Towards Privacy-Preserving Verification

Timos Antonopoulos[1], Ning Luo[2], and Ruzica Piskac[1(✉)]

[1] Yale University, New Haven, USA
{timos.antonopoulos,ruzica.piskac}@yale.edu
[2] University of Illinois Urbana-Champaign, Illinois, USA
nl27@illinois.edu

Abstract. Program verification provides stronger guarantees of correctness than standard testing. The verification process takes a program as input and derives a mathematical formula. Proving that a program is correct then reduces to establishing that this derived formula is unsatisfiable. Traditionally, automated reasoning tools can be used to determine unsatisfiability automatically. Furthermore, modern solvers can also produce a proof of unsatisfiability. However, these techniques typically rely on the proof and the underlying code being publicly available, which may not be desirable for certain applications.

This work shows how to address this problem. Our team initially developed a protocol for validating the unsatisfiability of Boolean formulas in privacy-preserving settings. Building on these initial results, we devised ZKSMT, a virtual machine for validating unsatisfiability results produced by SMT solvers in zero-knowledge settings. In this paper we describe the theoretical foundations of such virtual machines and demonstrate how they can be applied to the theories of uninterpreted functions and linear integer arithmetic, two of the most widely used theories in verification.

We conclude by outlining how the full formal verification workflow can be adapted to operate in privacy-preserving settings.

1 Introduction

Conventional approaches to verifying software take for granted that the code under scrutiny can be freely inspected. A verifier is given both the program and a formal description of how it ought to behave, and then works to establish that the program lives up to that description. Open-source projects lend themselves naturally to this paradigm, as do proprietary systems in which the organization performing the verification has full access to the underlying codebase.

Reality, however, presents cases that fall outside this comfortable assumption of code accessibility. There are often circumstances in which the very party that needs reassurance about a piece of software is also the one that should be kept away from the software source code. A company's IP may reside in its codebase, proprietary methods, hard-won algorithms, and competitive secrets, making it impossible to hand over that code to an outside organization for auditing, even

A. Sampaio and M. Stoelinga (Eds.): FM 2026, LNCS 16556, pp. 21–43, 2026.
https://doi.org/10.1007/978-3-032-26204-2_2

if the organization constantly runs the software in some encrypted or obfuscated form. In this setting, verification must be delegated to a trusted third party—one granted access to the code for inspection, yet equally bound to preserve its confidentiality.

Privacy-preserving verification tools have been proposed to eliminate this dependence on a trusted intermediary entirely. By integrating cryptographic techniques into formal verification methods, such tools enable an untrusted party to verify the correctness of a confidential program without learning anything about its internals. In privacy-preserving verification setting, two mutually distrusting parties interact: the program owner and the verifier. The program owner wishes to convince the verifier that her code satisfies some desired property, such as safety or soundness, without exposing the code itself. The verifier, for his part, wants a guarantee that the program behaves exactly as the owner claims. Each party can act maliciously: a dishonest program owner might cheat about her program, for instance by implementing only a subset of the functionality the verifier expects, while a dishonest verifier might attempt to extract information about the program from the verification process itself.

Drawing inspiration from classical program verification, privacy-preserving verification builds on the same logical foundation that underlies most modern verification tools: automated reasoning and formula solving. In the standard, non-private setting, the claim a program owner establishes is that the program cannot violate a given property under any circumstances. When the program and its specification are encoded as logical formulas, this reduces to showing that a certain formula is *unsatisfiable*. When these formulae are indeed unsatisfiable, the certificate that asserts the formula's unsatisfiability, also known as the refutation proof, can be output by the formula solvers. Privacy-preserving verification lifts this foundation into a private setting. Rather than handing over the formula or its refutation proof, the program owner proves to the verifiers that a formula is unsatisfiable, without revealing any information on formulae or proofs.

Proving the unsatisfiability of a formula in a privacy-preserving setting is achieved via *zero-knowledge proof* (ZKP) [20,22]. ZKP is a cryptographic protocol between a prover and a verifier in which the prover convinces the verifier that a statement, which can contain confidential components, is true without revealing any information beyond the truth of the statement itself. In our setting, the statement is that a valid refutation proof for a private formula exists; the prover establishes this fact without disclosing the formula, the proof, or anything else about the program under verification.

The logical formalisms employed in program verification differ in expressive power. This paper explains the ZKP of refutation for the three principal automated-reasoning formalisms employed in software verification: Boolean unsatisfiability (SAT/UNSAT), Satisfiability Modulo Theories (SMT), and Quantified Boolean Formulas (QBF). In this paper, we survey these recent lines of work that collectively establish a coherent methodology for privacy-preserving automated reasoning.

1. **Verifying UNSAT of propositional Boolean formulae.** Luo et al. [34] introduced the first practical ZKP for proving propositional unsatisfiability, addressing a problem that had resisted practical treatment despite the theoretical existence of ZK proofs for all of PSPACE. The key insight is to prove, in ZKP, the validity of a resolution refutation proof. Clauses are encoded as low-degree polynomials over a finite field, where the roots correspond to the field-element encodings of the clause's literals. This polynomial representation is compatible with a wide class of efficient ZKP backends. The resolution rule then translates naturally into an arithmetic divisibility check over the committed polynomials.
2. **Verifying unsatisfiability of SMT formulae.** Luick et al. [33] extended this approach to the full SMT setting, which is the language in which realistic program verification tools actually operate. The central challenge is that SMT refutation proofs employ dozens of theory-specific inference rules beyond resolution, such as Farkas' lemma certificates, congruence closure steps, and others, making the proof objects far more complex than their purely propositional counterparts and rendering the machinery of ZKUNSAT inapplicable. ZKSMT overcomes this by introducing a tailored zero-knowledge virtual machine (ZKVM) modeled on the Von Neumann architecture. Theory-specific proof rules are compiled into VM instructions, and the main checker performs step-by-step proof validation in ZKP. Instantiated with the theories of equality and linear integer arithmetic and evaluated on Boogie-generated verification benchmarks, the system achieves a three-order-of-magnitude speedup over a general-purpose ZKP baseline.
3. **ZK proofs for unsatisfiability of QBF (PSPACE).** Karthikeyan et al. [27] push the frontier further, presenting the first *practical* ZK protocols for PSPACE-complete statements by targeting QBF evaluation. The core idea is to validate *quantified resolution* (Q-Res) proofs in ZKP, extending the polynomial encoding of [34] to handle universal and existential quantifier alternation. The work also introduces a complementary protocol for proving knowledge of a *winning strategy* for a QBF game, which is often the object of interest in hardware synthesis and reactive verification. Evaluated on QBFEVAL benchmarks, the system verifies 72% of instances via Q-Res proof and 82% of instances' winning strategies within 100 s.

2 Background

In deductive verification, the verification process typically involves proving the validity of a logical implication of the form $F \Rightarrow G$, where F represents the program semantics and G denotes the desired safety or correctness properties. To prove this validity, automated solvers typically demonstrate that the negation of the implication, $F \wedge \neg G$, is unsatisfiable (UNSAT). This section provides the necessary background for privacy-preserving automated reasoning, specifically focusing on resolution proofs, proofs of unsatisfiability, and zero-knowledge proofs.

2.1 Automated Reasoning and Proofs of Unsatifiability

The ability to automatically determine the satisfiability of a given formula, either by finding a satisfying assignment or providing a proof of its non-existence, remains a fundamental challenge in computer science. A significant milestone in this domain was the development of SAT solvers and the DPLL (Davis-Putnam-Logemann-Loveland) algorithm [15]. A SAT solver accepts a Boolean formula, typically in Conjunctive Normal Form (CNF), and determines if a satisfying assignment exists. While SAT solvers operate within the bounds of decidable propositional logic, first-order theorem provers extend these capabilities to tackle formulas expressed in first-order logic, which offers greater expressivity but introduces undecidability in the general case [43].

Despite the inherent undecidability of first-order logic, resolution-based first-order theorem provers have been highly effective in proving formulas coming from mathematics and formal verification [42]. All these systems are based on the resolution rule [38], which defines a refutationally complete calculus for first-order logic. In its simplest propositional form, the resolution rule is defined as:

$$\frac{A \vee B \quad \neg A \vee C}{B \vee C}$$

In the first-order case, this rule is also combined with unification to handle variables.

The refutational completeness of the resolution calculus means that for every unsatisfiable formula, a proof of its unsatisfiability can be systematically derived [3]. These resolution-based derivations serve as the foundational framework for a wide variety of modern proof formats. In the context of SAT solvers, for instance, the DRAT (Deletion Resolution Asymmetric Tautology) format has become the de facto standard for certifying solver outputs [46].

However, reasoning algorithms for first-order theories, in particular those used in SMT solvers, is often not purely resolution-based. SMT solvers employ specialized decision procedures to handle specific theories such as linear arithmetic, bit-vectors, or uninterpreted functions. Consequently, generating and verifying proofs in SMT is a significantly more intricate task than in propositional logic, as the proof must encapsulate the interactions between the SAT engine and multiple theory-specific solvers. Typical proof formats used by modern SMT solvers to address this complexity include LFSC (Logical Framework with Side Conditions) [41] and the Alethe proof format [39].

While standard SMT and resolution proofs are designed for public verifiability, where the verifier has full access to the formula and its derivation, certain industrial applications require privacy-preserving automated reasoning. In such scenarios, a prover must demonstrate that a formula is unsatisfiable (or satisfiable) without revealing the internal structure of the program F, the specific properties G, or the intermediate steps of the proof derivation process. This requirement leads to the integration of zero-knowledge proofs into the automated reasoning pipeline.

2.2 Zero Knowledge Proof

A standard interactive Zero-Knowledge Proof (ZKP) [20,22] protocol involves two parties: the *prover* and the *verifier*. It is important to stress that these two terms have slightly different meanings in the cryptography and formal verification communities, which can occasionally lead to misunderstanding. In automated reasoning, a "prover" typically refers to the algorithmic engine (such as a SAT or SMT solver) that searches for a proof. In the context of ZKPs, however, the prover is a party that already possesses a witness (a resolution proof or a satisfying assignment) and has to convince the verifier of a statement's validity. In these settings, the witness remains strictly private. The goal of the protocol is for the prover to demonstrate the correctness of the statement, while maintaining that the verifier learns nothing about the underlying witness, specific axioms, or intermediate resolution steps used to derive the result, other than the fact that the statement itself is indeed true.

Formally, a ZKP is defined as follows: given a public predicate P and a public input x, a prover must convince a verifier that it holds a private input w such that $P(w, x) = 1$, while revealing no additional information about w. Both prover and verifier may behave maliciously: the prover may not possess a valid witness w satisfying the predicate, and the verifier may attempt to extract information about the prover's private input. A secure ZKP must satisfy two properties. *Soundness* ensures that a malicious prover cannot convince the verifier of a false statement. *Zero-knowledge* ensures that a malicious verifier learns nothing about the private input w beyond the fact that the statement is true.

The soundness and zero-knowledge properties of ZKP make them a natural cryptographic foundation for privacy-preserving verification. Soundness plays the role of a trust anchor for the verifier: it guarantees that no matter how a dishonest program owner constructs the proof, the prover cannot convince the verifier that an incorrect or unsafe program satisfies the specification. Zero-knowledge, on the other hand, protects the program owner: the verifier learns nothing about the program, its specification, or the intermediate steps of the verification beyond the single bit of information that the program is correct. There have been many lines of work designing practically efficient and secure ZK protocols under different settings and assumptions [21,23–25].

We next describe a widely adopted ZKP paradigm known as *commit-and-prove* [9], which underlies all of the privacy-preserving verification protocols surveyed in this paper.

Commit and Prove. In the commit-and-prove ZKP, the interaction between the prover and verifier is split into two phases. In the *commitment phase*, the prover cryptographically commits to its private witness, using a binding and hiding commitment scheme. In our setting, the witness consists of the formula and the refutation proof. Binding ensures that the prover cannot alter the committed values after the fact. Hiding ensures that the commitments reveal nothing about the underlying witness to the verifier. Conceptually, commitment can be understood as a process where the prover encrypts and "locks" the proof. This

ensures that the proof cannot be altered during the protocol, while its contents remain hidden from the verifier. We use $[\![\cdot]\!]$ to denote a committed value.

In the *proving phase*, the prover demonstrates to the verifier that the committed values satisfy the desired predicate, for instance, that the committed proof is a valid refutation of the committed formula. Crucially, all interaction occurs only through the commitments: the underlying values are never revealed to the verifier at any point during the protocol.

The commit-and-prove ZKP can be applied, in principle, on any predicate P. Therefore, we apply it to validate refutation proofs for program verification. The protocol begins with the prover *committing* to the entire proof (for example, a resolution proof), via a cryptographic commitment scheme. After this commitment phase, the prover and verifier engage in a protocol to validate each step of the committed proof. Proof steps, such as resolution or redundancy checks, are individually checked without revealing the actual clauses or literals involved. By applying a zero-knowledge protocol to each proof step, the prover can demonstrate that a specific resolvent $B \vee C$ follows logically from its committed parents $A \vee B$ and $\neg A \vee C$, or that a clause meets the criteria for a DRAT-style deletion. This granular verification ensures that the verifier is convinced of the overall proof's validity while the underlying logic, axioms, and intermediate lemmas remain entirely private.

Despite the theoretical feasibility, the cost of verifying each proof step in ZK can be prohibitive without careful design. To make commit-and-prove ZKP practical for refutation proof, the proof must first be *arithmetized*, i.e. translated from its logical form into an algebraic one, so that each proof step reduces to a relation over committed field elements that a ZKP backend can verify efficiently. The central challenge and the primary technical contribution of the protocols surveyed in this paper is to find arithmetizations that are simultaneously faithful to the logical semantics of the underlying proof system and amenable to the algebraic operations supported by ZKP backends. In what follows, we describe the key arithmetization techniques and algebraic relations that have been developed and exploited in privacy-preserving verification.

Polynomial Equivalence in ZKP. A polynomial is committed by committing all its coefficients. Given two committed polynomials, denoted by $[\![P(X)]\!]$ and $[\![Q(X)]\!]$, the prover can convince the verifier that $P(X) = Q(X)$ without revealing the polynomials themselves. The key tool is the *Schwartz–Zippel lemma* [40]: if P and Q are distinct polynomials of degree at most d over a finite field $\mathbb{F}$, then they agree on at most d points. Consequently, if both polynomials evaluate, using their commitments, to the same value at a uniformly random point $r \in \mathbb{F}$ chosen by the verifier, then $P = Q$ holds with overwhelming probability when $|\mathbb{F}| \gg d$. This reduces a global structural equality between two polynomials to a single field equality checking, making it exceptionally cheap to verify in ZK.

More generally, given committed polynomials $[\![P(X)]\!]$, $[\![Q(X)]\!]$, and $[\![R(X)]\!]$, the prover can demonstrate that $[\![P(X) \cdot Q(X)]\!] = [\![R(X)]\!]$ by again invoking the Schwartz–Zippel lemma at a random evaluation point. This primitive is partic-

ularly useful in privacy-preserving verification because many structural properties of proofs, such as clause containment and resolution steps, reduce naturally to divisibility relations between polynomials, which in turn reduce to product checks.

Encoding logical objects, such as resolution proofs, as polynomials and reducing logical relations to polynomial relations verified in ZK via Schwartz–Zippel, is the arithmetization backbone shared by ZKUNSAT, ZKSMT, and the QBF protocols described in this paper. It enables the prover to commit once to the entire proof structure and then discharge each proof step with a small, constant number of field operations, yielding the practical efficiency that makes privacy-preserving verification feasible at scale.

3 Proving Unsatisfiability of Propositional Boolean Formulae in Privacy Preserving Settings

In program verification, the assertion that a program is *safe* reduces precisely to the unsatisfiability of the formula encoding the program together with the negation of the safety property. An efficient ZKP for UNSAT is therefore a direct cryptographic primitive for privacy-preserving program verification. However, most ZKP systems focus on the converse proving knowledge of a satisfying assignment to a Boolean formula, circuit, or program. Luo et al. [34]'s work provides the first practical ZKP for proving UNSAT of propositional Boolean formulae. Formally, to prove unsatisfiability, the statement to be proved in ZK is:

$$\exists\, \pi \ : \ \pi \text{ is a valid resolution refutation of } \varphi, \tag{1}$$

where φ is a CNF formula known only to the formula or program owner.

3.1 Polynomial Encoding of Clauses

The central technical contribution of [34] is an algebraic encoding that represents clauses as low-degree polynomials over a finite field, enabling resolution steps to be validated via arithmetic checks that are amenable to zero-knowledge proof.

The literal encoding $\phi : \mathit{Lits} \to \mathbb{F}$ satisfies the negation constraint

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

so that a literal and its negation sum to the field identity. This is required to enable efficient ZK operations.

A clause $C = \ell_1 \vee \cdots \vee \ell_k$ is then encoded as the polynomial

$$\gamma(C)(X) = \prod_{j=1}^{k} \big(X - \phi(\ell_j)\big) \ \in \ \mathbb{F}[X], \tag{3}$$

whose roots are exactly the field-element encodings of the clause's literals. For example, the clause $C = x_1 \vee \neg x_2 \vee \neg x_3$ is encoded as:

$$\gamma(C)(X) = \big(X - \phi(x_1)\big)\big(X - \phi(\neg x_2)\big)\big(X - \phi(\neg x_3)\big). \tag{4}$$

The degree of $\gamma(C)$ equals the width of C, so for a proof of width w, all polynomials have degree at most w.

We now explain the intuition behind the polynomial encoding. A clause is a disjunction of literals, and as such it can be viewed as a *set* of literals with no inherent ordering. Under the literal encoding ϕ, this set of literals corresponds to a set of field elements in $\mathbb{F}$. A finite set of field elements is uniquely represented, up to a scalar factor, by the monic univariate polynomial whose roots are exactly those elements. This is precisely the clause encoding γ in (3): the polynomial $\gamma(C)$ has as its roots the field-element images $\phi(\ell)$ of the literals $\ell \in C$.

The key observation is that the central operation over clauses, namely, resolution relation, translates cleanly into this polynomial representation. Recall that resolution derives a new clause C_r from two premise clauses C_0 and C_1 by canceling a complementary pivot literal pair. At the set level, this corresponds to taking the union of the two literal sets after removing the pivot literal and its negation. At the polynomial level, this set-theoretic relationship is captured by the divisibility relations, which we explain in detail next.

3.2 Algebraic Characterization of Resolution

Lemma 1 (Implication as Divisibility [34]). *For clauses C and C', $C \to C'$ if and only if $\gamma(C) \mid \gamma(C')$ in $\mathbb{F}[X]$.*

This follows immediately from the fact that $C \subseteq C'$ implies every root of $\gamma(C)$ is a root of $\gamma(C')$, so the linear factors of $\gamma(C)$ divide $\gamma(C')$.

Weak Resolution. Rather than checking standard resolution directly, ZKUNSAT introduces *weak resolution*, which relaxes the standard rule while remaining sound and complete for refutation. A non-tautology clause C_r is a *weak resolvent* of C_0 and C_1 on pivot variable x if

$$C_0 \to C_r \vee x \qquad \text{and} \qquad C_1 \to C_r \vee \neg x. \tag{5}$$

mmmmmmm Standard resolution is the special case in which C_r is the smallest such clause.

Weak resolution improves the efficiency of verifying unsatisfiability in ZK without sacrificing the soundness. In standard resolution, the resolvent C_r must be *exactly* the clause obtained by merging the two premises and removing the pivot literal pair. Enforcing such equivalence in ZK is expensive. Weak resolution drops the exactness requirement and replaces it with a one-sided check: the "resolvent" C_r is allowed to contain more literals than the standard resolvent, i.e., it is only required to be a superset of the standard resolvent. This reduces the verification condition to a pure subset relation, which at the polynomial level corresponds to the two divisibility relations. Last, as established by Lemma 1, this relaxation preserves soundness: a formula is unsatisfiable if and only if it admits a weak resolution proof. A weak resolution proof mirrors a standard resolution proof, except that each resolution step is replaced by weak resolution

To verify a weak resolution step in zero knowledge, the prover commits to the polynomial encodings $q_0 = \gamma(C_0)$, $q_1 = \gamma(C_1)$, $q_r = \gamma(C_r)$, together with linear pivot polynomials $\rho(X) = X - \phi(x)$ and $\bar{\rho}(X) = X - \phi(\neg x)$. The *weak resolution test* asserts:

$$q_0 | q_r \cdot \rho, \tag{6}$$

$$q_1 | q_r \cdot \bar{\rho}, \tag{7}$$

$$\rho(X) + \bar{\rho}(1_{\mathbb{F}} - X) = 0. \tag{8}$$

Equation (6) encodes $C_0 \to C_r \vee x$ as polynomial divisibility: $q_0 \mid q_r \cdot \rho$, with w_0 the explicit quotient. Equation (7) encodes $C_1 \to C_r \vee \neg x$ analogously. Equation (8) enforces the negation consistency (2) between ρ and $\bar{\rho}$, ensuring that the two pivot polynomials genuinely encode complementary literals. All three equations are checked probabilistically via the Schwartz–Zippel lemma by evaluating at a random field point.

3.3 The ZKUNSAT Protocol

Setup. The prover holds a CNF formula $\varphi = C_1 \wedge \cdots \wedge C_m$ over n variables, together with a weak resolution refutation $\pi = (D_1, \ldots, D_\ell)$ of width w and length ℓ. Each D_j is either an input clause $C_i \in \varphi$ or a weak resolvent of two earlier clauses D_a and D_b in π on some pivot variable x, and the final clause $D_\ell = \bot$ is the empty clause. The verifier knows only the claimed dimensions (m, n, w, ℓ) of the formula and the proof.

Commit Phase. The prover encodes every input clause C_i and every derived clause D_j as a polynomial via (3), and commits to each polynomial using a commitment scheme. We denote them as $[\![\gamma(C]\!]$ for the committed polynomial of clause C. The prover also commits to additional quotient polynomials $w_0^{(j)}, w_1^{(j)}$ and the linear pivot polynomials $\rho^{(j)}, \bar{\rho}^{(j)}$ for each resolution step j.

Proving Phase. The prover convinces the verifier of the following three conditions, each checked in ZKP over the commitments.

1. *Resolution correctness.* For each derivation step $D_j = \mathrm{Res}(D_a, D_b, x)$, the committed polynomials satisfy the weak resolution test (6)–(8):

$$[\![w_0^{(j)}]\!] \cdot [\![D_a]\!] = [\![D_j]\!] \cdot [\![\rho^{(j)}]\!],$$

$$[\![w_1^{(j)}]\!] \cdot [\![D_b]\!] = [\![D_j]\!] \cdot [\![\bar{\rho}^{(j)}]\!],$$

$$[\![\rho^{(j)}(X)]\!] + [\![\bar{\rho}^{(j)}(1_{\mathbb{F}} - X)]\!] = 0.$$

   Each check reduces to a polynomial product equality verified via the Schwartz-Zippel lemma at a single random evaluation point, as we explained in Sect. 3.2 making the per-step cost essentially a constant number of field operations.

2. *Contradiction.* The final derived clause D_ℓ encodes the empty clause, i.e., $\gamma(D_\ell) = 1_{\mathbb{F}}$, the constant polynomial with no roots. The prover demonstrates that the committed polynomial $[\![D_\ell]\!]$ evaluates to $1_{\mathbb{F}}$ at every point, confirming that the derivation terminates in a contradiction.

Random-Access Memory. A central challenge in the proof phase is that each resolution step $D_j = \mathrm{Res}(D_a, D_b, x)$ may reference arbitrary earlier steps D_a and D_b at non-consecutive positions in π, requiring random access to the committed clause table. The protocol handles this using a *zero-knowledge random-access memory* (ZK-RAM) [18], which allows the prover to demonstrate, in zero knowledge, that the polynomial retrieved at a given index is consistent with the committed table, without revealing the index or the contents of the table. The ZK-RAM argument contributes a $O(\ell \log \ell)$ overhead to the proof size, which in practice is dominated by the $O(\ell \cdot w)$ cost of the resolution checks.

Security. Soundness follows from the binding property of the commitment scheme and Lemma 1: if the weak resolution test passes at a random evaluation point, the committed polynomials decode via p^* to a valid weak resolution step with overwhelming probability by the Schwartz–Zippel lemma. Zero-knowledge follows from the hiding property of the commitment scheme: the verifier's view consists entirely of commitments and polynomial evaluations at a single verifier-chosen random point, from which no information about φ, π, or the program they encode can be extracted. The only information revealed is the dimensions (m, n, w, ℓ) of the formula and the proof, which the verifier must know to bound its own verification cost.

4 Proving First-Order Unsatisfiability in Privacy-Preserving Settings

The propositional setting covered in Sect. 3 provides an approach to proving unsatisfiability of propositional Boolean formulae, but practical program verification rarely operates at the level of pure Boolean logic. Modern verification toolchains, such as Boogie [5], CBMC [10], Dafny [31], and their relatives, reduce program correctness to the unsatisfiability of *Satisfiability Modulo Theories* (SMT) formulas: Boolean combinations of atoms drawn from rich first-order theories such as equality with uninterpreted functions (EUF) and linear integer arithmetic (LIA). Extending the ZKUNSAT approach to this setting is the goal of ZKSMT [33].

The central difficulty is that SMT refutation proofs are structurally far more complex than their propositional counterparts. A resolution refutation of a CNF formula applies a single, uniform rule at every step, which is why the polynomial encoding of Sect. 3 suffices: every step reduces to the same divisibility check. An SMT refutation proof, by contrast, may invoke dozens of theory-specific inference rules: congruence closure, Farkas' lemma certificates, arithmetic normalization

steps, and many others, each with its own side conditions, and the number and shape of its premises.

ZKSMT overcomes this challenge by introducing a purpose-built *zero-knowledge virtual machine* (ZKVM) whose instruction set is precisely the set of SMT proof rules required to validate a given refutation. We describe its architecture and its instantiation on the theories of propositional logic, EUF, and LIA below.

4.1 The ZKSMT Virtual Machine

An SMT refutation proof can be viewed as a sequence of *proof steps*, each consisting of a rule identifier, a list of premises (references to earlier steps), and a conclusion formula. This sequential structure closely mirrors the execution model of a Von Neumann processor: rules correspond to instructions, proof steps correspond to program statements, and the formula being refuted corresponds to the program input. ZKSMT exploits this analogy directly.

Encoding Formulas: The Expression Table. Every formula that appears anywhere in the proof is stored, in abstract syntax tree (AST) form, in a read-only table M_e called the *expression table*. Each row of M_e represents one AST node and contains three fields: a *NodeID* identifying the operator (e.g., `Eq`, `Lt`, `And`), an *immediate address list* `ImmAddr` for constants and variable names, and an *indirect address list* `IndAddr` holding indices into M_e for the node's children. Sharing is explicit: if the same subformula t appears in ten different formulas, M_e contains exactly one entry for t, and all ten formulas reference it by index. This sharing is crucial for efficiency in the ZK setting, since equality of two subformulas reduces to equality of their indices, a single field comparison rather than a full structural traversal.

Encoding Proofs: The Step Table. The proof steps themselves are stored in a second read-only table M_p. Each row of M_p records a *StepID* (the logical ordering of the step), a *RuleID* (identifying the theory and the rule applied), a *Result* (an index into M_e for the conclusion formula), and a list of *Premises* (StepIDs of the proof steps whose conclusions are used as premises). The size of the machine is parameterized by π (the maximum number of proof steps), χ (the number of expressions in M_e), μ (the maximum number of premises in any rule), α (the maximum argument list length of any expression), and ρ (the number of distinct rules used).

Machine Execution. The main verification loop iterates over all π proof steps. For each step, the machine (i) fetches the rule identifier and the conclusion from M_p and M_e, (ii) fetches the conclusion expressions of each listed premise by reading the corresponding entries of M_p and M_e, (iii) dispatches to the *checking instruction* for the given `RuleID`, which asserts that the conclusion is a valid consequence of the premises under that rule, and (iv) records the step's `StepID` in an append-only array D. After the loop, a permutation check confirms that D

is a permutation of $\{0, 1, \ldots, \pi - 1\}$, ensuring that every proof step was validated exactly once. The acyclicity of the proof graph is enforced by requiring that the `StepID` of every premise is strictly smaller than the `StepID` of the step that uses it.

This architecture is both *complete* and *sound*. Any valid SMT refutation proof can be verified by some ZKSMT instance with appropriate size parameters. Any proof accepted by the machine corresponds to a valid derivation in the underlying theory. These properties are proved formally in [33].

4.2 Theory Instantiations

To make the ZKVM useful in practice, one must supply a checking instruction for each inference rule of each theory. We describe the most significant examples.

Propositional Rules. Simple tautology rules, such as the law of the excluded middle ($a \vee \neg a$), require only pattern-matching on the structure of the conclusion node: the checking instruction confirms that the `NodeID` is `Or`, retrieves its two children from M_e, and verifies that one is the negation of the other by comparing indices. The Resolution rule is more involved: its checking instruction must verify that the multiset of literals in one premise, minus the pivot literal p, is contained in the conclusion's literal multiset, and similarly for the other premise and the negation $\neg p$ of the pivot. The natural language of these conditions is that of *multiset containment*.

Equality with Uninterpreted Functions (EUF). The congruence rule, which derives $f(a_1, \ldots, a_n) = f(b_1, \ldots, b_n)$ from pairwise equalities $a_i = b_i$, is checked by ZKSMT in an alternative but logically equivalent formulation: a disjunction is derived from no premises, asserting that either the two function applications are equal or some argument pair is not. The checking instruction confirms that the conclusion has the correct disjunctive structure, identifies the pair of function applications, and verifies that the remaining disjuncts match the corresponding argument pairs by index comparison in M_e.

Linear Integer Arithmetic (LIA). The two most important LIA rules are multiplication distribution (MulDist) and Farkas' lemma. The MulDist checking instruction validates the identity $c \cdot (\sum_i d_i \cdot x_i) = \sum_i (c \cdot d_i) \cdot x_i$ by iterating over the children of the sum nodes in lockstep, verifying that scaling factors combine correctly and that child expressions match by index. Farkas' lemma is a linear arithmetic rule that derives a strict inequality from a collection of hypotheses; its checking instruction confirms the structural pattern of the premise and conclusion and verifies that the coefficients satisfy the required non-negativity conditions.

4.3 Zero-Knowledge Instantiation

Running the ZKSMT machine in zero knowledge requires hiding the contents of M_e and M_p, that is, the formula and the proof, while still allowing the verifier

to be convinced that every step was correctly executed. ZKSMT is instantiated using the VOLE-based ZK backend of [6,45], which supports efficient polynomial operations and read-only memory accesses.

Committing the Proof. Every integer field of every entry in M_e and M_p is committed bit-by-bit using the VOLE commitment scheme. To avoid leaking the `NodeID` of an expression from the sizes of its argument lists, all `ImmAddr` and `IndAddr` lists are padded to the uniform maximum length α.

Group Checking. A naive implementation would, at each step, multiplex over all ρ possible checking instructions—executing all of them and selecting the result corresponding to the private `RuleID`. This incurs a ρ-fold overhead. ZKSMT instead uses *group checking*: all steps that use the same rule are verified consecutively, invoking only that rule's checking instruction. The verifier therefore learns how many times each rule is applied—the *rule histogram*—but nothing about which specific steps use which rule. This is the primary privacy–efficiency trade-off in ZKSMT: revealing the rule histogram enables SIMD-style batch optimizations (verifying many instances of the same rule in a single amortized pass) at the cost of leaking aggregate rule usage counts.

Multiset Checks via Polynomial Arithmetic. Many checking instructions reduce to verifying multiset containment relations among the argument lists of the relevant M_e entries—as seen in the Resolution and DeDup rules above. ZKSMT encodes multisets as univariate polynomials over a finite field $\mathbb{F}$: a multiset $\{\ell_0, \ldots, \ell_d\}$ is represented as $\prod_i (X - \psi(\ell_i))$, where ψ is an injective encoding of expression indices into $\mathbb{F}$. Multiset containment $A \subseteq B$ then corresponds to polynomial divisibility, and the verifier can check this at a single random evaluation point via the Schwartz–Zippel lemma. A more refined relation, containment of distinct elements, ignoring multiplicity, written $A \subseteq_d B$, is checked using a bivariate polynomial protocol from [37]. Together, these two relations suffice to implement the checking instructions for all propositional, EUF, and LIA rules in ZKSMT.

Read-Only Memory Accesses. Fetching entries from M_e and M_p by committed (private) index requires a ZK read-only memory (ROM) protocol. ZKSMT uses the constant-overhead ZK-RAM construction of [19], which provides amortized $O(1)$ cost per access. In practice, memory accesses dominate the cost of rules like Farkas' lemma (which must read every argument of its premise and conclusion to match their structure), while arithmetic operations dominate for rules like Resolution and Consolidate (which perform multiset checks but can work directly with expression indices without reading full node data).

5 Privacy-Preserving Reasoning for Other Formalisms

The protocols described in Sects. 3 and 4 establish that the unsatisfiability of propositional and SMT formulas can be certified in ZK with practical efficiency.

Many problems of genuine interest in formal verification lie strictly in PSPACE, beyond the complexity class within which both ZKUNSAT and ZKSMT operate. Hardware synthesis, reactive controller, and planning under uncertainty all give rise to PSPACE-complete decision problems.

The classical result IP = PSPACE ensures that zero-knowledge proofs exist for all PSPACE-complete problems, but the generic construction based on the sumcheck protocol requires evaluating high-degree multivariate polynomials over large fields, with degree scaling linearly in the number of variables. This renders it impractical for real-world instances. Karthikeyan et al. [27] bridge this gap by presenting the first practical ZK protocols for PSPACE-complete statements, targeting the evaluation of Quantified Boolean Formulas (QBFs).

5.1 Q-Resolution Proofs

A QBF in prenex conjunctive normal form (PCNF) has the structure

$$\Psi = Q_1 X_1\, Q_2 X_2 \cdots Q_k X_k.\ \psi(x_1, \ldots, x_n),$$

where each $Q_i \in \{\forall, \exists\}$ is a quantifier, each X_i is a block of Boolean variables, and ψ is a propositional CNF formula over those variables. QBF evaluation is PSPACE-complete, and many practically important problems reduce to it naturally.

The Q-resolution (Q-Res) proof system provides a complete and practical certificate format for establishing that a QBF is *false*. A Q-Res proof is a sequence of clauses derived from the matrix ψ by two inference rules, terminating in the empty clause $\bot$.

Universal Reduction ($\forall$*-Red).* Let y be the innermost existential literal in a clause C. Any universal literal $x_i \in C$ that appears *after* y in the quantifier prefix (i.e., $y \prec x_i$) can be safely removed from C without affecting the truth value of Ψ. Intuitively, the universal player's choice of x_i can always be made after observing the existential choice of y, so x_i provides no additional power to make the clause false.

Existential Resolution ($\exists$*-Res).* Given two clauses C_a and C_b and an existential variable $y \in X_\exists$ such that $y \in C_a$ and $\neg y \in C_b$, the Q-resolvent is $C_r = (C_a \cup C_b) \setminus \{y, \neg y\}$, provided C_r is non-tautological. This is standard propositional resolution restricted to existentially quantified pivot variables.

A complete Q-Res proof derives the empty clause by interleaving $\forall$-Red and $\exists$-Res steps. The soundness and completeness of Q-Res is classical: a closed PCNF formula is unsatisfiable if and only if a Q-Res proof exists [4,36].

5.2 Polynomial Encoding for Q-Res

The polynomial clause encoding introduced for propositional UNSAT (Sect. 3) extends to the QBF setting, but with a key augmentation: literals must carry quantifier-order information to enable checking of $\forall$-Red. The encoding of a literal ℓ is the concatenation of two binary strings,

$$\varepsilon(\ell) = \mathrm{order}(\ell)_{\mathbf{B}} \parallel \mathrm{sign}(\ell),$$

where $\text{sign}(\ell) \in \{0, 1\}$ indicates the polarity of the literal and $\text{order}(\ell)$ is the position of its variable in the quantifier prefix. This encoding is interpreted both as a finite field element (enabling polynomial clause encoding) and as a natural number (enabling comparison of quantifier levels). The same injective negation constraint $\varphi(\ell) + \varphi(\neg\ell) = 1_{\mathbb{F}}$ required for resolution checks continues to hold under this encoding. Two public sets $L_\forall$ and $L_\exists$ record the literals of universally and existentially quantified variables respectively, and their membership can be checked in ZK using standard set-argument protocols.

Checking ∃-Res. An existential resolution step is a standard (propositional) resolution step with the additional requirement that the pivot literal ℓ_p is existentially quantified. The pivot membership check $\ell_p \in L_\exists$ is appended to the resolution check from [34], adding negligible overhead.

Checking ∀-Red. A valid universal reduction step on clause C producing clause C_r is characterized by four conditions: the removed literals W_{rem} must all be universally quantified; they must all appear after the innermost existential literal w_ℓ in the prefix; the retained literals W_{res} must all appear before w_ℓ; and C decomposes as $W_{\text{res}} \cup W_{\text{rem}} \cup \{w_\ell\}$ while $C_r = W_{\text{res}} \cup \{w_\ell\}$. The prover commits all four components as extended witnesses. Quantifier membership is checked via $L_\forall$ and $L_\exists$; quantifier order comparisons exploit the integer interpretation of the literal encoding, checked via a range-proof gadget; and the clause decomposition is verified using the polynomial equality operations already needed for ∃-Res. Together, these checks add a small, fixed overhead per proof step relative to a propositional resolution step.

Protocol Structure. Both parties agree publicly on the proof length R, the maximum clause width w, and the maximum number of literals removed in any single ∀-Red step. The prover commits all clauses, both input clauses from ψ and derived clauses, via the polynomial encoding, stores them in a ZK append-only array (as in ZKUNSAT [34]), and iterates through each Q-Res step, invoking the ∃-Res and ∀-Red checks. After all R steps, the verifier confirms that the final derived clause is the empty clause. The ZKQRES protocol is proven to be a ZKP of knowledge for QBF falsification.

5.3 Winning Strategies in Zero Knowledge

Beyond certifying the evaluation of a QBF, the prover may wish to demonstrate possession of a concrete *winning strategy*. For a false QBF, this is a Herbrand strategy: a set of Boolean functions $H = \{g_x : \text{Pred}_\exists(x) \to \{0, 1\} \mid x \in X_\forall\}$ that assign values to universally quantified variables based on the existential variables preceding them in the prefix, in a way that forces the formula to be false regardless of the existential player's choices. Formally, substituting each universal variable x with its Herbrand function g_x in ψ yields a propositional formula ψ_H, the Herbrandization, that is unsatisfiable. Symmetrically, for a true QBF, a Skolem strategy assigns values to existential variables as functions of preceding universal variables, yielding a Skolemization ψ_S that is a tautology.

The ZKWS protocol in [33] verifies a winning strategy in zero knowledge by decomposing the task into three sub-problems. First, the Herbrand (or Skolem) function must be *well-formed*: each universal variable is defined exactly once, the dependency graph is acyclic, and each variable depends only on literals that precede it in the quantifier prefix. Second, the propositional formula ψ_H produced by applying the strategy must be a correct Tseitin CNF encoding of the Herbrand function. Third, ψ_H must be unsatisfiable, which is handled by invoking ZKUNSAT [34] as a sub-protocol.

The Herbrand function is represented as a list of triples $\{(x_i, \ell_i^a, \ell_i^b)\}$ encoding gates $x_i \Leftrightarrow \ell_i^a \wedge \ell_i^b$, committed as an And-Inverter Graph (AIG). Well-formedness is verified in ZK by checking uniqueness of output variables via set operations, acyclicity by confirming that each input literal refers to an earlier-indexed variable (leveraging the append-only array), and prefix-consistency by checking that the dependency level of each defined variable does not exceed its quantifier-prefix position, using the integer interpretation of literal encodings. Substitution correctness is verified by checking that the Tseitin clauses of ψ_H are consistent with the committed AIG gates, using the literal membership primitives of the clause commitment scheme.

Efficiency Advantage. When the prover is willing to reveal the size of the winning strategy, the ZKP for PSPACE reduces to ZKP for UNSAT with only a small overhead for well-formedness and substitution checks. On instances where compact winning strategies exist, this yields dramatic savings compared to Q-Res proof checking: the paper reports reductions in proving time of up to 200× on individual benchmarks.

ZKUNSAT Optimization. A practical complication arises when running ZKUNSAT on ψ_H: all clauses must be padded to the width of the widest clause, and the Herbrandization often produces many narrow clauses alongside a few wide ones. To mitigate this, the protocol introduces a *clause bucketing* scheme that partitions the proof into groups of clauses with similar width, running a separate ZKUNSAT instance per bucket. This exposes the bucket-width histogram (a mild leakage) but reduces padding waste substantially, yielding approximately a 2× runtime improvement in practice.

5.4 ZKPs for Regular Expression Equivalence

Proving that a QBF instance is unsatisfiable is a PSPACE-complete problem, and we have developed other frameworks that target problems in the same complexity class, namely, regular expression equivalence. More specifically, party A can convince party B that two private regular expressions are equivalent, or more generally that the language of one is contained in the language of the other, without revealing either expression unless explicitly chosen to do so [29]. A naïve approach would be to convert the regular expressions into finite automata and prove the existence of a simulation relation between them, which would require separate ZK protocols for the conversion step and for the simulation relation

itself, resulting in a more complex pipeline with longer proofs. Instead, our approach operates directly on regular expressions via a custom, sound and complete calculus of proof rules based on Brzozowski derivatives and coinduction, keeping individual proof step checking to constant or linear time and avoiding the overhead of validating format conversions. Intuitively, this approach fuses the automaton construction and the simulation argument into a single coinductive proof over the regular expressions themselves.

6 Challenges and Future Directions

The privacy-preserving verification techniques described above establish a strong foundation, but they leave open a number of important challenges. Some concern the broader verification pipeline: ensuring that the chain of trust extends from the logical formula all the way to the running binary. Others are foundational: extending the scope of what can be verified privately, from standard safety properties to richer relational and hyperproperties, and from propositional and first-order reasoning to problems in higher complexity classes. In this section we look at some of these challenges and outline directions for future work, including ongoing work from our group that begins to address some of them.

6.1 Completing the Verification Pipeline

Once a piece of code has been proved correct using the privacy-preserving techniques surveyed above, several further aspects of the deployment pipeline still require trust. The techniques described so far establish that a combined formula is unsatisfiable: that is, the formula encoding the program, conjoined with the negation of the specification, admits no satisfying assignment. This is precisely the statement that the program cannot violate the specification. However, this guarantee is only as strong as the chain of transformations that produced the formula in the first place. There are still many layers and sublayers that need to be handled:

Formalization Layer: One still needs to trust that the conversion from program source code to a logical formula was done faithfully. In real-world verification pipelines this is achieved by tools such as Boogie [5], CBMC [10], and Frama-C [13], among others. This is not simply a syntactic translation, but involves deeper verification condition generation: computing weakest preconditions, handling loops via invariants, and encoding memory models and heap reasoning into the formula. In order for the private program to be trusted, this transformation needs to be performed under the same Zero-Knowledge guarantees.

Deployment Layer: Even more importantly, once one is convinced that a particular piece of source code satisfies a given property, there remains a significant deployment gap: how can they be sure that the program actually being executed corresponds to the verified source code?

- **Compilation sublayer:** First, one needs to trust that the compiler preserves the semantics established at the source level, and moreover that this preservation can be attested to under zero-knowledge guarantees: one must be able to prove that a correct compilation took place without revealing the source. Certified compilation [30,32] addresses the first requirement but not the second.
- **Execution sublayer:** Second, even given a trustworthy and privacy-preserving compilation step, one needs assurance about the execution environment: specifically, that the binary being run is indeed the one produced from the verified source, and not a modified or substituted version. This second aspect requires some form of attestation from the execution environment, provided in practice either by hardware-based trusted execution environments (TEEs) such as Intel SGX [12] or AMD SEV [26], or by zkVM-based execution proofs [2,8].

Even assuming all three layers of the deployment pipeline are addressed, a more fundamental challenge remains. Consider a network equipment vendor that proves, using the privacy-preserving techniques described above, that their router firmware satisfies a given security property, compiles it under zero-knowledge guarantees to a binary, and deploys it with zkVM or TEE-based execution attestation. A customer receiving these proofs might reasonably ask how they can be certain that the program actually processing their network traffic is the one for which all these proofs were produced. Nothing in the proof chain prevents the vendor from running two separate instances: a verified one that produces attestations on a designated stream, and an unverified one that actually handles customer data. All proofs would remain valid, and the customer would have no cryptographic means of detecting the discrepancy. This is what we call the *binding problem*: the proofs establish correctness of an execution in the abstract, but they do not bind that execution to any specific interaction.

What is needed to address this challenge is a guarantee that this *particular execution*, processing this *particular input*, is the one for which the proof was produced. Establishing this requires orthogonal mechanisms from outside the proof-theoretic framework, such as network-level monitoring, hardware integrity guarantees, and auditability requirements.

6.2 Specifications

The type of users of such a system who can tolerate the higher costs of privacy-preserving verification is often more interested in security properties of the software under analysis. As a typical example, they may be interested in knowing that the output of the software will always be independent of its input, especially any secret inputs. Expressing such properties requires reasoning about more than one execution of the given program. There are many such properties,

often called hyperproperties or relational properties, which strictly subsume the standard non-relational ones. These properties often quantify over the inputs or executions of the program, with arbitrary quantifier complexity. The simplest, termed hypersafety properties, are those where all executions are universally quantified; therefore, refuting those properties requires only a finite collection of traces. The example above, *observational determinism*, is one such property: for any two executions π_1 and π_2 of the program, if the former results from feeding in secret input s_1 and public input p, and the latter results from feeding in secret input s_2 but the same public input p, then their public outputs, *i.e.*,what can be observed by a third party, are the same.

Existing tools geared towards non-relational properties are insufficient to support this richer formalism. Hypersafety properties can be reduced to non-relational ones via self-composition, where two or more copies of the program are stitched together to simulate tuples of executions. However, verifying such properties through this translation is often significantly harder in practice than reasoning about them natively, with an analysis designed with relational reasoning in mind. A natural next challenge then is to extend the privacy-preserving formula satisfiability and unsatisfiability approaches described above towards supporting such richer reasoning. Formalisms such as Relational Hoare Logic [7], HyperLTL [11], BiKAT [1], and Hyper Hoare Logic [14] are designed precisely for this purpose, offering native support for relational reasoning without the overhead of self-composition. The challenge is to lift the privacy-preserving verification machinery to operate within such formalisms, while retaining the efficiency and soundness guarantees established in the non-relational setting. With the ultimate goal of exploring Hyper Hoare Logic and BiKAT in a privacy-preserving setting, we have developed the protocol, briefly discussed at the end of Sect. 5, allowing one party to convince another that two (private) regular expressions are equivalent, in Zero Knowledge [29].

Verification of Machine Learning Systems: A further compelling direction concerns the formal verification of privacy guarantees for machine learning systems. The verification of neural network properties such as robustness and safety has seen substantial recent progress, with a growing ecosystem of tools including $\alpha\beta$-CROWN [44], NeuralSAT [16,17], and Marabou [28], and an annual competition (VNN-COMP) benchmarking their performance. Furthermore, a central tension is that ML models are most powerful when trained on large, representative datasets, yet such datasets frequently contain sensitive or proprietary records whose custodians are unwilling to release without assurances. The risk is not merely that the raw data is exposed during training: models can memorize their training data in subtle ways, making it possible for adversaries to recover sensitive information through inference attacks on the deployed model. What is needed is a way for model developers to give verifiable, cryptographic assurances that such leakage is bounded or impossible, without themselves having to reveal

the model architecture, weights, or training corpus. This connects naturally to the relational and hyperproperty verification challenges discussed above, since the relevant properties, such as various notions of non-leakage, are inherently statements about a model's behavior on different types of contents of its training set. Extending the privacy-preserving verification machinery of this paper to reason about such properties in a zero-knowledge setting, so that a model developer can convince a third party of a neural network property without revealing the model itself, is a significant open challenge.

Acknowledgments. The work presented in this paper provides an overview of results previously published in [27,29,33–35]. We acknowledge all co-authors of these works for their contributions and for the collaborations that significantly shaped the results reported here. The authors of this paper were partially supported by NSF awards CCF-2106845, CCF-2131476, CCF-2219995, CCF-2318974, and CNS-2245344 as well as by a grant from Google. We gratefully acknowledge this support.

References

1. Antonopoulos, T., Koskinen, E., Le, T.C., Nagasamudram, R., Naumann, D.A., Ngo, M.: An algebra of alignment for relational verification. Proc. ACM Program. Lang. **7**(POPL), 573–603 (2023)
2. Arun, A., Setty, S.T.V., Thaler, J.: Jolt: snarks for virtual machines via lookups. In: Joye, M., Leander, G. (eds.) Advances in Cryptology - EUROCRYPT 2024 - 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, 26–30 May 2024, Proceedings, Part VI. LNCS, pp. 3–33. Springer (2024)
3. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning (in 2 volumes), pages 19–99. Elsevier and MIT Press (2001)
4. Balabanov, V., Jiang, J.-H.R.: Resolution proofs and Skolem functions in QBF evaluation and applications. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 149–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_12
5. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17
6. Baum, C., Malozemoff, A.J., Rosen, M.B., Scholl, P.: Mac'n'cheese: zero-knowledge proofs for Boolean and arithmetic circuits with nested disjunctions, pp. 92–122 (2021)
7. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Jones, N.D., Leroy, X. (eds.) Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, 14–16 January 2004, pp. 14–25. ACM (2004)
8. Bruestle, J., Gafni, P., The RISC Zero Team: RISC zero zkVM: scalable, transparent arguments of RISC-V integrity. Technical report

9. Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. In: Proceedings 42nd IEEE Symposium on Foundations of Computer Science, pp. 136–145. IEEE (2001)
10. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_15
11. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Abadi, M., Kremer, S. (eds.) POST 2014. LNCS, vol. 8414, pp. 265–284. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54792-8_15
12. Costan, V., Devadas, S.: Intel SGX explained. IACR Cryptol. ePrint Arch. **2016**, 86 (2016)
13. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C - a software analysis perspective. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 233–247. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33826-7_16
14. Dardinier, T., Müller, P.: Hyper hoare logic: (dis-)proving program hyperproperties. Proc. ACM Program. Lang. **8**(PLDI), 1485–1509 (2024)
15. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (1962)
16. Duong, H., Nguyen, T., Dwyer, M.: A DPLL(T) framework for verifying deep neural networks (2024)
17. Duong, H., Xu, D., Nguyen, T., Dwyer, M.B.: Harnessing neuron stability to improve DNN verification. Proc. ACM Softw. Eng. **1**(FSE) (2024)
18. Franzese, N., Katz, J., Lu, S., Ostrovsky, R., Wang, Weng, C.: Constant-overhead zero-knowledge for ram programs. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pp. 178–191 (2021)
19. Franzese, N., Katz, J., Lu, S., Ostrovsky, R., Wang, X., Weng, C.: Constant-overhead zero-knowledge for RAM programs, pp. 178–191 (2021)
20. Goldreich, O., Micali, S., Wigderson, A.: Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. J. ACM **38**(3), 691–729 (1991)
21. Goldwasser, S., Kalai, Y.T., Rothblum, G.N.: Delegating computation: interactive proofs for muggles, pp. 113–122 (2008)
22. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof-systems (extended abstract), pp. 291–304 (1985)
23. Groth, J.: Short pairing-based non-interactive zero-knowledge arguments, pp. 321–340 (2010)
24. Ishai, Y., Kushilevitz, E., Ostrovsky, R., Sahai, A.: Zero-knowledge from secure multiparty computation, pp. 21–30 (2007)
25. Jawurek, M., Kerschbaum, F., Orlandi, C.: Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently, pp. 955–966 (2013)
26. Kaplan, D., Powell, J., Woller, T.: AMD SEV-SNP: strengthening vm isolation-with integrity protection and more. White Paper, Advanced Micro Devices Inc, 65 (2020)
27. Karthikeyan, A., Liu, H., Meel, K.S., Luo, N.: Towards practical zero-knowledge proof for PSPACE. arXiv preprint arXiv:2511.15071 (2025)
28. Katz, G., et al.: The Marabou framework for verification and analysis of deep neural networks. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 443–452. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_26

29. Kolesar, J.C., Ali, S., Antonopoulos, T., Piskac, R.: Coinductive proofs of regular expression equivalence in zero knowledge. Proc. ACM Program. Lang. **9**(OOPSLA2), 357–385 (2025)
30. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, San Diego, CA, USA, 20–21 January 2014, pp. 179–192. ACM (2014)
31. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
32. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: Gregory Morrisett, J., Jones, S.L.P. (eds.) Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, 11–13 January 2006, pp. 42–54. ACM (2006)
33. Luick, D., et al.: ZKSMT: a VM for proving SMT theorems in zero knowledge. In: 33rd USENIX Security Symposium (USENIX Security 2024), pp. 3837–3845 (2024)
34. Luo, N., Antonopoulos, T., Harris, W.R., Piskac, R., Tromer, E., Wang, X.: Proving unsat in zero knowledge. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, pp. 2203–2217 (2022)
35. Luo, N., Judson, S., Antonopoulos, T., Piskac, R., Wang, X.: {ppSAT}: towards {Two-Party} private {SAT} solving. In: 31st USENIX Security Symposium (USENIX Security 2022), pp. 2983–3000 (2022)
36. Niemetz, A., Preiner, M., Lonsing, F., Seidl, M., Biere, A.: Resolution-based certificate extraction for QBF. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 430–435. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_33
37. Pearson, L., Fitzgerald, J., Ardévol, H.M., Muñoz, M.B., Tapia, J.L.M.: Reconciling plonk with plookup, Plonkup (2022)
38. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM **12**(1), 23–41 (1965)
39. Schurr, H.-J., Fleury, M., Barbosa, H., Fontaine, P.: Alethe: towards a generic SMT proof format (extended abstract). In: Keller, C., Fleury, M. (eds.) Proceedings Seventh Workshop on Proof eXchange for Theorem Proving, PxTP 2021, Pittsburg, PA, USA, 11 July 2021, vol. 336. EPTCS, pp. 49–54 (2021)
40. Schwartz, J.T.: Fast probabilistic algorithms for verification of polynomial identities. J. ACM (JACM) **27**(4), 701–717 (1980)
41. Stump, A.: Proof checking technology for satisfiability modulo theories. In: Abel, A., Urban, C. (eds.) Proceedings of the International Workshop on Logical Frameworks and Metalanguages: Theory and Practice, LFMTP@LICS 2008, Pittsburgh, PA, USA, 23 June 2008. Electronic Notes in Theoretical Computer Science, vol. 228, pp. 121–133. Elsevier (2008)
42. Sutcliffe, G.: The TPTP problem library and associated infrastructure - from CNF to th0, TPTP v6.4.0. J. Autom. Reason. **59**(4), 483–502 (2017)
43. Turing, A.M.: On computable numbers, with an application to the Entscheidungsproblem. Proc. Lond. Math. Soc. **42**(1), 230–265 (1936). Published 1936–37

44. Wang, S., et al.: Beta-crown: efficient bound propagation with per-neuron split constraints for neural network robustness verification. In: Ranzato, M.A., Beygelzimer, A., Dauphin, Y.N., Liang, P., Vaughan, J.W. (eds.) Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, 6–14 December 2021, Virtual, pp. 29909–29921 (2021)
45. Weng, C., Yang, K., Katz, J., Wang, X.: Wolverine: fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits, pp. 1074–1091 (2021)
46. Wetzler, N., Heule, M.J.H., Hunt, W.A.: DRAT-trim: efficient checking and trimming using expressive clausal proofs. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 422–429. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09284-3_31