Proving UNSAT in Zero Knowledge

Ning Luo
Yale University
New Haven, USA
ning.luo@yale.edu

Timos Antonopoulos
Yale University
New Haven, USA
timos.antonopoulos@yale.edu

William R. Harris*
Galois, Inc
Portland, USA
bll.hrris@gmail.com

Ruzica Piskac
Yale University
New Haven, USA
ruzica.piskac@yale.edu

Eran Tromer
Columbia University
New York City, USA
et2555@columbia.edu

Xiao Wang
Northwestern University
Evanston, USA
wangxiao@cs.northwestern.edu

ABSTRACT

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.

We 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.

CCS CONCEPTS

• **Theory of computation → Cryptographic protocols**; **Logic and verification**.

KEYWORDS

Zero-knowledge proofs, Propositional unsatisfiability

ACM Reference Format:
Ning Luo, Timos Antonopoulos, William R. Harris*, Ruzica Piskac, Eran Tromer, and Xiao Wang. 2022. Proving UNSAT in Zero Knowledge. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3548606.3559373

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9450-5/22/11.
https://doi.org/10.1145/3548606.3559373

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 [39] and perhaps beyond [12]. 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., [4, 14, 16, 21, 22, 24, 36, 38, 43, 46, 47, 49, 56, 59, 66]). These found a rapidly-expanding set of applications, including: blockchain privacy [13, 25] and scalability [23, 65], legal systems [35] and anonymous networks [6].

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 [55].

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], 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 [8, 50].

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

*Author now employed at Google LLC

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 [41] 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 [48] 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 [34] 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 [14, 16, 21, 22, 24, 26, 36, 46, 47, 49, 56, 66].

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.

We evaluated our protocol empirically by implementing it via the EMP framework [67] 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 [17] benchmark set. This includes verification of Linux device drivers, Windows NT device drivers, and C implementations of floating-point computation.

Our 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 paper. We provide more discussion at the end of the paper.

Organization The remainder of this paper is organized as follows: Section 2 presents an overview of our protocol by example; Section 3 reviews foundational definitions and results on which our work is based; Section 4 presents our protocol in technical detail; Section 5 describes our implementation and empirical evaluation of the protocol; Section 6 compares our contribution to related work, and Section 7 concludes.

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 We 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 formulae, we use the small C program `sum3` given in Figure 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 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

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

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

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

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

Figure 1: An example program and Boolean formula that characterizes its executions.

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.

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 [28].

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* [60]. A formula is unsatisfiable if and only if we can derive $\perp$ (false) by applying *resolution steps*, according to the fundamental theorem about refutational completeness of first-order logic [7] (which applies also to the propositional logic we employ here). Resolution proofs are reviewed in formal detail in Section 3.2.1, 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 $\perp$ formula has been derived.

Although simple, the resolution rule is the basis of modern automated first-order reasoners [61], and their applications to program verification. Indeed, we proceed to show its use to prove that sum3 does not overflow.

We 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 $\perp$ 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}:

$$\dfrac{\dfrac{o_1 \vee o_2 \qquad \dfrac{\ldots}{\neg o_1}}{o_2} \qquad \dfrac{\ldots}{\neg o_2}}{\perp}$$

We managed to derive $\perp$, 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 $\perp$ (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 $\perp$.

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

[1] Had the formula been satisfiable, applying the resolution rules could never have derived $\perp$, and moreover (for propositional logic), the process would have eventually terminated and let us read a satisfying assignment out of the derived clauses [7], revealing inputs to sum3 that cause an overflow.

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

$$\neg b_0 \vee \neg acc_0 \vee \neg a_1. \tag{1}$$

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 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 $\perp$ clause.

In our example we produced a proof derivation that only derived clauses needed to derive $\perp$. 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 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 5, 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 (1) 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. We describe this encoding in detail in Section 4.1.

.

3 TECHNICAL PRELIMINARIES

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 paper 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 polynomial. 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 p^* has no repeated root, and can be divided by *every* completely reducible divisor of p that has no repeated root;

3.2 Boolean logic

In this work, 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 Lits = Vars × $\mathbb{B}$, where $\mathbb{B}$ denotes the Booleans); a positive occurrence of variable $x \in$ 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 : 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.

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

Definition 3.1. 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 3.2. 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 [30]. 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 [44]. 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 [63].

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

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

Instance: On receiving (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 (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 (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 2: Functionality for zero-knowledge proofs of circuit satisfiability and polynomials.

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 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. We 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 paper [11, 32, 68, 70], 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 benefit from this modular design because the security follows immediately from the composition theorem [27].

Zero-knowledge proofs of random accesses. There has been a long line of works [14, 16, 21, 22, 24, 26, 36, 46, 47, 49, 56, 66] 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 [46, 47, 49, 56] combine ZK protocols with oblivious RAMs [42]: 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 [14, 16, 24, 26, 36, 66] 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.

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 the full version of the paper [53].

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 C that only contains all clauses in ϕ. In each iteration, we fetch two clauses from C as premise clauses, compute their resolvent, and append the resulting clause to C. If the resolution completes, the last clause added to C should be $\bot$.

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 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 4.1 discusses our encoding methods for both literals and clauses. It provides huge improvement compared to a bit-vector-based representation. In Section 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 4.3, we discuss our solution for the second task.

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.

4.1.1 Naive encoding methods. As discussed in Section 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.

4.1.2 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 $x \in \mathsf{Vars}$,

$$\phi(x) + \phi(\neg x) = 1_{\mathbb{F}} \tag{2}$$

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 (2) is a *literal encoding* into $\mathbb{F}$.

For the rest of this paper, 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 $\bot$ is $\gamma_\phi(\bot) = 1_{\mathbb{F}}$, where $1_{\mathbb{F}}$ denotes a polynomial with only a constant term, which is distinct from the field element in Equation 2.

For the rest of this paper, 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 (2).

Lemma 4.1. *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 4.2. *For clauses C and C', if $C \to C'$, then*

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

4.1.3 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 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

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

Input: On receiving (Input, $\ell_0, \cdots, \ell_{k-1}, w$) from prover and (Input, w) from verifier where $\ell_i \in$ 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 (Equal, $[C_0]$, $[C_1]$) from both parties, check if $C_0 = C_1$; if not, the functionality aborts.

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

Figure 3: Functionality for ZK operations on clauses.

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 $\mathcal{F}_{\text{ZK}}$ 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. [58].

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.

4.2.1 Resolution with weakening. To define our encoding scheme, we first define a set of derivations of SAT formulas that slightly generalizes resolution derivations (Section 3.2.1). 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 4.3. A *weak resolvent* of clauses C and C' on pivot variable x is a clause C'' such that

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

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. 3.1).

A weakened resolution derivation is a sequence of weak resolvents, analogous to how a resolution derivation (Defn. 3.2) is a sequence of resolvents:

Definition 4.4. 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 4.2.2 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.

4.2.2 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 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 4.4) by checking divisibility of polynomials, which itself is checked by checking equality of 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:

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. $\phi : \text{Lits} \to \mathbb{F}$ 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 4: Our protocol to instantiate $\mathcal{F}_{\mathsf{Clause}}$.

$$w_0 \cdot q_0 = q_r \cdot \rho \tag{3}$$

$$w_1 \cdot q_1 = q_r \cdot \bar{\rho} \tag{4}$$

$$\rho(X) + \bar{\rho}(1_{\mathbb{F}} - X) = 0_{\mathbb{F}} \tag{5}$$

The verifier also attests that ρ and $\bar{\rho}$ have degrees of at most one. Equations (3) to (5) 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 6) is complete.

Lemma 4.5. *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 4.4, and uses *maximal completely reducible divisors*, introduced in Section 3.1.

Lemma 4.6. *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 $\gamma^{-1}(q_1^*)$.*

The full version of the paper [53] contains a complete proof of Lemma 4.6 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. 3.1) and γ is a bijection into the completely reducible polynomials.

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. 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 $\log|\text{Lits}|$ bits to encode. As a result each clause needs at least $w \log|\text{Lits}|$ 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 [36] 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 [36], 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.

4.4 Putting everything together

In Figure 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.

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

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

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

Array read: On receiving $(\text{Read}, \ell, d, t)$ 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 $f := \text{cheating}$.

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

Figure 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 $\mathcal{F}_{\text{Clause}}$; since ϕ is known to both parties, it uses 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 $\mathcal{F}_{\text{Clause}}$ using witness authenticating the coefficients.
(3) The two parties send $(\text{Init}, |\phi| + R - 1, [C_0], \ldots, [C_{|\phi|+R-1}])$ to $\mathcal{F}_{\text{FlexZKArray}}$.
(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 $(\text{Read}, l_i, C_{l_i}, i)$ to $\mathcal{F}_{\text{FlexZKArray}}$; $\mathcal{V}$ sends (Read, i) to $\mathcal{F}_{\text{FlexZKArray}}$, 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 $(\text{Xres}, [C_{l_i}], [C_{k_i}], [C_i])$ to $\mathcal{F}_{\text{Clause}}$.
(5) After R iterations, two parties use $\mathcal{F}_{\text{Clause}}$ to check that $[C_R]$ equals $\bot$; if the functionality aborts, $\mathcal{V}$ aborts.
(6) Two parties send (check) to $\mathcal{F}_{\text{FlexZKArray}}$, if the functionality aborts, $\mathcal{V}$ aborts.

Figure 6: Protocol for checking resolution proof.

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 4.7. *The protocol in Figure 6 is a zero-knowledge proof of knowledge of refutation proof.*

We provide a proof of sketch of this theorem in the full version [53] . 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 [11, 32, 51, 68, 70] and was formally discussed by Hazay and Lindell [45].

5 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.

5.1 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 [67] and the high-performance library NTL [62] 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 [36]. 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 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.

5.2 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 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 $|\varphi| \in \{2000, 2200, \cdots, 3000\}$,

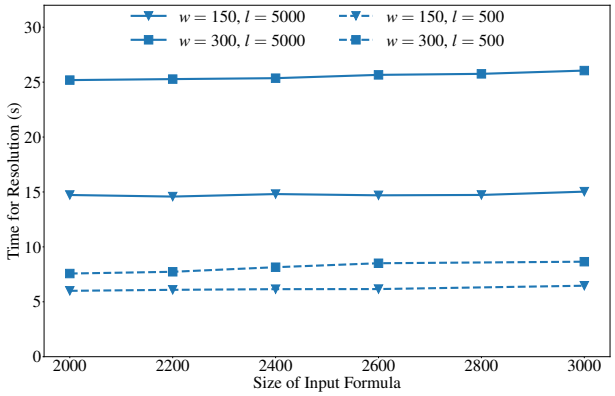

Figure 7: Clause verification time vs. size of input formula. The total time for verifying a resolution proof changes negligibly with an increase in the size of the input formula, under various fixed refutation lengths l and widths w.

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 increase under fixed lengths l and widths w; Figure 7 contains the evaluation's 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 the full version [53]).

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

Figure 8 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 9 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 premises are adequately visible.

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

Communication cost We evaluated the communication costs for verifying refutations of different length and width; Table 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 4.1.1). 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 10 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 |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 $|\text{Lits}| = 2^{11}$. A proof with number of literals $|\text{Lits}| = 2^{15}$ 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.

5.3 Verifying safety-critical proofs in ZK

We evaluated ZkUnsat on refutations generated from benchmarks in corpus of the *Competition on Software Verification (SV-COMP)* [17], 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 [9, 10], 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

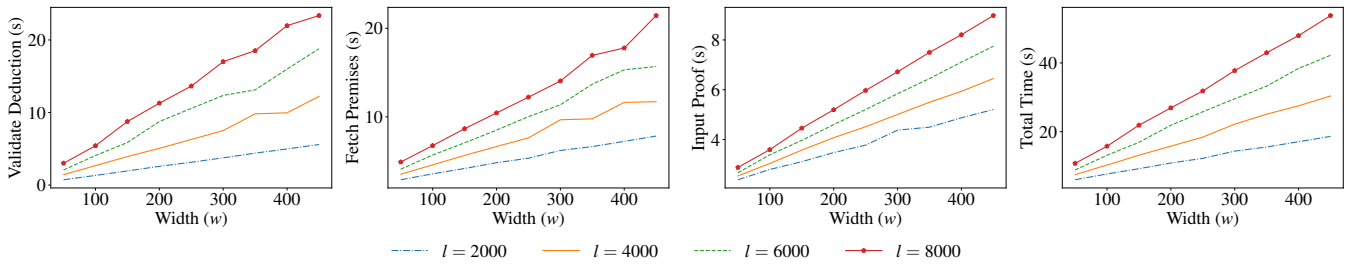

Figure 8: Verification time vs. refutation width. Plots of phase time and total performance vs. width w, for various refutation lengths $l \in [2,000, 8,000]$, with a fixed formula size of $|\varphi| = 3,000$. The times spent inputting the proof, fetching premises, and checking resolution steps are all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

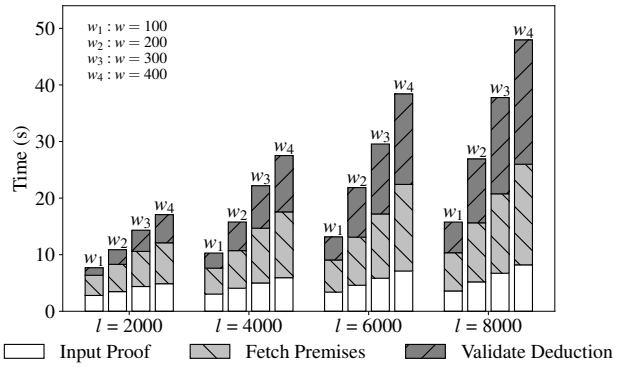

Figure 9: Verification time vs. refutation length. For different fixed refutation widths w, verification time is linear in the refutation's length l. As the length grows, the increase in time of inputting proof is less than the increase for fetching premises and checking resolution. Furthermore, as length increases, the time for fetching premises and checking resolution dominates verification time.

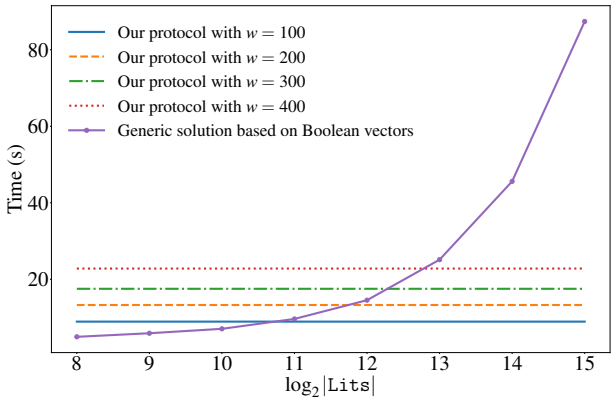

Figure 10: Time vs. number of literals, per clause representation. A plot of verification time of different protocols vs. the number of variables used by the input formula, on refutations with fixed length 3,000, which was chosen as sufficiently large to observe an effect. The purple line depicts the 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.

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) [52]. 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) [19] 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 [5, 57]. 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`.

Refutations of the generated formulas were generated using the PicoSAT SAT solver [18]. Figure 11 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

[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

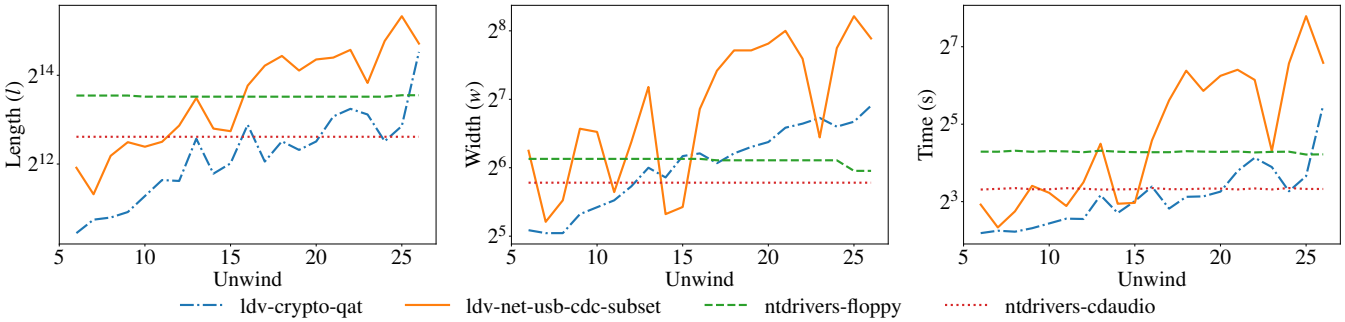

Figure 11: Verification features vs. bound on loop unwindings for drivers. Plots of refutation length, width, and verification time vs. bound on loop unwindings for a set of Windows NT and Linux drivers.

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 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.

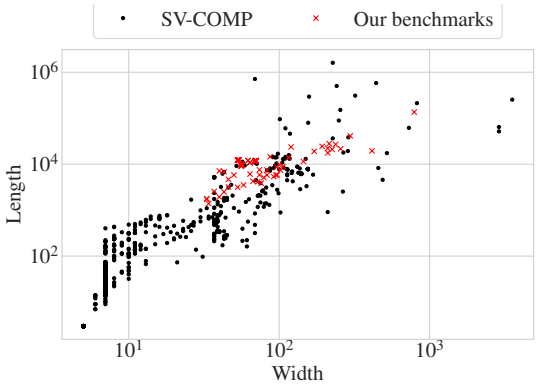

Figure 12: Distribution of the clausal length and width of formulae for real programs in SV-COMP.

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 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 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 12, alongside the paper's examples. The result shows that the scale of formulae in our benchmarks can cover 804 of 814 (98.7%) verification tasks from SV-COMP.

6 RELATED WORK

The previous work closest to our goal addresses approaches to static program analysis in zero knowledge [34]. When the proven invariants of programs are used to establish that the secret program satisfies a specification of correctness, such static analyses effectively prove that safety of a program in zero knowledge. The contribution of this work is complementary to such approaches: definitions of static analyses in ZK describe how to generate a ZK proof statement about a potentially unbounded program, given a definition of an *abstract domain* of the facts, equipped with operations that describe how to merge multiple facts soundly. Current implementations of such schemes have used encouraging but relatively lightweight abstract domains, which typically are used to prove simple program properties. In contrast, our approach for verifying resolution proofs in ZK can be used to instantiate such schemes with a comparatively powerful abstract *symbolic domain* of facts as Boolean formulas. Within such a scheme, the symbolic domain could be used to deep safety and correctness properties of unbounded programs.

In [54], the authors present ppSAT, a privacy-preserving satisfiability solver, where two parties can contribute two private, respectively to each party, formula and the tool employs Multi-Party Computation (MPC) techniques to determine if the conjunction of

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

these two formulas is satisfiable. The approach taken in that work is finely tuned for proving the satisfiability of formulas. As such, although the tool could be used for showing unsatisfiability of the conjunction of the input formulas, it would have to check all the possible variable assignments that are exponential in number.

Resolution proofs are well-studied systems for formally proving the validity of, or refuting, statements in formal logics. Classical results have established that they are a sound and complete system for refuting propositional formulas [60], that there are families of unsatisfiable formulas without short refutations in resolution-based systems [44], and that in general there may be a fundamental tradeoff between a refutations dimensions, namely its length and its width [63]. Practical implementations of many modern SAT solvers can be configured so that, upon determining that a formula φ is unsatisfiable, they generate a refutation of φ as a resolution proof [18, 31, 33, 37]. In this work, we have introduced a slight variation of a standard resolution proof system for Boolean logic; the proposed system retains the soundness and completeness of standard systems, but its refutations can be verified more efficiently than proofs in systems that are equivalent in expressive power but that imposes stricter requirements on the structure of its proofs. Our approach does not rely on novel, tight bounds on the resolution proofs' dimensions: instead, we have defined a optimized ZK verifier that reveals only the refutation's dimensions. Proofs in standard systems directly correspond to proofs in our relaxed system: thus, our approach can be used to verify proofs generated by all existing SAT solvers without modification to the underlying solver.

An extensive line of work has investigated reducing problems in verification to solving or refuting SAT formulas [10, 19, 52, 69]. Such approaches, given a program P and property Q, generate a propositional formula φ such that P (or a bounded approximation of P) satisfies Q if and only if φ is unsatisfiable. Our approach for validating a proof of unsatisfiability can be combined with any such model checker and any process that generates resolution proofs as refutations to prove that a program satisfies a desired property without revealing information about proof itself.

Zero-knowledge proofs in the RAM model has been studied extensively in recent years [14, 16, 21, 22, 24, 26, 36, 46, 47, 49, 56, 66]. Most of these works focus on designing a general-purpose RAM machine or random access structure to be used for any computation. To support efficient fetching of premise clauses, we optimize a prior RAM construction [36] in our setting. Our construction is no longer general-purpose, but it provides improved efficiency in our application.

While this paper studies cryptographic proofs composed with resolution proofs, a different notion of "proofs about proofs" is recursively composing cryptographic proofs with cryptographic proofs, as in Incrementally Verifiable Computation [64] and Proof-Carrying Data [15, 20].

7 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)* [40, 71], 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 [29] 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.

ACKNOWLEDGEMENTS

Work by William Harris and Eran Tromer is supported in part by DARPA under Contract No. HR001120C0085. Work by Xiao Wang is supported in part by DARPA under Contract No. HR001120C0087, NSF award CNS-2016240, and research awards from Meta and Google. Work by Timos Antonopoulos has been supported in part by ONR under Grant N00014-17-1-2787 and by NSF awards CCF-2106845, CCF-2131476. Work by Ruzica Piskac and Ning Luo is supported in part by NSF award CNS-1562888 and CCF-2131476. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

REFERENCES

[1] 2002. Coverity. https://coverity.com
[2] 2008. SonarQube. https://www.sonarqube.org
[3] 2016. ShiftLeft. https://www.shiftleft.io
[4] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. 2017. Ligero: Lightweight Sublinear Arguments Without a Trusted Setup. In *ACM CCS 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, Dallas, TX, USA, 2087–2104. https://doi.org/10.1145/3133956.3134104
[5] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. 2003. Evaluating the "small scope hypothesis". In *In Popl*, Vol. 2. Citeseer.
[6] Elli Androulaki, Seung Geol Choi, Steven M Bellovin, and Tal Malkin. 2008. Reputation systems for anonymous networks. In *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 202–218.
[7] Leo Bachmair and Harald Ganzinger. 2001. Resolution Theorem Proving. In *Handbook of Automated Reasoning (in 2 volumes)*, John Alan Robinson and Andrei Voronkov (Eds.). Elsevier and MIT Press, 19–99. https://doi.org/10.1016/b978-044450813-3/50004-7
[8] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K Rajamani. 2004. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *International Conference on Integrated Formal Methods*. Springer, 1–20.
[9] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. 2001. Automatic Predicate Abstraction of C Programs. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, Michael Burke and Mary Lou Soffa (Eds.). ACM, 203–213. https://doi.org/10.1145/378795.378846
[10] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. 2001. Boolean and Cartesian Abstraction for Model Checking C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2031)*, Tiziana Margaria and Wang Yi (Eds.). Springer, 268–283. https://doi.org/10.1007/3-540-45319-9_19
[11] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. 2021. Mac'n'Cheese: Zero-Knowledge Proofs for Boolean and Arithmetic Circuits with Nested Disjunctions. In *CRYPTO 2021, Part IV (LNCS, Vol. 12828)*, Tal Malkin and Chris Peikert (Eds.). Springer, Heidelberg, Germany, Virtual Event, 92–122. https://doi.org/10.1007/978-3-030-84259-8_4
[12] Michael Ben-Or, Oded Goldreich, Shafi Goldwasser, Johan Håstad, Joe Kilian, Silvio Micali, and Phillip Rogaway. 1990. Everything Provable is Provable in Zero-Knowledge. In *CRYPTO'88 (LNCS, Vol. 403)*, Shafi Goldwasser (Ed.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 37–56. https://doi.org/10.1007/0-387-34799-2_4
[13] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, Berkeley, CA, USA, 459–474. https://doi.org/10.1109/SP.2014.36
[14] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. 2013. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In *CRYPTO 2013, Part II (LNCS, Vol. 8043)*, Ran Canetti and Juan A. Garay (Eds.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 90–108. https://doi.org/10.1007/978-3-642-40084-1_6
[15] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Scalable Zero Knowledge via Cycles of Elliptic Curves. In *CRYPTO 2014, Part II (LNCS, Vol. 8617)*, Juan A. Garay and Rosario Gennaro (Eds.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 276–294. https://doi.org/10.1007/978-3-662-44381-1_16
[16] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. In *USENIX Security 2014*, Kevin Fu and Jaeyeon Jung (Eds.). USENIX Association, San Diego, CA, USA, 781–796.
[17] Dirk Beyer. 2017. Software verification with validation of results. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 331–349.
[18] Armin Biere. 2008. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation* 4, 2-4 (2008), 75–97.
[19] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. 2003. Bounded model checking. (2003).
[20] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. 2013. Recursive composition and bootstrapping for SNARKS and proof-carrying data. In *45th ACM STOC*, Dan Boneh, Tim Roughgarden, and Joan Feigenbaum (Eds.). ACM Press, Palo Alto, CA, USA, 111–120. https://doi.org/10.1145/2488608.2488623
[21] Alexander R. Block, Justin Holmgren, Alon Rosen, Ron D. Rothblum, and Pratik Soni. 2020. Public-Coin Zero-Knowledge Arguments with (almost) Minimal Time and Space Overheads. In *TCC 2020, Part II (LNCS, Vol. 12551)*, Rafael Pass and Krzysztof Pietrzak (Eds.). Springer, Heidelberg, Germany, Durham, NC, USA, 168–197. https://doi.org/10.1007/978-3-030-64378-2_7
[22] Alexander R. Block, Justin Holmgren, Alon Rosen, Ron D. Rothblum, and Pratik Soni. 2021. Time- and Space-Efficient Arguments from Groups of Unknown Order. In *CRYPTO 2021, Part IV (LNCS, Vol. 12828)*, Tal Malkin and Chris Peikert (Eds.). Springer, Heidelberg, Germany, Virtual Event, 123–152. https://doi.org/10.1007/978-3-030-84259-8_5
[23] Joseph Bonneau, Izaak Meckler, Vanishree Rao, and Evan Shapiro. 2020. Coda: Decentralized Cryptocurrency at Scale. Cryptology ePrint Archive, Report 2020/352. https://eprint.iacr.org/2020/352.
[24] Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune K. Jakobsen, and Mary Maller. 2018. Arya: Nearly Linear-Time Zero-Knowledge Proofs for Correct Program Execution. In *ASIACRYPT 2018, Part I (LNCS, Vol. 11272)*, Thomas Peyrin and Steven Galbraith (Eds.). Springer, Heidelberg, Germany, Brisbane, Queensland, Australia, 595–626. https://doi.org/10.1007/978-3-030-03326-2_20
[25] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. 2020. ZEXE: Enabling Decentralized Private Computation. In *2020 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 947–964. https://doi.org/10.1109/SP40000.2020.00050
[26] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. 2013. Verifying Computations with State. In *SOSP '17*. 341–357.
[27] Ran Canetti. 2000. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology* 13, 1 (Jan. 2000), 143–202. https://doi.org/10.1007/s001459910006
[28] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A tool for checking ANSI-C programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 168–176.
[29] Matthew Clegg, Jeff Edmonds, and Russell Impagliazzo. 1996. Using the Groebner Basis Algorithm to Find Proofs of Unsatisfiability. In *28th ACM STOC*. ACM Press, Philadephia, PA, USA, 174–183. https://doi.org/10.1145/237814.237860
[30] Martin Davis and Hilary Putnam. 1960. A computing procedure for quantification theory. *Journal of the ACM (JACM)* 7, 3 (1960), 201–215.
[31] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Proofs and Refutations, and Z3.. In *LPAR Workshops*, Vol. 418. Citeseer, 123–132.
[32] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. 2021. Line-Point Zero Knowledge and Its Applications. In *2nd Conference on Information-Theoretic Cryptography*.
[33] Niklas Eén and Niklas Sörensson. 2003. An extensible SAT-solver. In *International conference on theory and applications of satisfiability testing*. Springer, 502–518.
[34] Zhiyong Fang, David Darais, Joseph P. Near, and Yupeng Zhang. 2021. Zero Knowledge Static Program Analysis. In *ACM CCS 2021*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, Virtual Event, USA, 2951–2967. https://doi.org/10.1145/3460120.3484795
[35] Jonathan Frankle, Sunoo Park, Daniel Shaar, Shafi Goldwasser, and Daniel J. Weitzner. 2018. Practical Accountability of Secret Processes. In *USENIX Security 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, Baltimore, MD, USA, 657–674.
[36] Nicholas Franzese, Jonathan Katz, Steve Lu, Rafail Ostrovsky, Xiao Wang, and Chenkai Weng. 2021. Constant-Overhead Zero-Knowledge for RAM Programs. In *ACM CCS 2021*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, Virtual Event, USA, 178–191. https://doi.org/10.1145/3460120.3484800
[37] Zhaohui Fu, Yogesh Marhajan, and Sharad Malik. 2004. Zchaff sat solver. *Princeton University. Princeton, NJ* 8544 (2004).
[38] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. 2013. Quadratic Span Programs and Succinct NIZKs without PCPs. In *EUROCRYPT 2013 (LNCS, Vol. 7881)*, Thomas Johansson and Phong Q. Nguyen (Eds.). Springer, Heidelberg, Germany, Athens, Greece, 626–645. https://doi.org/10.1007/978-3-642-38348-9_37
[39] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1986. Proofs that Yield Nothing But their Validity and a Methodology of Cryptographic Protocol Design (Extended Abstract). In *27th FOCS*. IEEE Computer Society Press, Toronto, Ontario, Canada, 174–187. https://doi.org/10.1109/SFCS.1986.47
[40] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *19th ACM STOC*, Alfred Aho (Ed.). ACM Press, New York City, NY, USA, 218–229. https://doi.org/10.1145/28395.28420
[41] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1991. Proofs That Yield Nothing But Their Validity Or All Languages in NP Have Zero-Knowledge Proof Systems. *Journal of the ACM* 38, 3 (1991), 691–729.
[42] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (may 1996), 431–473. https://doi.org/10.1145/233551.233553
[43] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *EUROCRYPT 2016, Part II (LNCS, Vol. 9666)*, Marc Fischlin and Jean-Sébastien Coron (Eds.). Springer, Heidelberg, Germany, Vienna, Austria, 305–326. https://doi.org/10.1007/978-3-662-49896-5_11
[44] Armin Haken. 1985. The intractability of resolution. *Theoretical computer science* 39 (1985), 297–308.

[45] Carmit Hazay and Yehuda Lindell. 2010. A Note on Zero-Knowledge Proofs of Knowledge and the ZKPOK Ideal Functionality. Cryptology ePrint Archive, Report 2010/552. https://eprint.iacr.org/2010/552.

[46] David Heath and Vladimir Kolesnikov. 2020. A 2.1 KHz Zero-Knowledge Processor with BubbleRAM. In *ACM CCS 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM Press, Virtual Event, USA, 2055–2074. https://doi.org/10.1145/3372297.3417283

[47] David Heath, Yibin Yang, David Devecsery, and Vladimir Kolesnikov. 2021. Zero Knowledge for Everything and Everyone: Fast ZK Processor with Cached ORAM for ANSI C Programs. In *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 1538–1556. https://doi.org/10.1109/SP40001.2021.00089

[48] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. 2012. Secure two-party computations in ANSI C. In *ACM CCS 2012*, Ting Yu, George Danezis, and Virgil D. Gligor (Eds.). ACM Press, Raleigh, NC, USA, 772–783. https://doi.org/10.1145/2382196.2382278

[49] Zhangxiang Hu, Payman Mohassel, and Mike Rosulek. 2015. Efficient Zero-Knowledge Proofs of Non-algebraic Statements with Sublinear Amortized Cost. In *CRYPTO 2015, Part II (LNCS, Vol. 9216)*, Rosario Gennaro and Matthew J. B. Robshaw (Eds.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 150–169. https://doi.org/10.1007/978-3-662-48000-7_8

[50] Franjo Ivančić, Zijiang Yang, Malay K Ganai, Aarti Gupta, Ilya Shlyakhter, and Pranav Ashar. 2005. F-Soft: Software verification platform. In *International Conference on Computer Aided Verification*. Springer, 301–306.

[51] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. 2013. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *ACM CCS 2013*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM Press, Berlin, Germany, 955–966. https://doi.org/10.1145/2508859.2516662

[52] Daniel Kroening and Michael Tautschnig. 2014. CBMC–C bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 389–391.

[53] Ning Luo, Timos Antonopoulos, William Harris, Ruzica Piskac, Eran Tromer, and Xiao Wang. 2022. Proving UNSAT in Zero Knowledge. Cryptology ePrint Archive, Paper 2022/206. https://eprint.iacr.org/2022/206.

[54] Ning Luo, Samuel Judson, Timos Antonopoulos, Ruzica Piskac, and Xiao Wang. 2022. ppSAT: Towards Two-Party Private SAT Solving. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association.

[55] Kenneth L McMillan. 2003. Interpolation and SAT-based model checking. In *International Conference on Computer Aided Verification*. Springer, 1–13.

[56] Payman Mohassel, Mike Rosulek, and Alessandra Scafuro. 2017. Sublinear Zero-Knowledge Arguments for RAM Programs. In *EUROCRYPT 2017, Part I (LNCS, Vol. 10210)*, Jean-Sébastien Coron and Jesper Buus Nielsen (Eds.). Springer, Heidelberg, Germany, Paris, France, 501–531. https://doi.org/10.1007/978-3-319-56620-7_18

[57] Madanlal Musuvathi and Shaz Qadeer. 2007. Iterative context bounding for systematic testing of multithreaded programs. *ACM Sigplan Notices* 42, 6 (2007), 446–455.

[58] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. 2011. Optimal Verification of Operations on Dynamic Sets. In *CRYPTO 2011 (LNCS, Vol. 6841)*, Phillip Rogaway (Ed.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 91–110. https://doi.org/10.1007/978-3-642-22792-9_6

[59] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2013. Pinocchio: Nearly Practical Verifiable Computation. In *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, Berkeley, CA, USA, 238–252. https://doi.org/10.1109/SP.2013.47

[60] J. A. Robinson. 1965. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM* 12, 1 (jan 1965), 23–41. https://doi.org/10.1145/321250.321253

[61] John Alan Robinson and Andrei Voronkov (Eds.). 2001. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press. https://www.sciencedirect.com/book/9780444508133/handbook-of-automated-reasoning

[62] Victor Shoup et al. 2001. NTL: A library for doing number theory.

[63] Neil Thapen. 2016. A tradeoff between length and width in resolution. *Theory of Computing* 12, 1 (2016), 1–14.

[64] Paul Valiant. 2008. Incrementally Verifiable Computation or Proofs of Knowledge Imply Time/Space Efficiency. In *TCC 2008 (LNCS, Vol. 4948)*, Ran Canetti (Ed.). Springer, Heidelberg, Germany, San Francisco, CA, USA, 1–18. https://doi.org/10.1007/978-3-540-78524-8_1

[65] Psi Vesely, Kobi Gurkan, Michael Straka, Ariel Gabizon, Philipp Jovanovic, Georgios Konstantopoulos, Asa Oines, Marek Olszewski, and Eran Tromer. 2022. Plumo: An Ultralight Blockchain Client. In *FC 2022 (LNCS)*. Springer, Heidelberg, Germany.

[66] Riad S. Wahby, Srinath T. V. Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. 2015. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS 2015*. The Internet Society, San Diego, CA, USA.

[67] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. 2016. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit.

[68] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. 2021. Wolverine: Fast, Scalable, and Communication-Efficient Zero-Knowledge Proofs for Boolean and Arithmetic Circuits. In *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 1074–1091. https://doi.org/10.1109/SP40001.2021.00056

[69] Yichen Xie and Alex Aiken. 2007. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29, 3 (2007), 16–es.

[70] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. 2021. QuickSilver: Efficient and Affordable Zero-Knowledge Proofs for Circuits and Polynomials over Any Field. In *ACM CCS 2021*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, Virtual Event, USA, 2986–3001. https://doi.org/10.1145/3460120.3484556

[71] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets (Extended Abstract). In *27th FOCS*. IEEE Computer Society Press, Toronto, Ontario, Canada, 162–167. https://doi.org/10.1109/SFCS.1986.25