all polynomials p and q with root $a \in \mathbb{F}$, the polynomial $p \cdot q$ has a as a *repeated root*. For each polynomial p, we can construct a unique completely reducible divisor p^* as by having $p^* = \prod_{i=0}^{k}(X - a_k)$, where $a_0, \ldots, a_k$ are *all* the roots of p. Notice that p^* does not have a repeated root and can be divided by *every* completely reducible divisor of p that does not have a repeated root;

3.3.2 Boolean logic

In this chapter, we primarily consider Boolean formulas in a clausal form. A *literal* over a set of variables Vars (whose elements are denoted using lowercase letters) is an element in Vars paired with a bit that denotes if the variable occurs positively or negatively (the set of literals over Vars is denoted $\mathsf{Lits} = \mathsf{Vars} \times \mathbb{B}$, where $\mathbb{B}$ denotes the Booleans); a positive occurrence of variable $x \in \mathsf{Vars}$ is denoted as simply x, while a negative occurrence of x is denoted $\neg x$. A *clause* is a set of literals and it denotes the logical disjunction of the literals that is contains. The empty clause is denoted $\bot$; the union of clauses C and C' is denoted $C \vee C'$ and C extended with a single literal ℓ is denoted $C \vee \ell$. Note that because clauses are *sets* of literals (and not general multisets or sequences), a given clause can contain at most one occurrence of a given literal. As one consequence,

$$(C \vee \ell) \vee \ell = C \vee \ell$$

for each clause C and literal ℓ.

A *formula* is a set of clauses, which denotes their conjunction; the set of formulas is denoted $\mathcal{F}$. An assignment $f : \mathsf{Vars} \to \mathbb{B}$, satisfies a positive (negative) literal l if it assigns l's variable to True (False); it satisfies a clause C if and only if it satisfies some literal in C. As such, an empty clause $\bot$ cannot be satisfied by any assignment. f satisfies formula $\varphi \in \mathcal{F}$ if and only if it satisfies each clause in φ, and the formula φ is *unsatisfiable* if it is not satisfied by any assignment.

Resolution proofs

Resolution proofs are formal arguments that a given clause is implied by a given formula.

Definition 1. *For clauses C and C', the* resolvent *of* premise *clauses $x \vee C$ and $\neg x \vee C'$ on* pivot *variable x is the clause $C \vee C'$.*

Resolution derivations are sequences of clauses in which each clause in the sequence is the resolvent of the two preceding two clauses.

Definition 2. *A* (resolution) derivation *from formula φ is a finite sequence of clauses $\langle C_i \rangle$ in which each C_i is either **(1)** a clause in φ or **(2)** the resolvent of two clauses $j, k < i$. A* (resolution) refutation *of φ is a derivation from φ in which the final clause is $\bot$.*

Resolution derivations are *sound*: i.e., if a clause C can be derived from a formula φ then each assignment that satisfies φ also satisfies C. As an immediate consequence, if there is a refutation of φ, then φ is unsatisfiable. Conversely, resolution is *complete* for proving unsatisfiability: if a formula φ is unsatisfiable, then there is a refutation of φ [71]. However, unsatisfiable formulas may not have resolution refutations that are *short*: there is an infinite set of unsatisfiable formulas with no resolution refutation of size bounded by a polynomial over the size of the formula [113]. The *length* of a derivation is the number of clauses that it contains. The *width* of a derivation is the maximum number of literals that occur over all of its clauses; the product of a refutation's length with it's width is the refutation's *area*. In general, there is a trade-off between a proof's dimensions: there is an infinite set of formulas in which all refutations have length or width exponential in the size of the formula [208].

3.3.3 Efficient zero-knowledge protocols

The focus of this work is not to design a general-purpose zero-knowledge proof protocol but to apply existing protocols to build applications with significant practical importance and to explore its efficiency. To this end, we present in Figure 3.2 a ZK functionality ($\mathcal{F}_{\mathsf{ZK}}$) required for performing clause resolution in zero-knowledge. The functionality is reactive and allows the prover to commit to witnesses and later prove circuit satisfiability over the specified field. I use $[x]$ to represent an idealized commitment of the value; its real data depends on the underlying ZK protocol that instantiates $\mathcal{F}_{\mathsf{ZK}}$. In VOLE-based ZK that we use in this chapter [220, 75, 29, 225], the underlying commitment is information-theoretic MAC. The last two instructions in $\mathcal{F}_{\mathsf{ZK}}$ prove relationships about polynomials. It is well known that the equality of two committed polynomials over a large field can be efficiently checked in zero-knowledge using Schwartz–Zippel lemma with the cost of evaluating a random point on two committed polynomials. we include an extended instruction PoPDegCheck to prove that the products of two sets of polynomials are equal. All ZK protocols in the commit-and-prove paradigm can be used to instantiate this functionality, with $[x]$ representing a commitment of x. As a result, our clause resolution protocol has the potential to be connected to many different ZK backends. By designing our protocol in the $\mathcal{F}_{\mathsf{ZK}}$ hybrid world, future ZKUNSAT works based on other ZKPs will

Functionality $\mathcal{F}_{\mathsf{ZK}}$

Witness: On receiving $(\mathsf{Witness}, x)$ from the prover, where $x \in \mathbb{F}$, store x and send $[x]$ to each party.

Instance: On receiving $(\mathsf{Instance}, x)$ from both parties, where $x \in \mathbb{F}$, store x and send $[x]$ to each party. If the inputs sent by the two parties do not match, the functionality aborts.

Circuit relation: On receiving $(\mathsf{Relation}, C, [x_0], \ldots, [x_{n-1}])$ from both parties, where $x_i \in \mathbb{F}$ and $C \in \mathbb{F}^n \to \mathbb{F}^m$, compute $y_1, \ldots, y_m := C(x_0, \ldots, x_{n-1})$ and send $\{[y_1], \ldots, [y_m]\}$ to both parties.

Productions-of-polynomial equality check: On receiving $(\mathsf{PoPEqCheck}, n, \{[P_i(X)]\}_{i\in[n]}, \{[Q_i(X)]\}_{i\in[n]})$ from both parties, where $[P_i(x)]$ and $[Q_i(x)]$ are polynomials with their coefficient committed: if $\Pi_i P_i(x) \neq \Pi_i Q_i(x)$, the functionality aborts.

Figure 3.2: Functionality for zero-knowledge proofs of circuit satisfiability and polynomials.

benefit from this modular design because the security follows immediately from the composition theorem [55].

Zero-knowledge proofs of random accesses. There has been a long line of works [37, 50, 38, 215, 126, 177, 49, 46, 119, 37, 121, 87] in supporting ZK proofs over RAM programs. Here, we are only interested in the mechanisms that enable RAM accesses in ZK rather than the overall RAM architecture, which involves many other aspects like designing an instruction set. Existing works enable RAM accesses in roughly two ways. Some prior works [126, 177, 119, 121] combine ZK protocols with oblivious RAMs [99]: the prover proves in ZK the computation of an oblivious RAM client that translates each private access to a set of public accesses. The second approach [37, 50, 38, 215, 49, 87] is to prove all RAM accesses in a batch: by gathering all accesses and their results, the correctness validation can be expressed in a circuit of quasi-linear size.

3.4 Encoding Scheme and Protocol

This section describes our protocol in technical detail. The protocol's key correctness and security properties, along with key lemmas that support them, are stated as lemmas and theorems; their proofs are included in Section 3.5.1.

A proof of refutation of a formula ϕ can be viewed as a list of tuples, each of which specifies two clauses. The process of a resolution derivation can be viewed as an iterative procedure. We start with a list of clauses $\mathcal{C}$ that only contains all clauses in ϕ. In each iteration, we fetch two clauses from $\mathcal{C}$ as premise clauses, compute their resolvent, and append the resulting clause to $\mathcal{C}$. If the resolution completes, the last clause added to $\mathcal{C}$ should be $\perp$.

To perform the derivation in zero-knowledge, we need to pay attention to two

core tasks: 1) efficiently perform clause resolution given two clauses; and 2) efficiently fetch clauses from $\mathcal{C}$ in ZK while keeping indices private. Below, we will introduce the technical details in how our solutions work and why they improve efficiency. Section 3.4.1 discusses our encoding methods for both literals and clauses. It provides huge improvement compared to a bit-vector-based representation. In Section 3.4.2, we further improve the efficiency of clause resolution by introducing a weakened version of resolution. It provides more flexibility with prover when providing premise clauses and thus there fewer conditions to check in zero-knowledge proof. Finally, in Section 3.4.3, we discuss our solution for the second task.

3.4.1 Clause representation

To improve the efficiency of the aforementioned procedures, the central task is finding a suitable way to represent clauses. Ideally the representation should be compact so that the overhead when storing in a random-access array in ZK would not be too high; other the other hand, it should preserve the structure of a clause so that clause resolution could be done efficiently.

Naive encoding methods

As discussed in Section 3.3.2, a clause is essentially a set (of literals). Therefore, clause encoding resembles a lot in set encoding, which has been studied in numerous scenarios. Our first attempt was to use bit vectors inspired by the bit-vector representation of sets. Assuming that $|\mathsf{Lits}|$ is public, then a clause can be represented as a bit vector of length $|\mathsf{Lits}|$, such that the i-th bit indicates if the i-th literal appears in the clause. This representation is very intuitive as Boolean operations on bit vectors are closely related to Boolean logic on clauses: element-wise AND (resp., OR) on two vectors is the conjunction (resp., disjunction) of the underlying clauses. However, the downside of this approach is also obvious. Every operation on a clause has a complexity of $O(|\mathsf{Lits}|)$, even if the number of literals in the clause is significantly less. Therefore this encoding does not really scale for large formulas.

The bit vector representation is not good for sparse clauses (where the number of literals is much less than $|\mathsf{Lits}|$), but it can be improved using a better encoding. A natural next step is to instead use an enumeration-based representation for a set (and thus clause). For example, if we map every literal $\ell \in \mathsf{Lits}$ to an integer in $[\mathsf{Lits}]$, any clause with d literals can be represented in $\log |\mathsf{Lits}|$ bits. The downside of this approach is that operations on this representation are more complicated to instantiate. For example, to compute the conjunction of two clauses represented in this way, we would need to compute the intersection of two sets.

Encoding clauses as polynomials

To enable compact representation and efficient operations at the same time, our protocol encodes clauses as polynomials over some finite field. Such representation

has a small encoding size while operations, including clause resolution can still be done efficiently by representing them as operations on polynomials.

As the first step, we need to encode literals to field elements. In addition to completeness (i.e., different literals should be encoded to different field elements), we also want the encoding to support efficient negation of a literal, which is useful when doing clause resolution. For a field $\mathbb{F}$ where $|\mathbb{F}| > |\mathsf{Lits}| = 2|\mathsf{Vars}|$, we want to find an injective function $\phi : \mathsf{Lits} \to \mathbb{F}$ such that for each variable $x \in \mathsf{Vars}$,

$$\phi(x) + \phi(\neg x) = 1_{\mathbb{F}} \tag{1}$$

The definition can be adjusted to use field elements $a \in \mathbb{F}$ other than $1_{\mathbb{F}}$, so long as a ensures that ϕ is injective. Each ϕ satisfying Equation (1) is a *literal encoding* into $\mathbb{F}$.

For the rest of this chapter, let $\mathbb{F}$ denote an arbitrary field that satisfies such conditions for Vars and let ϕ refer to an arbitrary literal encoding of $\mathbb{F}$.

Given a concrete encoding of literals as field elements, we can encode a clause (which is a set of literals) as a field polynomial. From literal encoding ϕ, we define an encoding $\gamma_\phi : \mathsf{Clauses} \to \mathbb{F}[X]$ of clauses as (univariate) polynomials over $\mathbb{F}$ such that the image under ϕ of the literals in each clause C are the roots of the image of C under γ_ϕ:

$$\gamma_\phi(\ell_0 \vee \cdots \vee \ell_d) = (X - \phi(\ell_0)) \ldots (X - \phi(\ell_d))$$

for literals $\ell_0, \ldots, \ell_d \in \mathsf{Lits}$. As an important special case, the encoding of the clause $\perp$ is $\gamma_\phi(\perp) = 1_{\mathbb{F}}$, where $1_{\mathbb{F}}$ denotes a polynomial with only a constant term, which is distinct from the field element in Equation 1.

For the rest of this chapter, we will only be using only one field and one literal encodings; thus we will omit the subscript and write simply $\gamma(C)$ to denote the encoding of a clause C, whenever the field and literal are unambiguous from the context.

The key property of ϕ and γ_ϕ introduced above is stated formally as follows. It only requires the fact that ϕ is injective, not that ϕ additionally satisfies Equation (1).

Lemma 1. *For each literal ℓ and clause C, $\ell \in C$ if and only if $\phi(\ell)$ is a root of the polynomial $\gamma(C)$.*

As a corollary, logical implication over clauses corresponds to divisibility of clauses, under literal and clause encodings.

Corollary 1. *For clauses C and C', if $C \to C'$, then*

$$\gamma(C) \mid \gamma(C')$$

ZK operations on polynomial-encoded clauses

We are now ready to put clause operations inside a ZK protocol. The first operations is to allow the prover to commit to a clause. A clause with d literals can be encoded as a

Functionality $\mathcal{F}_{\mathsf{Clause}}$

Input: On receiving $(\mathsf{Input}, \ell_0, \cdots, \ell_{k-1}, w)$ from prover and (Input, w) from verifier where $\ell_i \in \mathsf{Lits}$, the functionality check that $k \leq w$ and abort if it does not hold. Otherwise store $C = \ell_0 \vee \cdots \vee \ell_{k-1}$, and send $[C]$ to each party.

Equal: On receiving $(\mathsf{Equal}, [C_0], [C_1])$ from both parties, check if $C_0 = C_1$; if not, the functionality aborts.

X-Res: On receiving $(\mathsf{Xres}, [C_0], [C_1], [C_r])$ from both parties, check if $\{C_0, C_1\} \vdash_{\text{X-Res}} C_r$; if not the functionality aborts.

IsFalse: On receiving $(\mathsf{IsFalse}, [C])$ from both parties, check if $C = \bot$; if not, the functionality aborts.

Figure 3.3: Functionality for ZK operations on clauses.

degree-d polynomial; however, in some cases even the degree could reveal information about the prover's witness (i.e., the refutation proof). To commit a clause C without revealing its real degree, the prover, after obtaining the coefficients of $C(x)$, can simply use zeros as high-order coefficients. Another caveat is that a cheating prover could potentially commit an irreducible polynomials, which cannot be factorized; this would make witness-extraction fail. To ensure extractability of clause commitments, we need the prover to commit all root of the polynomial again and two parties can use $\mathcal{F}_{\mathsf{ZK}}$ to ensure the validity of the polynomial.

Another important operation is clause resolution. To check that clause C_r is a resolvent of clauses C_0 and C_1, we must check that there is a variable x such that $C_0 = x \vee C$, $C_1 = \neg x \vee C'$, and $C_r = C_0 \vee C_1$. When translated to our polynomial-based encoding, we need to check the above relationship on roots of the polynomial. While polynomial division can be easily checked by the prover providing an extended witness and proving the equality of polynomial product, checking intersection of the roots from two polynomials would require extra effort, e.g., incorporating techniques from Papamanthou et al. [183].

3.4.2 Improved resolution via weakening

This section proposes a more efficient way of ZK resolution derivation without hurting security at all. Our key idea is a new way to weaken the properties checked by resolution while maintaining the soundness of such a check.

Resolution with weakening

To define our encoding scheme, we first define a set of derivations of SAT formulas that slightly generalizes resolution derivations (Section 3.3.2). The only differences are that in a weak resolution, **(1)** a pivot variable need not necessarily occur in the premises and **(2)** the resolvent need only be implied by resolvent of the premises

(potentially weakened with literals built from the pivot variable).

Definition 3. *A* weak resolvent *of clauses C and C' on pivot variable x is a clause C'' such that*

$$C \to C'' \vee x \qquad \textit{and} \qquad C' \to C'' \vee \neg x$$

As a special case, one weak resolvent of clauses $C \vee x$ and $\neg x \vee C'$ on pivot variable x is their resolvent, $C \vee C'$ (Defn. 1).

A weakened resolution derivation is a sequence of weak resolvents, analogous to how a resolution derivation (Defn. 2) is a sequence of resolvents:

Definition 4. *A* weak (resolution) derivation *from formula φ is a finite sequence of clauses $\langle C_i \rangle$ in which each C_i is either **(1)** in φ or **(2)** a weak resolvent of two clauses $j, k < i$.*

Weak *refutations* are similarly defined as instances of weak derivations. It is straightforward to show that weak resolution derivations are both a sound and complete system for refuting Boolean formulas: i.e., a Boolean formula is unsatisfiable if and only if it has a weak refutation. Soundness follows from the fact that resolution refutations are sound and every refutation is a weak refutation. Completeness can be proved by interleaving each step of resolution in a given weak refutation with a (potentially empty) sequence of resolutions that derives the weakening of a resolvent from the resolvent itself.

Compared to derivations, weak derivations do not have any apparent interesting proof-theoretic properties. However, in Section 3.4.2 we will introduce a scheme specifically for encoding and validating weak resolvents; the validation cannot apparently be adjusted to validate exactly resolvents without more than doubling the size of the encoding of each validation. Moreover, a practical consequence of the fact that each refutation is a weak refutation is that any refutation generated by existing SAT theorem provers can be directly encoded by our scheme. In principle, such refutations could potentially be minimized by replacing multiple steps of resolution that derive a weakening of a resolvent with a single step of weak resolution; however, our current implementation does not perform such an optimization.

Proving weakened resolution in ZK

A weak resolution derivation can be efficiently checked using field arithmetic: clauses in the derivation are represented as polynomials and the fact that a clause is a weak resolvent of two clauses can be checked efficiently by testing equality of polynomials. We present our protocol in Figure 3.4.

A clause can be checked to be a weak resolvent to two other clauses by checking equalities of the clauses encodings as polynomials. The key idea behind the protocol is to check the implications over clauses that define a weak resolution (Definition 4) by checking divisibility of polynomials, which itself is checked by checking equality of

Protocol Π_{Clause}

Parameters: A set Lits of all possible literals and a finite field $\mathbb{F}$. An integer w and a set of clauses $\mathbf{C}_w$ that contains all clauses no more than w literals of Lits. $\phi : \mathsf{Lits} \to \mathbb{F}$ is injective.

Inputs:

1. $\mathcal{P}$ holds a clauses $C = \ell_0 \vee \cdots \vee \ell_{k-1} \in \mathbf{C}_w$, defines $\gamma(C)(X) = (X - \phi(\ell_0)) \cdots (X - \phi(\ell_{k-1}))$ and locally computes $c_0, \ldots, c_w$ such that $\gamma(C)(X) = \sum_{i \in [0,w]} c_i X^i$.
2. For each $i \in [0, w]$, two parties use $\mathcal{F}_{\mathsf{ZK}}$ to get $[c_i]$. Two parties output $[\gamma(C)] = \{[c_i]\}_{i \in [0,w]}$

Equal: Both parties send $(\mathsf{PoPEqCheck}, 1, [\gamma(C_0)(X)], [\gamma(C_1)(X)])$ to $\mathcal{F}_{\mathsf{ZK}}$.

X-Res:

1. $\mathcal{P}$ locally computes $W_0(X), W_1(X)$ and ℓ_p, such that $W_0(X) \cdot \gamma(C_0)(X) = \gamma(C_r)(X) \cdot (X + \phi(\ell_p))$ and $W_1(X) \cdot \gamma(C_1)(X) = \gamma(C_r)(X) \cdot (X + \phi(\neg \ell_p))$. Note that the degree of $W_0(X)$ and $W_1(X)$ are bounded by w.
2. $\mathcal{P}$ locally computes $\rho(X) = X - \phi(\ell_p)$, of which the degree is bounded by 1.
3. Two parties use $\mathcal{F}_{\mathsf{ZK}}$ to authenticate all $w + 1$ polynomial coefficients in $W_0(X)$ and $W_1(X)$, and two polynomial coefficients in $\rho(X)$. As a result, two parties get $[W_0(X)]$, $[W_1(X)]$ and $[\rho(X)]$.
4. Using $\mathcal{F}_{\mathsf{ZK}}$, two parties check that the highest coefficient in $[\rho(X)]$ is non-zero, this make sense that $[\rho(X)]$ has degree exactly 1.
5. The prover locally computes polynomial $\bar{\rho}(X) = \rho(1_{\mathbb{F}} - X)$ and commits its 2 coefficients to obtain $[\bar{\rho}(X)]$. Then two parties check that the committed coefficients satisfy $\bar{\rho}(X) = \rho(1_{\mathbb{F}} - X)$.
6. Both parties send $(\mathsf{PoPEqCheck}, 2, ([W_0(X)], [\gamma(C_0)(X)]), ([\gamma(C_r)(X)], [\rho(X)]))$ to $\mathcal{F}_{\mathsf{ZK}}$.
7. Both parties send $(\mathsf{PoPEqCheck}, 2, ([W_1(X)], [\gamma(C_1)(X)]), ([\gamma(C_r)(X)], [\bar{\rho}(X)]))$ to $\mathcal{F}_{\mathsf{ZK}}$.

IsFalse: Both parties send $(\mathsf{PoPEqCheck}, 1_{\mathbb{F}}, [\gamma(C)(X)], [1])$.

Figure 3.4: Our protocol to instantiate $\mathcal{F}_{\mathsf{Clause}}$.

polynomials using a secret witness divisor. The prover can efficiently construct such witnesses, using the pivot variable of the step of resolution.

In detail, for the prover to prove that committed clause C_r is a weak resolvent of clauses C_0 and C_1 on pivot variable X, the prover finds clauses W_0 and W_1 such that

$$W_0 \vee C_0 = C_r \vee x \qquad \text{and} \qquad W_1 \vee C_1 = C_r \vee \neg x$$

W_0 and W_1 can always be defined to be:

$$W_0 = (C_r \cup \{x\}) \setminus C_0 \qquad \text{and} \qquad W_1 = (C_r \cup \{\neg x\}) \setminus C_1$$

The prover then commits polynomials p_0, w_0, p_1, w_1, and p_r, that encode C_0, W_0, C_1, W_1, and C_r, respectively, along with the following polynomial encodings of the literals with variable x:

$$\rho(X) = X - \phi(\ell_p) \qquad \text{and} \qquad \bar{\rho}(X) = X - \phi(\neg \ell_p)$$

The verifier validates the prover has committed encodings of clauses C_0 and C_1 with weak resolvent C_r by attesting the following polynomial equalities over the committed polynomials:

$$\begin{aligned} w_0 \cdot q_0 &= q_r \cdot \rho \\ w_1 \cdot q_1 &= q_r \cdot \bar{\rho} \\ \rho(X) + \bar{\rho}(1_{\mathbb{F}} - X) &= 0_{\mathbb{F}} \end{aligned}$$

The verifier also attests that ρ and $\bar{\rho}$ have degrees of at most one. Sections 3.4.2 to 3.4.2 combined with the attestation of degrees are referred to as the *weak resolution test*.

The following lemma establishes that encodings of clauses in a step of weakened resolution, combined with additional witness polynomials, are solutions to the weak resolution test. It is a key lemma used to prove that the overall protocol (Figure 3.6) is complete.

Lemma 2. *If clause C_r is a weak resolvent of clauses C_0 and C_1 on variable x, then there are polynomials ρ and $\bar{\rho}$ of degree at most one, and polynomials w_0 and w_1 that combined with*

$$q_0 = \gamma(C_0) \qquad q_1 = \gamma(C_1) \qquad q_r = \gamma(C_r)$$

satisfy the weak resolution test.

The following lemma establishes that each solution to the weak resolution test corresponds to some step of weakened resolution. It is a key lemma used to show that the overall protocol is sound in Section 3.4.4, and uses *maximal completely reducible divisors*, introduced in Section 3.3.1.

Lemma 3. *For polynomials q_0 q_1, q_r, w_0, w_1, ρ, and $\bar{\rho}$ that satisfy the weak resolution test, clause $\gamma^{-1}(q_r^*)$ is a weak resolvent of clause $\gamma^{-1}(q_0^*)$ and clause $\gamma^{-1}(q_1^*)$.*

Section 3.5.1 contains a complete proof of Lemma 3 but to see that the lemma is well-defined, note that for each polynomial p, the clause $\gamma^{-1}(p^*)$ is well-defined, because the polynomial p^* is completely reducible (Sec. 3.3.1) and γ is a bijection into the completely reducible polynomials.

3.4.3 Weakened random array access

Our protocol to check resolution proof requires an array to store all literals in all intermediate clauses and the ability to access array elements where the index is private to the prover. This could be instantiated using prior works discussed in Section 3.3.3. However, the overhead would be too high since the bit representation of clause is fairly large: every clause contains up to w literals, each of which requires at least $\log |\mathsf{Lits}|$ bits to encode. As a result each clause needs at least $w \log |\mathsf{Lits}|$ bits to represent. All existing RAM constructions need some sort of bit decomposition on the payload of the array and thus this quickly becomes an huge overhead.

We improved upon a recent prior work [87] for efficient RAM access in ZK in multiple ways. First, as described at the beginning of this section, we only need two operations to the array: append a value to the array and read. In the context of ZK, the prover could precompute all values and thus prepare the whole array ahead of time. During the execution of the protocol, if we need to append v, we read from the location to be written and check that the value equals to v. This way, we only need to support read.

Second, we relax the functionality so that the prover can freely choose the read indices as long it does not read values not appended to the array yet; thus the functionality is significantly weakened. E.g., we can no longer ensure if the prover read the same element twice or not. However, in the context of ZK refutation proof, this weak functionality is sufficient: as long as the protocol arrives to $\perp$, we can always extract a valid UNSAT proof of the formula.

Third, each memory cell contains a complete clause, which consists of w field elements. In [87], the number of AND gates is proportional to the bit-length of the payload; so larger elements lead to a high cost. We improve the access time by applying a universal hash function before the accesses are checked so that the effective bit-length is much shorter. To ensure the soundness, the universal hash function is picked only right before the batch checking.

3.4.4 Putting everything together

In Figure 3.6, we put together our main protocol in the ($\mathcal{F}_{\mathsf{ZK}}$, $\mathcal{F}_{\mathsf{Clause}}$, $\mathcal{F}_{\mathsf{FlexZKArray}}$)-hybrid model. Our protocol assumes that the number of steps in the refutation proof and the width of the proof are public. It proves to the verifier in ZK that the prover has a valid refutation proof.

Functionality $\mathcal{F}_{\mathsf{FlexZKArray}}$

Array initialization: On receiving $(\mathsf{Init}, N, [m_0], \ldots, [m_{N-1}])$ from $\mathcal{P}$ and $\mathcal{V}$, where $m_i \in \mathbb{F}$, store the $\{m_i\}$ and set $f := \mathsf{honest}$ and ignore subsequent initialization calls.

Array read: On receiving $(\mathsf{Read}, \ell, d, t)$ from $\mathcal{P}$, and (Read, t) from $\mathcal{V}$, where $d \in \mathbb{F}$ and $\ell, t \in \mathbb{N}$, send $[d]$ to each party. If $d \neq m_\ell$ or t from both parties do not match or $\ell \geq t$ then set $f := \mathsf{cheating}$.

Array check: Upon receiving (check) from $\mathcal{V}$ do: If $\mathcal{P}$ sends (cheat) then send $\mathsf{cheating}$ to $\mathcal{V}$. If $\mathcal{P}$ sends $(\mathsf{continue})$ then send f to $\mathcal{V}$.

Figure 3.5: Functionality for weak random access arrays in ZK.

Protocol CheckProof

Inputs: Both parties have formula $\phi = C_0 \wedge \cdots \wedge C_{|\phi|-1}$. $\mathcal{P}$ has a proof of refutation $((k_0, l_0), \ldots (k_{R-1}, l_{R-1}))$; Both parties know the length of the refutation proof R and the width of the proof $w = \max_i\{|C_i|\}$.

Protocol:

1. The two parties obtain $[C_i]_{i\in[0,|\phi|-1]}$ using $\mathcal{F}_{\mathsf{Clause}}$; since ϕ is known to both parties, it uses $\mathsf{instance}$ to authenticate the coefficients.
2. $\mathcal{P}$ locally runs the refutation proof verification process and gets $C_{|\phi|-1+i}$ from the i-th iteration. The two parties obtain $[C_i]_{i\in[|\phi|-1,|\phi|-1+R]}$ using $\mathcal{F}_{\mathsf{Clause}}$ using $\mathsf{witness}$ authenticating the coefficients.
3. The two parties send $(\mathsf{Init}, |\phi| + R - 1, [C_0], \ldots, [C_{|\phi|+R-1}])$ to $\mathcal{F}_{\mathsf{FlexZKArray}}$.
4. For the i-th iteration, the two parties advance the proof check by doing the following.
   (a) The prover looks up the tuple (k_i, l_i) from the refutation proof such that $\{C_{k_i}, C_{l_i}\} \vdash_{\text{X-RES}} C_i$.
   (b) Fetching the premise: the prover sends $(\mathsf{Read}, l_i, C_{l_i}, i)$ to $\mathcal{F}_{\mathsf{FlexZKArray}}$; $\mathcal{V}$ sends (Read, i) to $\mathcal{F}_{\mathsf{FlexZKArray}}$, from which the two parties obtain $[C_{l_i}]$. Similarly, the two parties obtain $[C_{k_i}]$ and $[C_i]$.
   (c) Checking the inference: the two parties send $(\mathsf{Xres}, [C_{l_i}], [C_{k_i}], [C_i])$ to $\mathcal{F}_{\mathsf{Clause}}$.
5. After R iterations, two parties use $\mathcal{F}_{\mathsf{Clause}}$ to check that $[C_R]$ equals $\perp$; if the functionality aborts, $\mathcal{V}$ aborts.
6. Two parties send (check) to $\mathcal{F}_{\mathsf{FlexZKArray}}$, if the functionality aborts, $\mathcal{V}$ aborts.

Figure 3.6: Protocol for checking resolution proof.

The protocol consists of three parts: 1) the prover run the verification locally and prepare $C_1, \ldots, C_{R+|\varphi|-1}$; the first $|\varphi|$ clauses are the original formula and the

rest are intermediate clauses; In the i-th iteration, the prover verifies one step of the refutation in ZK by: 2) fetching relevant existing clauses and 3) proving that they derive to C_i. The proof is accepted if the last clause if False.

Theorem 1. *The protocol in Figure 3.6 is a zero-knowledge proof of knowledge of refutation proof.*

We provide a proof of sketch of this theorem in section 3.5.2. Because we model the zero-knowledge proof as a functionality, the simulator plays the role of knowledge extractor in the case of a corrupted prover and plays the role of ZK simulator in the case of a corrupted verifier. Such a formulation was adopted in prior works [132, 220, 75, 29, 225] and was formally discussed by Hazay and Lindell [117].

3.5 Proofs

3.5.1 Proofs of correctness

In this section, we prove the key theorems and lemmas concerning the correctness and security of our protocol, stated in Section 3.4. The following properties of each literal encoding ϕ are used in the proofs of key lemmas and theorems. For each clause C, $\gamma_\phi(C)$ is completely reducible and contains no repeated roots, because each clause is a set of literals and ϕ is injective. For all clauses C_0 and C_1,

$$\gamma_\phi(C_0 \vee C_1) = \gamma_\phi(C_0 \cap C_1)^2 \cdot \gamma_\phi(C_0 \setminus C_1) \cdot \gamma_\phi(C_1 \setminus C_0)$$

Thus $\gamma_\phi(C_0 \vee C_1) \mid \gamma_\phi(C_0) \cdot \gamma_\phi(C_1)$ and

$$\gamma_\phi(C_0 \vee C_1) = \gamma_\phi(C_0) \cdot \gamma_\phi(C_1)$$

when C_0 and C_1 are disjoint. Conversely, for completely reduced polynomials p_0 and p_1 that do not share roots,

$$\gamma^{-1}(p_0 \cdot p_1) = \gamma^{-1}(p_0) \vee \gamma^{-1}(p_1)$$

A proof of Lemma 1:

Proof. Suppose that $C = \ell_1 \vee \ldots \vee \ell_d$. For the *only if* direction note that for all $0 \leq i \leq d$, $\phi(\ell_i)$ is indeed a root of $(X - \phi(\ell_0)) \cdots (X - \phi(\ell_d))$. For the *if* direction. Suppose ℓ is different from all literals in $\{\ell_0, \ldots, \ell_d\}$. The zeros of $\gamma(C)$ are exactly the values $\{\phi(\ell_0), \ldots, \phi(\ell_d)\}$. Since ϕ is injective, it follows that $\phi(\ell)$ is different from all $\{\phi(\ell_0), \ldots, \phi(\ell_d)\}$. □

A proof of Corollary 1:

Proof. Clause C' satisfies the equality $C' = (C' \setminus C) \uplus C$ by the assumption that $C \to C'$. Thus $\gamma(C') = \gamma(C' \setminus C) \cdot \gamma(C)$ because γ distributes over disjoint unions. Thus $\gamma(C) \mid \gamma(C')$, with factor $\gamma(C' \setminus C)$. □

The following lemma will be useful for proving the key lemma for protocol soundness:

Lemma 4. *For completely reducible polynomials p and p' with no repeating roots, if $p \mid p'$, then*

$$\gamma^{-1}(p) \to \gamma^{-1}(p')$$

is valid.

Proof. There is some completely reducible polynomial w with no repeating roots and no common roots with p such that

$$w \cdot p = p'$$

by the assumptions that $p \mid p'$ and that p' is fully reducible with no repeating roots. The clause $\gamma^{-1}(w \cdot p)$ is well-defined, because w and p do not share roots. Thus

$$\begin{aligned} \gamma^{-1}(w \cdot p) &= \gamma^{-1}(p') \\ \gamma^{-1}(w) \vee \gamma^{-1}(p) &= \gamma^{-1}(p') \end{aligned}$$

by Lemma 4. Thus $\gamma^{-1}(p) \to \gamma^{-1}(p')$ is valid. □

A proof of the completeness lemma, Lemma 2:

Proof. The clausal implications $C_0 \to C_r \vee x$ and $C_1 \to C_r \vee \neg x$ are valid by Definition 3. Thus

$$\begin{array}{lll} \gamma(C_0) & \mid \gamma(C_r \vee x) & \mid \gamma(C_r) \cdot \gamma(x) \\ \gamma(C_1) & \mid \gamma(C_r \vee \neg x) & \mid \gamma(C_r) \cdot \gamma(\neg x) \end{array}$$

by Corollary 1 and the definition of γ. Thus there are polynomials w_0 and w_1 such that

$$\begin{aligned} w_0 \cdot \gamma(C_0) &= \gamma(C_r) \cdot \gamma(x) \\ w_1 \cdot \gamma(C_1) &= \gamma(C_r) \cdot \gamma(\neg x) \end{aligned}$$

by the definition of polynomial division. Thus for polynomials $\rho = \gamma(x)$ and $\bar{\rho} = \gamma(\neg x)$, q_0, w_0, ρ, q_1, w_1, and $\bar{\rho}$ satisfy the weak resolution test. □

A proof of the soundness lemma, Lemma 3:

Proof. By the assumption that the given polynomials satisfy the weak resolution test,

$$\begin{aligned} q_0 &\mid q_r \cdot \rho \\ q_1 &\mid q_r \cdot \bar{\rho} \end{aligned}$$

Functionality $\mathcal{F}_{\mathsf{ZKUNSAT}}$

On receiving (ϕ, x) from the prover and (ϕ) from the verifier, where $x \in \{0,1\}^n$, parse x as a refutation proof and check if x is a valid refutation proof of ϕ. The functionality aborts if it is invalid and send pass to both parties otherwise.

Figure 3.7: Functionality for zero-knowledge proofs of refutation.

Thus

$$q_0^* \mid (q_r \cdot \rho)^*$$
$$q_1^* \mid (q_r \cdot \bar{\rho})^*$$

by the definition of maximal completely reducible divisors.

ρ and $\bar{\rho}$ have unique roots a and b such that such that for ϕ the literal encoding that defines γ, $\phi^{-1}(b) = \neg\phi^{-1}(a)$, by the assumption that ρ and $r\bar{h}o$ satisfy the weak resolution test. Thus

$$\gamma^{-1}(q_0^*) \to \gamma^{-1}(q_r^*) \vee \phi^{-1}(a)$$
$$\gamma^{-1}(q_1^*) \to \gamma^{-1}(q_r^*) \vee \neg\phi^{-1}(a)$$

Thus $\gamma^{-1}(q_r^*)$ is a weak resolvent of $\gamma^{-1}(q_0^*)$ and $\gamma^{-1}(q_1^*)$ on the pivot variable that defines literals $\phi^{-1}(a)$ and $\phi^{-1}(b)$, by Defn. 3. □

3.5.2 Proof Sketch of Theorem 4.7

In this section, we provide a proof sketch of Theorem 4.7. We will show that the protocol in Figure 3.6 securely instantiates the ZKUNSAT functionality presented in Figure 3.7, which is a standard ZKPoK ideal functionality where the relation is hard-coded to refutation verification. To prove the security, we will consider both cases of a corrupted prover and a corrupted verifier respectively. In each case, we will construct a polynomial-time simulator $\mathcal{S}$, such that the joint distribution of the output of the adversary and the honest party is indistinguishable between the real world (where our protocol is executed) and the ideal world (where $\mathcal{F}_{\mathsf{ZKUNSAT}}$ is used).

Corrupted prover. In this case, we construct a simulator $\mathcal{S}$ that interactive with the corrupted prover $\mathcal{A}$ and the ideal functionality $\mathcal{F}_{\mathsf{ZKUNSAT}}$. The simulator works as follows:

(1-2) $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{Clause}}$, and record the clauses that $\mathcal{A}$ sent to $\mathcal{F}_{\mathsf{Clause}}$, namely C_i for $u \in [0, |\phi| + R - 1]$. If $\phi \neq C_0 \wedge ... \wedge C_{|\phi|-1}$, $\mathcal{S}$ aborts.

(3) $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{FlexZKArray}}$.

(4-6) $\mathcal{S}$ interactive with $\mathcal{A}$ as an honest verifier and plays the role of $\mathcal{F}_{\mathsf{Clause}}$ and $\mathcal{F}_{\mathsf{FlexZKRAM}}$.

At the end of step (6), $\mathcal{S}$ sends $C_{i\in[|\phi|-1} \wedge \ldots \wedge C_{|\phi|-1+R]}$ to $\mathcal{F}_{\mathsf{ZKUNSAT}}$ and aborts if $\mathcal{F}_{\mathsf{ZKUNSAT}}$ aborts.

Since $\mathcal{F}_{\mathsf{Clause}}$ and $\mathcal{F}_{\mathsf{ZKFlexArray}}$ are always correct, the indistinguishability boils down to the validity of the extracted proof, which is proven in Lemma 3 in Section 3.4.2.

Corrupted verifier.

3.6 ZK verification vs. generation

As mentioned in Section 3.8.2, we constructed Boolean formulas whose unsatisfiability is hard to prove in plaintext, but with small enough refutation proofs so that their unsatisfiability can be demonstrated in ZK in much shorter time. In particular we constructed a formula which the PicoSAT SAT solver [41] can prove unsatisfiable only after 180 seconds, but it takes ZkUnsat roughly 5 seconds to establish its unsatisfiability, once the prover possesses the proof.

We proceed by describing the construction. Let k and n be arbitrarily large integers. We can construct a graph $G_W = (V_W, E_W)$ with $K > k$ nodes, such that it is not 3-colorable, but removing any small number of edges between a few of those nodes, results in a 3-colorable graph. We call this graph the "witness cycle". Define $G_i = (V_{i,1} \cup V_{i,2} \cup V_{i,3}, E_i)$, for $i \leq K$, to be a complete 3-partite graph of size $3n$, where each partition $V_{i,j}$ of the graph is of size n. For each node v_p in the set of nodes V_W of the witness graph, for $1 \leq p \leq K$, let v_p be connected to all the nodes in $V_{p,1}$ and $V_{p,2}$, but none of the nodes in $V_{p,3}$. We call each of these complete 3-partite graphs a "noise graph". Notice that connecting the nodes of a graph H to the copies of the noise graphs in this way, does not affect the 3-colorability of the overall graph. In other words the resulting graph is 3-colorable if and only if the original graph H is 3-colorable. As, such, given that the witness cycle G_W is not 3-colorable by construction, the resulting graph is also not 3-colorable.

Using standard reduction techniques we can convert each instance for any K, n, into a SAT formula (whose size is polynomially bounded) that is satisfiable if and only if the constructed graph is 3-colorable. The subformula that corresponds to the witness cycle G_W, is also a witness to the unsatisfiability of the input formula. The purpose of the noise graphs G_i is to distract the SAT solvers away from quickly producing a refutation proof. A small K corresponds to a smaller proof size, and a larger n corresponds to longer times for state-of-the-art SAT solvers in finding a refutation. As such, by keeping K small enough and increasing n arbitrarily enough, we can construct a formula where a SAT solver takes longer to find the solution and generate a proof than the time it takes to verify in ZK that the formula is unsatisfiable, when the prover already knows the refutation proof.

3.7 ZK proofs of secret programs

ZK validation of resolution refutations could enable multiple realistic protocols for ZK proofs of safety of a secret program. One protocol that would directly extend known protocols for encoding programs as SAT formulas [85] would be to encode a program consisting of n instructions as **(1)** n copies of a public formula that multiplexes over designated selector variables to execute some valid instruction and **(2)** secret clauses that constrain the selector variables. Such an encoding reveals no information about the program other than the number of instructions that it contains. A resolution refutation of the conjunction of the formula and a formula satisfied only by erroneous executions proves that the secret program is safe. For i the maximum size of a formula needed to model any of the m instructions, the refuted formula has size $O(i \cdot m \cdot n)$.

However, the ability to validate resolution refutations enables protocols for proving safety of secret programs that in some cases, would be even more efficient; instead of imposing a strong requirement on the *intensional* structure on a relatively large formula, we can impose independent *extensional* requirements on a smaller formula. In particular, we can designate particular propositional variables as modeling state before and after each of the n instructions and ensure that each of the instructions is valid by validating a logical implication, whose validity is witnessed by a resolution refutation. The resulting protocol involves validating refutations of n formulas each of size $O(i \cdot n)$, which can be proved in parallel.

3.8 Implementation and Evaluation

This section contains details of our implementation and the results of its empirical evaluation. We will openly release our implementation to accompany the final publication of our results. All of our benchmarks were performed on AWS instances of type `r5b.2xlarge` with 64 GB of memory, 16 vCPUs and a 10 Gbps network connection between the prover and the verifier. We used an instance with a large amount of memory because our largest benchmark (described below) uses more than 32 GB of memory.

3.8.1 Implementation and optimization

We implemented and evaluated our protocol as a tool, named ZkUnsat, using the EMP-toolkit interactive zero-knowledge proof library for Boolean/arithmetic circuits and polynomials [217] and the high-performance library NTL [200] for arithmetic on polynomials over finite fields. Because the underlying ZK protocol in EMP is a constant-round interactive ZK, our whole protocol is also constant-round. In ZkUnsat, we instantiated the protocol on the binary field $\mathbb{F}_{2^{128}}$, under which field operations can be efficiently implemented using the `CLMUL` instruction; we represented the indices of clauses using 20-bit integers, which support refutation proofs of length up

to one million.

To verify refutations of practical formulas, we aggressively optimized our implementation's memory usage. When verifying practical resolution proofs in the clear, memory usage is typically moderate; however, when verifying them in ZK, it is significantly higher due to the use of information-theoretic MACs [87]. We implemented protocol components to to store only data that is essential to complete the rest of validation. Recall that for each resolvent, the prover must prepare and commit a set of polynomials (see Section 3.4). Storing witnesses for all resolvents simultaneously would consume a prohibitive amount of memory. However, the witness of a resolvent is only used when that resolvent is being validated. Thus, in our implementation, the prover generates and commits the witness only before checking the corresponding resolvents. Moreover, the witness is stored in memory only during the validation of its corresponding resolvent.

3.8.2 Performance per phase

Verifying a refutation of a formula φ consists of three phases: **(1)** loading all clauses deduced in the refutation; **(2)** fetching clauses as premises; and **(3)** validating steps of deduction (see Figure 3.6). We empirically evaluated the relationship between the cost of performing each of the phases and the size of practical refutations, specifically the size of the formulas $|\varphi|$, the refutation's length l, and the refutation's width w, in addition to their effect on overall performance.

Instance generation In order to benchmark the distinct phases of our protocol, we generated refutations of particular sizes by repeating clauses in a small proof. In more detail, starting from a refutation of formula φ of length l, we generated a refutation of formula φ' with $|\varphi'| \geq |\varphi|$, of length $l' \geq l$. To do so, we added $|\varphi'| - |\varphi|$ copies of an arbitrary clause in φ and added $l' - l$ copies of an arbitrary resolvent in the proof. Because the width of a proof is a public parameter provided by the prover, we generated one proof for each combination of formula size $|\varphi| \in \{2000, 2200, \cdots, 3000\}$, small length $l = 50$ or large length $l \in \{2000, 3000, \cdots, 8000\}$, and width $w \in \{100, 150, 300, 450\}$. They cover a large range of parameters that can be accurately evaluated and can also tell us the performance trend of our protocol.

Input formulas size We measured the growth of the total verification time when the size of input formulas increases under fixed lengths l and widths w; Figure 3.8 contains the evaluation results. For each length and width, verification time changes negligibly as the size of the input formula increases. Furthermore, to demonstrate that showing unsatisfiability of a large formula in plaintext can be harder than verifying an existing refutation proof in ZK, we constructed formulas where the former process takes more than 180 seconds using PicoSAT, whereas the latter takes roughly 5 seconds with ZkUnsat (see Section 3.6).

Refutation width A refutation's width determines the degree of the polynomials that encode clauses maintained by the protocol. To evaluate the effect of width on

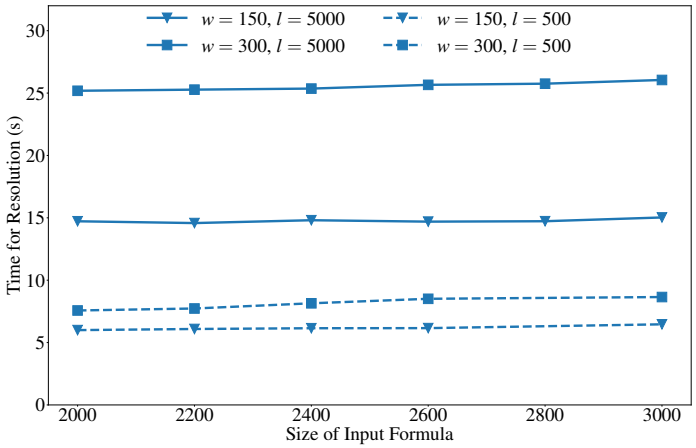


Figure 3.8: Clause verification time vs. size of input formula. The total time for verifying a resolution proof changes negligibly with an increase in the size of the input formula, under various fixed refutation lengths l and widths w.

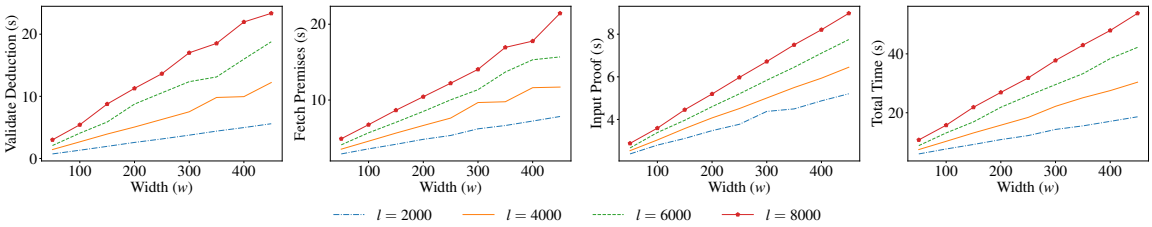


Figure 3.9: Verification time vs. refutation width. Plots of phase time and total performance vs. width w, for various refutation lengths $l \in [2,000, 8,000]$, with a fixed formula size of $|\varphi| = 3,000$. The times spent inputting the proof, fetching premises, and checking resolution steps are all linear in the width.

protocol performance, we measured the protocol's verification time under varying widths, with fixed input formula size $|\varphi| = 3000$.

Figure 3.9 contains the evaluation's results. In practice, verification time is linear in the refutation's width. Furthermore, the times of each of the protocol's three phases are linear in the width, as well. We can also see that the majority of the time is spent on validating deduction and fetching premises, two main parts that our work optimized. In addition, compared to the protocol's other phases, the time taken to input the proof rises less significantly with width.

Refutation length A refutation contains a series of resolvents, where the deduction of each by resolution must be verified. In principle, the refutation's length l determines the number of groups of either bit-vectors or polynomials that are verified as encodings of steps of resolution is linear in the refutation length l. We evaluated our implementation's actual performance versus refutation length, under different fixed refutation widths. Figure 3.10 contains the results of our evaluation, which demonstrate that in practice, verification time is indeed linear in refutation length. Moreover, the cost for inputting the proof only shows a limited increase when the length l grows, while the increase of time cost for checking inference and fetching

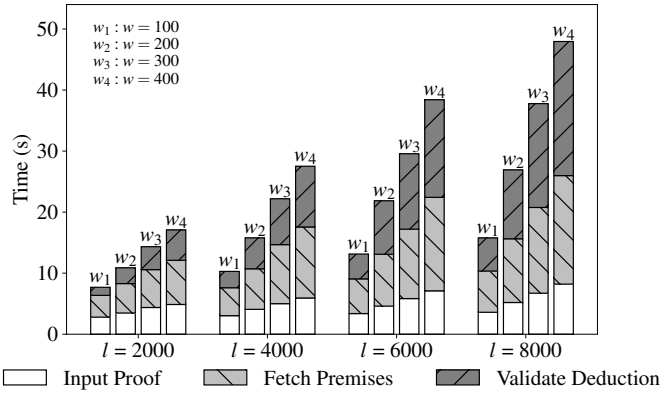


Figure 3.10: Verification time vs. refutation length. For different fixed refutation widths w, verification time is linear in the refutation's length l. As the length grows, the increase in time of inputting proof is less than the increase for fetching premises and checking resolution. Furthermore, as length increases, the time for fetching premises and checking resolution dominates verification time.

Len.	Width	Comm. (MB)	Len.	Width	Comm. (MB)
2,000	150	75.68	3,000	100	72.91
2,000	300	142.40	3,000	200	136.20
2,000	450	200.87	3,000	300	209.95

Table 3.1: Communication cost vs. length and width. The amount of data communicated is nearly proportional to the refutation's area.

premises are adequately visible.

Communication cost We evaluated the communication costs for verifying refutations of different length and width; Table 3.1 contains the evaluation's results. Similar to verification time, the amount of communicated data grows proportionally to the refutation's length and width; refutations with similar areas were verified with similar communication costs.

Clause representations To evaluate the effect of representing refutation clauses as polynomials, we compared protocols that use polynomials to a generic protocol that represents clauses as bit-vectors (see Section 3.4.1). To do so, we increased the number of literals Lits from 2^8 to 2^{15} and measured the time required by the generic protocol with length $l = 3,000$ and input formula of size $|\varphi| = 1000$.

Figure 3.11 contains the evaluation's results. As expected from a complexity analysis of the generic protocol, the time used by its implementation in practice increases linearly with $|\mathsf{Lits}|$, while the polynomial-based protocol's verification time is unaffected. The polynomial-based protocols perform better when the set of literals is suitably large: the polynomial-based protocol with $w = 100$ outperforms the generic methods when $|\mathsf{Lits}| = 2^{11}$. A proof with number of literals $|\mathsf{Lits}| = 2^{15}$ and large width $w = 400$ is verified by the generic protocol in over 80 seconds, but verified by the polynomial-based protocol in only 20 seconds.

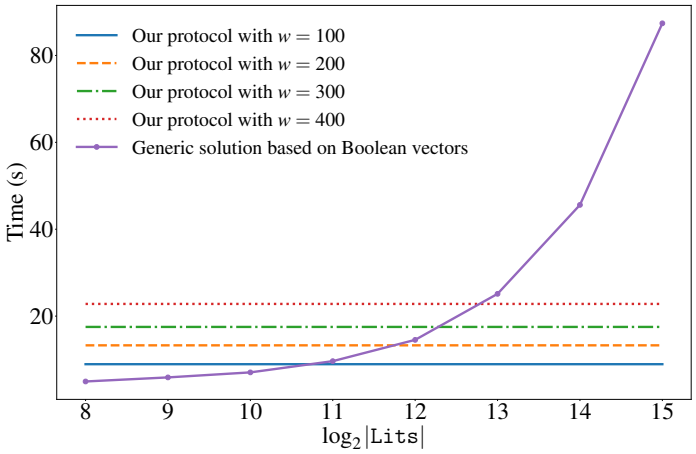


Figure 3.11: Time vs. number of literals, per clause representation. A plot of verification time of different protocols vs. the number of variables used by the input formula, on refutations with fixed length $3,000$, which was chosen as sufficiently large to observe an effect. The purple line depicts the performance of a protocol that represents clauses as bit-vectors and reveals nothing about the proof; the other lines depict the performance of protocols that represent clauses as polynomials and additionally reveal various upper bounds on the refutation's width.

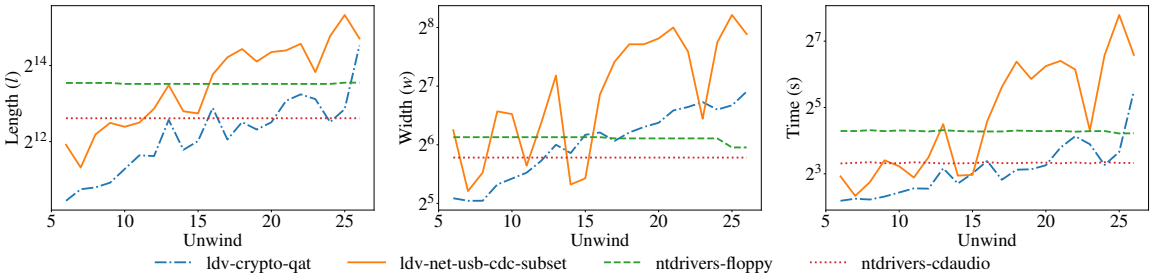


Figure 3.12: Verification features vs. bound on loop unwindings for drivers. Plots of refutation length, width, and verification time vs. bound on loop unwindings for a set of Windows NT and Linux drivers.

3.8.3 Verifying safety-critical proofs in ZK

We evaluated ZkUnsat on refutations generated from benchmarks in corpus of the *Competition on Software Verification (SV-COMP)* [40], and major competition for evaluating program verifiers on practical and challenging programs. From the complete SV-COMP corpus, we selected benchmarks of two types: **(1)** system drivers, selected to evaluate ZkUnsat's practicality and **(2)** programs that induce large refutations, to evaluate ZkUnsat's scalability. The system drivers benchmarks are real-world implementations of drivers, instrumented with code annotations that define the correct behavior. As an illustration, consider the following example: if at some point in a program two system variables need to be equal, the program is instrumented with the if statement that checks this equality. If they are not equal, then this should raise an alert. These alerts are typically implemented as a call to a special "error-code" procedure. In this example, to verify that two variables are equal at the given

program point means to formally prove that the error procedure is never invoked in the instrumented code. In the jargon of the verification community, we need to prove that the error code is never reached.

One prominent approach to program verification [23, 24], given program P, compiles it to a Boolean formula φ such that each execution of P corresponds to an satisfying assignment of φ. Additionally, the program property is compiled to a second Boolean predicate ψ that is satisfied by all program runs in which the property is preserved. Thus, the program is safe if the formula $\varphi \rightarrow \psi$ is valid or, equivalently, the formula $\varphi \wedge \neg\psi$ is unsatisfiable. A refutation of $F \wedge \neg P$ is this a formal argument that the program P is correct.

The SV-COMP verification benchmarks are compiled to Boolean formulas using the C Bounded Model Checker (CBMC) [148]. Compilation from C code to a Boolean formula is relatively straightforward, with the exception of unbounded looping or iteration. To cope with such control structures, a *Bounded Model Checker (BMC)* (BMC) [43] takes an additional non-negative integer unwind and unwinds all loops at most unwind times, generating the program that safely halting if it to attempts to execute unwind $+ 1$ iterations. The resulting program does not model all of the given program's executions, but in practice there is considerable practical value in verifying even bounded programs up to even just a few unwindings.

We evaluated ZKUNSAT's performance on refutations corresponding to verification problems for proving unreachability of error locations, with unwindings of unwind in $\{6, 7, \cdots, 26\}$. In practice, the small unwinding is usually sufficient to test properties of the program [180, 14]. All of the verification problems that we evaluated were obtained from the public SV-COMP repository:

- `ldv-crypto-qat`[2]: verification of safety for Intel(R) QuickAssist (QAT) crypto poll mode driver for analysis of pointer aliases and function pointers.

- `ldv-net-usb-cdc-subset`[3]: safety verification for the Linux Simple USB Network Links (CDC Ethernet subset) driver by analysis of pointer aliases and function pointers.

- `ntdriver-floppy`[4]: The code is instrumented with control labels that describe the correctness behavior of a Window NT floppy disk driver. The verification task boils down to reachability analysis and proving that the error code is never reached..

- `ntdriver-cdaudio`[5]: The specification and verification problems are defined similarly to the case of `ntdriver-floppy`.

[2] `github.com/sosy-lab/sv-benchmarks/blob/master/c/ldv-linux-4.2-rc1/linux-4.2-rc1.tar.xz-08_1a-drivers--crypto--qat--qat_common--intel_qat.ko-entry_point.cil.out.c`

[3] `github.com/sosy-lab/sv-benchmarks/blob/master/c/ldv-linux-4.2-rc1/linux-4.2-rc1.tar.xz-32_7a-drivers--net--usb--cdc_subset.ko-entry_point.cil.out.c`

[4] `github.com/sosy-lab/sv-benchmarks/blob/master/c/ntdrivers/floppy.i.cil-1.c`

[5] `github.com/sosy-lab/sv-benchmarks/blob/master/c/ntdrivers/cdaudio.i.cil-1.c`

Program	Len. (K)	Width	Time (s)
inv-square-int	194	414	172.5
rlim-invariant	481	198	1943.3
sin-interpolated-smallrange	375	308	2571.8
interpolation	135	790	3771.6
inv-sqrt-quake	182	749	5764.1
zonotope-loose	35	2887	9996.9
zonotope-tight	64	2887	11143.3
interpolation2	600	1047	OOM

Table 3.2: Length, width, and verification time in the large. The performance of ZkUnsat on large proofs for proving properties of benchmark programs with floating point computation. Column "Time (s)" contains the performance of ZkUnsat in seconds; column "Len. (K)" contains the refutation's length, in multiples of $1,000$; column "Width" contains the refutation's width. The value "OOM" denotes that ZkUnsat ran out of memory.

Refutations of the generated formulas were generated using the PicoSAT SAT solver [41]. Figure 3.12 reports the features of refutations and the performance of ZkUnsat vs. the chosen unwinding bounds. Refutation length and width either increased sharply with unwinding bounds or remained constant. We expect that the latter occurs due to optimizations within both CBMC and PicoSAT. Verification time is determined by refutation area, as in the evaluations described above.

The results demonstrate that ZkUnsat can be used to verify arguments of safety of practical programs in ZK; ZkUnsat can verify the safety and correctness of all the presented drivers in under five minutes. The largest refutation corresponds to the verification of `ldv-net-usb-cdc-subset` with loops unwound 256 times; ZkUnsat verifies this refutation in under 256 seconds.

To evaluate ZkUnsat's scalability, we evaluated its performance on large refutations of formulas corresponding to the verification of programs that use floating-point operations.[6]. Out of a total of 58 benchmarks, we selected benchmarks whose formulas could be extracted from the program and solved in under 30 minutes, and whose proofs have length at least $l \geq 10,000$ and a width of at least $w \geq 100$. We omitted benchmarks whose generated refutations were too large to be parsed within allocated memory.

The results, given in Table 3.2, demonstrate that ZkUnsat can verify proofs of moderate length and of width as large as 2.8K in an amount of time that would be useful in multiple cases: under three hours. The results also give insight into ZkUnsat's current limitations: when attempting to verify a refutation containing 600K resolvents and with width $1,047$, our implementation exhausted the allocated memory.

The verification time and memory requirements depend on the clausal length and width of the proof. To see if ZkUnsat is practical, it is also important to learn the distribution of proof length/width for real programs. We uniformly sampled a set

[6] `github.com/sosy-lab/sv-benchmarks/tree/master/c/float-benchs`

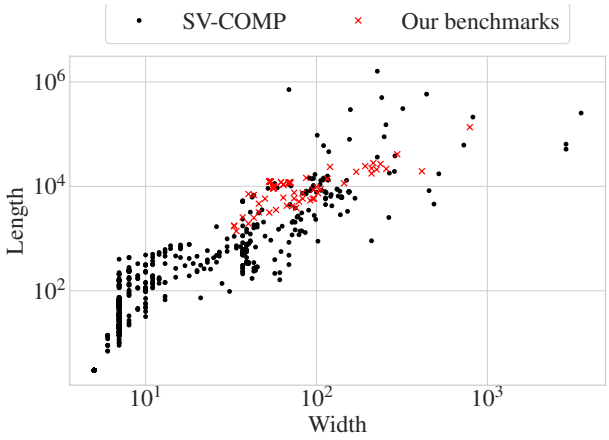


Figure 3.13: Distribution of the clausal length and width of formulas for real programs in SV-COMP.

of SV-COMP verification tasks that generate unsatisfiable SAT formulas, setting the parameter unwind to the standard value 2. The distribution of proof length/width is depicted in Figure 3.13, alongside the chapter's examples. The result shows that the scale of formulas in our benchmarks can cover 804 of 814 (98.7%) verification tasks from SV-COMP.

3.9 Related work

There are several implementations of secure 2 party pattern matching protocols [27, 228, 213, 116, 145, 68, 194]. They support typical string operations, such as exact matching, substring matching, approximate matching and wildcard matching. These protocols do not support more expressive regular expression, and hence have limited applicability in many pattern matching applications. In particular, DNA tandem repeats that serve as genetic markers to track inheritance in families cannot be expressed as a simple string matching; they require a formalism that supports the repetition.

Contrary to those implementations, there are also implementations that support regular expressions [231, 176, 151, 89, 211]. They are based on the secure evaluation of deterministic finite automata (DFA). The complexity of these protocols is at least linear to the size of the DFA, which can be exponential to the size of the regular expression. Some works [151, 195] do present secure protocols for general NFA evaluation, which can be used for regular expression pattern matching. However, they have expensive homomorphic encryptions or they have communication complexity at least $\Omega(mn^2\kappa)$ (where κ is the security parameter) which is asymptotically inferior to both our proposed protocols.

To the best of our knowledge, there is no prior work studying zero-knowledge pattern matching for regular expressions. However, there are works on proving the properties of committed data in zero knowledge, which can be viewed as a more general functionality, but with less attention on the formalization of the pattern match. In the work of zero-knowledge middleboxes (ZKMB) [110], the authors provide

protocols based on zk-SNARK to prove the compliance check of domains hidden in encrypted DNS, e.g. DNS-over-TLS (DOT) or DNS-over-HTTPS. The work of DECO [234] builds a decentralized oracle which allows a user to prove to third parties the provenance of data encrypted in a TLS session. Additionally, a recent work Mystique [221] shows an application that a party can first publicly commits to a machine learning model, then prove that it correctly conducts an image inference in zero-knowledge without revealing the model. In the domain of proving knowledge of committed values, the LegoSNARK [54] is a framework for commit-and-prove zk-SNARKs (CP-SNARKs) which proposes efficient CP-SNARKs for popular Pedersen-like commitments and polynomial commitments.

3.10 Conclusion

We have presented a novel protocol for proving knowledge that a given propositional formula is unsatisfiable while revealing minimal information about the known supporting argument, structured as a resolution refutation. The protocol's key features are the use of **(1)** a sub-protocol for efficiently executing RAM programs in zero knowledge, used to hide which facts derived from the formula are used at which steps of the argument and **(2)** an encoding of propositional clauses as arithmetic polynomials, which allows us to aggressively minimize costs by revealing only the refutation's length and width. Our empirical evaluation of a prototype implementation indicates that the protocol can be used to prove the safety and correctness of safety-critical software (specifically, system device drivers) while keeping secret the details of why the software is correct.

Future work and challenges. A compelling direction for future work is to develop a protocol that proves the safety of a program that is itself kept secret: this could be achieved by extending the presented protocol to verifiably translate a secret program to a formula satisfied by the hypothetical unsafe executions, and use the existing protocol to prove that no such assignment in fact exists. We believe that such a formula could be generated either by relating a secret formula to the syntactic structure of a secret program that at each control point steps by executing some instruction secretly chosen from a public set of faithful instruction models, or by validating additional resolution proofs that prove that each instruction formula models program instruction semantics faithfully. By including public instruction models or symbolically proving that each instruction formula faithfully models error-triggering conditions, secret programs could be proved to satisfy properties that require that no instruction in the program performs an error, e.g. accessing memory out of bounds, overflowing arithmetic, or dividing by zero. Both such strategies would draw on the wealth of existing work in automated theorem proving and symbolic reasoning driven by the software verification community. In general, verifying stateful program properties may require verifying a program in which assertions have effectively been inlined. A verifier could potentially inline assertions correctly but blindly by following

a protocol based on *Multi-Party Computation(MPC)* [227, 96], where the prover and verifier input assertions and programs, respectively.

Resolution is one of the proof systems for the unsatisfiability problem that is well-studied and implemented. Other alternatives remain unexplored, among which Groebner proof system [64] is of particular interest. In a Groebner proof system, the witnesses are in the form of polynomials over a finite field and thus could have natural encodings in ZK. On the other hand, the translation from clauses to polynomials will introduce additional overhead that could affect the overall performance.

Chapter 4

Private regular expression pattern matching

Regular expression pattern matching is one of the fundamental query operations in computer science and formal languages: checking if an input string is in the language defined by an input regular expression. A wide range of applications, such as network intrusion detection and policy checking for DNS queries, depend on such queries and furthermore, are increasingly concerned with privacy issues. In this chapter, we provide two protocols for private regular expression pattern matching based on the oblivious stack and oblivious transfer protocols, respectively. The former results in the best-known asymptotic complexity for regular expression pattern matching, while the latter leads to optimal numerical performance. In these protocols, the privacy of both strings and regular expressions is protected. In addition, we introduce a protocol for zero-knowledge proofs of pattern matching, where the given regular expression is public to both parties, but the input string is kept private. This effort initializes the work on zero-knowledge proofs in this domain, and our experimental results show that our protocol is of practical proof size under the setting of policy checking for encrypted DNS queries.

4.1 Introduction

A pattern matching algorithm takes as input a string and a pattern and checks whether the given pattern appears in the string. It has been widely used in many areas, including bioinformatics [39], database search engines [52], intrusion detection systems [212], text processing [149], and digital forensics [33]. Matching patterns are commonly represented as regular expressions because they are able to match text using a structured pattern and are commonly used in search engines, word processors, and programming languages. The practicality of regular expression matching is demonstrated in two key scenarios that drive this research: monitoring and enforcing DNS query policies and detecting network intrusions. DNS queries can be monitored for banned URLs and rejected by network administrators, leveraging the

expressiveness of regular expressions with low false positive rates and high expressiveness [175, 16, 224]. Intrusion detection systems (IDS) can also use regular expressions to identify potential malicious network packets, reducing the task of detecting network attacks to a pattern-matching problem for incoming packets [203].

Many of these applications have strict privacy requirements. In the DNS setting, exposing the DNS queries to the network administrator would violate the privacy of users. Enforcing the policy check by examining all outbound packets also prevents the deployment of recent privacy-preserving DNS techniques such as DNS-over-HTTPS [122]. However, in this case, the policy (e.g. blocklist) may not need to be kept secret. For example, under the Children's Internet Protection Act, schools are suggested to censor Internet browsing from the school network but should make public the criteria used to block websites [172, 109]. On the contrary, the network intrusion detection example would prefer keeping the secrecy for both the input string and pattern. The users tend to keep packets secret from IDS. Meanwhile, the IDS may view its regular expression database as its intellectual property, so there is a strong motivation for them to limit the database's access [199]. In summary, solving a pattern-matching problem in privacy-preserving settings is becoming an increasingly important problem, while the privacy requirements vary across different use cases.

In this paper, we focus on privacy-preserving regular expression matching. We first consider the settings where the pattern is publicly known. In these settings, a prover holds a private string and generates a proof, which is used to validate if that string is accepted or rejected by the given public pattern. It is required that the proof reveals no information about the input string. We refer to this as *ZK-regex* and realize it by using zero-knowledge proofs (ZKP) [100]. A typical application is the blocklisting of DNS queries demonstrated by zero-knowledge middlebox (ZKMB) [109]. We next consider the settings where a pattern holder and a string holder want to verify whether the pattern matches the string. Both parties should not learn anything beyond the result (true or false) and the length of inputs. We refer to this as *secure-regex* and realize it by secure two-party computation (2PC) protocols [226]. A typical application is the privacy-preserving network intrusion detection described before.

Before introducing our solution, we first describe how to do regular expression matching in plaintext efficiently. Thompson proposed a method [209], in which the regular expression is converted into a Thompson nondeterministic finite automata (TNFA) with a size linear in the size of the expression (cf. Section 4.2.1). TNFAs can be regarded as directed graphs with states as nodes and transitions as labeled edges. This representation reduces pattern matching to finding a valid path from the initial state to an accepting state in the graph for the input string. Thompson's algorithm constructs a set $\mathcal{S}_i$ of all reachable nodes through valid paths for the first i characters of the input string. If the pattern in the regular expression matches the input string, an accepting state should be in $\mathcal{S}_m$ where m is the length of the input string. We use m and n to denote the string and pattern length throughout this paper.

We demonstrate the TNFA simulation for pattern matching in regular expressions using the example in Figure 4.1. The figure shows a TNFA for the regular expression $((A)^*B|C)$ where S_0 is the initial state and S_8 is the accepting state. For the input string AB, which matches $((A)^*B|C)$, there is a valid path $S_0 \xrightarrow{\varepsilon} S_1 \xrightarrow{\varepsilon} S_2 \xrightarrow{A} S_3 \xrightarrow{\varepsilon} S_4 \xrightarrow{\varepsilon} S_5 \xrightarrow{B} S_6 \xrightarrow{\varepsilon} S_8$. The path is considered valid as the labels along the path from the sequence $\varepsilon^*A\varepsilon^*B\varepsilon^*$, where the superscript '$*$' indicates any number of repetitions. The following steps explain the process of determining if the given TNFA accepts AB:

1. Set $\mathcal{S}_0$ is initialized to $\{S_0\}$, and then all states reachable from $\mathcal{S}_0$ through ε-labeled edges are added. This results in $\mathcal{S}_0 = \{S_0, S_1, S_2, S_4, S_5, S_7\}$.
2. As $S_2 \in \mathcal{S}_0$ and there is an edge (S_2, S_3) labeled with A, set $\mathcal{S}_1$ is initialized to S_3 for the first character A in the string AB. This is then extended to $\mathcal{S}_1 = \{S_1, S_2, S_3, S_4, S_5\}$ by including states reachable through ε-labeled edges.
3. The same is repeated for the second character B in the string AB, resulting in $\mathcal{S}_2 = \{S_8\}$.
4. Since the accepting state S_8 is in $\mathcal{S}_8$, there is a valid path in the TNFA from S_0 to S_8 for AB, meaning $((A)^*B|C)$ matches AB.

The example demonstrates the two-step process of constructing each set $\mathcal{S}_i$: initializing $\mathcal{S}_i$ using $\mathcal{S}_{i-1}$ and character-labeled edges, and then extending $\mathcal{S}_i$ by adding states reachable through ε-labeled edges. The first step is called the *character transition* and the second step is the *epsilon transition*. Note, however, that from the sequence of sets $\mathcal{S}_0, \mathcal{S}_1, \ldots$ one can reconstruct both, the input string and the TNFA. In this paper we develop efficient protocols for regular expression matching, based on the above TNFA simulation, while maintaining privacy of input.

Challenges. Both ZKP and 2PC securely process an algorithm in the circuit model, by first translating any algorithm into a Boolean or arithmetic circuit. Since the topology of the circuit needs to be public, it requires that the circuit itself does not reveal any private information. Equivalently, it means the access pattern of the underlying algorithm should be independent of inputs except for their length. In the case when the algorithm of plaintext evaluation leaks inputs by its access pattern (as shown in the previous example), additional oblivious data structure or oblivious algorithms are utilized to enhance the privacy.

The pattern matching in plaintext consists of character transition and epsilon transition (details in Section 4.3). In privacy-preserving settings, the character transition can be executed directly through linear scans and equality checks, which are easily handled by ZKP or 2PC protocols. Thus, the main challenge of this work is a design of a data-oblivious epsilon transition. Epsilon transitions involve searching for all reachable states in the TNFA. In the plaintext execution, this search can be performed using a depth-first search (DFS). However, the best-known oblivious DFS for general graphs requires a circuit size of $O(n^2)$, leading to a total complexity of $O(mn^2)$ to process a length-n regular expression and a length-m string [45]. This complexity is not feasible for practical real-world applications.

Our contributions. We briefly describe how we tackle the above challenges.

TNFA simulation via two linear scans. We propose a pattern-matching algorithm that leverages two passes of linear scans for epsilon transition. Our key insight is that the TNFA is a sparse graph, where all accessible nodes can be reached through a path with at most one backward edge in the epsilon transition. This observation enables us derive the elements of the set $\mathcal{S}_i$ through two passes of linear scans, with the access pattern of the epsilon transition being independent of the input string. This property makes our algorithm directly applicable to ZK-regex scenarios. However, the access pattern is dependent on the TNFA graph during the retrieval of previous states in epsilon transitions, requiring further protection in secure-regex scenarios, where the TNFA graph is derived from the input regular expression.

ZK-regex. We present a ZK-regex protocol that enables a prover to prove whether a public regular expression matches a string without revealing any information about the string. Define $\log|\Sigma|$ to be the bit length of the alphabet. The circuit for our ZK-regex protocol is derived from our two-linear scan algorithm and is of size $\mathcal{O}(mn\log|\Sigma|)$. We leverage the fact that our TNFA simulation algorithm is already oblivious to input strings so that no additional data-oblivious algorithm is needed to protect the access patterns. Our ZK-regex circuit fits into any general-purpose ZKP framework. In addition, we describe the application of examining encrypted Transport Layer Security (TLS) packets using ZK-regex. It allows a packet sender to prove whether a specific pattern matches her packet without exposing the packet's content, enabling effective filtering of malicious traffic while preserving the privacy of users.

Secure-regex. To minimize the need for additional data-oblivious operations to protect the graph structure, we further investigate the properties of TNFAs and prove a bound on the maximum degree of each node. By limiting the number of predecessors and successors of each node, we develop two efficient two-party protocols for secure-regex. Define κ to be the computational security parameter and σ to be the alphabet. The first protocol, based on the oblivious stack (OS), is a constant-round protocol with a communication complexity of $O(\kappa mn(\log n + \log|\Sigma|))$, outperforming the previous best-known result of $O(\kappa n^2(m + |\Sigma|))$ [195]. The second protocol, based on 1-out-of-n+1 oblivious transfer (OT), achieves a communication overhead of $\mathcal{O}(mn(n + \kappa\log|\Sigma|))$ with a linear number of round-trips, while incurring a higher asymptotic complexity. However, it outperforms the first protocol in scenarios with low network latency and short input regular expressions.

Implementation and evaluation. To empirically evaluate our proposed protocols, we implemented ZK-regex over TLS and measured the proof size and prover time. We demonstrate the practicality of our ZK-regex protocol by running our implementation on a set of regular expressions used for DNS filter Pi-hole [6]. For a 128-byte string, a proof for the longest pattern in Pi-hole (97 bytes) can be generated in 0.57s with a size of 397KB. Our secure-regex protocols attain the best-known results for two-party regular expression matching, and their practicality is demonstrated in real-world

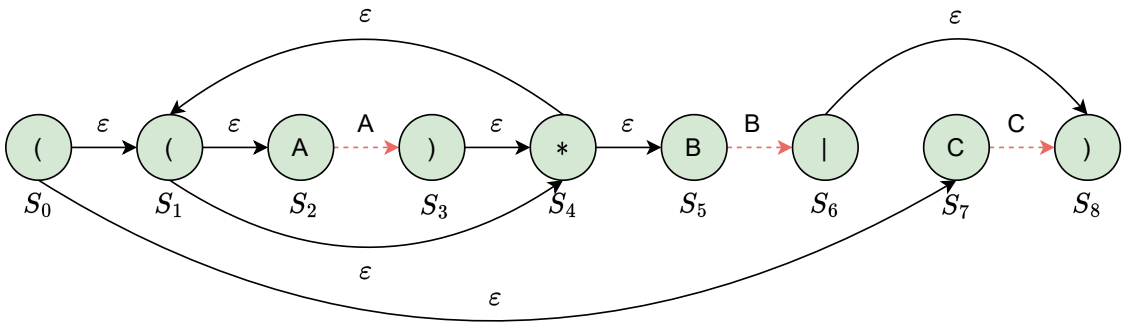


Figure 4.1: Example TNFA for regular expression $\mathsf{re} = ((A)^*B|C)$. The solid arrows are directed edges in G and define epsilon transitions of the TNFA. The dashed arrows are the character transition.

intrusion detection based on SNORT PCRE [58]. The OT-based protocol performs matching on a 512-byte string and a 63-byte pattern in 4s in a LAN with 100Mbps bandwidth. In a WAN with 60ms latency, our OS-based protocol performs matching on a 512-byte string and a 15-byte pattern in 4s. We also compare our secure-regex implementations that use OS-based and OT-based protocols. We test them on varying network and input conditions, and we report the performance difference.

4.2 Preliminaries

4.2.1 Regular Expressions and Thompson NFAs

In this section, we introduce formal definitions for regular expressions and for NFAs. We discuss how NFAs can also be viewed as a labeled directed graph which will be critical for our constructions. Furthermore, we present Thomspon's construction [209] which shows how any regular expression can be converted into an "equivalent" NFA. By equivalent, we mean the NFA accepts a string if and only if the corresponding string can be generated by the regular expression.

Regular expression. A regular expression (regex) on the alphabet Σ consists of characters in Σ and meta characters $\{\ |, (,), *\ \}$. Let ε and ϕ denote the empty string and the empty set, respectively. A regular expression re describes a set of strings (or language) $L(\mathsf{re})$ over Σ recursively:
1. basic case: $L(\phi) = \phi$, $L(\varepsilon) = \{\varepsilon\}$ and $L(a) = \{a\}$;
2. concatenation: $L(\mathsf{re}\ \mathsf{re}') = \{xx' | x \in L(\mathsf{re}), x \in L(\mathsf{re}')\}$;
3. union : $L((\mathsf{re}|\mathsf{re}')) = L(\mathsf{re}) \cup L(\mathsf{re}')$;
4. loop: $L((\mathsf{re})^*) = \bigcup_{k \in \mathbf{N}} \{x_0 x_1 \cdots x_k | x_i \in L(\mathsf{re})\} \cup \{\varepsilon\}$.

Nondeterministic finite automata. A nondeterministic finite automata (NFA) N is defined as a 5-tuple $(\mathcal{S}, \Sigma, \delta, S_0, \mathcal{S}_A)$. S is a set of states and Σ is the alphabet

set. $\delta : \mathcal{S} \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^S$ is a transition function that maps a state along with character to a set of states. For states S, S' we write $S \xrightarrow{\varepsilon} S'$ if either $S = S'$ or there exists a sequence of states $S_1, \ldots, S_n$ with $S = S_1$ and $S_n = S'$ such that for all $1 \leq i < n$, $S_{i+1} \in \delta(S_i, \varepsilon)$. For $x \in \Sigma$ and states S, S', we write $S \xrightarrow{x} S'$ when there are states S_1 and S_2 such that $S \xrightarrow{\varepsilon} S_1$ and $S_2 \in \delta(S_1, x)$ and $S_2 \xrightarrow{\varepsilon} S'$. An NFA N accepts a string $x_0 \cdots x_{m-1}$ on Σ if there exists a sequence of states $S_0, S_1, S_2, \ldots, S_m$, such that 1) S_0 is the starting state; 2) for $0 \leq i \leq m-1$, $S_i \xrightarrow{x_i} S_{i+1}$; and 3) $S_n \in \mathcal{S}_A$ is an accepting state.

Thompson NFA construction. An NFA, N, is equivalent to a regular expression, re, if $L(N) = L(re)$. It is well known that any regular expression can be converted to an equivalent NFA using Thompson's construction, which is called a Thompson NFA (TNFA). There are variations of Thompson's construction, and in this work, we adopt a specific variant [198]. The TNFA construction algorithm is explained in Algorithm 11. Given a regular expression, re, the algorithm processes each character in re in order and outputs a directed graph, $G = (V, E = E_\Sigma \cup E_\varepsilon)$, which can be interpreted as a TNFA as follows:

- Each $i \in V$ represents a state in the resulting NFA, i.e., $\mathcal{S} = V$. Node 0 and n are the initial and accepting states, respectively.
- Edges in E_ε specify epsilon transitions. If $(i, j) \in E_\varepsilon$, then $\delta(i, \varepsilon) = j$.
- Edges in E_Σ specify character transitions. If $(i, j) \in E_\Sigma$ is labeled by $c \in \Sigma$, then $\delta(i, c) = j$.
- Edges in E_Σ specify the character transition. If $(i, j) \in E_\Sigma$ is labeled by $c \in \Sigma$, then define $\delta(i, c) = \{j\}$;
- For the rest of the undefined $(i, c) \in V \times (\Sigma \cup \varepsilon)$, set $\delta(i, c) = \phi$.

Thus, the graph $G = (V, E)$ defines a TNFA, $N = (V, \Sigma \cup \varepsilon, \delta, 0, n)$.

As the set of nodes in the directed graph is the set of states, we do not differentiate states and nodes without raising confusion in the rest of this paper. The output TNFA is equivalent to the language defined by re [198].

TNFA simulation. Taking as input $G = (V, E)$ for re and a string $x = x_0 \cdots x_{m-1}$, the TNFA simulation decides if x is an element of $L(\mathsf{re})$ by checking whether or not the TNFA defined by G accepts x. The TNFA simulation algorithm is listed in Algorithm 13. It simulates the operation of an NFA by computing $\mathcal{S}_i$, which includes all states that are reachable after processing the first i characters of x. The algorithm finally outputs if S_n is in $\mathcal{S}_m$.

Computing $\mathcal{S}i$ from $\mathcal{S}i - 1$ involves finding all states that can be accessed through a path $e_0, \cdots, e_\ell$, where $e_0 \in E_\Sigma$ is labeled x_{i-1} and $e_i \in E_\varepsilon$ for $i > 0$. The process is divided into two steps. First, the algorithm sets $\mathcal{S}i$ to all states that can be reached from $\mathcal{S}i - 1$ through $e_0 \in E_\Sigma$ (lines 5-7 in Algorithm 13). As all edges in E_Σ have the form $(j - 1, j)$ for some j (line 22 in Algorithm 11), $\mathcal{S}_i$ can be set up through a linear scan. This step is referred to as the character transition. Second, the algorithm

Algorithm 11: Thompson NFA construction

Input: $\mathsf{re} \in \Sigma^n$
Output: $G(V, E), V = 0, \cdots, n, E, \mathsf{re} \in \Sigma^n$

```
1  Initialize an empty stack 'ops of integers
2  Initialize a graph G(V, E = E_Σ ∪ E_ε) with nodes {0, ··· , n}, Both E_Σ and E_ε
     are initialized to be empty
3  for i = 0 to n do
4      lp ← i
5      if re_i ?= '(' or re_i ?= '|' then
6          'ops.push(i)
7      else
8          if re_i ?= ')' then
9              or ←' ops.pop()
10             if re_or ?= '|' then
11                 lp ←' ops.pop()
12                 add (lp, or + 1) and (or, i) to E_ε
13             else
14                 if re_or ?= '(' then
15                     lp ← or
16     if i < n − 1 and re_{i+1} ?= '*' then
17         add (lp, i + 1) and (i + 1, lp) to E_ε
18     if re_i ?= '(' or re_i ?= '*' or re_i ?= ')' then
19         add (i, i + 1) to E_ε
20 for i = 0 to n − 1 do
21     if re_i ∈ Σ then
22         label the edge (i, i + 1) by re_i and add it to E_Σ
```

extends $\mathcal{S}_i$ to all states reachable from $\mathcal{S}_i$ through any number of epsilon transitions (line 8 in Algorithm 13). This is done by finding all reachable nodes through a depth-first search (DFS). This step is referred to as the epsilon transition.

For simplicity of presentation of our protocols, from here onward we will represent a regular expression of size n by a graph of epsilon transitions $G = (V, E_\varepsilon)$ and an array re of size $n + 1$, such that $\mathsf{re}[j]$ equals the character transition for the edge $(j, j + 1)$ if it exists, else it is set to $\perp$.

4.2.2 Cryptographic Preliminaries

Oblivious transfer. Oblivious transfer (OT) is a fundamental primitive for secure computation [189]. It takes N messages $(m_0, \ldots, m_{N-1})$ from a sender and a choice index b from a receiver. The receiver receives m_b without knowing $\{m_i\}_{i \neq b}$ while the sender has no information on b. The random oblivious transfer (ROT) differs from

Algorithm 12: Oblivious transfer (OT) functionality $\mathcal{F}_{\mathsf{OT}}$

1 Upon receiving $(\mathsf{ot}, N, \ell, \{m_i\}_{i\in[0,n)})$ from the sender and $(\mathsf{ot}, N, \ell, b)$ from the receiver such that $m_i \in \{0,1\}^\ell$ and $b \in [0, N)$, send m_b to the receiver.

2 Upon receiving (rot, N, ℓ) from the sender and receiver, uniformly sample $(m_0, \ldots, m_{N-1}) \leftarrow \{0,1\}^{n\times\ell}$ and $b \in [0, N)$. Send $\{m_i\}_{i\in[0,N)}$ to the sender and (b, m_b) to the receiver.

OT by letting the functionality sample uniform messages for the sender and an index for the receiver. The functionalities are formally stated in Functionality 12.

Two-party computation. Two-party computation (2PC) allows two mutually untrusted parties to jointly compute a public function over their private inputs without leaking anything beyond the output. We rely on the *Universal Composability* (UC) framework [56] to prove our two-party protocols are secure against passive adversaries, i.e., corrupt parties do not deviate from the protocol but they try learning the honest parties' inputs from the view of the protocol execution.

Garbled circuit. We employ Yao's garbled circuits [226] (GC) to handle invocations to $\mathcal{F}$2PC, which is defined as a generic 2PC functionality. The function to evaluate is represented as a Boolean circuit consisting of XOR and AND gates. A garbler P_0 constructs garbled tables that represent the wire values by random labels. An evaluator P_1 receives the input labels of the circuit and decrypts the output labels of each gate following the topological order. They derive the output by having the garbler interpret the output labels of the circuit.

Secret shares and their conversion. A binary value $x \in \{0,1\}$ can be secretly shared by two parties P_0 and P_1 through various sharing schemes. The share of secret held by each party does not reveal any information of x. We utilize two types of secret sharing schemes: Yao's garbled circuits mentioned above and the additive secret shares [98], as well as a practical approach to convert between these two types of sharings [73].

- Yao's share ($[b]^Y$): P_0 holds a map $[b]^Y_0 := \{k_0 : 0; k_1 : 1\}$ and P_1 holds the key $[b]^Y_1 := k_b$.
- Additive share ($[b]^B$): P_0 holds $[b]^B_0$ and P_1 holds $[b]^B_1$ such that $[b]^B_0 \oplus [b]^B_1 = b$.
- Y2B ($[b]^Y \rightarrow [b]^B$): the conversion from Yao's shares to additive shares (local operation [73]).
- B2Y ($[b]^B \rightarrow [b]^Y$): the conversion from additive shares to Yao's shares.

Oblivious stack. An oblivious stack data structure can be realized by the garbled circuits protocol. It allows conditional push and pop that take a secret Boolean value dictating whether the operation should be performed or disguised by a dummy execution [218]. We rely on the following operations.

- $\mathsf{ObStack} \leftarrow \mathsf{stack}()$: initialize an oblivious stack;
- $(\cdot) \leftarrow \mathsf{ObStack.CondPush}([b], [x])$: push the element x to the oblivious stack if $b = 1$, else skip.

Algorithm 13: TNFA simulation in plain text

Input: An TNFA defined by $G(V, E)$ and string $x = x_0 \cdots x_{m-1}$
Output: True/False

Initiate a set $\mathcal{S}_0 = \{S_0\}$
Add all nodes that is reachable by S_0 in G to $\mathcal{S}_0$
for $i = 0$ *to* $m - 1$ **do**
    Initiate a empty set $\mathcal{S}_i = \phi$
    for $j = 1$ *to* n **do**
        if $S_{j-1} \in \mathcal{S}_{i-1}$ **and** $(j-1, j) \in E_\Sigma$ *is labeled by* x_{i-1} **then**
            Put S_j to $\mathcal{S}_i$
                ▷ Character transition
    $\mathcal{S}_i \leftarrow$**DFS** $(G_\varepsilon = (V, E_\varepsilon), \mathcal{S}_i)$ ▷ Epsilon transition
return True if $S_n \in \mathcal{S}_m$, and False otherwise.

- $[x] \leftarrow \mathsf{ObStack.CondPop}([b])$: pop and return the top element x if $b = 1$, otherwise return a pre-defined dummy value $\perp$.

Zero-knowledge proofs. For a relation $\mathcal{R}$, public statement x, and private witness w, zero-knowledge proof of knowledge (ZKPoK) allows a prover $\mathcal{P}$ who proves $(w, x) \in \mathcal{R}$ to a verifier $\mathcal{V}$ without revealing any information of w. It enables $\mathcal{P}$ to validate a statement without compromising her privacy.

In terms of its security requirements: *Completeness* states that a valid witness always makes $\mathcal{V}$ accept. *Proof of knowledge* requires that, if $\mathcal{V}$ accepts, there is overwhelming probability that $\mathcal{P}$ holds a valid witness. *Zero-knowledge* means that no information related to the witness is leaked to $\mathcal{V}$ during the proving phase.

4.3 TNFA Simulation via Two Linear Scans

In this section, we describe our new TNFA simulation algorithm that is oblivious to the input string. The classical TNFA simulation algorithm 13 has an input-dependent memory access pattern and therefore is not suitable for private evaluation in circuit-based 2PC or ZKP. The input of the TNFA simulation algorithm includes the regular expression and the string. The privacy of either the input string or both the string and the regular expression is required. We first make the algorithm's access pattern independent of the input string by implementing the epsilon transition via linear scans. In Section 4.4, we explain how to utilize this linear scan-based simulation algorithm to perform regular expression matching in the ZKP setting. Later we show how to protect the privacy of both inputs via the oblivious stack or oblivious transfer in the 2PC scenario in Section 4.5.

The source of input dependence of the access pattern of the classical NFA simulation algorithm is the epsilon transition. In the epsilon transition, a depth-first

search (DFS) over $G(V, E_\varepsilon)$ is invoked to add all reachable nodes from a set of nodes. The DFS algorithm results in memory access patterns dependent on the set of active states and the input graph.

We found that the DFS operation in the epsilon transition can be replaced by two linear scans. In algorithm 14, we represent the states of nodes using a bit vector $\boldsymbol{s}$ of length $n+1$. The vector is initialized to $\boldsymbol{s} = (10000\ldots)$. A node S_i is said active if $s_i = 1$. The epsilon transition updates the ith bit from 0 to 1 if node S_i is reachable from a node S_j in the graph $G(V, E\varepsilon)$ of the TNFA, where $s_j = 1$. To check reachability, it's sufficient to look at the states of a node's predecessors in the linear scan (line 12, Algorithm 14), as paths from u to v contain at most one backward edge if v is reachable from u. Nodes reachable by paths with only forward edges, such as S_5, are activated in the first linear scan, as their predecessors are activated first. Nodes reachable by paths with backward edges are activated in the second scan, as at least one of their predecessors is activated in the first scan. For example, in Figure 4.2, S_4 is activated in the first scan, making S_2 active in the second scan.

Definition 5. *An edge $(u_0, u_1) \in E_\varepsilon$ is deemed a backward edge if $u_1 < u_0$, otherwise it is a forward edge.*

Definition 6. *A pair of edges $(u_0, u_1), (v_0, v_1) \in E_\varepsilon$ are called **nested edges**, if $u_0 < v_0 < v_1 < u_1$ or $v_0 < u_0 < u_1 < v_1$, otherwise they are called a pair of **cross edges**.*

Lemma 5. *For $u, v \in V$, if v is reachable from u in $G(V, E_\varepsilon)$, then there exists a path from u to v that contains a maximum of one backward edge.*

Proof. It is sufficient to show that if there exist a path from u to v containing $t > 1$ backward edges, we can always find an alternative path from u to v that containing only $t-1$ backward edges. Let the path be $P = (P'||u_0, \cdots u_h, w_0, \cdots, w_k, v_0, \cdots, v_\ell)$ where $\{u_i\}$, v_i, $\{w_i\}$ are all ascending lists, (u_h, w_0), (w_k, v_0) are backward edges, and $v_\ell = v$. We now discuss all three possible cases:

- $w_0 < u_h \leq v_0 < w_k$: (u_h, w_0) and (w_k, v_0) are neither crossed nor nested. Then $(w_0 \leq v_0 < w_k)$. Therefore, there exists $0 \leq c \leq k$ an edge $(w_c \leq v_0 < w_{c+1})$. Furthermore, w_c should be equal to v_0, otherwise the edge (w_c, w_{c+1}) crosses the backward edge (w_k, v_0). Therefore, the path $P = (P'||u_0, \cdots u_h, w_0, \cdots, w_c = v_0, \cdots, v_\ell)$ is also a path from u to v with $t-1$ backward edges.

- $v_0 \leq w_0 < u_h \leq w_k$: (u_h, w_0) is nested within (w_k, v_0) are neither crossed nor nested. we thus have $w_0 < u_h \leq w_k$. Then there exists $0 \leq c \leq k$ an edge $(w_c < u_h \leq w_{c+1})$. Furthermore, w_{c+1} should be equal to u_h, otherwise the edge (w_c, w_{c+1}) crosses the backward edge (u_h, w_0). Therefore, the path $P = (P'||u_0, \cdots u_h = w_{c+1}, w_{c+2}, \cdots, w_k = v_0, \cdots, v_\ell)$ is also a path from u to v with $t-1$ backward edges.

- $w_0 \leq v_0 < w_k \leq u_h$: (w_k, v_0) is nested within (u_h, w_0). Using an argument similar to that before, we have $v_0 = w_c$ for some c. Thus, the path $P = (P' \mid u_0, \cdots, u_h, w_0, \cdots, w_c = v_0, \cdots, v_\ell)$ is also a path from u to v with $t-1$ backward edges.

Theorem 2. *Algorithm 14 is correct, i.e., for any $x \in \Sigma^*$ and TNFA N, the Algorithm 14 outputs $N(x)$.*

Proof. The only difference between the transition TNFA simulation and 2 linear scan algorithm is in the epsilon transition step. To show correctness, we essentially need to prove, if a node v is reachable from an active node u at the start of the epsilon transition step, then s_v is set to 1 by the end of the epsilon transition step. From Lemma 5 there exist a path P with at max one backward edge from u to v in $G(V, E_\varepsilon)$.

1. Case 1: If P contains no backward edge:
   The path from u to v consists of only forward edges. Let the path be $u = u_1, u_2, \ldots, u_k = v$, where we have $_i < u_i$ if $i < j$. Hence in the iteration $t = 0$ on Step 13 of the algorithm, we know s_{u_2} is set to 1, since its predecessor s_{u_1} is set to 1. Inductively, s_{u_t} for any t is set to 1 since $s_{u_{t-1}}$ is set to 1.

2. Case 2: If P contains 1 backward edge:
   Let the one backward edge on the path P be (w, z).
   Case 2.1: $u \neq w$: There exist a path from u to w consisting only of forward edges, and hence s_w is set to 1 by the end of the first iteration of the For loop on Step 13. Variable s_z is set to 1 during the 2^{nd} iteration of the For loop, which will further helps set s_v to 1 in the same iteration, since there is a path of forward edges from node z to v.
   Case 2.2: $u = w$: Since $z < u$, s_z is set to 1 during first iteration of the For loop in Step 13, which would further set s_v to 1 in the same iteration since there exist a path consisting of forward edges from z to v.

Optimization for wildcard. The wildcard metacharacter "?" can be used to represent any single character in a string, making it a powerful tool for pattern matching. For example, the wildcard pattern "a?b" would match any string that starts with "a" and ends with "b" with one letter in between, such as "acb" and "a1b". Classical TNFA simulation can perform pattern matching with wildcards using union operations for a finite alphabet, but this can be inefficient when the alphabet is large. Algorithm 14 avoids this issue by treating the wildcard metacharacter as a character that matches any character in character transition (See Line 10 in Algorithm 14) so that the cost of wildcard matching is independent of the size of the alphabet.

Complexity. Each invocation of the epsilon transition requires us to scan nodes sequentially and update their active bit twice. This depends on whether any of its predecessors are active and each node has at maximum 2 predecessors (Lemma 7). Thus, the epsilon transition requires $O(n)$ time, which makes the TNFA evaluation run in $O(nm)$ time in total since it makes m invocations of the epsilon transition procedure.

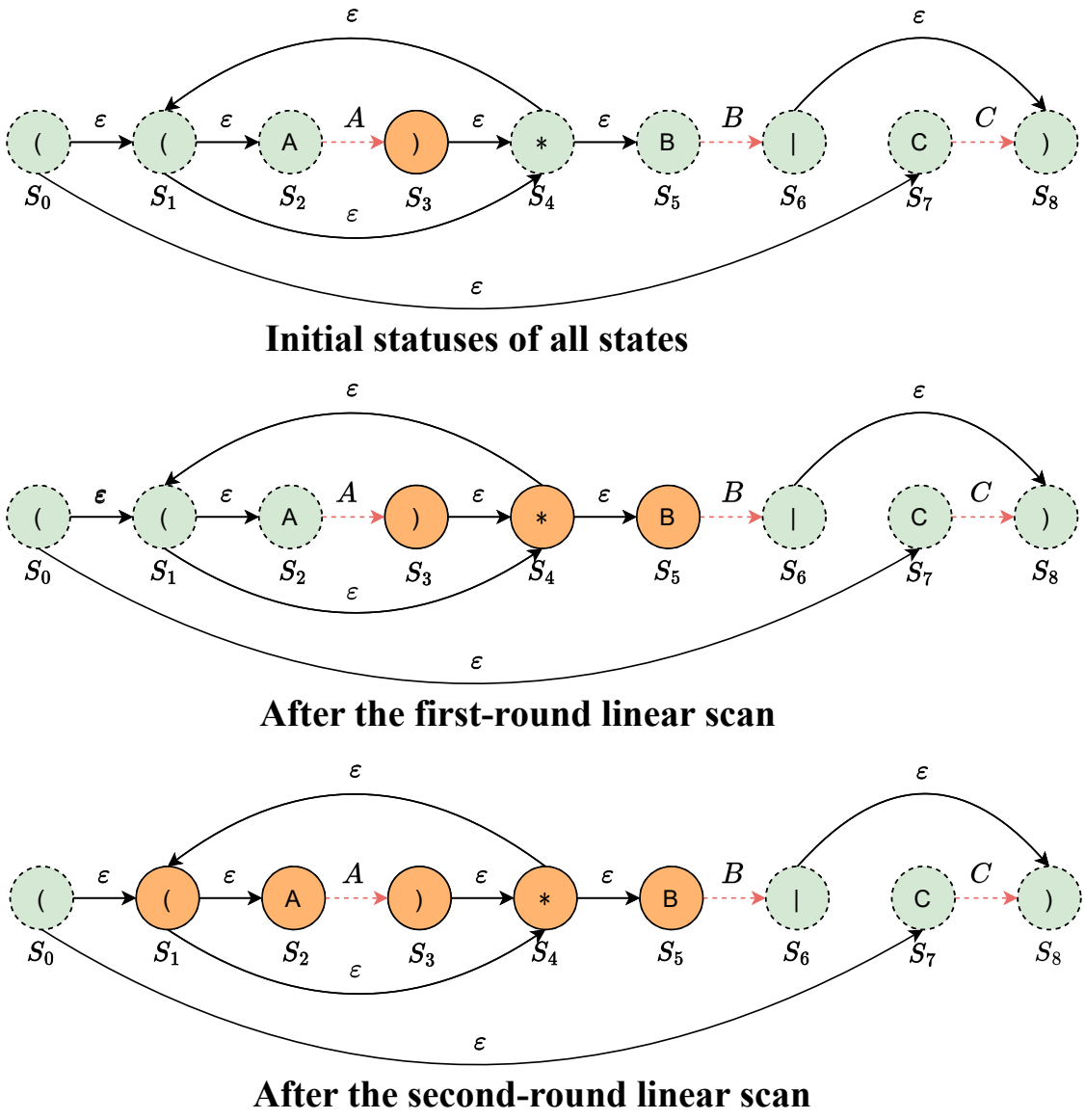


Figure 4.2: This figure explains the activation of reachable states from a currently active state S_3. During the first linear scan, states S_1 and S_2 are not activated as they are reachable from state S_3 through a backward edge between S_4 and S_1. At the end of the first linear scan, state S_4 is activated and causes states S_1 and S_2 to become activated in the second linear scan.

Algorithm 14: TNFA simulation via two linear scans

Input: String holder's input $\boldsymbol{x} = x_0 \dots x_{m-1}$ and Pattern holder inputs a TNFA $N = (G(V, E_\varepsilon), \mathsf{re})$

Output: String holder and pattern holder shares $N(\boldsymbol{x})$

Function `2ScanTNFAEval`(x, N)**:**
  $\boldsymbol{s} \leftarrow (100\dots0) \in \{0,1\}^{n+1}$
  EpsilonTransitions$(G(V, E_\varepsilon), \boldsymbol{s})$
  for $i = 0$ *to* $(m-1)$ **do**
    CharacterTransitions$(x_i, \mathsf{re}, \boldsymbol{s})$
    EpsilonTransitions$(G(V, E_\varepsilon), \boldsymbol{s})$
  return s_n
Function `CharacterTransitions`$(x_i, \mathsf{re}, \boldsymbol{s})$**:**
  for $j = n$ *to* 1 **do**
    $s_j \leftarrow (s_{j-1} \wedge (\mathsf{re}_j \stackrel{?}{=} x_i)) \vee (\mathsf{re}_j \stackrel{?}{=} '?')$
  $s_0 \leftarrow 0$
Function `EpsilonTransitions`$(G(V, E_\varepsilon), \boldsymbol{s})$**:**
  for $t = 0$ *to* 1 **do**
    for $j = 0$ *to* n **do**
      $s_j \leftarrow s_j \vee (\vee_{(\alpha,j)\in E_\varepsilon} s_\alpha)$

4.4 ZK Regular Expression Matching

In this section, we introduce the Zero-knowledge Regular Expression (ZK-Regex) matching protocol. This protocol is designed for scenarios where the pattern for the regular expression is publicly known. The goal for the string holder (prover) is to prove to another party (verifier) that their private string matches or does not match the public regular expression, while keeping the string confidential.

The ZK-Regex protocol leverages the TNFA simulation algorithm presented in Section 4.3 in a zero-knowledge proof setting. We also present a practical application of the ZK-Regex protocol, where a packet sender can demonstrate to a network middlebox that an encrypted TLS message complies with a public regular expression without revealing its content. The network middleboxes such as firewalls or deep packet inspection devices could use this technique to enforce regulations while preserving the privacy of packet owners. This setting is relevant to systems like ZKMB [109] and DECO [234].

4.4.1 Standalone ZK-Regex

Our ZK-Regex protocol, detailed in Algorithm 15, is a zero-knowledge version of Algorithm 14. The notation $[x]$ represents a private witness in zero-knowledge proof. In this protocol, the input string $[\mathbf{x}]$ is kept confidential, while the pattern defined by the TNFA $N = (G(V, E_\epsilon), \mathsf{re})$ is public. The algorithm can be represented as

Algorithm 15: Zero-knowledge version of the Algorithm 14 (ZK-Regex)

Private Input: $[\mathbf{x}] = ([\mathbf{x}_0] \dots [\mathbf{x}_{m-1}])$, $\mathbf{x} \in \Sigma^m$
Public Input: TNFA $N = (G(V, E_\varepsilon), \mathsf{re})$
Output: String holder and pattern holder shares $N(\mathbf{x})$

Function `2ScanTNFAEval`$([\mathbf{x}], N)$**:**
  $[\boldsymbol{s}] \leftarrow ([1][0][0] \dots [0])$ where $\boldsymbol{s} \in \{0,1\}^{n+1}$
  EpsilonTransitions$(G(V, E_\varepsilon), [\boldsymbol{s}])$
  for $i = 0$ *to* $(m-1)$ **do**
    CharacterTransitions$([\mathbf{x}_i], \mathsf{re}, [\boldsymbol{s}])$
    EpsilonTransitions$(G(V, E_\varepsilon), [\boldsymbol{s}])$
  return $[s_n]$
Function `CharacterTransitions`$([\mathbf{x}_i], \mathsf{re}, [\boldsymbol{s}])$**:**
  for $j = n$ *to* 1 **do**
    $[s_j] \leftarrow ([s_{j-1}] \wedge (\mathsf{re}_j \stackrel{?}{=} [\mathbf{x}_i])) \vee (\mathsf{re}_j \stackrel{?}{=} '?')$
  $[s_0] \leftarrow 0$
Function `EpsilonTransitions`$(G(V, E_\varepsilon), [\boldsymbol{s}])$**:**
  for $t = 0$ *to 1* **do**
    for $j = 0$ *to* n **do**
      $[s_j] \leftarrow [s_j] \vee (\vee_{(\alpha,j) \in E_\varepsilon} [s_\alpha])$

a Boolean circuit that includes m invocations of character transitions and $m+1$ invocations of epsilon transitions. Each character transition requires $2n$ equality checks, with each check requiring $\log|\Sigma| - 1$ logical AND gates. Additionally, $2n$ extra logical AND gates are needed to combine the results. The epsilon transition involves $2 \times (n+1)$ iterations, each using at most 3 logical OR gates, based on the fact that each state has a maximum of two predecessors, as stated in Section 4.3. The total circuit for the zero-knowledge TNFA simulation requires $2mn \log|\Sigma| + 6(m+1)(n+1)$ logical AND gates for an input string of length m and a size-n regular expression.

4.4.2 ZK-Regex over TLS

In ZK-regex, a string holder proves to a verifier that her private string meets a public regular expression without revealing the string. However, simply using a standalone ZK pattern matching protocol may not be adequate for practical applications as the prover could always present an input string that matches or doesn't match any pattern. To apply ZK-Regex in real-world scenarios, it's essential to link the private string to a particular message of interest.

We illustrate the practicality of ZK-regex in the context of Transport Layer Security (TLS) by first demonstrating that the private string corresponds to an encrypted TLS message. The rest of the proof then shows the relationship between this private string and the public regular expression. This allows network middleboxes (such as firewalls or deep packet inspection systems) to inspect encrypted payloads without

decrypting them, thereby preserving the privacy of the sender.

To formalize the idea of ZK-Regex-over-TLS, we define the following functions, which are summarized from ZKMB [109].

- $\mathsf{TLS.Handshake}(\mathsf{HSSec}) \rightarrow \mathsf{EK}$ takes input partial handshake secrets HSSec in TLS handshake and outputs an encryption key EK for the TLS record layer.
- $\mathsf{TLS.RecordDec}(\mathbf{c}, \mathsf{EK}) \rightarrow \mathbf{x}$ takes input a TLS ciphertext $\mathbf{c}$ and the encryption key EK, and outputs a string $\mathbf{x}$.

Assume the verifier inspects the network communication and records all TLS messages transited between the packet sender and receiver. At the first step, the prover (packet sender) first takes input the secrets HSSec derived during the TLS handshake and proves the knowledge of secret keys EK for TLS record layers. It ensures that EK is the one derived during the handshake. Otherwise, the prover could use arbitrary $\mathsf{EK}' \neq \mathsf{EK}$ to decrypt $\mathbf{c}$ to any $\mathbf{x}' \neq \mathbf{x}$ without being detected, since the cipher suites in TLS do not provide the property of key binding. Next, the prover extracts the encrypted messages $\mathbf{x}$ in ZK, which is ready to be fed into Algorithm 15 for the proof of pattern match. We define the ZK-Regex-over-TLS as follows.

Definition 7. *The functionality of ZK-Regex-over-TLS takes private input* $\mathbf{x}$*, handshake secrets* HSSec*, and public TLS ciphertext* $\mathbf{c}$*. Let both* re *and* Σ *be public. It outputs* accept *to* $\mathcal{V}$ *if* $\mathbf{x} \in L(\mathsf{re}) \subseteq \Sigma^*$ *and* $\mathbf{x} = \mathsf{TLS.RecordDec}(\mathbf{c}, \mathsf{TLS.Handshake}(\mathsf{HSSec}))$*. Otherwise, it outputs* reject*.*

We refer to [109] for the detailed protocols that realize $\mathsf{TLS.Handshake}$ and $\mathsf{TLS.RecordDec}$. In Section 4.6, we describe our end-to-end implementation of the above applications and provide the performance evaluation.

4.4.3 Advantage of ZK Policy Check from Regular Expression

Several recent works have demonstrated how zero-knowledge policy check contributes to privacy-enhancing applications. In the work of zero-knowledge middleboxes [109], the authors study the compliance check of domains hidden in encrypted DNS, e.g. DNS-over-TLS (DOT) or DNS-over-HTTPS. The problem is important because at one hand, the encrypted DNS provides huge benefit for protecting privacy of internet users while at the other hand, systems such as K12 education are forced to set middlebox to check and filter the domain queries from children. The zero knowledge policy check perfectly handles the check while preserving the privacy. The work of DECO [234] builds a decentralized oracle which allows a user to prove to third parties the provenance of data encrypted in a TLS session.

However, the policy check methods in these works are insufficient. For example, ZKMB uses Merkle tree non-membership proofs to prove that a domain is not in a public blocklist [109]. In detail, it sorts the blocklist in alphabetical order and compute a Merkle tree from it. Then it proves that a domain is within the range of two adjacent Merkle tree leaves [144]. Though the blocklist based system is straightforward and efficient, studies show that this coarse-grained approach does not offer accurate

classification [230]. It is common for the harmless web applications and malicious ones to have the same domain suffix, thus creating false alarms. To solve this issue, the blocklist based systems are evolving into the control system based on regular expressions which offers finer-grained filtering. This is the main reason that our regular expression based pattern matching algorithms are more suitable for zero-knowledge policy checks, as it provides low false positive rates and high expressiveness.

4.5 Secure Two-Party Regular Expression Matching

In many practical situations, both the regular expression and input string must be kept private. In this section, we introduce our protocols for secure two-party regular expression matching (secure-regex). Based on the TNFA simulation detailed in Algorithm 14, secure-regex consists of two components: character transition and epsilon transition. The character transition involves comparing each character of a length-n regular expression to a single character of the input string and updating the states based on the comparison results. This process is represented by n comparison circuits of size $\log|\Sigma| - 1$ and $2n$ logical AND gates for state updates. These character transitions can be easily expressed as a Boolean circuit and securely evaluated through 2PC. The communication complexity for privately performing all character transitions in a pattern matching task is $\mathcal{O}(\kappa mn \log|\Sigma|)$ when 2PC is implemented through the Garbled Circuit (GC) method.

Implementing epsilon transitions in Algorithm 14 directly through GC is difficult due to the need to keep the interconnection of the graph $G(V, E_\varepsilon)$ secret from the string holder. The state of a node in the graph is propagated from its predecessors, and the 2PC protocol must perform an oblivious retrieval of a state from a list of secretly shared states. This type of oblivious table lookup operation is often challenging. While using oblivious RAM could solve the problem generically, it would bring significant overhead and make the protocol inefficient [99].

The two methods for efficient handling of the epsilon transition are presented in the subsequent sections: one based on the oblivious stack (OS) and another based on the oblivious transfer (OT). The OT-based approach incurs lower communication overhead for short input regular expression, but its cost increases faster than OS-based protocol, thus is less efficient for long regular expressions. Also, the OS-based approach incurs only a constant number of round-trip communication and has a significant advantage in the high-latency network. We defer our detailed performance evaluation and comparison in Section 4.6.

4.5.1 Epsilon Transition via Oblivious Stack

This section presents a constant-round two-party computation protocol that employs an oblivious stack for epsilon transitions. The protocol requires $\mathcal{O}(n)$ stack operations with an average communication complexity of $O(\log n)$ per operation. First, we

introduce the following definitions and lemmas that specify these patterns.

Definition 8. *An edge $(u_0, u_1) \in E_\varepsilon$ is termed a **long edge** if $|u_1 - u_0| > 1$. In addition, (u_0, u_1) is called u_0's outgoing edge or u_1's incoming edge.*

Lemma 6. *For any two cross edges in E_ε, they are both forward edges. In other words, no edge "crosses" any backward edge in the TNFA graph.*

Proof. No backward edge crosses any other edge in the TNFA graph by construction (Algorithm 11). We can show this property hold for all regular expressions by giving an induction proof using the recursive regex definition. Given two regular expressions re, re satisfying this property, we have:

1. concatenation: the regular expression (re re′) just introduces a "new" forward edge in the TNFA graph, from the end node of re TNFA to the start node of re′ TNFA. Hence the TNFA for (re re′) satisfies this property too.
2. union : The TNFA for (re|re′) introduces new forward edges, but the all have end points being either the first node or the last node of re or re′ TNFA. Hence no backward edge in re′ or re′ crosses any of the newly introduced forward edges.
3. loop: (re)*, introduces a new forward edges and a single backward edges - all of them have end points being either the first node or the last node of re or re′ TNFA.

Lemma 7. *Every node in the TNFA graph G has at most one incoming long-forward edge and at most one outgoing long-forward edge in E_ε.*

Lemma 8. *Every pair of backward edges is nested.*

Proof. The lemma can be obtained directly as a collary of Lemma 6.

Lemma 9. *A pair of long-forward edges (u_0, u_1) and (v_0, v_1) with $v_0 < u_0$ are cross edges if and only if $v_1 = u_0 + 1$ and $re[u_0] =' |'$.*

Proof. We can prove this by induction, similar to proof of Lemma 6. We can show this property hold for all regular expressions by giving an induction proof using the recursive regex definition. Given two regular expressions re, re satisfying this property, we have:

1. concatenation: the regular expression (re re′) just introduces a "new" forward edge in the TNFA graph, from the end node of re TNFA to the start node of re′ TNFA. Hence the TNFA for (re re′) satisfies this property too.
2. union : The TNFA for (re|re′) introduces a new pair of forward cross edges. Where one edge is from the first node of re to the first node of re′, and the other edge is from the node labeled | to the last node of re′. Hence the induction property holds.
3. loop: (re)*, introduces a new forward edges and a single backward edges - all of them have end points being either the first node or the last node of re or re′ TNFA. Hence no new cross edges are introduced.

Lemma 8 and Lemma 9 illustrate that all backward edges are pairwise nested, and forward edges are almost nested with the exception of the scenario described in Lemma 9.

Oblivious graph representation. Using Lemma 7, we can limit the number of incoming and outgoing edges for each node in $G = (V, E_\varepsilon)$ by 2. In addition, each node has one incoming and outgoing long edge. Therefore, we can represent G using two adjacency lists of $n+1$ pairs of indices: one for the incoming edge and one for the outgoing edge. In particular, the pattern holder can set the ith entry of $\{(I_0^i, I_1^i)\}_{i=0}^n$ and $\{(O_0^i, O_1^i)\}_{i=0}^n$ as the following:
- I_0^i is $i-1$ if there is an edge in E_ε from S_{i-1} to S_i, and otherwise $\perp$ as the dummy;
- O_0^i is $i+1$ if there is an edge in E_ε from S_i to S_{i+1}, and otherwise $\perp$ as the dummy;
- I_1^i is j if there is a long edge in E_ε from S_j to S_i, and otherwise $\perp$ as the dummy;
- O_1^i is j if there is a long edge in E_ε from S_i to S_j, and otherwise $\perp$ as the dummy.

With the oblivious graph representation of the TNFA, we present the formal outline of our oblivious stack-based circuit in Algorithm 18. The algorithm consists of two subprotocols: the *forward scan* (Algorithm 16) and the *backward scan* (Algorithm 17).

Algorithm 16: ForwardScan

Input: $G = (V, E_\varepsilon)$ presented by $\{([I_0^i], [I_1^i])\}_{i=0}^n$ and $\{([O_0^i], [O_1^i])\}_{i=0}^n$; re; and $\{[s_0] \cdots, [s_n]\}$
$\mathsf{ostack} \leftarrow \mathsf{stack}()$
$[\mathsf{cross}] \leftarrow \mathsf{false}, \mathsf{tmp} \leftarrow \mathsf{false}$
for $j = 1$ *to* n **do**
  $\mathsf{hasLFIE} \leftarrow ([I_1^j] \neq \perp) \wedge ([I_1^j] < j)$
  $\mathsf{isOR} \leftarrow ([\mathsf{re}_j] \stackrel{?}{=} \text{'|'})$
  $\mathsf{popElmt} \leftarrow \mathsf{ostack.pop}((\mathsf{hasLFIE} \wedge \neg\mathsf{cross}) \vee \mathsf{isOR})$
  $[s_\alpha] \leftarrow \mathbf{Ite}(\mathsf{cross}, \mathsf{tmp}, \mathsf{popElmt})$ *// If-then-else*
  $[s_\alpha] \leftarrow \mathbf{Ite}(\mathsf{hasLFIE}, [s_\alpha], \mathsf{false})$
  $[s'_\alpha] \leftarrow \mathbf{Ite}([I_0^j] \neq \perp, [s_{j-1}], \mathsf{false})$
  $[s_j] \leftarrow [s_j] \vee [s_\alpha] \vee [s_{\alpha'}]$
  $\mathsf{hasLFOE} \leftarrow ([O_1^j] \neq \perp) \wedge ([O_1^j] > j)$
  $\mathsf{ostack.push}(\mathsf{hasLFOE}, [\mathsf{s_j}])$
  $\mathsf{cross} \leftarrow \mathsf{isOR}$
  $\mathsf{tmp} \leftarrow \mathbf{Ite}(\mathsf{cross}, \mathsf{popElmt}, \mathsf{false})$

In the backward scan subprotocol, nodes are processed from n to 0 and activated (by setting their s entry to true) if they have an incoming backward edge from an active node. When reaching an outgoing backward edge from state j, s_j is pushed into the stack. When the current state s_j has an incoming backward edge, a state bit is popped from the stack and its status is propagated to s_j. Due to the nested property and stack data structure, the backward scan ensures correctness, meaning any node reachable from an active node via a backward edge is activated.

The objective of the forward scan is to activate a node if there is a path to it from an active node that consists only of forward edges. Table 4.1 provides an

Algorithm 17: BackwardScan

Input: $G = (V, E_\varepsilon)$ presented by $\{([I_0^i], [I_1^i])\}_{i=0}^n$ and $\{([O_0^i], [O_1^i])\}_{i=0}^n$; $\{[s_0] \cdots, [s_n]\}$
ostack $\leftarrow$ stack()
for $j = n$ *to* 0 **do**
  $\alpha \leftarrow 0$
  hasBIE $\leftarrow ([I_1^j] \neq \perp) \wedge ([I_1^j] > j)$
  $\alpha \leftarrow$ ostack.pop(hasBIE)
  hasBOE $\leftarrow ([O_1^j] \neq \perp) \wedge ([O_1^j] < j)$
  $[s_j] \leftarrow [s_j] \vee \alpha$
  ostack.push(hasBOE, $[s_j]$)

illustration of the forward scan subprotocol. Unlike backward edges, forward edges can cross, as shown by the crossing of S_6 and S_7 in Figure 4.1. However, according to Lemma 9, these crossings can only occur between adjacent nodes and can be identified by examining if the character is $'|'$ in re.

To handle these special crossings, the stack-based approach used for backward edges can be adjusted. When the stack contains two nodes with crossed forward edges, such as s_0, s_6 in the processing of S_6, the top two elements can be swapped to s_6, s_0. This allows s_0 to be popped from the stack and used for updating the next state with a long forward incoming edge (LFIE), S_7 in this case. However, swapping the elements causes three operations on the stack. To reduce the number of operations, the variable tmp is introduced to store the top bit of the stack instead of swapping. After processing S_6, the stack will have s_6 and tmp $= s_0$, which will be used immediately for the next state with an LFIE. This approach is valid due to the structure of the TNFA presented in Lemma 9.

Optimization. In our oblivious graph representation, each node is represented by four integers for its incoming and outgoing edges. To improve the protocols, we replace these 4 integers with just 5 bits. Algorithm 16 and 17's binary values hasLFIE and hasBIE depend only on the secret input I_1^j and public information j, which can be computed by the pattern holder locally and fed into $\mathcal{F}$2PC instead of inputting I_j^1. Additionally, pattern holder directly inputs the binary values hasLFOE and hasBOE rather than an integer O_1^j. The pattern holder also inputs a bit indicating the predecessor relationship between S_{i-1} and S_i, used in Line 10 of Algorithm 16. The binary value isOR in Algorithm 16 depends only on the regular expression, so the pattern holder can compute it locally and feed it into the GC. With these optimizations, the OS-based epsilon transition is independent of the character size $\log |\Sigma|$.

Security. All components in the Algorithms 16, Algorithm17, and Algorithm 18, are realized in $\mathcal{F}$2PC. Thus the security follows the instantiation of the underlying two-party computation.

State	stack	s_α	s'_α	cross	tmp	Long Edge
S_0	s_0	F	F	F	F	Has LFOE
S_1	s_0, s_1	F	s_0	F	F	Has LFOE
S_2	s_0, s_1	F	s_1	F	F	None
S_3	s_0, s_1	F	F	F	F	None
S_4	s_0	s_1	s_3	F	F	Has LIFE
S_5	s_0	F	s_4	F	F	Has LIFE
S_6	s_6	F	F	T	s_0	Has LFOE
S_7	s_7	s_0	F	F	F	Has LFIE
S_8		s_6	F	F	F	Has LFIE

Table 4.1: Example of one forward scan for the TNFA in Algorithm 16. The ith row in the table shows the status of the stack and variable values after processing the ith state. If a state has a long forward outgoing edge (LFOE), its status is always pushed onto the stack. If it has a long forward incoming edge (LFIE), its status is updated using either a bit popped from the stack or stored in tmp. For example, S_4 in Figure 4.1 has a LFIE caused by $'*'$, so S_α is assigned the bit popped from the stack. Meanwhile, S_7 has a LFIE caused by $'|'$, so tmp stores S_0 after processing S_6 and is used to update S_7.

Algorithm 18: Epsilon Transition via Oblivious Stack

1 **Input:** $N = (G = (V, E_\varepsilon), \mathsf{re})$ and states' statuses $\{[s_0], \cdots, [s_n]\}$ that are shared by two parties.
2 ForwardScan($G = (V, E_\varepsilon), \mathsf{re}$)
3 BackwardScan($G = (V, E_\varepsilon)$)
4 ForwardScan($G = (V, E_\varepsilon), \mathsf{re}$)
Output: Updated shared states' statuses $\{[s_0], \cdots, [s_n]\}$

Complexity. An epsilon transition requires a total of $6n + 2$ stack accesses. The efficient construction of oblivious stacks [232] for a stack of size $\mathcal{O}(n)$ incurs $O(\log n)$ amortized cost for each oblivious access. Thus the net circuit size for oblivious stack is $O(n \log n)$. Additionally, both forward and backward scans takes $O(n)$ AND gates for non-stack operations. Hence, the epsilon transition has $O(n \log n)$ circuit complexity (in the number of AND gates). While instantiating $\mathcal{F}$2PC with GC, it results in the net communication complexity of the regular expression matching task $O(\kappa mn(\log n + \log |\Sigma|))$.

4.5.2 Epsilon Transition via 1-out-n+1 OT

An important observation that helps in optimizing the OS-based protocol is that the predecessors of each state in a TNFA are solely determined by the input regular expression. As a result, the pattern holder knows the exact states (i.e., the value of α at line 15 in Algorithm 14) that need to be retrieved when updating a state during epsilon transitions. Based on this, we propose an alternative OT-based approach for secure epsilon transition that has a total communication overhead of

Algorithm 19: Epsilon Transition via 1-out-of-n+1 OTs

Input: String holder inputs Boolean shares of states' statuses $\{[s_0]_0^B, \ldots, [s_n]_0^B, \}$

Input: Pattern holder inputs $G(V, E_\varepsilon)$, and her Boolean shares of states' statuses $\{[s_0]_1^B, \ldots [s_n]_1^B\}$

1 **for** $t = 0$ *to 1;* $j = 0$ *to* n **do**

2 String holder samples a random bit r_b, and sends $(\mathsf{ot}, n+1, 1, \{[s_j]_0^B \oplus r_b\}_{j\in[0,n]})$ to $\mathcal{F}_{\mathsf{OT}}$

3 Denote α as the index of j-th state's predecessor. Pattern holder sends $(\mathsf{ot}, n+1, 1, \alpha)$ to $\mathcal{F}_{\mathsf{OT}}$

4 $\mathcal{F}_{\mathsf{OT}}$ returns $m = [s_\alpha]_0^B \oplus r_b$ to the pattern holder

5 Pattern holder sets her new share $[s_\alpha]_1^B \leftarrow m \oplus [s_\alpha]_1^B$

6 String holder sets her new share $[s_\alpha]_0^B \leftarrow r_b$

7 For another predecessor of j-th state α', two parties repeat the above to get $[s_{\alpha'}]_0^B$ and $[s_{\alpha'}]_1^B$ respectively

8 Invoke $\mathcal{F}_{\mathsf{2pc}}$ with B2Y to compute $[s_j]_0^Y, [s_j]_1^Y \leftarrow ([s_j]_0^B \oplus [s_j]_1^B) \vee ([s_\alpha]_0^B \oplus [s_\alpha]_1^B) \vee ([s'_\alpha]_0^B \oplus [s'_\alpha]_1^B)$

9 Translate Yao's share to additive share as $[s_j]_0^B, [s_j]_1^B \leftarrow \text{Y2B}([s_j]_0^Y, [s_j]_1^Y)$

Output: String holder holds shares of updated states $\{[s_0]_0^B, \ldots, [s_n]_0^B\}$

Output: Pattern holder holds shares of updated states $\{[s_0]_1^B, \ldots, [s_n]_1^B\}$

$\mathcal{O}(mn(n + \kappa \log |\Sigma|))$. While its asymptotic overhead is not as good as the OS-based approach, it is still practical for matching with short regular expressions or in low-latency networks. A comprehensive comparison with the OS-based approach will be provided in Section 4.6.

The protocol utilizes the functionalities $\mathcal{F}$2PC, $\mathcal{F}$OT, and the conversion functions Y2B and B2Y discussed in Section 4.2.2. It is based on the bound on the number of predecessors for each state established in Lemma 7, which states that a state can have at most 2 predecessors. The protocol starts by having the two parties secretly share the states $(s_0, \ldots, s_n) \in 0, 1$. The goal is to update a state s_j based on its two predecessors s_α and $s_{\alpha'}$ if they exist. To do this, the parties use 1-out-of-n+1 OT twice to retrieve the secret shares of α-th and α'-th state (lines 2-7). Then they employ $\mathcal{F}_{\mathsf{2PC}}$ to update the secret shared state s_j (line 8). Y2B and B2Y are utilized to convert between the forms of the secret shares when transforming between $\mathcal{F}_{\mathsf{OT}}$ and $\mathcal{F}_{\mathsf{2PC}}$ (lines 8-9).

Optimization through 1-out-of-n+1 ROT. To reduce the number of round-trip communications and minimize the impact of network delay in Algorithm 19, we optimize the 1-out-of-n+1 OT by compiling it from the 1-out-of-n+1 ROT. This change is shown in Algorithm 20 and results in a reduction of the number of round-trips to $\mathcal{O}(1)$, leading to improved performance compared to the original implementation.

The total number of required 1-out-of-n+1 OT during the pattern matching task

Algorithm 20: Optimization via 1-out-of-n+1 ROT

Initialization $(t, \{\alpha_i\}_{i\in[t]})$**:**

String holder and pattern holder send $(\mathsf{rot}, n+1, 1)$ to $\mathcal{F}_{\mathsf{OT}}$ for t times, which returns $\{r_0^i, \cdots, r_n^i\}_{i=0}^t$ to string holder and $\{(\gamma_i, r_{\gamma_i}^i)\}_{i=0}^t$ to pattern holder

Pattern holder sends $\{\delta_i\}_{i=0}^t$ to string holder such that $\delta_i = \alpha_i - \gamma_i \mod n+1$.

Online *When invoking the ith 1-out-n+1 OT in Algorithm 19*:

String holder computes the list $\{[s_0]_0^B \oplus r_b \oplus r_{i_0}^i, \cdots, [s_n]_0^B \oplus r_b \oplus r_{i_n}^i\}$, where $i_k = k - \delta_i \mod n+1$ for $k \in [0, n]$. She sends this list to the pattern holder

Pattern holder selects $c = [s_\alpha]_0^B \oplus r_b \oplus r_\alpha^i$, and gets $[s_\alpha]_0^B \oplus r_b$ by computing $[s_\alpha]_0^B \oplus r_b \oplus r_\alpha^i \oplus r_{\gamma_i}^i$.

is denoted by t. At the start, the pattern holder and the string holder perform t ROTs in a batch. This leaves the string holder with the random OT messages $r_0^i, \cdots, r_{n}^{i\,t}{}_{i=0}$ and the pattern holder with the random choice indices and the corresponding OT messages $\{(r_{\gamma_i}^i, \gamma_i)\}_{i=0}^t$. The pattern holder knows the actual choice indices $\{\alpha_i\}_{i\in[t]}$ because they are determined by her input regular expression. She sends the string holder the offset δ_i between the actual and random indices. The real OT messages are then obtained by rotating the ROT messages by the offset, i.e., the OT messages for the real choice index α_i are $(r_{n-\delta_i+1}^i, \ldots, r_n^i, r_0^i, \ldots, r_{n-\delta_i}^i)$. This initialization phase requires only $\mathcal{O}(1)$ round-trip communication.

In the online phase, when the string holder collects the input messages $(m_0, \ldots, m_n)$ for the i-th 1-out-of-n+1 OT, she sends $(m_0 \oplus r_{n-\delta_i+1}^i, \ldots, m_{\delta_i} \oplus r_n^i, m_{\delta_{i+1}} \oplus r_0^i, \ldots, m_n \oplus r_{n-\delta_i}^i)$ to the pattern holder. The pattern holder can obtain m_{α_i} without any round-trip communication.

Instantiation and Security of 1-out-of-N ROT. Our design of 1-out-of-N ROT is shown in Figure 21. We define the following theorem and provide its proof.

Theorem 3. *The 1-out-of-N random oblivious transfer protocol (in Figure 21) securely realizes the ROT function from the functionality $\mathcal{F}_{\mathsf{OT}}$ with semi-honest security in the $\mathcal{F}_{\mathsf{ROT}}$-hybrid model.*

Proof. We prove the security by simulating the views for P_0 and P_1. Define a simulator $\mathcal{S}_0$ and an adversary $\mathcal{A}_0$ who controls P_0. $\mathcal{S}_0$ emulates the functionality $\mathcal{F}_{\mathsf{OT}}$. It samples uniform $\{(y_i^0, y_i^1)\}_{i\in[0,d)}$ and sends them to $\mathcal{A}_0$. It receives a_0^0 from $\mathcal{A}_0$. Then $\mathcal{S}_0$ continues to execute as an honest P_0. The simulated view of $\mathcal{A}_0$ is indistinguishable to its view in the real-world. Also, define a simulator $\mathcal{S}_1$ and an adversary $\mathcal{A}_1$ who controls P_1. $\mathcal{S}_1$ emulates the functionality $\mathcal{F}_{\mathsf{OT}}$. It samples uniform $\{(x_i, y_i^{x_i})\}_{i\in[0,d)}$ and a_0^0, and sends them to $\mathcal{A}_1$. It executes the protocol as an honest P_1 would do. The view of $\mathcal{A}_0$ is indistinguishable to the view of honest P_0 in the real-world.

Algorithm 21: 1-out-of-N ROT protocol

Input: Define parameter $d := \lceil \log N \rceil$ and security parameter κ and message length ℓ. Assume a hash function H and a function $G : \{0,1\}^\kappa \to \{0,1\}^\ell$.

Output: P_0 outputs $(m_0, \ldots, m_{N-1}) \in \{0,1\}^{N \times \ell}$. P_1 outputs (b, m_b) where $b \in [0, N)$

1 For $i \in [0, d)$, P_0 and P_1 send $(\mathsf{rot}, 2, \kappa)$ to $\mathcal{F}_{\mathsf{OT}}$, which returns (y_i^0, y_i^1) to P_0 and $(x_i, y_i^{x_i})$ to P_1.

2 P_0 samples $a_0^0 \in \mathbb{F}_{2^\kappa}$ and sends it to P_1.

3 **for** $i = 0$ *to* $d - 1$ **do**

4 | **for** $j = 0$ *to* $2^i - 1$ **do**

5 | | P_0 computes $(a_i^{2j}, a_i^{2j+1}) := (H(a_{i-1}^j, y_i^0), H(a_{i-1}^j, y_i^1))$.

6 | P_1 computes $k_i = \sum_{t=0}^{i} x_t \cdot 2^t$ and $a_i^{k_i} := H(a_{i-1}^{\lfloor k/2 \rfloor}, y_i^{x_i})$.

7 P_0 outputs $\{G(a_{d-1}^i)\}_{i \in [0,N)}$ and P_1 outputs $(k_{d-1}, G(a_i^{k_{d-1}}))$.

Security. The OT based epsilon transition protocol invokes the $\mathcal{F}_{\mathsf{OT}}$ (Lines 2-4,7) and the $\mathcal{F}_{\mathsf{2PC}}$ (Line 8) functionalities. The security guarantee is stated and proved below.

Theorem 4. *The protocol shown in Algorithm 19 securely realizes the function* EpsilonTransition *described in Algorithm 14 (Lines 12-15) in the two-party computation setting.*

Proof. We follow the simulation-based paradigm to prove the Theorem 4. We first consider a corrupted string holder $\mathcal{A}$, who acts as the sender in $\mathcal{F}_{\mathsf{OT}}$ and a garbler in $\mathcal{F}_{\mathsf{2PC}}$. A simulator $\mathcal{S}1$ emulates the functionality $\mathcal{F}$OT and $\mathcal{F}$2PC. When emulating OT, it receives and stores the OT input messages from $\mathcal{A}$. When simulating the 2PC functionality, it receives the inputs from $\mathcal{A}$, samples a random bit $b \in \{0,1\}$ and returns it to $\mathcal{A}$. We consider a corrupted pattern holder $\mathcal{A}$. A simulator $\mathcal{S}2$ emulates the functionality $\mathcal{F}$OT and $\mathcal{F}$2PC. When emulating OT, it receives and stores the choice indices from $\mathcal{A}$ and returns a random bit c as the output for OT receiver. When simulating the 2PC functionality, it receives the inputs from $\mathcal{A}$, samples a random bit $d \in \{0,1\}$ and returns it to $\mathcal{A}$. Both views of corrupted string holder and corrupted pattern holder are perfectly simulated.

Complexity. Algorithm 19 makes $4(n+1)$ invocations of 1-out-of-n+1 OT. The corresponding garbled circuit(GC) contains $4(n+1)$ inputs from both the garbler and evaluator and $4(n+1)$ AND gates. With optimizations described in Algirithm 20, the $4(n+1)$ invocations of 1-out-of-n+1 OT only require the string holder sending $4(n+1)^2$ bits during the online phase, and do not incur any round-trip communication. Assume that the string holder is the garbler and the pattern holder is the evaluator of GC. Among $2(n+1)$ rounds, the garbler sends 4κ bits and the evaluator sends 2 bits per round. This epsilon transition protocol incurs a total of $4(n+1)^2+16(n+1)\kappa+4(n+1)$

bits communication overhead and $2(n+1)$ round trips. When applying this OT-based epsilon transition to the TNFA simulation described in Algorithm 14, the overall communication complexity is $\mathcal{O}(mn(n + \kappa \log |\Sigma|))$ and it requires $\mathcal{O}(mn)$ round-trips. Note that the total preprocessing cost for 1-out-of-n+1 OT is $\mathcal{O}(n \log n)$ and thus does not depend on string length m. Next, we describe how it is achieved by batch 1-out-of-n+1 ROTs.

It appears multiple times in Algorithm 19 that the pattern holder and string holder invoke multiple 1-out-of-n+1 OTs with the same OT receiver's choice. We take advantage of this to further optimize the batch generation of ROT, which reduce the number of invocation of 1-out-of-n ROT by a factor of m (length of the string). In detail, to batch t 1-out-of-n+1 ROT with the same receiver's choice, two parties first instantiate Algorithm 21 to generate one 1-out-of-n+1 ROT. Assume a PRG $\mathsf{G}(\cdot) : \{0,1\}^\kappa \to \{0,1\}^{t\times\kappa}$, P_0 computes $\{m_i^j\}_{j\in[0,t)} := \mathsf{G}(m_i)$ for $i \in [0,n]$ and P_1 computes $\{m_b^j\}_{j\in[0,t)} := \mathsf{G}(m_b)$. In this way, each pair of $(\{m_i^j\}_{i\in[0,n)}, (b, r_b^j))$ is an output of 1-out-of-n+1 ROT.

4.6 Performance Evaluation

We first describe the detailed instantiation of cryptographic protocols. Then we provide concrete performance evaluations of our privacy-preserving pattern matching protocols.

4.6.1 Instantiate Cryptographic Building Blocks

1-out-of-N ROT. We describe an efficient protocol for 1-out-of-N ROT as an optimization for our OT-based protocol. The ROT functionality is included in Algorithm 12. Define a security parameter κ and messages length ℓ. The functionality samples uniform $(m_0, \ldots, m_{N-1}) \in \{0,1\}^{N\times\ell}$ and choice index $b \in [0, N)$. It sends $\{m_i\}_{i\in[0,N)}$ to P_0 and (b, m_b) to P_1. Its cost is dominated by $\mathcal{O}(\log n)$ communication and $\mathcal{O}(n)$ AES encryption.

Oblivious stack. We use the oblivious stack protocol proposed in [232]. Its each conditional push or conditional pop operation incurs $\mathcal{O}(\kappa \log n)$ amortized communication complexity when instantiated in garbled circuits. At a high level, it builds a layered data structure with increased capacity from the lower layers to the upper layers. The pushed elements are first stored at lower layers. All elements at one layer will be propagated to the upper layer when the current layer is possibly full. The pop operations reverse this process. We refer to [232] for more details about the protocol. In our prototype, we use the implementation open-sourced by [163].

MPC-in-the-Head. We instantiate the MPCitH protocol by the Limbo framework [72]. Limbo adopts an MPC verification protocol that is based on additive secret

sharing and a sublinear distributed multiplication triple checking scheme [47, 104]. After being compiled to a ZKP, it results in high efficiency, non-interactiveness, and proof size linear to the circuit size. It is currently the state-of-the-art MPCitH protocol. The only work in this regime that achieves sublinear proof size is Ligero [13]. However, its computation is more expensive than Limbo.

Channel opening for TLS 1.3. Channel opening refers to the circuit that proves a private string is the decryption of a public TLS ciphertext. Since TLS 1.3 allows non-committing encryption schemes, the circuit must re-derive the session key to prevent key equivocation. A naive approach is to repeat all client key derivations from the TLS Handshake, but this leads to inefficiency due to costly group operations and hashing of a potentially long transcript, making the circuit size dependent on the longest possible transcript. ZKMB [109] offers a shortcut by only re-executing essential intermediate operations from the handshake process, eliminating those expensive operations. Our implementation takes this approach, resulting in a circuit that does not contain any group operations, and its size is independent of the transcript length.

4.6.2 Performance Evaluation of ZK-Regex

We now evaluate the performance of our ZK-regex protocol. First, we measure the concrete prover time and proof size of our standalone ZK-regex protocol. Then, we deploy our protocol over TLS and evaluate its performance in a realistic and practical setting: end-to-end TLS-encrypted packet inspection. We show our ZK-regex over TLS is practical by evaluating the concrete proof size and prover time in a realistic DNS policy checking setting.

We implemented ZK-regex protocol and channel opening for TLS using emp-toolkit [217] and benchmarked our circuits in Limbo framework [72] with parameters that achieve 2^{-40} statistical security. Specifically, we fix the compression factor to 32 and MPC party numbers to 16 for all runs. All experiments in this section use a single Amazon EC2 `m5.4xlarge` instance with 64 gigabytes of RAM.

r1	97	379KB	0.57s	r2	39	169KB	0.23s
r3	66	267KB	0.38s	r4	69	277KB	0.39s
r5	83	328KB	0.47s	r6	37	161KB	0.22s
r7	23	111KB	0.14s	r8	21	103KB	0.13s
r9	59	241KB	0.35s	r10	65	263KB	0.37s
r11	6	49KB	0.05s	r12	16	85KB	0.1s
r13	68	274KB	0.39s	Avg.	49	208KB	0.29s

Table 4.2: The Length of the regular expression, proof size and prover time of ZK-regex applied to Pi-hole regular expression set. The patterns (r1,...,r13) are provided in [4]. We omitted one pattern because it cannot be efficiently translated.

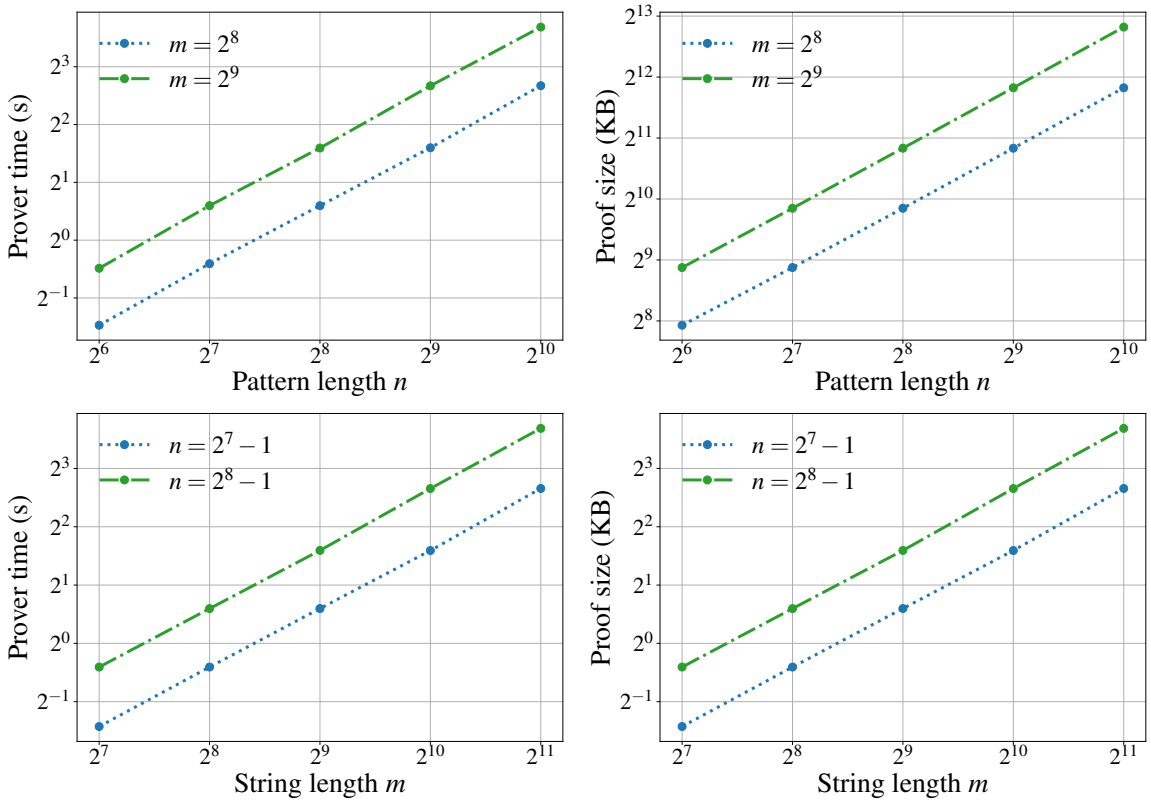


Figure 4.3: The prover time and proof size of our ZK-regex protocol (average over 10 executions.

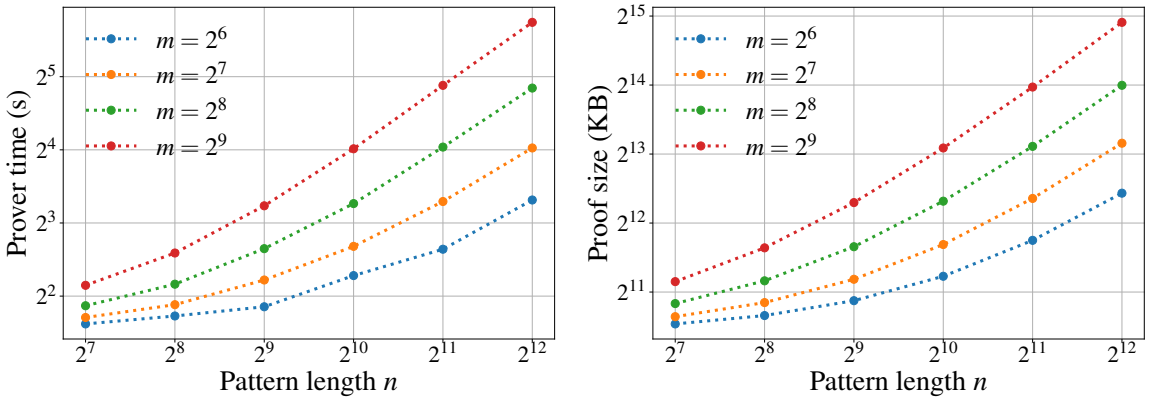


Figure 4.4: The prover time and proof size of our ZK-regex protocol deployed over TLS 1.3 (average over 10 executions).

Benchmark the protocol. We assume $|\Sigma| = 256$ so that both pattern length and string length are in bytes. We measure the proof size and prover time with varying input strings and regular expression length. The result appears in Figure 4.3. The proof size and prover time grow linearly with both the input string and regular expression length.

Next, we apply ZK-regex to a realistic scenario: policy checking for encrypted DNS queries. Here we use the regular expressions for blocking filters provided by Pi-hole [6]. The evaluation result appears in Table 4.2. For the longest pattern r1, ZK-regex generates a proof of size 379 KB in just 0.57 s.

Benchmark the ZK-regex over TLS. We implement our ZK-regex protocol over TLS by integrating the shortcut channel opening in our ZK-regex circuit. We evaluate the concrete proof size and prover time with varying m and n. Cost of our proof now consists of two parts: proving a private string is the decryption of a TLS ciphertext and proving a public regular expression matches that private string. Although the addition of a channel opening subcircuit imposes some overhead, we show it is still practical by evaluating our ZK-regex over TLS in a realistic DNS policy checking setting.

We evaluated the prover time and size of our ZK-regex deployed over TLS 1.3 with various input strings and regular expressions length. The result appears in Figure 4.4. When ZK-regex over TLS is applied to DNS policy checking using the Pi-hole pattern set, the longest pattern with $n = 97$ and $m = 128$ yielded a proof that took only 2.88 seconds to generate and had a size of just 1372 KB.

We remark that the overhead imposed by the channel opening circuit is independent of n and only linear to m. We further remark that there is a natural trade-off between proof size and prover time: increasing the number of parties in MPCitH will decrease the proof size but it will also increase prover time. This trade-off makes ZK-regex versatile and well-suited for a range of settings with different priorities.

Bandwidth(Mbps)	50	100	500	1K	5K
OT	**18.2**	**9.1**	**8.9**	**8.8**	**8.4**
OS	568.4	284.2	56.8	28.4	11.8

Table 4.3: The running time (seconds) of the protocol with subject to the bandwidth. The pattern length $n = 255$ and the string length $m = 256$.

4.6.3 Performance Evaluation of Secure-Regex

We implement our privacy-preserving pattern match protocols and demonstrate their performance through a series of experiments. Our use of garbled circuits protocol is built on top of the emp-toolkit [217]. We implement the 1-out-of-$n + 1$ oblivious transfer (OT) protocol described in Section 4.6.1, and use the oblivious stack proposed in [232] and open sourced from [163]. We use two Amazon EC2 `m5.xlarge` instances located in the same region to act as the string holder and the pattern holder. The instances are equipped with 4 vCPUs with clock speed up to 3.1 GHz, 16 GiB RAM and up to 10 Gbps network bandwidth. We use only a single thread. Meanwhile, we use the Linux traffic control tool `tc` to control the network bandwidth and create latency to simulate the wide area network (WAN).

Benchmark the protocols. We first fix the network bandwidth to 1 Gbps and show the performance with regard to variables n (pattern length) and m (string length). We report the running time and communication overhead for each execution for both approaches (OT-based or OS-based). Recall that asymptotically, the communication complexity of the OT-based approach is $\mathcal{O}(mn(n + \kappa \log |\Sigma|))$ and the OS-based approach is $O(\kappa mn(\log n + \log |\Sigma|))$. Figure 4.5 demonstrates how the protocols scale with the increasing of the input string length m. We choose the parameters $(n = 2^i - 1, m = 2^j)$ where $i \in \{6, 8\}$ and $j \in [5, 12]$. Both the running time and the communication overhead for two protocols are linear to m, which aligns with our analysis. Figure 4.6 shows the performance with the increased pattern length n. We choose the parameters $(n = 2^i - 1, m = 2^j)$ where $i \in [5, 13]$ and $j \in \{6, 8\}$. For short patterns, the communication overhead of OT-based approach is dominated by the character transition, thus is still linear to n. As n increases, the cost of epsilon transition dominates thus the overhead becomes quadratic to n. The performance of OS-based approach complies with our analysis. Overall the OT-based approach is more efficient for short patterns, but is outperformed by OS-based when $\log n > 12$.

Next, we fix the pattern length $n = 255$ and string length $m = 256$. To demonstrate how our protocols react to the change in the network bandwidth, we report the running time under different network settings. In Table 4.3, we report the running time of two protocols while increasing the network bandwidth. The performance of the OT-based protocol does not change when the bandwidth is higher than 100 Mbps due to its low communication overhead. The running time of the OS-based protocol decrease as the bandwidth is raised from 50 Mbps to 5000 Mbps, which shows that communication is its bottleneck.

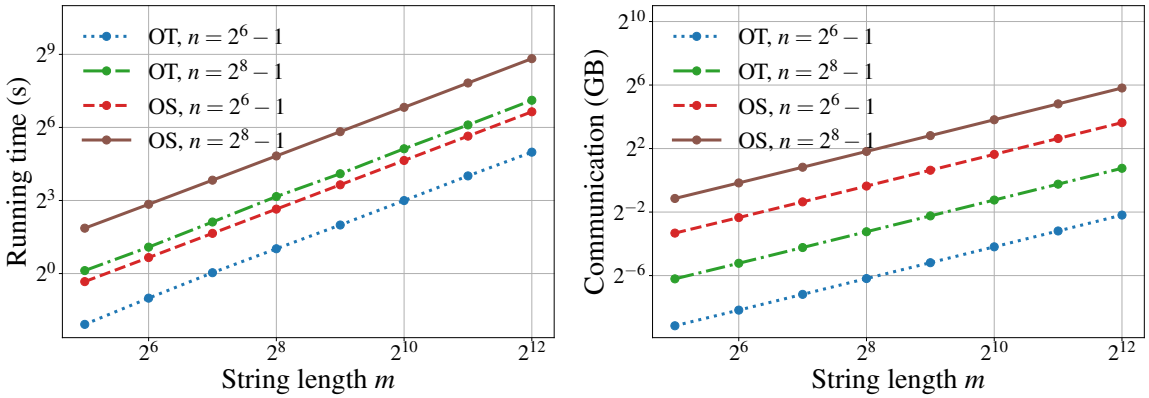


Figure 4.5: The running time (in seconds) and communication overhead (in gigabytes) of our OT-based and OS-based protocol with subject to the change of string length (m). The network bandwidth is fixed to 1 Gbps.

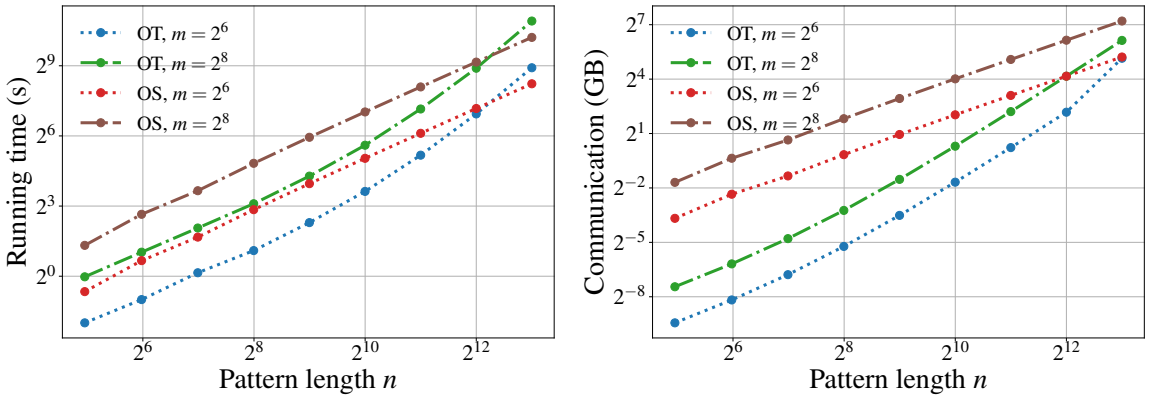


Figure 4.6: The running time (in seconds) and communication overhead (in gigabytes) of our OT-based and OS-based protocol subject to changes in pattern length (n). The network bandwidth is fixed to 1 Gbps.

Scheme	0 ms	2 ms	20 ms	40 ms	60 ms
OT	**0.5**	17.2	167.3	333.75	500.2
OS	1.5	**1.6**	**1.7**	**2.8**	**14.2**

Table 4.4: The running time (seconds) of the protocol with subject to the network latency. The pattern length $n = 63$ and the string length is $m = 64$.

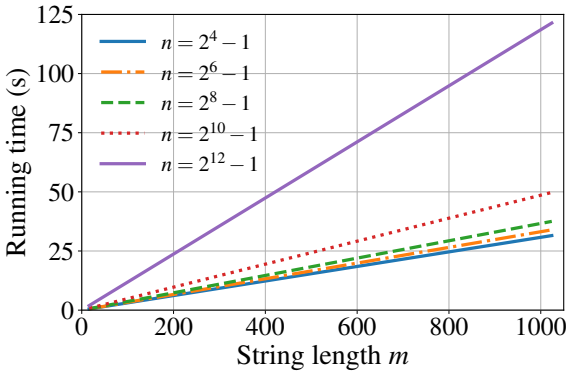


Figure 4.7: The running time (in seconds) of our OT-based protocol applied to regular expressions (RE) from the SNORT system. The network bandwidth is set to be 1 Gbps.

The influence of the network latency on our protocol is demonstrated in Table 4.4. We fix the pattern length $n = 63$ and string length $m = 64$. We choose the maximum latency to be 60 milliseconds, which is the round-trip delay between the US west and east coast. This is a proper simulation of real-world WAN. The OT-based protocol has a number of round-trips linear to the pattern length n and string length m, thus it deteriorates quickly when the latency increases. The OS-based protocol is less affected by the latency due to its constant round of communication and performs better than OT-based approach in all delayed network settings.

Overall, the OT-based approach shows better performance in low-bandwidth, low-latency, and short-pattern settings, while the OS-based protocol is more efficient in high-bandwidth, high-latency and long-pattern settings.

Benchmarks against SNORT. We demonstrate the use of secure-regex in resolving privacy concerns in real-world intrusion detection. A pattern holder holds rule-based regular expressions and can identify potential threats from incoming network packets, while a string holder holds confidential network packets from clients. To protect both the rules and packet contents, the two parties can use secure-regex protocols to perform secure pattern matching.

We use the open-source SNORT PCRE (Perl Compatible Regular Expressions) to showcase the use of regular expressions in intrusion detection [58]. SNORT PCRE is a sophisticated form of regular expressions with advanced features like backreferences and recursive patterns. Our focus is on classical regular expressions, which typically support only disjunction, concatenation, and loop operations. We rewrite the SNORT PCRE into our format described in Section 4.2.1. This results in 416 SNORT PCRE that are suitable for use in our secure-regex protocol, with 356 (85.5%) of them having

a length of $\leq 2^{12}$ bytes.

We benchmark our secure-regex protocol against regular expressions of SNORT, ignoring the regex that has length $> 2^{12}$. We show in Figure 4.7 the running time of secure-regex (OT-based approach) with partial SNORT PCRE against input string of varied length. Note that our secure-regex implementation requires the length of regular expressions padded to almost a power of 2. For clarity we only benchmark for $n = 2^j - 1, j \in \{4, 6, 8, 10, 12\}$.

4.7 Related Work

Previous works have proposed various secure two-party pattern matching protocols (e.g., [27, 228, 213, 116, 145, 68, 194]), which support common string operations such as exact matching, substring matching, approximate matching, and wildcard matching. However, these methods are less expressive than regular expression pattern matching, limiting their real-world applicability. For example, DNA tandem repeats, which are genetic markers used to track inheritance in families, cannot be expressed by simple string matching and require a formalism that supports repetition.

Protocol	Comm. Complexity
[151]	$O(\kappa n(n\lvert\Sigma\rvert + n + 1)m)$
[195]	$O(\kappa n^2(m + \lvert\Sigma\rvert))$
This work (from OS)	$O(\kappa mn(\log n + \log \lvert\Sigma\rvert)$
This work (from OT)	$O(mn(n + \kappa \log \lvert\Sigma\rvert))$

Table 4.5: Complexities for secure two-party regular expression pattern matching protocols. All protocols except for [195] mostly use symmetric key primitives as building blocks. We only consider NFA-based algorithms, as DFA-based algorithms exhibit complexity proportional to the size of the DFA, which can be exponential in the size of the regular expression.

In contrast, some works support regular expression pattern matching (e.g., [139, 211, 89, 176, 151, 231]) through secure evaluation of deterministic finite automata (DFA). However, their complexity is at least linear to the size of the DFA, which can be exponential to the size of the regular expression. Some works (e.g., [151, 195]) have presented secure protocols for general NFA evaluation, but they either use computationally-intensive homomorphic encryption or incur communication complexity of $\Omega(mn^2\kappa)$ (where κ is the security parameter), making them less practical than our protocols. A comparison to other NFA-based pattern matching protocols can be found in Table 4.5.

To the best of our knowledge, no prior work provides a complete solution for regular expression pattern matching in the ZKP setting. Both [206] and [141] mention regular expression pattern matching in their ZK algorithms without instantiating this functionality. Toots et al. design a ZK algorithm that proves whether a string is accepted by a nondeterministic finite automata (NFA) [210]. However, it does not support private epsilon transition thus its application is limited.

4.8 Conclusion

In this paper, we present secure-regex and ZK-regex, two protocols for private regular expression pattern matching in different privacy settings. Our OT-based and OS-based secure-regex protocol can efficiently perform pattern matching even when the regular expression needs to be private. Our Zero-knowledge proof pattern matching protocol ZK-regex further boosts the efficiency in settings where the regular expression can be made public.

Based on the encouraging theoretical and concretely efficient results we obtained, we aim to continue this line of work in two ways: (i) other more expressive language grammars, such as regular expressions with back-references, resembling common formalisms found in many programming languages, and (ii) quantitative pattern matching outputs, such as the number of substrings in the input strings matching the given regular expression.

We have presented two protocols for privacy-preserving regular expression matching in various privacy settings. Our secure-regex protocol efficiently performs pattern matching while ensuring the privacy of the regular expression. Meanwhile, our ZK-regex protocol further improves efficiency in cases when the regular expression is public. Our results demonstrate the viability of these protocols in practical deployment, and we look forward to exploring further applications of this work in other areas, such as quantitative pattern matching and pattern matching of languages with more expressive grammar.

Chapter 5

Privacy-preserving CTL model checking

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.

5.1 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 [26] as the use of cryptography to secure user data has come under regulations such as the GDPR [83]. A classic domain for formal methods is cyberphysical systems in aerospace engineering, transportation, medicine, and industrial control systems [79, 222]. 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 [79, 222]. 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 [130].

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 [18, 80, 81, 161]. The use of such tools during review may prevent proliferation of harmful software for the benefit of both users and platforms. Further, the example of app store maintainers raises that privacy concerns regarding verification may go in both directions. A tool such as PiOS [80] 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) [60, 63, 62]. 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) [35, 95, 99, 103, 156, 157, 174, 227], 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 [45].

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 §5.7.

In summary, this chapter 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 §5.2 we introduce both model checking of CTL and our necessary cryptographic primitives. Our contributions begin in §5.3, with data-oblivious model checking subroutines based on prior work for oblivious graph algorithms. We then give our full model checking construction in §5.4. We follow with discussion of our implementation and experimentation in §5.5, cover related work in §5.6, and consider potential future work and conclude in §5.7.

5.2 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* [63, 62, 60, 188]. 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 [61, 202]. 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 [62, 63].

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 chapter. 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 [35, 95, 99, 103, 156, 157, 174, 227]. 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.

5.2.1 Model Checking

A *Kripke structure* [62, 63] 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 $L : S \to 2^q$ a labeling function mapping states to subsets of the q available labels. We note that $O(n^2) = |\mathcal{M}|$.

Example 1. *Consider a toy authentication program implementing session management. When an unknown user arrives they are prompted for credentials, and re-prompted 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 5.1.*

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* $\mathbf{A}\ \ell_1\ \mathbf{U}\ \ell_3$. *Our example structure does not meet this specification as the user need not ever successfully authenticate, in which case a session is never established.*

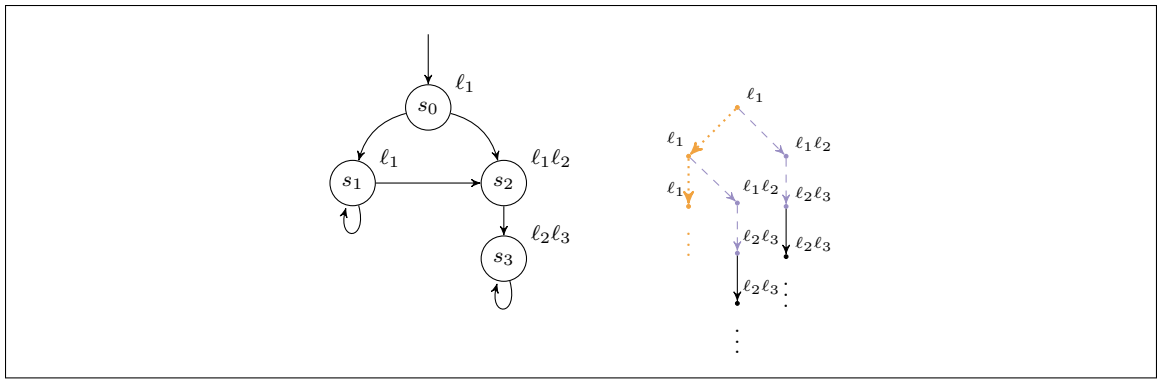


Figure 5.1: A Kripke structure (left) modeling the program given in Example 1, and its corresponding computation tree (right). The vertices and edges in the computation tree show a failed checking for $\mathbf{A}\ \ell_1\ \mathbf{U}\ \ell_3$, which holds at s_2 and s_3 (dashed blue) but not s_0 or s_1 (dotted orange).

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 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 [60], 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

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

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 $\texttt{check}_{\text{CTL}}(\mathcal{M}, \phi) = 1$ if and only if $\mathcal{M} \models \phi$. Our example formula $\phi = \mathbf{AG}\ (userdata \rightarrow \mathbf{AG}\ \neg network)$ 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

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

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 [62, 63] 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 22, up to the 'quantified until' operators of **EU** and **AU** which are given in Algorithm 1 and Algorithm 5 respectively. That every temporal operator in CTL is quantified has the crucial quality that all CTL formulas are *state formulas* — 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 [62, 63] for discussion of state and path formulas.

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 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,

Algorithm 22: The $\texttt{check}_{\text{CTL}}$ algorithm up to the quantified until operator subroutines and various helper functions.

```
1
2  Function check_CTL(M, φ ):
3      o^φ ← rec(M, φ)
4      sat ← 1
5      for i ∈ [n] do
6          sat ← sat ∧ (o^φ[i] ∨ ¬M.S[i].inI)
7      return sat
8
9  Function checkAND(M, l^ψ, r^φ):
10     for i ∈ [M.n] do
11         o[i] ← l^ψ[i] ∧ r^φ[i]
12     return o
13
14 Function checkNOT (M, r^φ):
15     for i ∈ [M.n] do
16         o[i] ← ¬r^φ[i]
17     return o
18
19 Function checkEX (M, r^φ):
20     for i ∈ [M.n] do
21         for j ∈ [M.n] do
22             o[i] ← o[i] ∨ (M.δ[i][j] ∧ r^φ[j])
23     return o
24
25 Function rec (M, φ):
26     (op, ψ, φ) ← parse(φ)
27     if op = ∧ then
28         return checkAND(M, rec(M, ψ), rec(M, φ))
29     else if op = ¬ then
30         return checkNOT(M, rec(M, φ))
31     else if op = EX then
32         return checkEX(M, rec(M, φ))
33     else if op = EU then
34         return checkEU(M, rec(M, ψ), rec(M, φ))
35     else if op = AU then
36         return checkAU(M, rec(M, ψ), rec(M, φ))
37     else
           /* op is an atom ℓ_k.                                    */
38         k ← label(op)
39         for i ∈ [M.n] do
40             o[i] ← M.ℓ̄_k(M.S[i])
41         return o
```

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 1 and Algorithm 5 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 [62, 63], and we refer to them the interested reader who finds our discussion overly terse. We conclude with the following theorem, and refer to [61] for a proof.

Theorem 5.2.1. *For any Kripke structure $\mathcal{M} = (S, I, \delta, L)$ and CTL formula ϕ*

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

5.2.2 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*

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

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 $\texttt{view}_A(\Pi, x_0, B(1^\lambda, x_1))$ 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 $\texttt{view}_A(\Pi, x_0, B(1^\lambda, x_1))$ and $\texttt{view}_A(\Pi, x_0, \textsc{Sim}_B(1^\lambda, x_0, y))$ 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 $x_1 \in X_1$, $\texttt{view}_A(\Pi, x_0, B(1^\lambda, x_1))$ and $\texttt{view}_A(\Pi, x_0, \textsc{Sim}_B(1^\lambda, x_0, y))$ are *computationally indistinguishable*, or

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

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

We notate the 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 [93, 156, 157] 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 [35, 95, 99, 103, 156, 157, 174, 227]. Of these primitives, our work will make use of garbled circuits with oblivious transfer in the semihonest model due to Yao [156, 227]. However, we will not simply be

rendering the decision procedure $\texttt{check}_{\text{CTL}}(\mathcal{M}, \phi)$ as a circuit. Rather, we will employ constant-sized binary circuits for certain intermediary computations and then use data-oblivious computation [99] and compositionality [55] to combine these intermediary results to execute the full checking.

For concision we will not delve into the details of garbled circuits or their underlying cryptographic components. Rather, for the remainder of the discussion we will assume access to a protocol $(y \parallel y) \leftarrow \text{GC}(\texttt{c}; x_0 \parallel x_1)$, 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 $\text{GC}(\texttt{c}; \cdot \parallel \cdot)$ maintains the asymptotic complexity of $\texttt{c}$.

Data-Oblivious Computation Our treatment of data-oblivious computation follows that of Goldreich and Ostrovsky in [99] 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 $\texttt{write}$, $\texttt{read}$, or $\texttt{halt}$, $a \in [2^k]$ 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 $\texttt{halt}$ 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 [103]. 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 [35, 95, 156, 157, 227]. 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. $\texttt{Reconstruct}^{otp}(a, b)$ 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 $\Pi_S^{prf} = (\texttt{Gen}^{prf}, \texttt{Share}^{prf}, \texttt{Reconstruct}^{prf})$ requires the existence of a *pseudorandom function* (PRF). This is a keyed function $\text{PRF} : \{0,1\}^\lambda \times \{0,1\}^y \to \{0,1\}^z$ for $\lambda, y, z \in \mathbb{N}$ for which the distribution of $\text{PRF}(sk, x)$ is computationally indistinguishable from uniformly random for an adversary which does not know sk. We let $\texttt{Gen}^{prf}(1^\lambda)$ be the key generation algorithm for the PRF which is run at setup. To share a value $x \in \{0,1\}^z$, denoted $[[x]]$, $\texttt{Share}_{sk}^{prf}(x)$ uniformly samples $r \xleftarrow{\$} \{0,1\}^y$ and computes $c \leftarrow \text{PRF}(sk, r) \oplus x$. One share is then sk, and the other is $\langle c, r\rangle$. To reconstruct the value, $\texttt{Reconstruct}_{sk}^{prf}(\langle c, r\rangle)$ computes $x \leftarrow \text{PRF}(sk, r) \oplus c$.

The sharing and reconstruction algorithms given are identical to a standard con-

```
1  Function checkEU(M, l^ψ, r^ϕ):
2      o ← r^ϕ
3      R ← {i | r^ϕ[i] = 1}
4      K ← ∅
5      while R ≠ ∅ do
6          i ← draw(R)
7          for j ∈ {j' | (s_j', s_i) ∈ δ} do
8              if l^ψ[j] ∧ j ∉ K then
9                  o[j] ← 1
10                 R ← R ∪ {j}
11                 K ← K ∪ {j}
12         R ← R \ {i}
13     return o
```

Algorithm 1: The `checkEU` algorithm.

```
1  Function
       obcheckEU(n, M_π1, l^ψ_π1, r^ϕ_π1, idxs_π2):
2      R̂ ← r^ϕ_π1; ψ̂ ← l^ψ_π1; K̂ ← 0^n
3      for t ∈ [n] do
4          i, i' ← ⊥; m, m' ← 0
5          for c ∈ [n] do
6              b1 ← R̂[c] ∧ ¬K̂[c]; b2 ← (idxs_π2[c] > m)
7              i ← c b1 b2 + i(1 − b1 b2)
8              m ← idxs_π2[c]·b1 b2 + m(1 − b1 b2)
9          for c' ∈ [n] do
10             b'1 ← ¬R̂[c'] ∧ ¬K̂[c']; b'2 ← (idxs_π2[c'] > m')
11             i' ← c' b'1 b'2 + i'(1 − b'1 b'2)
12             m' ← idxs_π2[c'] · b'1 b'2 + m'(1 − b'1 b'2)
13         b3 ← (i = ⊥)
14         i* ← i' b3 + i · (1 − b3)
15         for j ∈ [n] do
16             R̂[j] ← R̂[j] ∨ (R̂[i*] ∧ M_π1.δ[j][i*] ∧ ψ̂[j])
17         K̂[i*] ← 1
18     return R̂
```

Algorithm 2: The oblivious `obcheckEU` algorithm.

struction for producing a *semantically secure symmetric key scheme for multiple encryptions* out of a PRF [135]. 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 [186]. 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 [146, 233].

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

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

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.

5.2.3 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[3] — 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 $\texttt{view}_A(\Pi, x_0, B(1^\lambda, x_1))$ 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 $\texttt{view}_A(\Pi, x_0, B(1^\lambda, x_1))$ and $\texttt{view}_A(\Pi, x_0, \textsc{Sim}_B(1^\lambda, x_0, y))$ 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 $x_1 \in X_1$, $\texttt{view}_A(\Pi, x_0, B(1^\lambda, x_1))$

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

and $\texttt{view}_A(\Pi,\, x_0,\, \textsc{Sim}_B(1^\lambda,\, x_0,\, y))$ are *computationally indistinguishable*, or

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

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 [93, 156, 157] 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 [35, 95, 99, 103, 156, 157, 174, 227]. Of these primitives, our work will make use of garbled circuits with oblivious transfer in the semihonest model due to Yao [156, 227]. However, we will not simply be rendering the decision procedure $\texttt{check}_{\text{CTL}}(\mathcal{M},\, \phi)$ as a circuit. Rather, we will employ constant-sized binary circuits for certain intermediary computations, and then use data-oblivious computation [99] and compositionality [55] to combine these intermediary results to execute the full checking.

For concision we will not delve into the details of garbled circuits or their underlying cryptographic components. Rather, for the remainder of the discussion we will assume access to a protocol $(y \parallel y) \leftarrow \text{GC}(\texttt{c};\, x_0 \parallel x_1)$, 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 $\text{GC}(\texttt{c};\, \cdot \parallel \cdot)$ maintains the asymptotic complexity of $\texttt{c}$.

Data-Oblivious Computation Our treatment of data-oblivious computation follows that of Goldreich and Ostrovsky in [99] 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 (`write`, a, v) command MEM copies v into the word addressed by a, upon a (`read`, a, $\cdot$) command returns the current value in the word addressed by a, and upon (`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 [103]. 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 [35, 95, 156, 157, 227]. 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. $\texttt{Reconstruct}^{otp}(a, b)$ 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 $\Pi_S^{prf} = (\texttt{Gen}^{prf}, \texttt{Share}^{prf}, \texttt{Reconstruct}^{prf})$ requires the existence of a *pseudorandom function* (PRF). This is a keyed function $\text{PRF} : \{0,1\}^\lambda \times \{0,1\}^y \to \{0,1\}^z$ for $\lambda, y, z \in \mathbb{N}$ for which the distribution of $\text{PRF}(sk, x)$ is computationally indistinguishable from uniformly random for an adversary which does not know sk. We let $\texttt{Gen}^{prf}(1^\lambda)$ be the key generation algorithm for the PRF which is run at setup. To share a value $x \in \{0,1\}^z$, denoted $[[x]]$, $\texttt{Share}_{sk}^{prf}(x)$ uniformly samples $r \xleftarrow{\$} \{0,1\}^y$ and computes $c \leftarrow \text{PRF}(sk, r) \oplus x$. One share is then sk, and the other is $\langle c, r\rangle$. To reconstruct the value, $\texttt{Reconstruct}_{sk}^{prf}(\langle c, r\rangle)$ computes $x \leftarrow \text{PRF}(sk, r) \oplus c$.

The sharing and reconstruction algorithms given are identical to a standard construction for producing a *semantically secure symmetric key scheme for multiple encryptions* out of a PRF [135]. 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 [186]. 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 [146, 233].

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

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

```
1  Function checkEU(M, l^ψ, r^φ):
2      o ← r^φ
3      R ← {i | r^φ[i] = 1}
4      K ← ∅
5      while R ≠ ∅ do
6          i ← draw(R)
7          for j ∈ {j' | (s_j', s_i) ∈ δ} do
8              if l^ψ[j] ∧ j ∉ K then
9                  o[j] ← 1
10                 R ← R ∪ {j}
11                 K ← K ∪ {j}
12         R ← R \ {i}
13     return o
```

Algorithm 3: The `checkEU` algorithm.

```
1  Function obcheckEU
       (n, M_π1, l^ψ_π1, r^φ_π1, idxs_π2):
2      R̂ ← r^φ_π1; ψ̂ ← l^ψ_π1; K̂ ← 0^n for t ∈ [n] do
3          i, i' ← ⊥; m, m' ← 0
4          for c ∈ [n] do
5              b_1 ← R̂[c] ∧ ¬K̂[c]; b_2 ←
                 (idxs_π2[c] > m)
6              i ← cb_1b_2 + i(1 − b_1b_2)
7              m ← idxs_π2[c]·b_1b_2 + m(1 − b_1b_2)
8          for c' ∈ [n] do
9              b'_1 ← ¬R̂[c'] ∧ ¬K̂[c']; b'_2 ←
                 (idxs_π2[c'] > m')
10             i' ← c'b'_1b'_2 + i'(1 − b'_1b'_2)
11             m' ←
                 idxs_π2[c'] · b'_1b'_2 + m'(1 − b'_1b'_2)
12         b_3 ← (i = ⊥)
13         i* ← i'b_3 + i · (1 − b_3)
14         for j ∈ [n] do
15             R̂[j] ←
                 R̂[j] ∨ (R̂[i*] ∧ M_π1.δ[j][i*] ∧ ψ̂[j])
16         K̂[i*] ← 1
17     return R̂
```

Algorithm 4: The oblivious `obcheckEU` algorithm.

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.[4] 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.

5.3 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

[4]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.

$\texttt{check}_{\text{CTL}}$ are either data-oblivious or may be rewritten to be so. This allows us direct adaption of these subroutines into (a part of) an MPC protocol using the preferred Π_S^{otp} secret sharing scheme.

As shown in Algorithm 1 and Algorithm 5, the `checkEU` and `checkAU` subroutines branch in a manner dependent on the truth values of both their subformulas 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 22 are data-oblivious.[5]

As noted in §5.2, 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. [45]. 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 2 to follow the discussion as it formally presents the oblivious algorithm.

5.3.1 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 subformulas 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 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,

[5] 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.

$r^{\phi}_{\pi_1}[i]$ 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 2, if $\hat{R}[c] \wedge \neg\hat{K}[c]$ then $s_{\pi_1(c)} \models \mathbf{E}\ \psi\ \mathbf{U}\ \phi$ and, as per §5.2, $s_{\pi_1(c)}$ has been 'made ready' but has not yet been 'made active'. To match Line 6 of Algorithm 1, 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:

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

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 1, 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

```
1  Function checkAU(M, l^ψ, r^φ):
2      o ← r^φ
3      R ← {i | r^φ[i] = 1}
4      K ← ∅
5      d ← degrees(M)
6      while R ≠ ∅ do
7          i ← draw(R) for
             j ∈ {j' | (s_j', s_i) ∈ δ} do
8            if l^ψ[j] ∧ j ∉ K then
9                d[j] ← d[j] − 1
10               if d[j] = 0 then
11                   o[j] ← 1    R ← R ∪ {j}
12                   K ← K ∪ {j}
13         R ← R \ {i}
14     return o
```

Algorithm 5: The `checkAU` algorithm.

```
1  Function
     obcheckAU(n, M_π1, l^ψ_π1, r^φ_π1, idxs_π2):
2      R̂ ← r^φ_π1; ψ̂ ← l^ψ_π1; K̂ ← 0^n
3      for t ∈ [n] do
4          i, i' ← ⊥; m, m' ← 0
5          for c ∈ [n] do
6              b1 ← R̂[c] ∧ ¬K̂[c]; b2 ←
                 (idxs_π2[c] > m)
7              i ← cb1b2 + i(1 − b1b2)
8              m ← idxs_π2[c]·b1b2 + m(1 − b1b2)
9          for c' ∈ [n] do
10             b'1 ← ¬R̂[c'] ∧ ¬K̂[c']; b'2 ←
                 (idxs_π2[c'] > m')
               i' ← c'b'1b'2 + i'(1 − b'1b'2)m' ←
               idxs_π2[c'] · b'1b'2 + m'(1 − b'1b'2)
11         b3 ← (i = ⊥) i* ← i'b3 + i · (1 − b3)
           for j ∈ [n] do
12             b4 ← R̂[i*] ∧ M_π1.δ_π[j][i*]
               M_π1.d[j] ← M_π1.d[j] − b4
                   R̂[j] ←
               R̂[j] ∨ (ψ̂[j] ∧ M_π1.d[j] = 0)
13         K̂[i*] ← 1
14     return R̂
```

Algorithm 6: The oblivious `obcheckAU` algorithm.

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 $\texttt{obcheck}_{\mathrm{CTL}}(\mathcal{M}, \phi)$ be $\texttt{check}_{\mathrm{CTL}}(\mathcal{M}, \phi)$ where `checkEU` and `checkAU` are replaced with their oblivious variants. We now claim the following theorems.

Theorem 5.3.1. *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.

Theorem 5.3.2. *For any Kripke structure $\mathcal{M} = (S, I, \delta, L)$ and CTL formula ϕ*

1. $\texttt{obcheck}_{\mathrm{CTL}}(\mathcal{M}, \phi) = 1$ *if and only if* $\mathcal{M} \models \phi$*; and*

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

Although asymptotically Theorem 5.3.2 is equivalent to Theorem 5.2.1 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 [62, 63].

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 [138, 218]. 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.

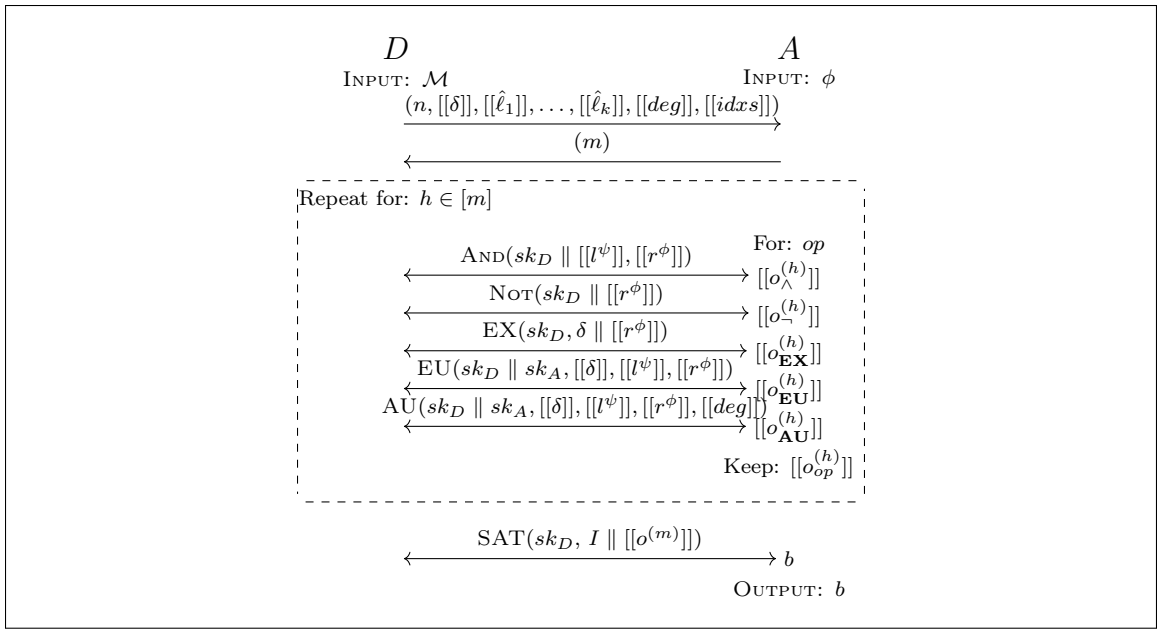
5.4 An MPC Protocol for CTL Model Checking

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

Function $\textsc{Init}(\mathcal{M} \parallel \phi)$**:**
  $D, A : sk_D \leftarrow \texttt{Gen}^{prf}(1^\lambda), sk_A \leftarrow \texttt{Gen}^{prf}(1^\lambda)$
  for $i \in [n]$ **do**
    $D :\ [[deg[i]]] \leftarrow \texttt{Share}_{sk_D}^{prf}(\mathcal{M}.S[i].deg)$
    $D : [[idxs[i]]] \leftarrow \texttt{Share}_{sk_D}^{prf}(i)$
    for $j \in [n]$ **do**
      $D : [[\delta[i][j]]] \leftarrow \texttt{Share}_{sk_D}^{prf}(\mathcal{M}.\delta[i][j])$
    for $k \in [q]$ **do**
      $D : [[\hat{\ell}_k[i]]] \leftarrow \texttt{Share}_{sk_D}^{prf}(\mathcal{M}.\overline{\ell}_k(s_i))$
  $D :$ send $\langle n \rangle$
  $D :$ send ciphertexts of $\langle [[\delta]], [[\hat{\ell}_1]], \ldots, [[\hat{\ell}_q]], [[deg]], [[idxs]] \rangle$
  $A :$ send $\langle m \rangle$

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 subformulas ϕ_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 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 **Function** AND($sk_D \parallel [[l^\psi]], [[r^\phi]]$)**:**
2 **for** $i \in [n]$ **do**
3 $[l^\psi[i]] \leftarrow$ SIMPLIFY$(sk_D, [[l^\psi[i]]])$ $[r^\phi[i]] \leftarrow$ SIMPLIFY$(sk_D, [[r^\psi[i]]])$
4 $[o[i]] \leftarrow [l^\psi[i]] \wedge [r^\phi[i]]$
5 $[[o[i]]] \leftarrow$ COMPLICATE$(sk_D, [o[i]])$
6 **Function** NOT($sk_D \parallel [[r^\phi]]$)**:**
7 **for** $i \in [n]$ **do**
8 $[r^\phi[i]] \leftarrow$ SIMPLIFY$(sk_D, [[r^\phi[i]]])$
9 $[o[i]] \leftarrow \neg[r^\psi[i]]$
10 $[[o[i]]] \leftarrow$ COMPLICATE$(sk_D, [o[i]])$
11 **Function** EX($sk_D, \delta \parallel [[r^\phi]]$)**:**
12 **for** $i \in [n]$ **do**
13 $[r^\phi[i]] \leftarrow$ SIMPLIFY$(sk_D, [[r^\phi[i]]])$
14 $[o[i]] \leftarrow 0$
15 **for** $j \in [n]$ **do**
16 $[o[i]] \leftarrow [o[i]] \vee ([r^\phi[j]] \wedge \delta[i][j])$
17 $[[o[i]]] \leftarrow$ 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 $\quad x \leftarrow \texttt{Reconstruct}^{prf}_{sk_t}([[x]])$`/* sk_t is key of 'to' party */`
3 $\quad [[x']] \leftarrow \texttt{Share}^{prf}_{sk_f}(x)$`/* sk_f is key of 'from' party */`
4 $\quad$ **return** $[[x']]$
5 **Function** PERM$(sk_D, \pi_D \parallel sk_A, \pi_A, \pi_{A'}, [[\hat{x}]]_{sk_D})$**:**
6 $\quad A : [[\hat{x}_{\pi_A}]]_{sk_D} \leftarrow \pi_A([[\hat{x}]]_{sk_D})$
7 $\quad$ **for** $i \in [n]$ **do**
8 $\quad\quad ([[\hat{x}_{\pi_A}[i]]]_{sk_A}) \leftarrow \mathrm{GC}(\texttt{transfer}; sk_D \parallel sk_A, [[\hat{x}_{\pi_A}[i]]]_{sk_D})$
9 $\quad D : [[\hat{x}_{\pi_A\pi_D}]]_{sk_A} \leftarrow \pi_D([[\hat{x}_{\pi_A}]]_{sk_A})$
10 $\quad$ **for** $i \in [n]$ **do**
11 $\quad\quad ([[\hat{x}_{\pi_A\pi_D}[i]]]_{sk_D}) \leftarrow \mathrm{GC}(\texttt{transfer}; sk_D, [[\hat{x}_{\pi_A\pi_D}[i]]]_{sk_A} \parallel sk_A)$
12 $\quad A : [[\hat{x}_{\pi_A\pi_D\pi_{A'}}]]_{sk_D} \leftarrow \pi_{A'}([[\hat{x}_{\pi_A\pi_D}]]_{sk_D})$
13 **Function** ALTPERM**:**
14 $\quad sk_D, \pi_D \parallel sk_A, \pi_A, [[\hat{x}]]_{sk_D}$
15 ...same as Lines 7-10...
16 **for** $i \in [n]$ **do**
17 $\quad ([\hat{x}_{\pi_A\pi_D}[i]]) \leftarrow \textsc{Simplify}(sk_A, [[\hat{x}_{\pi_A\pi_D}[i]]]_{sk_A})$

For 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 §5.3.

1 **Function** EU$(sk_D \parallel sk_A, [[\delta]], [[l^\psi]], [[r^\phi]])$**:**
2 $D, A : \pi_{1D}, \pi_{2D} \xleftarrow{\$} S_n, \pi_{1A}, \pi_{2A} \xleftarrow{\$} S_n$
3 $[idxs_{\pi_2}] \leftarrow \text{AltPerm}(sk_D, \pi_{2D}^{-1} \parallel sk_A, \pi_{2A}^{-1}, [[idxs]])$
4 $[l^\psi_{\pi_1}] \leftarrow \text{AltPerm}(sk_D, \pi_{1D}^{-1} \parallel sk_A, \pi_{1A}^{-1}, [[l^\psi]])$
5 $[r^\phi_{\pi_1}] \leftarrow \text{AltPerm}(sk_D, \pi_{1D}^{-1} \parallel sk_A, \pi_{1A}^{-1}, [[r^\phi]])$
6 $[\delta_{\pi_1}] \leftarrow \text{AltPerm}(sk_D, \pi_{1D}^{-1} \parallel sk_A, \pi_{1A}^{-1}, [[\delta]])$
7 ...same as Algorithm 2 with Π_S^{otp} shares and revealed i^*...
8 $[[\hat{R}]] \leftarrow \text{Complicate}(sk_D, [\hat{R}])$
9 $[[o]] \leftarrow \text{Perm}(sk_D, \pi_{1D} \parallel sk_A, \mathbf{1}, \pi_{1A}, [[\hat{R}]])$
10 **Function** AU$(sk_D \parallel sk_A, [[\delta]], [[l^\psi]], [[r^\phi]], [[deg]])$**:**
11 ...same as Lines 2-6...
12 $[deg] \leftarrow \text{AltPerm}(sk_D, \pi_{1D}^{-1} \parallel sk_A, \pi_{1A}^{-1}, [[deg]])$
13 ...same as Algorithm 6 with Π_S^{otp} shares and revealed i^*...
14 ...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 $[sat] \leftarrow 1$
3 **for** $i \in [n]$ **do**
4 $[o[i]] \leftarrow \text{Simplify}(sk_D, [[o^\phi[i]]])$
5 $[sat] \leftarrow [sat] \wedge ([o^\phi[i]] \vee \neg \mathcal{M}.S[i].inI)$
6 $(\cdot \parallel b) \leftarrow \text{Reveal}([sat])$

At Line 5, $\mathcal{M}.S[i].inI$ is a private input of D. The output of Line 6 completes the model checking protocol.

5.4.1 Correctness, Complexity, and Security

Our result with respect to correctness and complexity is an analogue of Theorem 5.3.2.

Theorem 5.4.1. *For any Kripke structure $\mathcal{M} = (S, I, \delta, L)$ and CTL formula ϕ, let $(\cdot \parallel b) \leftarrow \text{PPMC}(sk_D, \mathcal{M} \parallel sk_A, \phi)$. Then,*

1. *$b = 1$ if and only if $\mathcal{M} \models \phi$; and*
2. *$\text{PPMC}(\mathcal{M} \parallel \phi)$ 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 5.3.2, 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 5.4.2. *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* $\text{SIM}_D(1^\lambda, \phi, b)$ *such that*

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

2. *there exists a PPT simulator* $\text{SIM}_A(1^\lambda, \mathcal{M}, \cdot)$ *such that*

$$\mathit{view}_D(\text{PPMC}, \mathcal{M}, A(1^\lambda, \phi)) \approx \mathit{view}_D(\text{PPMC}, \mathcal{M}, \text{SIM}_A(1^\lambda, \mathcal{M}, m, \cdot)).$$

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 [156]. Since we are agnostic to the roles in the GC protocol, we just refer to the appropriate simulator as $\text{GCSIM}(1^\lambda, f, y)$. 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 [135]. 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 5.4.3. *Let $\Pi = (\texttt{Gen}, \texttt{Enc}, \texttt{Dec})$ be an encryption scheme and $\lambda \in \mathbb{N}$ a security parameter. We define the experiment* $\text{IND-CPA}_{\mathcal{A},\Pi}$ *between adversary $\mathcal{A}$ and a challenger $\mathcal{C}$ by*

1. *$\mathcal{C}$ runs $sk \leftarrow \texttt{Gen}(1^\lambda)$ and samples $b \xleftarrow{\$} \{0, 1\}$. $\mathcal{C}$ then invokes $\mathcal{A}(1^\lambda)$ and exposes an encryption oracle to it.*

2. *$\mathcal{A}$ sends a pair (m_0, m_1) to $\mathcal{C}$ 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* $\text{IND-CPA}_{\mathcal{A},\Pi}$ *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* $\mathcal{A}$,

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

We now have the necessary machinery to prove THEOREM 5.4.2. 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 $h \in [\tau]$.

$\text{SIM}_D(1^\lambda, \phi, n, b)$ first constructs a random Kripke structure $\mathcal{M}'$. To accomplish this, the simulator uniformly samples bits $b_1, \ldots, b_{n^2+(q+1)n} \stackrel{\$}{\leftarrow} \{0, 1\}$ 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 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 $\text{GCSIM}(1^\lambda, f, y_{Ah})$ 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 $\text{GCSIM}(\texttt{transfer}; \cdot \parallel \cdot)$ 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 $\mathcal{H}_0 = \texttt{view}_A($ PPMC, ϕ, $\text{SIM}_D(1^\lambda, \phi, n, b))$ 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 $\mathcal{A}$ 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 $\mathcal{A}'$ with non-negligible advantage in the IND-CPA security experiment, violating our assumption.

We let $\mathcal{A}'$ be parameterized by $\mathcal{M}^*$ (and n), and it is given 1^λ on start by $\mathcal{C}$. 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 $\mathcal{C}$, 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 $\mathcal{C}$. If $b^* = 0$, then $\mathcal{A}'$ perfectly instantiates $\mathcal{H}_0$ as it

1. executes a fixed order of garbled circuit simulators;
2. uniformly generates all Π_S^{otp} shares as required;
3. generates all Π_S^{prf} shares appropriately under the challenge sk using the oracle access from $\mathcal{C}$; and
4. 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 $b^* = 1$, $\mathcal{A}'$ perfectly instantiates $\mathcal{H}_1$ by an identical argument. So, upon receipt of the distinguishing bit b'' from $\mathcal{A}$, $\mathcal{A}'$ sets its own output bit $b' = b''$. It therefore retains the non-negligible advantage of $\mathcal{A}$, and so has a non-negligible advantage in the IND-CPA experiment. We conclude that $\mathcal{A}'$ may not exist as it derives a contradiction, and so neither may $\mathcal{A}$.

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 [55]. 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. □

5.5 Implementation

We implemented our protocol using the semihonest 2PC functionality within the EMP-Toolkit [219]. For AES, we used the key-expanded Bristol Format circuit,[6] 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 [146, 233].

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

We observe a consistent cost of $\approx$ 20-30ms 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 [10, 9, 11, 106].

[6] `https://homes.esat.kuleuven.be/~nsmart/MPC/old-circuits.html`

In addition, some of these primitives are naturally computed in arithmetic circuits, which may 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.

5.6 Related Work

Recent years have seen a proliferation of work applying MPC to real-world problems, with [17] an excellent overview. This work has been enabled by developments in the efficiency of primitives [20, 128, 137, 146, 233], by the creation of usable compilers and software libraries [115] 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 [186]. 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 [37, 120].

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 [10, 9, 11, 106]. Data-oblivious graph algorithms have also received attention both generically and within a variety of problem domains [45, 51, 82, 101, 102, 223]. Also relevant is work on generic oblivious data structures [138, 218]. 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 [115] for a comprehensive SoK. Work from the programming languages community has developed languages and compilers tailored to oblivious computation and MPC [67, 158, 159, 190]. Of particular note is [123], where a model checker is used in the compilation of C programs for MPC execution.

5.7 Conclusion

We have presented an oblivious algorithm for global explicit state model checking of CTL formulas, 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. **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 [155], *verifiable secret sharing* (VSS) is also required for composition [57]. 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 [37] — 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 [132, 136].

– 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.

Chapter 6

Conclusion and Future Work

Motivated by the tension between the verifiability and privacy of programs in the real-world setting, this thesis introduces and initiates the study of privacy-preserving formal methods. It then presents a suite of privacy-preserving formal methods techniques. In the thesis, we, in particular, focus on developing new protocols for SAT solving, verifying unsatisfiability, model checking, and regular expression pattern matching with enhanced privacy.

Given the centrality of SAT solving to computing and the importance of data privacy for society, both researchers and industry software developers can benefit from creating an efficient privacy-preserving SAT solver. In this thesis, we provide a privacy-preserving SAT solver, along with a formal security definition of the privacy-preserving SAT-solving problem. The core of our privacy-preserving SAT solver is an oblivious-variant DPLL algorithm that employs three private decision heuristics. The evaluation results show that our privacy-preserving SAT solver can resolve small but practical instances.

As deciding whether a formula is satisfiable, the problem of validating the unsatisfiability of a formula is also fundamental for program verification. In this thesis, we explain our protocol for verifying the unsatisfiability of a formula in zero knowledge so that the formula and the related resolution proof remain private. In addition, the thesis shows a method for encoding of propositional clauses and resolution proofs as arithmetic polynomials, which allows us to minimize costs aggressively by revealing only the refutation's length and width.

Beyond Boolean reasoning, the thesis also develops protocols for direct verification techniques, namely, model checking and regular expression pattern matching. The privacy-preserving model checking protocol in this thesis enables adversarial parties D and A holding $\mathcal{M}$ and ϕ, respectively, to check whether $\mathcal{M} \models \phi$ while maintaining the privacy of all other meaningful information. The construction of the protocol adapts oblivious graph algorithms to provide for secure computation of global explicit state model checking with specifications in *Computation Tree Logic* (CTL). In the end, our protocol design improves on the asymptotic overhead required by generic MPC schemes.

Regular expression pattern matching is the task of checking whether an input string is a member of the language defined by an input regular expression. Privacy issues are becoming increasingly relevant for a wide range of applications. In this thesis, we provide two protocols for private regular expression pattern matching based on the oblivious stack and the oblivious transfer protocols. The former achieves the best known asymptotic complexity for secure regular expression pattern matching, while the latter attains the optimal numerical performance. These protocols protect the privacy of both strings and regular expressions. In addition, we introduce a protocol for zero-knowledge proofs of pattern matching, where the given regular expression is public to both parties, but the input string is kept private.

6.1 Future Work

We conclude this dissertation by outlining possible directions for future research.

6.1.1 Extending and Improving ppSAT

§ 2 demonstrates the feasibility and potential scalability of our privacy-preserving SAT solver. However, the capacity and efficiency of the current implementation of ppSAT solver are insufficient for many real-world scenarios, such as distributed software marketplaces, where the ppSAT solver will be used to check the correctness of private programs. In the future, we plan to develop SAT-solving tools that can solve problems on a practical scale.

As mentioned in § 2, effective heuristics are the core of SAT solver research, and one of the most important is CDCL. The CDCL heuristic learns clauses implied by the original formula and backtracks non-chronologically when conflicts arise. Each learned clause is added to the input formula as a constraint and is used to shrink the search space. Supporting CDCL is a necessary step towards developing an efficient ppSAT solver and is also a challenging task. The difficulties arise from the fact that CDCL learns clauses when backtracking occurs. Continuing to hide those learned clauses would require a deterministic schedule for adding clauses. A simple approach is to add one clause per giant step in our current implementation of the ppSAT solver while padding the dummy when no clause is learned. However, every increase to the size of the formula makes every subsequent giant step more expensive. An alternative approach could be to add clauses less frequently, perhaps keeping the largest learned in the interim and discarding the rest. Exploring this frontier will be critical to enable CDCL within a ppSAT solver.

Additionally, the CDCL learning process itself would need to be made data-oblivious. Usually, it is understood as the building of an implication graph for which a suitable cut produces the assignments to negate. Though this process may be rendered instead as a sequence of resolution operations potentially amenable to data-oblivious formulation, finding a way to perform the resolution operations without

undue overhead may require further research.

6.1.2 Extending Zero-Knowledge Proofs of Unsatisfiability to SMT formulas

Program verification often involves SMT formulas. SMT formulas are first-order logic formulas involving many theories, such as equality logic with uninterpreted functions (EUF), linear integer arithmetic (LIA), bit-vector theory (BV), and array extensionality (ArraysEX). A refutation proof for an unsatisfiable SMT formula consists of two parts. The first part is the proof of unsatisfiability for a pure Boolean propositional formula containing lemma clauses. The second part is the proof of every lemma used in the pure Boolean propositional formula. The lemma clause is obtained by applying theory-specific axioms to a subset of clauses in the input formula. To construct a system that can verify SMT refutation proof in zero knowledge, designing protocols for both parts is necessary. ZKUNSAT presented in § 3, can handle Boolean propositional formulas and thus offers a solution to the first part. The other half of privacy-preserving SMT solving is a direction for future work: we need ZKP systems for checking proofs in various theories.

In particular, we will focus on three theories: EUF, ArraysEX, and LIA. First, uninterpreted functions are used widely in verification when certain functions' semantics can be ignored. EUF is relevant for proving the equivalence of two systems, such as a pair of chip designs, in hardware verification. Regarding ArrayEX, memory in software and memory components in hardware are usually modeled using arrays, and proving properties of the state of memory is a critical part of verifying the security and safety of programs. As for LIA, many verification problems boil down to constraints over integer variables. The goal is to design efficient representations of the verification procedures for these theories that can be proven in zero knowledge with low overhead.

6.1.3 Privacy-Preserving Program Compilation

This thesis offers a suite of verification tools with enhanced privacy. One missing piece arises from the fact that the inputs of the tools are all the abstract representations of a program, either a formula or a finite-state machine, rather than the program itself. To complete the toolchain for verifying private programs, we still need to verify the compilation from a program to these abstract representations in a privacy-preserving manner. In the future, we will design tools to ensure that each abstract representation accurately and adequately reflects the execution of the input private program. A potential direction would be to integrate the compilation with privacy-preserving construction. A zero-knowledge compiler will fulfill this requirement and provide a guarantee of producing the correct abstract representation based on the private input program. Having the binding between the private program and the

private abstract representation will prevent malicious parties from using a formula or finite state machine that does not honestly reflect her input private program.

Bibliography

[1] Coverity, 2002.

[2] SonarQube, 2008.

[3] ShiftLeft, 2016.

[4] Regex Filters for Pi-hole. `https://github.com/mmotti/pihole-regex/blob/master/regex.list`, 2020.

[5] Blumenthal, Blackburn & Klobuchar Introduce Bipartisan Antitrust Legislation to Promote App Store Competition, 2021. Press Release from the Office of Richard Blumenthal, US Senate. At `https://www.blumenthal.senate.gov/newsroom/press/release/blumenthal-blackburn-and-klobuchar-introduce-bipartisan-antitrust-legislation-to-promote-app-store-competition`. Accessed August 13th, 2021.

[6] Pi-hole:Network-wide protection. `https://pi-hole.net/`, 2022.

[7] Read the Antitrust Lawsuit Against Google, October 20th, 2020. At `https://www.nytimes.com/interactive/2020/10/20/us/doj-google-suit.html`. Accessed August 13th, 2021.

[8] Adam Satariano. Apple's App Store Draws E.U. Antitrust Charge. *The New York Times*, April 30th, 2021. At `https://www.nytimes.com/2021/04/30/technology/apple-antitrust-eu-app-store.html`. Accessed August 13th, 2021.

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

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

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

[12] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. pages 2087–2104, 2017.

[13] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2087–2104, New York, NY, USA, 2017. Association for Computing Machinery.

[14] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. Evaluating the "small scope hypothesis". In *In Popl*, volume 2. Citeseer, 2003.

[15] Elli Androulaki, Seung Geol Choi, Steven M Bellovin, and Tal Malkin. Reputation systems for anonymous networks. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 202–218. Springer, 2008.

[16] Manos Antonakakis, Roberto Perdisci, David Dagon, Wenke Lee, and Nick Feamster. Building a dynamic reputation system for {DNS}. In *19th USENIX Security Symposium (USENIX Security 10)*, 2010.

[17] David W. Archer, Dan Bogdanov, Yehuda Lindell, Liina Kamm, Kurt Nielsen, Jakob Illeborg Pagter, Nigel P. Smart, and Rebecca N. Wright. From keys to databases — real-world applications of secure multi-party computation. *The Computer Journal*, 61(12):1749–1771, 2018.

[18] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[19] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal Oblivious RAM. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT '20)*, 2020.

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

[21] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 19–99. Elsevier and MIT Press, 2001.

[22] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *International Conference on Integrated Formal Methods*, pages 1–20. Springer, 2004.

[23] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K Rajamani. Automatic predicate abstraction of c programs. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 203–213, 2001.

[24] Thomas Ball, Andreas Podelski, and Sriram K Rajamani. Boolean and cartesian abstraction for model checking c programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 268–283. Springer, 2001.

[25] Tomáš Balyo, Nils Froleyks, Marijn J.H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions. 2021.

[26] Manuel Barbosa, Gilles Barthe, Karthikeyan Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. Sok: Computer-aided cryptography. *IACR Cryptol. ePrint Arch.*, 2019:1393, 2019.

[27] Joshua Baron, Karim El Defrawy, Kirill Minkovich, Rafail Ostrovsky, and Eric Tressler. 5pm: Secure pattern matching. Cryptology ePrint Archive, Paper 2012/698, 2012. `https://eprint.iacr.org/2012/698`.

[28] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[29] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions. pages 92–122, 2021.

[30] Roberto J. Bayardo Jr. and Joseph Daniel Pehoushek. Counting Models Using Connected Components. In *National Conference on Artificial Intelligence (AAAI '00)*, 2000.

[31] Roberto J. Bayardo Jr and Robert C. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In *National Conference on Artificial Intelligence (AAAI '97)*, 1997.

[32] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 155–168, 2017.

[33] Nicole Beebe. Digital forensic research: The good, the bad and the unaddressed. In *IFIP International conference on digital forensics*, pages 17–36. Springer, 2009.

[34] Michael Ben-Or, Oded Goldreich, Shafi Goldwasser, Johan Håstad, Joe Kilian, Silvio Micali, and Phillip Rogaway. Everything provable is provable in zero-knowledge. pages 37–56, 1990.

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

[36] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, 2014.

[37] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, pages 90–108, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[38] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. pages 781–796, 2014.

[39] Gary Benson. Tandem repeats finder: a program to analyze dna sequences. *Nucleic acids research*, 27(2):573–580, 1999.

[40] Dirk Beyer. Software verification with validation of results. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 331–349. Springer, 2017.

[41] Armin Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(2-4):75–97, 2008.

[42] Armin Biere. Lingeling Essentials, A Tutorial on Design and Implementation Aspects of the the SAT Solver Lingeling. *Pragmatics of SAT (PoS@SAT '14)*, 2014.

[43] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. 2003.

[44] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of Satisfiability*. IOS press, 2009.

[45] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. Data-oblivious graph algorithms for secure computation and outsourcing. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 207–218, 2013.

[46] Alexander R. Block, Justin Holmgren, Alon Rosen, Ron D. Rothblum, and Pratik Soni. Public-coin zero-knowledge arguments with (almost) minimal time and space overheads. pages 168–197, 2020.

[47] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear pcps. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III*, pages 67–97. Springer, 2019.

[48] Joseph Bonneau, Izaak Meckler, Vanishree Rao, and Evan Shapiro. Coda: Decentralized cryptocurrency at scale. Cryptology ePrint Archive, Report 2020/352, 2020. `https://eprint.iacr.org/2020/352`.

[49] Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune K. Jakobsen, and Mary Maller. Arya: Nearly linear-time zero-knowledge proofs for correct program execution. pages 595–626, 2018.

[50] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *SOSP '17*, page 341–357, 2013.

[51] Justin Brickell and Vitaly Shmatikov. Privacy-preserving graph algorithms in the semi-honest model. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT '05)*, pages 236–252. Springer, 2005.

[52] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal xml pattern matching. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 310–321, 2002.

[53] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[54] Matteo Campanelli, Dario Fiore, and Anaïs Querol. Legosnark: Modular design and composition of succinct zero-knowledge proofs. CCS '19, page 2075–2092, New York, NY, USA, 2019. Association for Computing Machinery.

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

[56] Ran Canetti. Universally composable security. *J. ACM*, 67(5), sep 2020.

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

[58] CISCO. Snort intrution prevention system.

[59] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

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

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

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

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

[64] Matthew Clegg, Jeff Edmonds, and Russell Impagliazzo. Using the Groebner basis algorithm to find proofs of unsatisfiability. pages 174–183, 1996.

[65] Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing (STOC '71)*, 1971.

[66] Mark J. Daly, John D. Rioux, Stephen F. Schaffner, Thomas J. Hudson, and Eric S. Lander. High-Resolution Haplotype Structure in the Human Genome. *Nature Genetics*, (2), 2001.

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

[68] Javad Darivandpour and Mikhail J Atallah. Efficient and secure pattern matching with wildcards using lightweight cryptography. *Computers & Security*, 77:666–674, 2018.

[69] David McCabe and Daisuke Wakabayashi. Dozens of States Sue Google Over App Store Fees. *The New York Times*, July 7th, 2021. At `https://www.nytimes.com/2021/07/07/technology/google-play-store-antitrust-suit.html`. Accessed August 13th, 2021.

[70] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM (CACM)*, (7), 1962.

[71] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM (JACM)*, (3), 1960.

[72] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, and Titouan Tanguy. Limbo: Efficient zero-knowledge mpcith-based arguments. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 3022–3036, New York, NY, USA, 2021. Association for Computing Machinery.

[73] Daniel Demmler, Thomas Schneider, and Michael Zohner. Aby-a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[74] Damien Desfontaines and Balázs Pejó. Sok: Differential Privacies. *Proceedings on Privacy Enhancing Technologies (PoPETS '20)*, (2), 2020.

[75] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. Line-point zero knowledge and its applications. In *2nd Conference on Information-Theoretic Cryptography*, 2021.

[76] Jack Doerner and Abhi Shelat. Scaling ORAM for Secure Computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, 2017.

[77] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating Noise to Sensitivity in Private Data Analysis. In *Theory of Cryptography Conference (TCC '06)*. Springer, 2006.

[78] Cynthia Dwork and Aaron Roth. The Algorithmic Foundations of Differential Privacy. *Foundations and Trends in Theoretical Computer Science*, (3-4), 2014.

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

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

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

[82] David Eppstein, Michael T. Goodrich, and Roberto Tamassia. Privacy-preserving data-oblivious geometric algorithms for geographic data. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 13–22, 2010.

[83] European Parliament and Council. 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.

[84] Zhiyong Fang, David Darais, Joseph P. Near, and Yupeng Zhang. Zero knowledge static program analysis. pages 2951–2967, 2021.

[85] Joel Frank, Cornelius Aschermann, and Thorsten Holz. ETHBMC: A bounded model checker for smart contracts. pages 2757–2774, 2020.

[86] Jonathan Frankle, Sunoo Park, Daniel Shaar, Shafi Goldwasser, and Daniel J. Weitzner. Practical accountability of secret processes. pages 657–674, 2018.

[87] Nicholas Franzese, Jonathan Katz, Steve Lu, Rafail Ostrovsky, Xiao Wang, and Chenkai Weng. Constant-overhead zero-knowledge for RAM programs. pages 178–191, 2021.

[88] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient Private Matching and Set Intersection. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT '04)*. Springer, 2004.

[89] Keith B Frikken. Practical private dna string searching and matching through efficient oblivious automata evaluation. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 81–94. Springer, 2009.

[90] Vijay Ganesh and Moshe Vardi. On The Unreasonable Effectiveness of SAT Solvers. In Tim Roughgarden, editor, *Beyond the Worst-Case Analysis of Algorithms*. Cambridge University Press, 2020.

[91] Simson Garfinkel. History's worst software bugs, 2009.

[92] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. pages 626–645, 2013.

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

[94] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity and a methodology of cryptographic protocol design (extended abstract). pages 174–187, 1986.

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

[96] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. pages 218–229, 1987.

[97] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *Journal of the ACM*, 38(3):691–729, 1991.

[98] Oded Goldreich, Silvio Micali, and Avi Wigderson. *How to Play Any Mental Game, or a Completeness Theorem for Protocols with Honest Majority*, page 307–328. Association for Computing Machinery, New York, NY, USA, 2019.

[99] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, (3), 1996.

[100] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of computer and system sciences*, 28(2):270–299, 1984.

[101] Michael T. Goodrich, Olga Ohrimenko, and Roberto Tamassia. Data-oblivious graph drawing model and algorithms. *arXiv preprint arXiv:1209.0756*, 2012.

[102] Michael T. Goodrich and Joseph A. Simons. Data-oblivious graph algorithms in outsourced external memory. In *International Conference on Combinatorial Optimization and Applications*, pages 241–257. Springer, 2014.

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

[104] Vipul Goyal and Yifan Song. Malicious security comes free in honest-majority mpc. *Cryptology ePrint Archive*, 2020.

[105] Ana Graça, Inês Lynce, Joao Marques-Silva, and Arlindo L. Oliveira. Haplotype Inference by Pure Parsimony: a Survey. *Journal of Computational Biology*, (8), 2010.

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

[107] Adam Groce, Peter Rindal, and Mike Rosulek. Cheaper Private Set Intersection via Differentially Private Leakage. *Proceedings on Privacy Enhancing Technologies (PoPETS '19)*, (3), 2019.

[108] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 305–326, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[109] Paul Grubbs, Arasu Arun, Ye Zhang, Joseph Bonneau, and Michael Walfish. Zero-knowledge middleboxes. Cryptology ePrint Archive, Paper 2021/1022, 2021. `https://eprint.iacr.org/2021/1022`.

[110] Paul Grubbs, Arasu Arun, Ye Zhang, Joseph Bonneau, and Michael Walfish. {Zero-Knowledge} middleboxes. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4255–4272, 2022.

[111] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program Analysis as Constraint Solving. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*, 2008.

[112] Dan Gusfield. Haplotype Inference by Pure Parsimony. In *Annual Symposium on Combinatorial Pattern Matching (CPM '03)*. Springer, 2003.

[113] Armin Haken. The intractability of resolution. *Theoretical computer science*, 39:297–308, 1985.

[114] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. ManySAT: a Parallel SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, (4), 2010.

[115] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. Sok: General Purpose Compilers for Secure Multi-Party Computation. In *2019 IEEE Symposium on Security and Privacy (S&P '19)*, pages 1220–1237. IEEE, 2019.

[116] Carmit Hazay and Yehuda Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In *Theory of Cryptography Conference*, pages 155–175. Springer, 2008.

[117] Carmit Hazay and Yehuda Lindell. A note on zero-knowledge proofs of knowledge and the ZKPOK ideal functionality. Cryptology ePrint Archive, Report 2010/552, 2010. `https://eprint.iacr.org/2010/552`.

[118] Xi He, Ashwin Machanavajjhala, Cheryl Flynn, and Divesh Srivastava. Composing Differential Privacy and Secure Computation: A Case Study on Scaling Private Record Linkage. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, 2017.

[119] David Heath and Vladimir Kolesnikov. A 2.1 KHz zero-knowledge processor with BubbleRAM. pages 2055–2074, 2020.

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

[121] David Heath, Yibin Yang, David Devecsery, and Vladimir Kolesnikov. Zero knowledge for everything and everyone: Fast ZK processor with cached ORAM for ANSI C programs. pages 1538–1556, 2021.

[122] Paul Hoffman and Patrick McManus. Dns queries over https (doh). Technical report, 2018.

[123] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure two-party computations in ansi c. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*, pages 772–783, 2012.

[124] Nils Homer, Szabolcs Szelinger, Margot Redman, David Duggan, Waibhav Tembe, Jill Muehling, John V. Pearson, Dietrich A. Stephan, Stanley F. Nelson, and David W. Craig. Resolving Individuals Contributing Trace Amounts of DNA to Highly Complex Mixtures using High-Density SNP Genotyping Microarrays. *PLoS Genet*, (8), 2008.

[125] Holger H. Hoos and Thomas Stützle. SATLIB: An Online Resource for Research on SAT. *International Conference on Theory and Applications of Satisfiability Testing (SAT '00)*, 2000.

[126] Zhangxiang Hu, Payman Mohassel, and Mike Rosulek. Efficient zero-knowledge proofs of non-algebraic statements with sublinear amortized cost. pages 150–169, 2015.

[127] Russell Impagliazzo and Ramamohan Paturi. On the Complexity of k-SAT. *Journal of Computer and System Sciences*, (2), 2001.

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

[129] Franjo Ivančić, Zijiang Yang, Malay K Ganai, Aarti Gupta, Ilya Shlyakhter, and Pranav Ashar. F-soft: Software verification platform. In *International Conference on Computer Aided Verification*, pages 301–306. Springer, 2005.

[130] Jack Nicas. Apple Cracks Down on Apps That Fight iPhone Addiction. *The New York Times*, April 27th, 2019. At https://www.nytimes.com/2019/04/27/technology/apple-screen-time-trackers.html. Accessed August 13th, 2021.

[131] Jack Nicas and Keith Collins. How Apple's Apps Topped Rivals in the App Store It Controls. *The New York Times*, September 9th, 2019. At https://www.nytimes.com/interactive/2019/09/09/technology/apple-app-store-competition.html. Accessed August 13th, 2021.

[132] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. 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)*, pages 955–966, 2013.

[133] Samuel Judson, Ning Luo, Timos Antonopoulos, and Ruzica Piskac. Privacy preserving ctl model checking through oblivious graph algorithms. In *Proceedings of the 19th Workshop on Privacy in the Electronic Society*, pages 101–115, 2020.

[134] Richard M. Karp. Reducibility Among Combinatorial Problems. In *Complexity of Computer Computations*. Springer, 1972.

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

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

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

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

[139] Florian Kerschbaum. Practical private regular expression matching. In *Security and Privacy in Dynamic Environments: Proceedings of the IFIP TC-11 21st International Information Security Conference (SEC 2006), 22–24 May 2006, Karlstad, Sweden 21*, pages 461–470. Springer, 2006.

[140] Lina M. Khan. The Separation of Platforms and Commerce. *Columbia Law Review*, 119(4):973–1098, 2019.

[141] Franziskus Kiefer and Mark Manulis. Zero-knowledge password policy checks and verifier-based pake. In *Computer Security-ESORICS 2014: 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part II 19*, pages 295–312. Springer, 2014.

[142] Joe Kilian. Founding crytpography on oblivious transfer. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 20–31, 1988.

[143] Lea Kissner and Dawn Song. Privacy-Preserving Set Operations. In *Annual International Cryptology Conference (CRYPTO '05)*. Springer, 2005.

[144] Paul C Kocher. On certificate revocation and validation. In *International conference on financial cryptography*, pages 172–177. Springer, 1998.

[145] Vladimir Kolesnikov, Mike Rosulek, and Ni Trieu. Swim: Secure wildcard pattern matching from ot extension. In *International Conference on Financial Cryptography and Data Security*, pages 222–240. Springer, 2018.

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

[147] Ian Korf, Mark Yandell, and Joseph Bedell. *Blast.* " O'Reilly Media, Inc.", 2003.

[148] Daniel Kroening and Michael Tautschnig. Cbmc–c bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer, 2014.

[149] Karen Kukich. Techniques for automatically correcting words in text. *Acm Computing Surveys (CSUR)*, 24(4):377–439, 1992.

[150] Giuseppe Lancia, Maria Cristina Pinotti, and Romeo Rizzi. Haplotyping Populations by Pure Parsimony: Complexity of Exact and Approximation Algorithms. *INFORMS Journal on Computing*, (4), 2004.

[151] Peeter Laud and Jan Willemson. Universally composable privacy preserving finite automata execution with low online and offline complexity. *Cryptology ePrint Archive*, 2013.

[152] K. Rustan M. Leino. A SAT Characterization of Boolean-Program Correctness. In *International SPIN Workshop on Model Checking of Software (SPIN '03)*. Springer, 2003.

[153] Zhenping Li, Wenfeng Zhou, Xiang-Sun Zhang, and Luonan Chen. A Parsimonious Tree-Grow Method for Haplotype Inference. *Bioinformatics*, (17), 2005.

[154] Yehuda Lindell. How to Simulate It – a Tutorial on the Simulation Proof Technique. *Tutorials on the Foundations of Cryptography*, 2017.

[155] Yehuda Lindell and Benny Pinkas. 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)*, pages 52–78. Springer, 2007.

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

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

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

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

[160] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, and George Varghese. Network Verification in the Light of Program Verification. Technical report, Microsoft Research, 2013.

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

[162] Ning Luo, Timos Antonopoulos, William Harris, Ruzica Piskac, Eran Tromer, and Xiao Wang. Proving unsat in zero knowledge. *Cryptology ePrint Archive*, 2022.

[163] Ning Luo, Samuel Judson, Timos Antonopoulos, Ruzica Piskac, and Xiao Wang. ppsat: Towards two-party private sat solving. Cryptology ePrint Archive, Paper 2021/1584, 2021. `https://eprint.iacr.org/2021/1584`.

[164] Ning Luo, Samuel Judson, Timos Antonopoulos, Ruzica Piskac, and Xiao Wang. ppsat: Towards two-party private SAT solving. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, 2022.

[165] Ning Luo, Qiao Xiang, Timos Antonopoulos, Ruzica Piskac, Y. Richard Yang, and Franck Le. IVeri: Privacy-Preserving Interdomain Verification. *arXiv preprint arXiv:2202.02729*, 2022.

[166] Inês Lynce and Joao Marques-Silva. Efficient Haplotype Inference with Boolean Satisfiability. In *National Conference on Artificial Intelligence (AAAI '06)*. AAAI Press, 2006.

[167] Inês Lynce and João Marques-Silva. SAT in Bioinformatics: Making the Case with Haplotype Inference. In *International Conference on Theory and Applications of Satisfiability Testing (SAT '06)*. Springer, 2006.

[168] Jonathan Marchini, David Cutler, Nick Patterson, Matthew Stephens, Eleazar Eskin, Eran Halperin, Shin Lin, Zhaohui S Qin, Heather M. Munro, Gonçalo R. Abecasis, Peter Donnelly, and International HapMap Consortium. A Comparison of Phasing Algorithms for Trios and Unrelated Individuals. *The American Journal of Human Genetics*, (3), 2006.

[169] Piotr Mardziel, Michael Hicks, Jonathan Katz, and Mudhakar Srivatsa. Knowledge-oriented Secure Multiparty Computation. In *Proceedings of the 7th Workshop on Programming Languages and Analysis for Security (PLAS '12)*, pages 1–12, 2012.

[170] Joao Marques-Silva. The Impact of Branching Heuristics in Propositional Satisfiability Algorithms. In *Portuguese Conference on Artificial Intelligence*. Springer, 1999.

[171] Ruben Martins, Vasco Manquinho, and Inês Lynce. An Overview of Parallel SAT Solving. *Constraints*, (3), 2012.

[172] Martha M McCarthy. Filtering the internet: The children's internet protection act. *Educational Horizons*, 82(2):108–113, 2004.

[173] Kenneth L. McMillan. Interpolation and SAT-based Model Checking. In *International Conference on Computer Aided Verification (CAV '03)*. Springer, 2003.

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

[175] Paul V Mockapetris. Domain names-concepts and facilities. Technical report, 1987.

[176] Payman Mohassel, Salman Niksefat, Saeed Sadeghian, and Babak Sadeghiyan. An efficient protocol for oblivious dfa evaluation and applications. In *Cryptographers' Track at the RSA Conference*, pages 398–415. Springer, 2012.

[177] Payman Mohassel, Mike Rosulek, and Alessandra Scafuro. Sublinear zero-knowledge arguments for RAM programs. pages 501–531, 2017.

[178] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC '01)*, 2001.

[179] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[180] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *ACM Sigplan Notices*, 42(6):446–455, 2007.

[181] Paul Ohm. Broken Promises of Privacy: Responding to the Surprising Failure of Anonymization. *UCLA L. Rev.*, 2009.

[182] Emmanuel Olaoye. Knight capital filings show scant board duty for tech risk, 2012.

[183] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Optimal verification of operations on dynamic sets. pages 91–110, 2011.

[184] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. pages 238–252, 2013.

[185] Nila Patil, Anthony J. Berno, David A. Hinds, Wade A. Barrett, Jigna M. Doshi, Coleen R. Hacker, Curtis R. Kautzer, Danny H. Lee, Claire Marjoribanks, David P. McDonough, et al. Blocks of Limited Haplotype Diversity Revealed by High-Resolution Scanning of Human Chromosome 21. *Science*, (5547), 2001.

[186] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure Two-Party Computation is Practical. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT '09)*. Springer, 2009.

[187] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster Private Set Intersection Based on OT Extension. In *USENIX Security Symposium (USENIX Security '14)*, 2014.

[188] Amir Pnueli. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science (FOCS '77)*, pages 46–57. IEEE, 1977.

[189] Michael O Rabin. How to exchange secrets with oblivious transfer. *Cryptology ePrint Archive*, 2005.

[190] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. Wysteria: A Programming Language for Generic, Mixed-mode Multiparty Computations. In *2014 IEEE Symposium on Security and Privacy (S&P '14)*, pages 655–670. IEEE, 2014.

[191] Mark J. Rieder, Scott L. Taylor, Andrew G. Clark, and Deborah A. Nickerson. Sequence Variation in the Human Angiotensin Converting Enzyme. *Nature Genetics*, (1), 1999.

[192] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, jan 1965.

[193] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[194] Mohammad Hasan Samadani, Mehdi Berenjkoob, and Marina Blanton. Secure pattern matching based on bit parallelism. *International Journal of Information Security*, 18(3):371–391, 2019.

[195] Hirohito Sasakawa, Hiroki Harada, David duVerle, Hiroki Arimura, Koji Tsuda, and Jun Sakuma. Oblivious evaluation of non-deterministic finite automata with application to privacy-preserving virus genome detection. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, pages 21–30, 2014.

[196] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *2010 IEEE Symposium on Security and Privacy*, pages 513–528. IEEE, 2010.

[197] Thomas J. Schaefer. The Complexity of Satisfiability Problems. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing (STOC '78)*, 1978.

[198] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011.

[199] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *Proceedings of the 2015 ACM conference on special interest group on data communication*, pages 213–226, 2015.

[200] Victor Shoup et al. Ntl: A library for doing number theory, 2001.

[201] J.P. Marques Silva and K.A. Sakallah. GRASP-A New Search Algorithm for Satisfiability. In *Proceedings of International Conference on Computer Aided Design (ICCAD '96)*. IEEE, 1996.

[202] A. Prasad Sistla and Edmund M. Clarke. The Complexity of Propositional Linear Temporal Logics. *Journal of the ACM (JACM)*, 32(3):733–749, 1985.

[203] Robin Sommer and Vern Paxson. Enhancing byte-level network intrusion detection signatures with context. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 262–271, 2003.

[204] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT Solvers to Cryptographic Problems. In *International Conference on Theory and Applications of Satisfiability Testing (SAT '06)*. Springer, 2009.

[205] Niklas Sorensson and Niklas Een. Minisat v1. 13-A SAT Solver with Conflict-Clause Minimization. *SAT*, (53), 2005.

[206] Sheng Sun, Dr Wen, et al. zk-fabric, a polylithic syntax zero knowledge joint proof system. *arXiv preprint arXiv:2110.07449*, 2021.

[207] K. Sunder Rajan. *Biocapital: The Constitution of Postgenomic Life*. Duke University Press, 2006.

[208] Neil Thapen. A tradeoff between length and width in resolution. *Theory of Computing*, 12(1):1–14, 2016.

[209] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[210] Aivo Toots. Zero-knowledge proofs for business processes.

[211] Juan Ramón Troncoso-Pastoriza, Stefan Katzenbeisser, and Mehmet Celik. Privacy preserving error resilient dna searching through oblivious automata. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 519–528, 2007.

[212] Jan Van Lunteren. High-performance pattern-matching for intrusion detection. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, pages 1–13. Citeseer, 2006.

[213] Damien Vergnaud. Efficient and secure generalized pattern matching via fast fourier transform. In *International Conference on Cryptology in Africa*, pages 41–58. Springer, 2011.

[214] Psi Vesely, Kobi Gurkan, Michael Straka, Ariel Gabizon, Philipp Jovanovic, Georgios Konstantopoulos, Asa Oines, Marek Olszewski, and Eran Tromer. Plumo: An ultralight blockchain client. In *FC 2022*, LNCS. Springer, Heidelberg, Germany, 2022.

[215] Riad S. Wahby, Srinath T. V. Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. 2015.

[216] Rui Wang, Yong Fuga Li, XiaoFeng Wang, Haixu Tang, and Xiaoyong Zhou. Learning Your Identity and Disease from Research Papers: Information Leaks in Genome Wide Association Study. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*, 2009.

[217] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-Toolkit: Efficient MultiParty Computation Toolkit. `https://github.com/emp-toolkit`, 2016.

[218] Xiao Wang, Kartik Nayak, Chang Liu, T.H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious Data Structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*, 2014.

[219] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-Scale Secure Multiparty Computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, pages 39–56, 2017.

[220] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. pages 1074–1091, 2021.

[221] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. Mystique: Efficient conversions for {Zero-Knowledge} proofs with applications to machine learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 501–518, 2021.

[222] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and Experience. *ACM Computing Surveys (CSUR)*, 41(4):1–36, 2009.

[223] David J. Wu, Joe Zimmerman, Jérémy Planul, and John C. Mitchell. Privacy-Preserving Shortest Path Computation. In *23rd Annual Network and Distributed System Security Symposium (NDSS '16)*, 2016.

[224] Yinglian Xie, Fang Yu, Kannan Achan, Rina Panigrahy, Geoff Hulten, and Ivan Osipkov. Spamming botnets: signatures and characteristics. *ACM SIGCOMM Computer Communication Review*, 38(4):171–182, 2008.

[225] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. pages 2986–3001, 2021.

[226] Andrew Chi-Chih Yao. Protocols for secure computations. In *FOCS*, volume 82, pages 160–164, 1982.

[227] Andrew Chi-Chih Yao. How to Generate and Exchange Secrets. In *27th Annual Symposium on Foundations of Computer Science (FOCS '86)*. IEEE, 1986.

[228] Masaya Yasuda, Takeshi Shimoyama, Jun Kogure, Kazuhiro Yokoyama, and Takeshi Koshiba. Secure pattern matching using somewhat homomorphic encryption. In *Proceedings of the 2013 ACM workshop on Cloud computing security workshop*, pages 65–76, 2013.

[229] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla+ specifications. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 54–66. Springer, 1999.

[230] Zhonghao Yu, Sam Macbeth, Konark Modi, and Josep M Pujol. Tracking the trackers. In *Proceedings of the 25th International Conference on World Wide Web*, pages 121–132, 2016.

[231] Fatih Yucel, Kemal Akkaya, and Eyuphan Bulut. Efficient and privacy preserving supplier matching for electric vehicle charging. *Ad Hoc Networks*, 90:101730, 2019.

[232] Samee Zahur and David Evans. Circuit structures for improving efficiency of security and privacy tools. In *2013 IEEE Symposium on Security and Privacy*, pages 493–507, 2013.

[233] Samee Zahur, Mike Rosulek, and David Evans. Two Halves Make a Whole. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT '15)*, pages 220–250. Springer, 2015.

[234] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. Deco: Liberating web data using decentralized oracles for tls. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1919–1938, 2020.