ZKSMT: A VM for Proving SMT Theorems in Zero Knowledge

Daniel Luick
Yale University
daniel.luick@yale.edu

John Kolesar
Yale University
john.kolesar@yale.edu

Timos Antonopoulos
Yale University
timos.antonopoulos@yale.edu

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

James Parker
Galois, Inc.
james@galois.com

Ruzica Piskac
Yale University
ruzica.piskac@yale.edu

Eran Tromer
Boston University
tromer@bu.edu

Xiao Wang
Northwestern University
wangxiao@northwestern.edu

Ning Luo
Northwestern University
ning.k.luo@gmail.com

Abstract

Verification of program safety is often reducible to proving the unsatisfiability (i.e., validity) of a formula in Satisfiability Modulo Theories (SMT): Boolean logic combined with theories that formalize arbitrary first-order fragments. Zero-knowledge (ZK) proofs allow SMT formulas to be validated without revealing the underlying formulas or their proofs to other parties, which is a crucial building block for proving the safety of proprietary programs. Recently, Luo et al. (CCS 2022) studied the simpler problem of proving the unsatisfiability of pure Boolean formulas but does not support proofs generated by SMT solvers. This work presents ZKSMT, a novel framework for proving the validity of SMT formulas in ZK. We design a virtual machine (VM) tailored to efficiently represent the verification process of SMT validity proofs in ZK. Our VM can support the vast majority of popular theories when proving program safety while being complete and sound. To demonstrate this, we instantiate the commonly used theories of equality and linear integer arithmetic in our VM with theory-specific optimizations for proving them in ZK. ZKSMT achieves high practicality even when running on realistic SMT formulas generated by Boogie, a common tool for software verification. It achieves a three-order-of-magnitude improvement compared to a baseline that executes the proof verification code in a general ZK system.

1 Introduction

Formal verification is the process of using mathematical reasoning to prove the correctness of programs. It has been used to verify large-scale real-world programs like compilers [Ler09], operating systems [KEH+09, GSC+16], and the Transport Layer Security (TLS) protocol [BBD+17]. To confirm that a program adheres to some property, both are translated into some mathematical formalism so that the problem of reasoning about programs is reduced to reasoning about mathematical objects.

Boolean algebra is the simplest formalism used for verification, but almost all formal verification tasks need something beyond pure Boolean algebra. Satisfiability Modulo Theories (SMT) is a well-explored formalism that extends the concept of Boolean satisfiability with theories such as equality with uninterpreted functions and linear integer arithmetic. Tools known as SMT solvers [ORS09, CHN12, BBB+22, EMT+17, HS22, DMB08] are among the most widely used verification tools. SMT solvers reason about SMT formulas automatically: they can generate both proofs for valid SMT formulas and counterexamples for invalid ones.

Standard techniques for program verification require all relevant information to be completely public: both the program and the proof must be available to everyone who wants to check whether the program is safe. In practice, the owner of the program and the verifier of the program are not always the same entity, and the two may not trust each other. If a program contains sensitive intellectual property, the owner of the program cannot demonstrate the program's safety without revealing the program itself. This limitation results in real-world situations where vendors are forced to reveal their software. For example, cryptographic modules must be FIPS 140-2 [14001] certified to be allowed for use in US government systems. The certification process requires that auditors inspect the cryptographic software and run a series of test vectors on the software. Instead of requiring vendors to share their proprietary software for certification, a better approach would enable vendors to prove compliance while keeping their intellectual property secret.

Zero-knowledge (ZK) proofs are a cryptographic primitive that could in theory make this approach a reality. ZK proofs enable a prover to demonstrate that they know a witness w that satisfies a public predicate P without revealing anything about the value of w. In this context, the witness would be the private program and its SMT validity proof, while the predicate encodes a program that verifies the validity of the SMT formula. To instantiate such a system, one could take an existing tool that can convert any C-like program (e.g., [BCG+13, HYDK21, CHP+23, GHAH+23]) and apply it to execute the program that verifies the validation of the SMT formula in ZK. However, we observe that such an approach does not scale at all even on toy examples. When using a state-of-the-art ZK toolchain [CHP+23] to run on a short benchmark with only 6 steps, the end-to-end running time is almost *two hours* (Sec. 7.3)! Since typical SMT proofs often need hundreds of steps, this approach is clearly impractical, due to a few reasons:

1. To prove arbitrary programs in ZK, all tools adopt the von Neumann architecture, providing an execution environment that resembles the cleartext computation. However, translating the SMT proof verification program to such a format (e.g., TinyRAM [BCG+13]) incurs a huge overhead, a necessary cost to achieve the highest expressiveness.

2. Each SMT theory has its own verification techniques, which in turn means different optimization opportunities in ZK. Using a generic tool essentially prohibits theory-specific optimizations.

3. Supporting random access memory (RAM) in ZK protocols is often the most costly component [BCTV14, WSR+15, HMR15, MRS17, HK20, Set20, FKL+21, DdSGOTV22]. Although SMT verification features unique access patterns in how and when it reads from RAM, they cannot be captured in a generic toolchain.

Essentially, we need a framework that allows modular support of new theories (like cleartext SMT solvers) and the flexibility to introduce customized protocols for different rules. This framework should be compact, general, and compatible with common ZK optimizations simultaneously. The first two require reasonable proof size and the ability to express reasoning on first-order theories. Achieving a level of usability for these two features in SMT in cleartext took decades [DMB08, ORS09, CHN12, EMT+17, BBB+22, HS22]. Introducing a layer of ZK to the SMT-proof system should ideally maintain the same level of compactness and generality while being efficient; this requires a ZK mindset from the outset. Finally, the whole framework should allow incremental development, meaning that the support of different SMT theories could be added over time.

Our Contributions In this paper, we introduce a new proof framework, ZKSMT, designed to support proving SMT theorems, along with an efficient instantiation in ZK. We make three main contributions:

- We introduce ZKSMT, a virtual machine (VM) for validating refutation proofs of SMT formulas based on our proof representation. ZKSMT includes a new encoding of SMT refutation proofs that can express any refutation proof involving arbitrary first-order theories. ZKSMT is designed to be efficient when being instantiated in ZK where the privacy of the proof and formula can be upheld.
- We instantiate three common theories in ZKSMT: Boolean logic, Equivalence of uninterpreted functions (EUF), and Linear integer arithmetic (LIA). These theories require non-trivial checking procedures in ZK where we propose optimized arithmetizations based on multiset interpretation and polynomial encodings.
- We implement ZKSMT in ZK and benchmark it over formulas that are generated by the Boogie verification toolchain [BCD+06] and the Wisconsin Safety Analyzer [WiS] benchmark suite (from the official SMT-LIB benchmark set [smt]). The results for Boogie show that ZKSMT achieves a speed-up of more than three orders of magnitude compared to a state-of-the-art system [CHP+23]. ZKSMT can also verify an "ultra-large" proof instance from the Wisconsin Safety Analyzer set with 200,000 proof steps in about 3 hours.

Information Leakage Our system does reveal some size parameters of the proof (e.g., the number of proof steps). We also made some privacy-efficiency trade-offs by revealing the number of occurrences (but not the order) of each proof rule. Together with other techniques, our trade-offs enable the impressive improvement mentioned above.

2 Preliminaries

2.1 Quantifier-Free First-Order Logic

Formula Structure Logical formulas are mathematical statements that assert a property of functions and predicates; the class of formulas that we consider in this work have the following structure. A set of function symbols is a set in which each element has an arity, denoted $|f|$ for $f \in \mathcal{F}$. The arity of a function may be any natural number, including 0. The set of terms over function symbols $\mathcal{F}$ and variables $\mathcal{V}$, denoted $T_{\mathcal{F},\mathcal{V}}$, is the smallest set containing $\mathcal{V}$ and $f(t_0, \ldots, t_{|f|-1})$ for all function symbols $f \in \mathcal{F}$ and terms $t_i \in T_{\mathcal{F},\mathcal{V}}$. For instance, the function symbols for linear integer arithmetic include all integer literals $\mathtt{n}$, with $|\mathtt{n}| = 0$, the negation operator $-$, with $|-| = 1$, and the addition operator $+$, with $|+| = 2$. An example of a term over these function symbols and the variable $\mathtt{x}$ is $-(\mathtt{x} + \mathsf{3})$.

Predicate symbols, similar to function symbols, are a set equipped with arities. The set of atoms over variables $\mathcal{V}$, function symbols $\mathcal{F}$, and predicate symbols $\mathcal{P}$ is the set of all $P(t_0, \ldots, t_{|P|})$ for predicate symbols $P \in \mathcal{P}$ and terms $t_i \in T_{F,V}$. The formulas over $\mathcal{F}$, $\mathcal{P}$, and $\mathcal{V}$ are all Boolean combinations of atoms over $\mathcal{F}$, $\mathcal{P}$, and $\mathcal{V}$, i.e. all objects built from atoms using the distinguished formulas True and False and the constructors negation, conjunction, disjunction, and implication. For example, linear integer arithmetic has the predicate symbols $=$, $\leq$, and $<$, with $|=| = |\leq| = |<| = 2$. A formula over these function and predicate symbols and the variable $\mathtt{x}$ is $\mathtt{x} = \mathsf{0} \vee \mathsf{10} \leq \mathtt{x} + \mathsf{2}$.

The definitions of terms and formulas can be described by the following BNF grammar for terms

t and formulas b:

$$
\begin{aligned}
t &::= v \mid f(t_0, ..., t_n) \\
b &::= \mathsf{True} \mid \mathsf{False} \mid P(t_0, ..., t_n) \mid \neg b_0 \mid \bigwedge\{b_0, ..., b_n\} \mid \\
&\quad \bigvee\{b_0, ...b_n\} \mid b_0 \rightarrow b_1
\end{aligned}
$$

Formula Validity and Proofs One approach for assigning meaning to function and predicate symbols is to specify which of the formulas defined over them are conclusions of a given set of assumed formulas. Evidence that a formula is a conclusion of some assumptions $\mathcal{A}$ is represented as a proof: a tree-shaped argument whose nodes are formulas, each derived from its children by a step of inference.

More precisely, a theory is a set of automatically recognizable proof steps, each of which consists of: **(1)** a symbol, referred to as the rule identifier, which has a finite arity; **(2)** a set of formulas known as the premises; and **(3)** a formula referred to as the conclusion. The proofs of a formula φ in theory $\mathcal{T}$ under assumed formulas $\mathcal{A}$ are the smallest set such that **(1)** each assumption $\varphi \in \mathcal{A}$ is a proof of itself; **(2)** if $P_0, \dots, P_n$ are proofs of formulas $\varphi_0, \dots, \varphi_n$, and R is a proof step with φ' as its conclusion and $\varphi_0, \dots, \varphi_n$ as its premises, then R and $P_0, \dots, P_n$ form a proof of φ'. If φ has a proof in $\mathcal{T}$ under $\mathcal{A}$, then φ is derived in $\mathcal{T}$ from $\mathcal{A}$. A refutation of formula φ in theory $\mathcal{T}$ is a proof of False in $\mathcal{T}$ under assumption φ. Multiple theories can be combined into a single theory by combining the programs that recognize applications of their proof rules. Thus, when convenient, we may consider either individual theories in isolation (to explain facts that they can derive) or a combination of multiple theories (when describing benchmarks that use many theories in combination).

Defining a formal theory for a previously unformalized domain of interest, and obtaining assurance that it proves exactly the formulas of interest, can be non-trivial. Our work is applicable in a setting where each theory of interest is accompanied by a public definition of the theory as a set of inference rules that the prover and verifier have agreed allows the derivation of only desired conclusions from assumptions. App. A summarizes classical methods that provide such assurance in a theory; the methods generalize the fundamental concept of a satisfying assignment from Boolean satisfiability.

2.2 SMT Theories of Interest

We now introduce illustrative examples of inference rules that define three logical theories of central importance: those of propositional logic, equality with uninterpreted functions, and linear arithmetic. Each of these theories is commonly used by program verifiers to verify critical properties of software, and each is supported by the current implementation of our protocol. Each inference rule is presented using a standard notation where the rule's premises occur above a horizontal bar and its conclusion occurs below.

2.2.1 Propositional Logic

Propositional logic rules—i.e., how formulas constructed from conjunction, disjunction, and negation can be proved and used to prove other formulas—include the following.

ExclMid The rule $\mathsf{ExclMid}$ formalizes the law of the excluded middle, stating that each proposition or its negation must hold:

$$\frac{}{a \vee \neg a}$$

Resolution The rule Res formalizes the idea of reasoning by case splitting. Intuitively, if both $p \lor A$ and $\neg p \lor B$ hold, then either A must hold (when $\neg p$ holds) or B must hold (when p holds):

$$\frac{p \lor A \quad \neg p \lor B}{A \lor B}$$

DeDup The de-duplication rule DeDup prunes duplicated disjuncts. A weak form of it (which can be applied n times to prune disjuncts that are repeated n times) is

$$\frac{a \lor a \lor B}{a \lor B}$$

Given that resolution alone is complete for proving refutations in propositional logic and there are existing protocols that verify resolution proofs in ZK [LAH+22], it may be surprising that we consider a large collection of rules instead of a minimal subset. However, practical SMT theorem provers [CHN12] often generate proofs that use many distinct rules, both to minimize their tool's output and to simplify their implementations. While such proofs could be rewritten to use a more restricted rule set, the consequences for both the size of the resulting proof and the performance of a subsequent ZK proof that verifies it are non-obvious and well beyond the scope of the this work.

2.2.2 Equality with Uninterpreted Functions

The theory of Equality with Uninterpreted Functions (EUF) enables SMT to describe general properties of system operations without explicitly defining their complete behavior, which can be helpful for modeling complex systems that consist of multiple modules. EUF contains three rules—Reflexivity, Symmetry, and Transitivity—that express the fact that equality is reflexive, symmetric, and transitive (i.e., that it is, unsurprisingly, an equivalence relation); their definitions are straightforward. It also contains an infinite family of rules, $\mathsf{Cong_n}$ for all $n \in \mathbb{N}$, which express that applying an n-ary function f to n equal arguments produces equal results:

$$\frac{a_0 = b_0 \quad \dots \quad a_{n-1} = b_{n-1}}{f(a_0, \dots, a_{n-1}) = f(b_0, \dots, b_{n-1})}$$

2.2.3 Linear Integer Arithmetic

Linear Integer Arithmetic (LIA) is a commonly used first-order theory of integers that includes addition and multiplication by constants but does not permit multiplication between variables. It is used to model the semantics of both bounded and unbounded arithmetic.

Multiplication Distribution The rule MulDist is the general law that multiplication distributes over addition, specialized to the case of constant left factors. It can be applied to conclude, e.g., that the equation $4 * (2x + 3y) = 8x + 12y$ is valid. Its general form is:

$$\frac{}{c * (\sum_{i=0}^{n} d_i * x_i) = \sum_{i=0}^{n} c * d_i * x_i}$$

where $x_0, \dots, x_n$ are arbitrary terms; $c, d_0, \dots, d_n$ are constants.

Farkas' Lemma Farkas' Lemma derives strict inequalities over the coefficients of a given strict inequality. It can be expressed as the following family of inference rules, indexed by a term size n:

$$\frac{\sum_{i=0}^{n} c_i * a_i - c_i * b_i > 0}{\bigvee_{i=0}^{n} a_i > b_i}$$

A similar rule can be applied to derive a slightly more constrained disjunction when a linear term of the identical form is given to be equal to 0.

2.3 An Example Formalizing Software Safety

We can use EUF and LIA to model safety properties for numerical type conversion in languages like C. Let `f` be a function on integers. We do not have access to the source code for `f`, but we know that it respects negation: the identity `f(-x)` = `-f(x)` holds for any integer `x`. Suppose that we have another function `f-cast` that uses `f` as a helper:

```
extern int f(int x); // f(-x) = -f(x)

short f-cast(int a) {
   int b = -a;
   if (f(b) < SHRT_MIN || SHRT_MAX < f(b)) error;
   else return short (f(b));
}
```

`f-cast` returns a short as its output rather than an int. It includes a safeguard to ensure that its output lies within the bounds of the short type. `SHRT_MIN` and `SHRT_MAX` are the lowest and highest possible values for a signed 16-bit integer: -32768 and 32767, respectively. The check that the output lies between `SHRT_MIN` and `SHRT_MAX` is critical for the safety of `f-cast`. If `f(b)` does not fit within the bounds of the short type, the result will be truncated and will have a different value than it would in the original type. C does not throw any exception when truncations occur, so unguarded down-casting can silently introduce security vulnerabilities into a program, making memory corruption attacks possible [BCJ+07]. `f-cast` eliminates the vulnerability by throwing an exception on its own.

We can prove that `f-cast` must throw an exception if `f(a)` is out of bounds using the following SMT formula:

$$\texttt{SHRT\_MAX} = 32767 \land \texttt{SHRT\_MIN} = -32768 \tag{1}$$

$$f(-a) = -f(a) \land b = -a \tag{2}$$

$$\land(\neg(f(b) < \texttt{SHRT\_MIN} \lor \texttt{SHRT\_MAX} < f(b)) \lor err) \tag{3}$$

$$\land\ (f(b) < \texttt{SHRT\_MIN} \lor \texttt{SHRT\_MAX} < f(b) \lor ret = f(b)) \tag{4}$$

$$\land\ \neg err\ \ \land\ \ f(a) < \texttt{SHRT\_MIN} \tag{5}$$

Line (1) represents the upper and lower bounds of the short type. Line (2) represents our knowledge of the behavior of `f` and also the definition of `b`. Lines (3) and (4) represent the semantics of the conditional inside the function, where `err` is a Boolean variable indicating whether an exception has been thrown, and `ret` is the function's return value. Line (5) represents our assumptions for the specific scenario being analyzed: `f(a)` is out of bounds, but no exception has been thrown. We can give this formula as an input to an external SMT solver that produces a refutation proof that ZKSMT can use. In Sec. 3.1, we will discuss the encoding that ZKSMT uses to represent the refutation proof for this formula.

2.4 Zero-Knowledge Proofs

A zero-knowledge proof [GMR85, GMW91] allows a prover to convince a verifier that it possesses an input w such that $P(w) = 1$ for some public predicate P, while revealing no additional information about w. There have been many lines of work in designing practically efficient ZK protocols under different settings and assumptions (e.g., [IKOS07, GKR08, Gro10, JKO13]). ZKSMT uses a special type of ZK protocol commonly referred to as "commit-and-prove" ZK [CLOS02], which allows a witness to be committed and later proven over multiple predicates while ensuring consistency of the committed values.

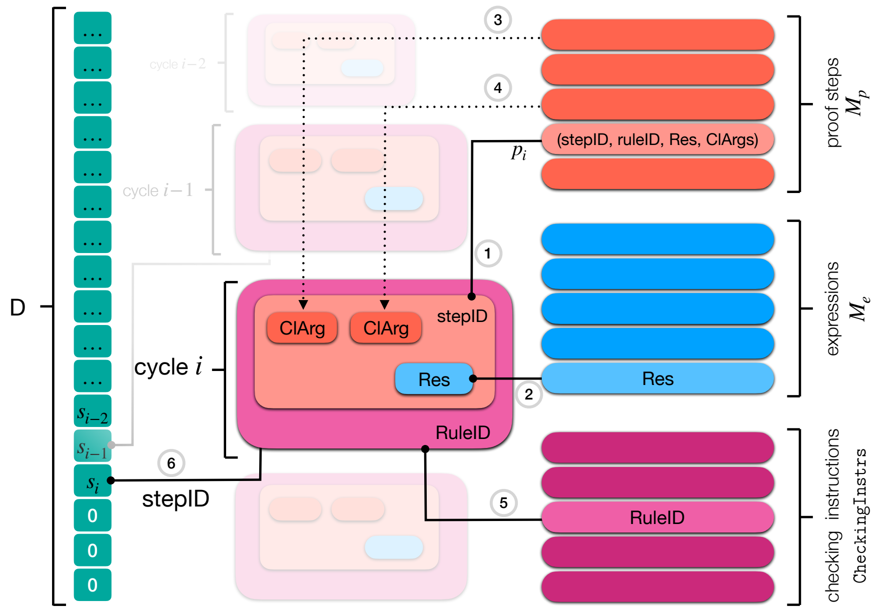


Figure 1: The retrieval and processing of information over one of ZKSMT's steps. Operations are numbered in the order of their occurrence. Data concerning rules applied and proof expressions to the right is used to check step validity, depicted in the middle. The result of the check is written to a storage cell in D, on the left.

Although ZKSMT can be instantiated with any commit-and-prove ZK, we use the recent VOLE-ZK series for maximum efficiency [WYKW21, BMRS21, DIO21] and, in particular, take advantage of optimizations for polynomials [YSWW21] and RAM operations [FKL+21]. We also use the permutation check originally proposed by [BEG+91].

Note that ZK proofs and refutation proofs are two different concepts, one in cryptography and one in formal methods. The verification procedure of a refutation proof is encoded as a statement proven by two parties using a ZK protocol.

3 ZKSMT Architecture

To verify an SMT refutation proof, ZKSMT examines the whole proof, step by step, in a loop: one such step is depicted in Fig. 1. In each iteration, ZKSMT **(1)** fetches the rule to be applied to the current step, **(2)** fetches the rule's premises, and **(3)** verifies that the derived formula is a valid conclusion of the proof rule. The overall structure resembles the design of a Von Neumann processor that executes only straight-line instructions (i.e., instructions that always transfer control to their successor). The available set of proof rules resembles a CPU's set of supported instructions. The proofs themselves are similar to programs composed of sequential instructions. In this analogy, the checking instruction responsible for each individual proof rule can be envisioned as analogous to a CPU's arithmetic-logic unit to handle specific computations.

Meanwhile, the main checker acts as the control unit, orchestrating the overall verification process. For each proof step, ZKSMT relies on a fixed-length array of formulas to store premises associated with the current step, functioning much like instruction operands. Furthermore, temporary storage is needed for a derived conclusion, a pointer to the next proof step, etc. The expression table, similar to the memory in CPU architecture, is read-only in this context. Our main philosophy is to develop a flexible VM that can efficiently encode and verify SMT refutation proofs when the underlying VM is instantiated using ZK protocols. This way, we can plug in any suitable ZK

Addr.	NodeID	ImmAddr	IndAddr	Meaning
&1	Var	SHRT_MIN	{}	SHRT_MIN
&2	Var	a	{}	a
&3	Apply	f	{&2}	$f(a)$
&4	Mul	-1	{&2}	$-1 * a$
&5	Var	b	{}	b
&6	Eq		{&5, &4}	$b = -a$
&7	Lt		{&3, &1}	$f(a) <$ SHRT_MIN
&8	Apply	f	{&5}	$f(b)$
&9	Apply	f	{&4}	$f(-a)$
&10	Mul	-1	{&3}	$-1 * f(a)$
&11	Eq		{&9, &10}	$f(-a) = -f(a)$
&12	Eq		{&8, &9}	$f(b) = f(-a)$
&13	Not		{&6}	$\neg(b = -a)$
&14	Eq		{&8, &10}	$f(b) = -f(a)$
&15	Or		{&12, &13}	$f(b) = f(-a)$ $\vee \neg(b = -a)$

Table 1: Part of the expression table M_e for the proof of the safety of `f-cast`. We use & to denote the addresses of expressions.

StepID	RuleID	Premises	Result
#1	Assume		&**11**: $f(-a) = -f(a)$
#2	Assume		&**6**: $b = -a$
#3	Assume		&**7**: $f(a) <$ SHRT_MIN
...			
#4	Cong		&**15**: $f(b) = f(-a) \vee \neg(b = -a)$
#5	Res	{#2, #4}	&**12**: $f(b) = f(-a)$
#6	Trans		$f(b) = -f(a)$ $\vee \neg(f(b) = f(-a))$ $\vee \neg(f(-a) = -f(a))$
#7	Res	{#1, #6}	$f(b) = -f(a)$ $\vee \neg(f(b) = f(-a))$
#8	Res	{#5, #7}	&**14**: $f(b) = -f(a)$
...			
#11	Res	{#9, #10}	$f(a) -$ SHRT_MIN $= 1$
#12	Farkas	{#11}	$\neg(f(a) <$ SHRT_MIN$)$
#13	Res	{#3, #12}	False

Table 2: Part of the array of proof steps M_p for the proof of the safety of `f-cast`. Not all conclusions' addresses are shown. We use # to denote the IDs of proof steps.

protocol for ZKSMT and bring in optimizations in CPU design. Below we introduce our VM's encoding for formulas and proofs and its execution strategy.

3.1 Encoding Formulas and Proofs

We first explain how ZKSMT represents SMT formulas and checks that particular formulas can be proved from others according to the rules of a logical theory. ZKSMT's encoding of SMT formulas, and the complex terms that they may contain, critically enables it to prove formulas in theories beyond what existing techniques can support.

Encoding Formulas in an Expression Table Every formula is constructed from an operator applied to a finite collection of sub-formulas; thus, it can be represented naturally as an AST. In particular, if we view formulas as being defined by the BNF grammar from Sec. 2.1, we can think of every individual production option used to produce a formula as a node in the formula's AST. The sub-productions are the node's children. Note that even semantically equivalent formulas can have distinct ASTs (e.g., False and $\neg\mathsf{True}$).

ZKSMT stores the ASTs for all formulas involved in the proof in a read-only table M_e, called the *expression table*. We refer to entries in the table as *expressions*. Each expression represents an individual node within the AST of a formula. A node of an AST has three fields: the Node ID (NodeID), the immediate addressing list (ImmAddr), and the indirect addressing list (IndAddr). NodeID specifies the operator being employed, such as Eq or Mul. ImmAddr is used to identify constants and immediate values, like an immediate value of an operand in a CPU. We also consider the variable names as immediate values. AST nodes with children store the indices of their children within the expression table under the IndAddr field. Most expressions, such as logical negation (Not) and equality (Eq), have a fixed number of children. Others, including Boolean conjunction (And), disjunction (Or), and applications of uninterpreted functions (Apply), can have a variable number of entries within IndAddr.

Note that not all nodes in the table are formulas: some entries simply represent sub-parts of other rows' formulas. A row that encodes a term or formula can have multiple other rows pointing to it; this happens when the term/formula appears in different formulas (which can even come from different theories). For example, in Table 1, a has only one entry even though it appears within $b = -a$, $f(a) < \texttt{SHRT\_MIN}$, and several other formulas.

Example 3.1. Table 1 shows a portion of the expression table for the proof in Table 2. The entry with address &7 in Table 1 represents the formula $f(a) < \texttt{SHRT\_MIN}$, whose NodeID is Lt (less than). The values indicated within the IndAddr field represent the indices for the sub-expression children of $f(a) < \texttt{SHRT\_MIN}$; specifically, the indices of $f(a)$ (entry &3) and $\texttt{SHRT\_MIN}$ (entry &1). The sub-expression $f(a)$ (entry &3) is a term rather than a formula and has one sub-expression child a (&2) and the label f as an immediate value stored in ImmAddr.

Encoding Proof Steps A proof step in a theory T consists of an application of a rule, labeled with an identifier with a fixed arity n to formulas $\varphi_0, \ldots, \varphi_n$ to conclude a formula φ (Sec. 2.1). The steps of a theory T of interest are checked in ZKSMT by a finite set of step *checking instructions*, each one checking steps identified by a corresponding rule of T. An occurrence p of a checking instruction has four fields:

- StepID: the position of the step in the execution order.
- RuleID: the identifier of the applied theory rule. Rule identifier r of theory T is identified as the pair (R, T).
- Premises: a list of the StepID's of $\varphi_0, \ldots, \varphi_n$. Each StepID points to a previous step, whose derived formula is a premise of p.
- Result: an index into the expression table to identify the conclusion φ of the current proof step.

A set of instructions $\texttt{T}$ is a *T-logical unit* if there is a bijection from rule identifiers of T to instructions in $\texttt{T}$ such that each instruction succeeds if and on if it is executed in a machine state in which it points encoding of premises and a conclusion that can be derived using its corresponding rule in T.

Size Parameters Five parameters bound the resources used by a ZKSMT instance. **(1)** π is the maximum number of proof steps. It parallels the concept of program size in CPU design and defines the extent of the proof structure that can be examined in a manner similar to how the size of a program in a CPU determines the number of instructions it can execute. **(2)** χ is the maximum number of expressions the proof can use, analogous to the size of the CPU's memory. **(3)** μ is the maximum number of premises in any rule, analogous to the number of registers in a CPU. **(4)** α is the maximum argument list size of any expression, where the argument list size of an expression e is defined as $|e.\mathsf{ImmAddr}|+|e.\mathsf{IndAddr}|$; it is analogous to the bit width of memory entries. **(5)** ρ is the number of distinct rules used in the proof, analogous to the size of the architecture's instruction set. The set of ZKSMT machines over checking instructions T on particular size parameters is denoted $\mathsf{ZKSMT}[\mathtt{T}](\pi, \chi, \mu, \alpha, \rho)$.

To define the necessary components of the machine, we often use the bit widths of these numbers: $\ell_p = \lceil \log(\pi) \rceil$, $\ell_e = \lceil \log(\chi) \rceil$, and $\ell_r = \lceil \log(\rho) \rceil$.

Example 3.2. Some of the entries of M_e and M_p for the refutation of the formula in Sec. 2.3 are shown in Table 1 and Table 2, respectively. The proof applies rules from EUF and LIA as well as rules for Boolean connectives. Most of the 1,038 steps in the proof are omitted, and some of the steps that we show are simplified. For example, we do not show the steps for adding and removing singleton disjunctions.

3.2 Machine Specification and Execution

Once the encodings are specified, we can build the VM on top of them. We show the overall architecture in Fig. 1.

Machine Specification ZKSMT has five main components:

- pc: the *proof counter*, an ℓ_p-bit integer.
- $\{r_i\}, \{t_i\}$: the list of *registers* that store information for the proof step currently being examined. The machine has $2\mu + 2$ registers in total: r_0 stores the conclusion, $r_1, \ldots, r_\mu$ store the premises, and r_{rule} stores the rule ID. The first $\mu + 1$ registers are of size ℓ_e, and r_{rule} is of size ℓ_r. The registers $\{t_1, \ldots, t_\mu\}$ store the addresses of $r_1, \ldots, r_\mu$. The main checker uses them when fetching the premises of a proof step from M_e. Each t_i is of size ℓ_e.
- M_e: the *expression table*, a read-only array of size χ that contains all expressions used in the proof, using the encoding system that we explained in Sec. 3.1.
- M_p: the *step table*, a read-only array of size π that contains all the proof steps used in the proof.
- D: the checking order of the proof. The checking order is the order in which proof steps are validated during the execution of the checker. If $\mathsf{D}[i] = j$, then the validation of the j^{th} proof step occurs on the i^{th} iteration of the main verification loop.

Machine Execution to Validate a Proof As mentioned above, ZKSMT's process of validating a proof closely resembles how a machine program is executed in the Von Neumann architecture (using a CPU, memory, etc.). To provide more flexibility in VM execution, we distinguish two orderings: the logical ordering and the checking ordering. The logical ordering is the original ordering of the proof as outlined in Sec. 2: a proof step should not use a result proven in a step that occurs later in the logical ordering. The StepID of each proof step is its logical ordering. However, the checking order, which is the order in which proof steps are validated during the execution of the VM, does not need to have any relationship with the logical ordering other than the former being a permutation of the latter. This could potentially provide huge opportunities in improving the performance when the VM is instantiated in ZK.

Algorithm 1 provides an overview of ZKSMT's algorithm, which iterates over the set of all proof steps (line 2). Each proof step is verified over five phases: proof step fetching, conclusion fetching, premise fetching, rule checking, and cycle checking. In the fetching phases (lines 3–10), ZKSMT fetches the relevant elements for that step from the tables M_p and M_e based on the values in the fields Result and Premises and stores them in $r_0, \ldots, r_\mu$. Next, the checker determines the checking instruction to execute by examining the value specified in RuleID (lines 11–12). It delegates the

Algorithm 1: $\mathsf{ZKSMT}[\mathtt{T}](\pi, \chi, \mu, \alpha, \rho)$'s execution

Output: True, False

1 $\mathsf{D} \leftarrow [0, \ldots, 0]$;
2 **for** $\mathsf{pc} = 0$ *to* $\pi - 1$ **do**
3   **Proof Step Fetch:**
4   $(\mathsf{StepID}, \mathsf{RuleID}, \mathsf{Res}, \mathsf{ClArgs}) \leftarrow M_p[\mathsf{pc}]$;
5   $r_{\mathsf{rule}} \leftarrow \mathsf{RuleID}$;
6   **Conclusion Fetch:**
7   $r_0 = M_e[\mathsf{Res}]$;
8   **Premise Fetch:**
9   $t_1, \cdots, t_\mu \leftarrow M_p[\mathsf{ClArgs}_0], \cdots, M_p[\mathsf{ClArgs}_{\mu-1}]$;
10   $r_1, \cdots, r_\mu \leftarrow M_e[t_1.\mathsf{Res}], \cdots, M_e[t_\mu.\mathsf{Res}]$;
11   **Rule Checking:**
12   $\mathtt{CheckingInstrs}[r_{\mathsf{rule}}](r_0, \{r_1, \cdots, r_\mu\})$;
13   **Cycle Checking:**
14   **for** $j = 1$ *to* μ **do**
15   **assert**$(t_j.\mathsf{StepID} < \mathsf{StepID})$;
16   $\mathsf{D}[i] \leftarrow \mathsf{StepID}$;
17 **assert**$(\mathtt{PermuteCheck}(\mathsf{D}, [0, \ldots, \pi - 1]))$;
18 $\mathsf{TypeCheck}(M_e)$;

responsibility of validating the proof step to the selected instruction and asserts the success of the validation (line 12). Formulas must be proven before being used as premises. Since StepID represents the logical ordering of a derived formula, we can confirm that in cycle checking that the StepID of every formula in Premises for a rule is strictly smaller than the StepID of the rule's conclusion. This is checked by iterating over all rule premises (lines 13–15). To conclude the iteration, the checker assigns StepID to $D[i]$ (line 16).

To address the potential discrepancies between orderings, we need to perform one more check. Every proof step needs to be verified at some point. The array D keeps track of which proof steps have been validated. When the main loop finishes execution, the main checker verifies that D is a permutation of the list $[0, \cdots, \pi - 1]$ (line 17). If it is, then every step in the refutation proof has been verified.

Well-Formed Expressions The soundness of ZKSMT also relies on the well-formedness of expressions in the table M_e. This can be ensured by a process analogous to proof validation. In particular, we type check each expression according to a set of type rules, which work similarly to proof rules and are provided as public configurations of ZKSMT. To forbid cyclic expressions, ZKSMT also checks for cycles in M_e, similarly to the check for cycles in proof steps.

3.3 Soundness and Completeness

The following are key properties of ZKSMT that establish that it produces exactly valid SMT formulas. Both theorems are defined over an arbitrary theory T and T-logical unit $\mathtt{T}$, formula φ, and size parameters $\pi, \chi, \mu, \alpha, \rho$ (Sec. 3.1)

In this context, we say that φ is *boundedly verifiable* if it has a derivation in $\mathcal{T}$ containing at most π steps, χ distinct expressions with at most α arguments, and using ρ rules which all have at most μ premises.

RuleID	Side Condition	Premises	Conclusion
Boolean			
Resolution	$\exists p. p \in \langle A\rangle$, $\neg p \in \langle B\rangle$, $\langle A\rangle \subseteq \langle C\rangle \uplus \langle p\rangle$, $\langle B\rangle \subseteq \langle C\rangle \uplus \langle \neg p\rangle$	$\bigvee A$, $\bigvee B$	$\bigvee C$
DeDup	$\forall a \in \langle A\rangle.\ a \in \langle B\rangle$	$\bigvee A$	$\bigvee B$
ExclMid			$\bigvee \{\neg a, a\}$
EUF			
Congruence	$\exists A, B, f.(f A = f B) \in C$, $\lvert A\rvert = \lvert B\rvert$, $\forall i \in \{0, \ldots, \lvert A\rvert - 1\}.\neg(A_i = B_i) \in C$		$\bigvee C$
LIA			
MulDist			$c * (\sum_{i=0}^{n} d_i * x_i) = \sum_{i=0}^{n} cd_i * x_i$
Flatten	$\exists \langle C\rangle, \langle D\rangle.\ \langle C\rangle \uplus \langle \sum D\rangle = \langle A\rangle$, $\langle C\rangle \uplus \langle D\rangle = \langle B\rangle$		$\sum A = \sum B$
Farkas	$\forall i \in \{0, \ldots, n\}.\ m_i \geq 0$ either $c > 0$, or $c = 0$ and $\exists j.\ \leq_j$ is $<$	$\sum_{i=0}^{n}(m_i * a_i) + (-m_i * b_i) = c$	$\bigvee_{i=0}^{n}\{\neg(a_i \leq_i b_i)\}$

Table 3: A selection of ZKSMT's rules for Boolean logic, EUF, and LIA that we cover in Sec. 4, Sec. 5, and Sec. 7. Tables 5 and 6 in the appendix show all of the proof rules omitted here.

Theorem 1 (Soundness)**.** *A VM in* ZKSMT$[\mathcal{T}](\pi, \chi, \mu, \alpha, \rho)$ *validates* φ *only if* φ *is boundedly verifiable.*

Theorem 2 (Completeness)**.** *If* φ *is boundedly verifiable, then some VM in* ZKSMT$[\mathcal{T}](\pi, \chi, \mu, \alpha, \rho)$ *validates it.*

App. C contains informal proofs of Thm. 1 and Thm. 2, specialized to the checking instructions `T` that we implemented to check the combination of the theories of propositional logic (Sec. 2.2.1), Equality with Uninterpreted Functions (EUF; Sec. 2.2.2) and Linear Integer Arithmetic (LIA; Sec. 2.2.3).

4 Instantiating ZKSMT on Practical Theories

In this section, we explain how to instantiate ZKSMT on propositional logic, equality with uninterpreted functions (EUF), and linear integer arithmetic (LIA). We discuss **(1)** the encoding of expressions in each theory, **(2)** the theories' proof rules, and **(3)** the implementations of the checking instructions for an illustrative selection of each theory's rules. Table 3 shows all of the rules that we cover in this section, along with a few others that we discuss later.

4.1 Checking Propositional Logic

We have implemented in ZKSMT an instruction unit that checks applications of the rules of propositional logic. We now describe implementations of checking instructions for selected example rules (Sec. 2.2.1).

ExclMid When the unit instruction that checks applications of ExclMid (the rule formalizing the law of the excluded middle) receives the conclusion expression r_0 from the main checker, it first confirms that r_0's NodeID is Or. Next, the checking instruction retrieves the first two entries a_0 and a_1 from list r_0.IndAddr and confirms that **(1)** the NodeID of a_0 is Not; and **(2)** the expression table index of a_0's child is the same as the expression table index of a_1. In general, the same technique is implemented by all checking instructions that must check that two expressions are identical: the instructions check the equality of indices in the expression table, instead of traversing the expressions' complete ASTs.

Many rules of propositional logic, as in the case of ExclMid, do not have premises. Instead of interacting with the results of previous steps, they introduce simple tautologies that other Boolean rules can use as premises later; the checking instructions for such rules need only to pattern match the rules' conclusions. However, in general, an instruction may need to validate non-trivial *side conditions* imposed by the a rule on the terms matched to its conclusion and premises (similar to the LFSC framework [ORS09]). One example of such a rule is Resolution, whose checking instruction we now describe.

Res The checking instruction for Res (the formalization of unit resolution) checks properties of the *multisets* of propositions that may be in each of its premise clauses. To describe the instruction's implementation, we employ the notation $\langle A \rangle$ (or $\langle a \rangle$) to represent the multiset containing the elements of a list A (or single element a).

The checking instruction interprets r_0.IndAddr—from the conclusion r_0—as a multiset $\langle C \rangle$ and interprets r_1.IndAddr and r_2.IndAddr—from the premises r_1 and r_2—as multisets $\langle A \rangle$ and $\langle B \rangle$, respectively. After checking that r_0, r_1, and r_2 are Or nodes, the instruction identifies the expression p, locates p within $\langle A \rangle$, and locates $\neg p$ within $\langle B \rangle$. Finally, the instruction checks the side conditions

$$\langle A \rangle \subseteq \langle C \rangle \uplus \langle p \rangle$$
$$\langle B \rangle \subseteq \langle C \rangle \uplus \langle \neg p \rangle$$

Note that p can be provided as an *extended witness* so that the checking instruction does not need to search for it.

In general, checking instructions for all propositional rules that have premises, as in the case of Res, must perform pattern matching on both the conclusion clause r_0 *and* the premise clauses $r_1, \ldots, r_k$ that they receive from the main checker.

Remark 4.1 (Extended witnesses)**.** In the context of zero-knowledge proofs, determining the value of p for the checking instruction for Resolution can be computationally expensive. To reduce the runtime cost, the proof itself can cache the value of p and provide it for the checking instruction directly. This value serves as an *extended witness*. When it receives an extended witness, the checking instruction only needs to test the side condition on the cached value of p rather than checking all possible options. Multiple other rules use extended witnesses for the same purpose.

DeDup It is straightforward to implement a checking instruction for applications of the deduplication rule DeDup as presented in Sec. 2.2.1: the instruction simply checks that its conclusion and premise are Or nodes, that the children of the conclusion's node occur in the premise, and that the children of the premise at corresponding positions are identical. However, checking DeDup strictly as presented would unfortunately require a proof to apply it multiple times to remove disjuncts that occur more than twice, and apply another rule formalizing the associativity of disjunction to arrange the premise in an expected form.

Instead, DeDup's actual checking instruction effectively checks repeated applications of such a rule in one step by checking that each distinct element in $\langle A \rangle$ is also in $\langle B \rangle$, where A is the argument list for the proof step's premise and B is the argument list for its conclusion.

4.2 Checking Equalities with Functions

We have instantiated ZKSMT as follows to refute proofs that use the theory of Equality with Uninterpreted Functions (EUF; Sec. 2.2.2). In particular, to check applications of a $\text{Congruence}_\text{i}$ rule, we model an alternative formulation, easily shown to be logically equivalent to the standard formalization, which derives a disjunctive clause from no premises. The ZKSMT checking instruction for Congruence begins by confirming that the NodeID of rule application r_0 is Or. Next, it retrieves the set of expressions indexed by r_0.IndAddr, identifies the pair of function applications, and verifies that the other disjuncts match the corresponding arguments of the function applications.

4.3 Checking Linear Integer Arithmetic

We now describe ZKSMT's representation of expressions from Linear Integer Arithmetic (LIA; Sec. 2.2.3). Then, we discuss implementations of checking instructions in ZKSMT for two LIA rules: MulDist and Farkas.

Expression Representation In ZKSMT's representation of LIA, addition is an n-ary operation, just like $\wedge$ and $\vee$. Singleton sums are allowed, and so are empty sums. The entries in a sum can be arbitrary integer-valued expressions, including other sums. Multiplication in LIA is shorthand for the repeated addition of an expression to itself. A multiplication node always has exactly one child, which can be an arbitrary integer-typed expression. It stores its scaling factor in the ImmAddr field, as we show in entries &4 and &10 in Table 1. The value of the scaling factor can be any integer, positive or negative. We store integer constants in M_e as multiples of a special variable `ONE` that represents 1. This representation enables checking instructions for rules such as MulDist to assume that the sums in their conclusions contain only Mul nodes rather than having a separate case for integer constants.

Multiplication Distribution MultDist's checking instruction can validate an application of MultDist by combining a bounded AST traversal and simple numerical computations with expression equality checks, implemented as checks for reference equality. Specifically, it first checks that the conclusion node r_0 is an Eq node whose children are **(1)** a Mul node with scaling factor denoted **(1.1)** and child denoted **(1.2)** and **(2)** an Add node. It then iterates over the children of nodes **(1.2)** and **(2)** in lockstep, checking that each child of node **(2)** is a Mul node with the same child as the corresponding Mul node in **(1.2)** and a scaling factor that is the product of **(1.1)** and the scaling factor of the same Mul node.

Farkas' Lemma Although Farkas' Lemma formalizes a somewhat subtle law of linear arithmetic, its application as a formal rule can be checked efficiently within ZKSMT's low-level design. The instruction checks that: **(1)** its conclusion operand is a node with operation Or whose children are negated inequalities; **(2)** its premise operand is a node with operation Eq whose children are a linear term matching the pattern given in the conclusion and a nonnegative constant; and **(3)** the sub-expressions of the linear term in the premise match the children of the inequalities in the disjuncts of the conclusion.

5 Zero-Knowledge Support

In this section we describe the technical details of ZKSMT's instantiation in ZK. Recall that the prover needs to demonstrate to the verifier that it knows a refutation proof of a formula without revealing the proof (or even the formula) to the verifier. We first explain how to commit ZKSMT's encoding of a refutation proof in Sec. 5.1. We discuss the details of how ZKSMT validates a committed refutation proof in ZK in Sec. 5.2. Finally, in Sec. 5.3, we explain the checking instruction protocols that have some non-trivial design component for the ZK setting, continuing our focus on the theories covered in Sec. 4.

5.1 Refutation Proof Commitment

Recall that a refutation proof consists of a set of clauses and a sequence of proof steps. Both the clauses and proof steps can be committed as fixed-length vectors of integers. In detail, for a k-bit integer, we commit each bit individually (i.e., $\mathbb{F}_2^k$) and they can be converted to an extension binary field element (i.e., $\mathbb{F}_{2^k}$) for free thanks to the structure of the VOLE commitment [FKL+21]. Let $\mathcal{I}$, $\mathcal{A}_{\text{imm}}$, and $\mathcal{A}_{\text{ind}}$ denote the set of all possible NodeID values, elements in ImmAddr, and elements in IndAddr, respectively. Given three injective functions $\epsilon_{\mathcal{I}} : \mathcal{I} \to \mathbb{N}$, $\epsilon_{\mathcal{A}_{\text{imm}}} : \mathcal{A}_{\text{imm}} \to \mathbb{N}_{>0}$, and

$\epsilon_{\mathcal{A}_{\text{ind}}} : \mathcal{A}_{\text{ind}} \to \mathbb{N}_{>0}$, an expression e specified by the tuple (NodeID, $\mathsf{ImmAddr}$, $\mathsf{IndAddr}$) can be mapped to the following vector of integers:

$$\{\epsilon_{\mathcal{I}}(\mathsf{NodeID})\} \| \{\epsilon_{\mathcal{A}_{\text{imm}}}(\mathsf{ImmAddr})\} \| \{\epsilon_{\mathcal{A}_{\text{ind}}}(\mathsf{IndAddr})\} \tag{6}$$

Here, $\epsilon_{\mathcal{A}_{\text{imm}}}$ and $\epsilon_{\mathcal{A}_{\text{ind}}}$ are applied element-wise on the two respective lists. Given concrete encoding schemes $\epsilon_{\mathcal{I}}$, $\epsilon_{\mathcal{A}_{\text{imm}}}$, and $\epsilon_{\mathcal{A}_{\text{imm}}}$, an expression can be committed by committing the vector in Eq. (6) element-wise. These encoding schemes are made known by both the prover and the verifier.

An expression's NodeID should be kept private. Different NodeIDs takes different numbers of operands. To avoid revealing an expression's NodeID from the size of its ImmAddr and IndAddr, we can pad both ImmAddr and IndAddr to the length α that is the upper bound of their size (Sec. 3.1).

Each proof step can be committed in a similar way. Recall that a proof step is encoded by four fields: StepID, RuleID, Result, and Premises which are either integers or lists of integers serving as pointers. The list Premises has its size bounded by μ. Hence, any proof step can be committed as a list of $\mu + 3$ integers.

5.2 Machine Execution in Zero Knowledge

We discuss how machine execution, i.e., the main checker, can be instantiated in ZK. Recall that the main checker performs three key operations:

1. **Fetching essential clauses and expressions.** To verify SMT proofs, we need to read entries from M_e and M_p using committed addresses. We can achieve this by instantiating M_e and M_p with any read-only memory (ROM) protocol [HK20, FKL+21, DdSGOTV22] in ZK that is compatible with the commitment scheme we use.

2. **Guaranteeing the proof is acyclic.** The proof can be regarded as a DAG with proof steps ordered by their logical order. Proving a graph is a DAG can be reduced to proving magnitude relationships between pairs of committed integers.

3. **Invoking the corresponding checking instructions.** To ensure the privacy of a proof, the proof rule employed by each proof step should be kept private. This can be achieved generically by multiplexing all checks, but that incurs a high cost and leads a to large overhead. Instead, ZKSMT uses *group checking*, as we will explain next.

Group Checking ZKSMT groups the verification of the proof steps with the same proof rule, where the real checking instruction is the *only* checking instruction that will be called. There is no multiplexing, and no other checking instructions are executed. For instance, all proof steps employing the Resolution rule are verified consecutively, and only the checking instruction of Resolution is invoked on them.

The RuleID of a proof step, which identifies the specific step being validated within a particular checking group, is private to the prover. The StepID of every step that has been verified so far appears in D. The array D is append-only, and at the end of each proof step, the step's committed StepID is appended to it. D can be implemented using a standard array containing commitments when ZKSMT is instantiated in ZK.

The soundness of ZKSMT relies on the permutation checking between D and $\{0, 1, \ldots, \pi - 1\}$ (see Algorithm 1, line 17). The permutation checking ensures that every proof step is validated in the end. When D contains committed values, permutation checking can be achieved efficiently using the Schwartz-Zippel lemma.

Remark 5.1 (Leakage and Optimization)**.** By group checking, we reveal the number of applications of each proof rule in the input proof. On the other hand, grouping checking for identical proof rules over different premises and conclusions offers a chance for optimization by using a ZK protocol optimized for batch proofs (i.e., single instruction multiple data (SIMD) optimizations), such as [WYY+22].

5.3 Checking Instructions in Zero Knowledge

Some checking instructions for Boolean, EUF, and LIA rules consist of only reading operations over the expression table and comparisons, such as ExclMid (Sec. 2.2.1). All necessary ZK operations are already needed by the main checker, and the same operations suffice for handling these simpler proof rules.

The instantiation of checking instructions becomes complex when the side condition of the proof rule involves traversing the IndAddr. In Sec. 4, we explain how these side conditions can be represented using the language of multisets. This level of abstraction further enables us to leverage the polynomial commitment scheme when instantiating these checking instructions in ZK. Next, we explain how to check two relations, subset and subset$_d$, between multisets using a polynomial commitment scheme. Following this, we will illustrate our implementations of the DeDup and Resolution checking instructions as examples.

Checking Multiset Relations To enable compact representation and efficient operations simultaneously, our protocol encodes multisets as polynomials over a finite field.

For the checking instructions we consider, we focus on two relations: subset and subset up to the number of occurrences (subset$_d$). The subset relation takes multiplicities into account. The multiset $\langle A \rangle$ is a subset of the multiset $\langle B \rangle$ if the multiplicities of all elements in $\langle A \rangle$ are less than or equal to their multiplicities in $\langle B \rangle$. On the other hand, $\langle A \rangle$ is a subset$_d$ of $\langle B \rangle$ if all distinct elements of $\langle A \rangle$ also appear in $\langle B \rangle$.

Checking the subset relation between two multisets is based on encoding multisets as univariate polynomials. Let Σ be a finite set, and $\mathbb{F}$ a finite field such that $|\mathbb{F}| > |\Sigma|$. Let $\langle \Sigma^* \rangle$ be the set of all possible multisets over Σ. Given an injective function $\psi : \Sigma \to \mathbb{F}$, we define an encoding $\gamma_\psi : \langle \Sigma^* \rangle \to \mathbb{F}[X]$ of a multiset as univariate polynomials over $\mathbb{F}$ such that for each multiset $\langle \ell \rangle$, the images under ψ of the Σ-elements ℓ_i in $\langle \ell \rangle$ are the roots of the image of $\langle \ell \rangle$ under γ_ψ:

$$\gamma_\psi(\langle \{\ell_0, \ldots, \ell_d\} \rangle) = (X - \psi(\ell_0)) \ldots (X - \psi(\ell_d))$$

To check the subset relation between two multisets $\langle \ell^{\text{sub}} \rangle$ and $\langle \ell^{\text{sup}} \rangle$, the prover commits their polynomial encodings, and the verifier checks that $\gamma_\psi(\langle \ell^{\text{sub}} \rangle)$ divides $\gamma_\psi(\langle \ell^{\text{sup}} \rangle)$ by attesting

$$\gamma_\psi(\langle \ell^{\text{sub}} \rangle) \cdot W = \gamma_\psi(\langle \ell^{\text{sup}} \rangle).$$

Here, W is a private polynomial committed by the prover as an extended witness. We use bivariate polynomials to verify the subset$_d$ relation between two multisets, leveraging the following observation in [GW20]. Let $\bar{\ell}^{\text{sub}}$, $\bar{\ell}^{\text{sup}}$ and $\bar{\ell}$ be permuted versions of ℓ^{sub}, ℓ^{sup} and $\ell = \ell^{\text{sub}} \uplus \ell^{\text{sup}}$ respectively with the d' being the size of $\bar{\ell}^{\text{sub}}$ and d being the size of $\bar{\ell}^{\text{sup}}$. Given the same ψ we use for subset checking, define the following two polynomials:

$$\begin{aligned}
\alpha_\psi(\langle \bar{\ell}^{\text{sub}} \rangle, \langle \bar{\ell}^{\text{sup}} \rangle) &:= (1+X)^{d'} \cdot \Pi_{i=0}^{d'-1}(Y + \psi(\bar{\ell}_i^{\text{sub}})) \\
&\cdot \Pi_{i=0}^{d-2}(Y \cdot (1+X) + \psi(\bar{\ell}_i^{\text{sup}}) + X \cdot \psi(\bar{\ell}_{i+1}^{\text{sup}})) \\
\beta_\psi(\langle \bar{\ell} \rangle) &:= \Pi_{i=0}^{d'+d-1}((1+X) \cdot Y + \psi(\bar{\ell}_i) + \psi(\bar{\ell}_{i+1}) \cdot X)
\end{aligned}$$

It is proved that $\alpha_\psi(\langle\bar{\ell}^{\mathrm{sub}}\rangle, \langle\bar{\ell}^{\mathrm{sup}}\rangle)(X,Y)$ equals $\beta_\psi(\langle\bar{\ell}\rangle)(X,Y)$ if and only if **(1)** $\langle\bar{\ell}^{\mathrm{sup}}\rangle$ is a subset$_d$ of $\langle\bar{\ell}^{\mathrm{sup}}\rangle$; and **(2)** $\bar{\ell}^{\mathrm{sub}}$, $\bar{\ell}^{\mathrm{sup}}$ and $\bar{\ell}$ are order-consistent[1]. A set of lists is order-consistent if values appear in the same order across all lists in the set. Putting it all together, to check if the subset$_d$ relation between $\langle\ell^{\mathrm{sup}}\rangle$ and $\langle\ell^{\mathrm{sub}}\rangle$ holds, the verifier attests the following relation between polynomials:

$$\begin{aligned}
\alpha_\psi(\langle\bar{\ell}^{\mathrm{sub}}\rangle, \langle\bar{\ell}^{\mathrm{sup}}\rangle) &= \beta_\psi(\langle\bar{\ell}\rangle)\\
\gamma_\psi(\langle\bar{\ell}^{\mathrm{sub}}\rangle) &= \gamma_\psi(\langle\ell^{\mathrm{sub}})\rangle)\\
\gamma_\psi(\langle\bar{\ell}^{\mathrm{sup}}\rangle) &= \gamma_\psi(\langle\ell^{\mathrm{sup}})\rangle)\\
\gamma_\psi(\langle\bar{\ell}\rangle) &= \gamma_\psi(\langle\ell^{\mathrm{sup}})\rangle) \cdot \gamma_\psi(\langle\ell^{\mathrm{sub}})\rangle)
\end{aligned}$$

Here, the prover computes and commits $\bar{\ell}$, $\bar{\ell}^{\mathrm{sub}}$ and $\bar{\ell}^{\mathrm{sup}}$ using some proper order over Σ.

Resolution Recall that the side condition of Resolution on premise clauses $\bigvee A$, $\bigvee B$ and conclusion clause $\bigvee C$ is the following:

$$\langle A\rangle \subseteq \langle C\rangle \uplus \langle p\rangle, \langle B\rangle \subseteq \langle C\rangle \uplus \langle \neg p\rangle$$

Here, A, B, and C are lists of addresses of the expression table and p is an address. Given that the size of the expression table is bounded by χ, we can restrict the co-domain of $\epsilon_{\mathcal{I}}$ to $\mathbb{N}_{\leq\chi}$, i.e., $\epsilon_{\mathcal{I}} : \mathcal{I} \to \mathbb{N}_{\leq\chi}$. We further fix an injective function $\psi_{\mathcal{I}} : \mathbb{N}_{\leq\chi} \to \mathbb{F}$ for a given proof. Then the checking instruction of the resolution rule can be implemented by verifying the subset relation between multisets $\epsilon_{\mathcal{I}}(\langle A\rangle)$, $\epsilon_{\mathcal{I}}(\langle C\rangle \uplus \langle p\rangle)$[2] and between $\epsilon_{\mathcal{I}}(B)$ and $\epsilon_{\mathcal{I}}(\langle C\rangle \uplus \langle\neg p\rangle)$ using the approach mentioned above, with ψ is concretized by $\psi_{\mathcal{I}}$. By applying $\epsilon_{\mathcal{I}}$ to the multisets, we mean element-wise application.

DeDup The side condition of DeDup asserts that for all $a \in \langle A\rangle$ it holds that $a \in \langle B\rangle$ given the premise clause $\bigvee A$ and the conclusion clause $\bigvee B$. This side condition can be validated by checking if $\langle A\rangle$ is a subset$_d$ of $\langle B\rangle$. Using the same encoding scheme as is used for the Resolution rule, we can implement the checking instruction of the DeDup rule by checking the subset$_d$ relation between $\epsilon_{\mathcal{I}}(\langle A\rangle)$ and $\epsilon_{\mathcal{I}}(\langle B\rangle)$. This relation checking can be achieved using the protocol we explain at the beginning of this section.

6 Implementation

We implement our protocol using the EMP-toolkit [WMK16] for ZKP operations (circuits, polynomials, read-only memory access). We instantiated the arithmetic field as the extension field $\mathbb{F}_{2^{128}}$, under which field operations (and their ZK counterparts) can be efficiently implemented. The indices of proof steps are 32-bit integers, which support refutation proofs with more than one billion steps. In addition, for performance optimization, we use an array M_a known as the *expression list table* to store argument lists for expressions that can take variable numbers of children. Expressions that take a fixed number of children (which is always 1 or 2 for the theories that we cover) store pointers to their children directly in M_e, but nodes that take variable numbers of children store a pointer to an entry in M_a that contains pointers to the expression's children. It allows us to keep the individual entries of M_e small and to avoid the cost of scanning a variable-length argument list for nodes like Not and Eq. We use η to denote the number of lists in M_a. This is not to be confused with α, which is the maximum length of an individual list within M_a.

[1] See Claim 3.1 [GW20] and its proof.

[2] When verifying equivalences that involve the combination of multisets through union operations, we compute the product of the two corresponding polynomials.

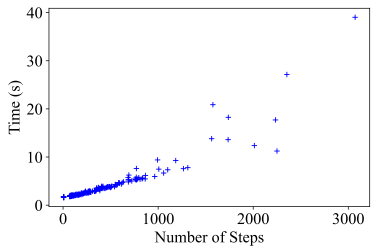


Figure 2: Step counts and time costs for validating SMT statements from the Boogie test suite in ZKSMT.

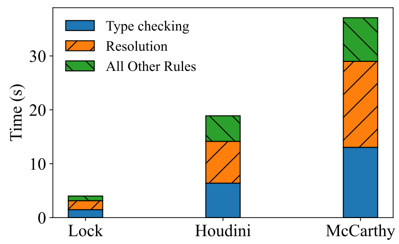


Figure 3: Time cost decomposition for Lock, Houdini, and McCarthy benchmarks.

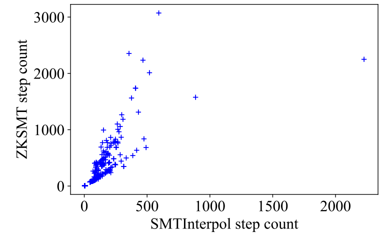


Figure 4: Relative step counts of ZKSMT's proofs and the original SMTInterpol proofs.

7 Evaluation

We evaluate ZKSMT to compare our protocol with the prior state of the art. We intend to answer three key questions with our experiments:

Q1 Does our protocol efficiently validate SMT formulas that formalize the safety and security of practical software?

Q2 Does our protocol scale well in response to increases in proof size?

Q3 Is ZKSMT more efficient than a zkVM running a commodity SMT proof validator?

The results of our experiments allow us to report an affirmative answer for all three questions. For all benchmarks, we ran ZKSMT on AWS instances of type `r5b.4xlarge` with 128 GB of memory, 16 vCPUs, and a 10 Gbps network connection between the prover and verifier. However, the underlying ZK protocols that we use only consume about 100 Mbps bandwidth. We also configured ZKSMT to use 8 threads. Our methodology and results for Q1, Q2, and Q3 appear in Sec. 7.1, Sec. 7.2, and Sec. 7.3, respectively.

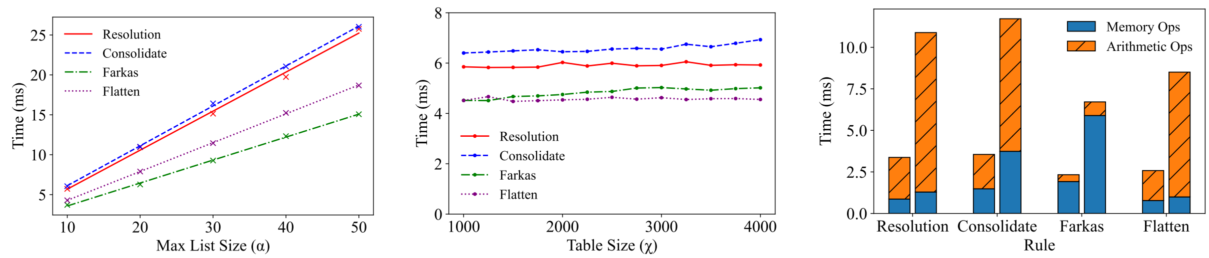
7.1 Verifying Practical Software

To answer Q1, we collected a set of SMT formulas whose validity formalizes program correctness. Specifically, the SMT formulas were generated by the Boogie verification toolchain [BCD+06]. The Boogie toolchain contains an intermediate language for expressing low-level programs annotated with function requirements and guarantees, along with compilation passes to an intermediate language from high-level languages including C, Spec#, and Dafny [Lei10]. Boogie generates verification conditions from the annotated intermediate programs in SMT-LIB 2.0 format, which can be validated by SMT solvers like Z3 [DMB08]. We ran Boogie on its test suite to collect the corresponding SMT formulas, and we validated the SMT formulas using the solver SMTInterpol [CHN12, HS22] to generate proof certificates that ZKSMT can process.

Fig. 2 shows the runtime of ZKSMT versus the number of proof steps used in each of the SMT statements in the Boogie test suite. ZKSMT is able to verify most of the test suite SMT statements in ZK within a few seconds, but the largest benchmark takes 39 seconds. We also observe a general linear trend between the running time and the number of steps, which is expected. The fluctuation is due to the use of different rules in each instance.

7.2 Scalability

To determine how our protocol scales in response to increases in proof size (Q2), we microbenchmark various aspects of ZKSMT while varying the size of input SMT statements.



(a) (b) (c)

Figure 5: Scalability and rule breakdown of our protocol. Fig. 5a and Fig. 5b contain the running times of a single proof step across different rules for changing values of expression table size χ. Fig. 5c shows the time cost decomposition across the four rules with max list size $\alpha = 5, 21$.

Proof Breakdown To assess the relative time consumption of different parts of our protocol, we run three of our Boogie benchmarks and separate the timing results into three phases: type checking, Resolution, and all other proof rules. We place Resolution in a phase of its own because, for each of the examples, it takes more time than checking all of the other rules combined. Fig. 3 shows the performance decomposition for the three benchmarks. All of them are related to program safety verification: Lock is a Boogie benchmark for verification of a lock; Houdini is a benchmark on modular contract checking [LV11]; and McCarthy is an adaptation of a standard benchmark for verification of recursive functions [MM69].

We observe that type checking can be as time-consuming as the main checking loop itself. This is due to the fact that ZKSMT needs to fetch every entry of every list in M_a at least once to confirm that its type fits with the list's type.

Max List Size To understand how α, the maximum list size, affects the running time of our rules, we benchmark the running time of our individual proof rules in isolation. To find the amortized cost of each rule, we run the rule 1,000 times and average the result. Most of our rules are simple, so we present the results for only our four most performance-intensive rules: Resolution, Consolidate, Farkas, and Flatten.

The results of varying α with values ranging from 10 to 50 are presented in Fig. 5a. We ran a linear regression and determined that all four rules scale linearly, with R^2 values above 0.99. This occurs because all four rules contain loops or procedures which iterate $O(\alpha)$ times.

Many of ZKSMT's rules are affected by the size of the longest list in the proof because all argument lists are padded to be the same size. The worst-case scenario for ZKSMT would be a proof that operates mainly on short lists but contains one extra-long list that forces all list-traversing rules to perform a large number of iterations. Fortunately, our benchmarks demonstrate that this degenerate case does not appear in practice. In the future we plan to mitigate the effect of α on a proof's overall running time by breaking down list-based rules into smaller pieces, which will improve runtime even more by eliminating the impact of large maximum list sizes.

Table Size Next, we consider how χ, the size of M_e, affects the running time of our four main rules. For each trial, we ran 10,000 instances of a rule with an α value of 10, which was a common value among our benchmarks, an η value of 1,000, and a π value of 10 while varying the value of χ to between 1,000 and 4,000. The results are plotted in Fig. 5b. Unlike our results for α, the running time does not change appreciably. This is because the main operation in these rules that is affected by a change in table size is the cost of accessing an element from ROM, for which the amortized access time does not depend on the number of elements. For similar reasons, the value of η does not significantly change the running time.

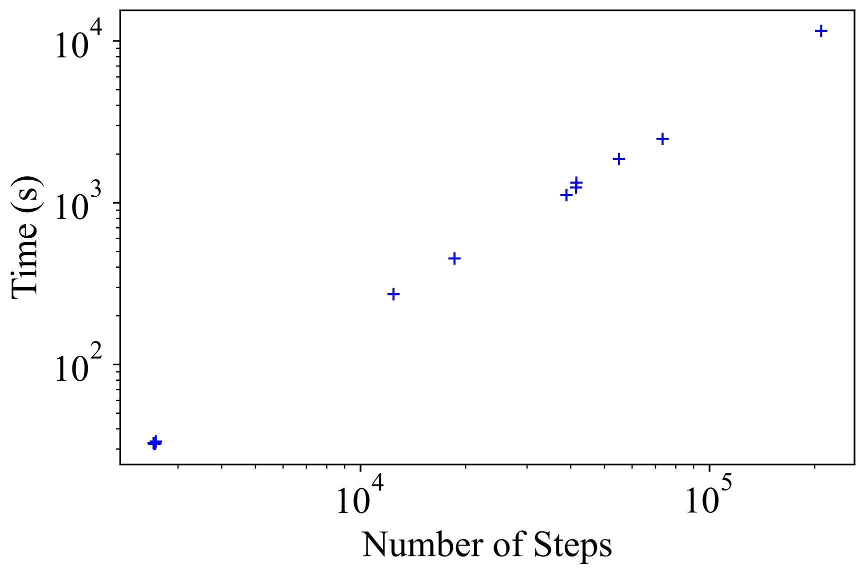


Figure 6: Step counts and time costs for validating SMT statements from the Wisconsin Safety Analyzer, plotted on a logarithmic scale.

Rule Breakdown We also consider which operations make up the running time of our four main rules. With α values of 5 and 21, we divide the running times for each rule into the time taken for memory operations (retrieving entries from M_e or M_a) and the time taken for arithmetic operations (everything else). 5 is a small but still realistic value for α, and 21 is the highest value of α that appears in our Boogie benchmarks. The results appear in Fig. 5c. Arithmetic operations dominate the running time for Resolution, Consolidate, and Flatten, but memory operations dominate the running time for Farkas. This makes sense because, unlike the other three rules, Farkas does not perform any multiset equivalence or containment checks. Multiset checks can work directly with expressions' addresses, but Farkas needs to fetch every entry in its premise and conclusion to pattern-match their NodeIDs and arguments.

Original Proof Size Our work uses a compiler to convert the output of SMTInterpol to the format accepted by ZKSMT. To enable evaluation in zero knowledge, some rules in SMTInterpol, particularly the LIA rules, must be broken down into simpler rules. This increases the proof size. A comparison between the number of proof steps in SMTInterpol and ZKSMT is given in Fig. 4 for the Boogie test suite. The proof size increases by a factor from 1 to 7, which is not problematic because ZKSMT is still vastly more efficient than the generic zkVM solution.

Stress Test To stress test ZKSMT, we ran it against a series of larger tests from the Wisconsin Safety Analyzer [WiS] benchmark suite found in the official SMT-LIB benchmarks repository [smt]. The resulting running times are plotted in Fig. 6. The largest test which passed uses 200K steps, 380K expressions, and a maximum list size α of 97. This verified in about 3 hours, requiring more than 22.9 billion $\mathbb{F}_2$ multiplications and 336 million $\mathbb{F}_{2^{128}}$ multiplications. Larger tests ran out of memory. This demonstrates that ZKSMT can scale up to proofs of a larger size, and gives insight into ZKSMT's current limitations.

7.3 Comparison with Alternative Protocols

Instead of developing a custom ZK protocol to validate SMT formulas, a simpler approach would be to take a commodity SMT proof validator and compile it to a ZK statement using a ZK virtual

Tool	MicroRAM cycles	$\mathbb{F}_2$ muls	$\mathbb{F}_{2^{128}}$ muls	Time
Baseline	183K	14B	421K	1h 51m
ZKSMT	—	108K	770	2s
Improvement	—	129,629×	546×	3,330×

Table 4: Comparison of ZKSMT's performance against the baseline of Cheesecloth and Diet Mac'n'cheese on the shortest Boogie benchmark.

machine (zkVM). We benchmark the performance of ZKSMT against such a zkVM to determine whether the benefits of a custom ZK protocol are worth the effort (Q3).

In our evaluation, we used Cheesecloth [CHP+23] and Diet Mac'n'cheese [Gal19, BMRS21] as the baseline zkVM. Cheesecloth is a general-purpose tool for generating zero-knowledge proof statements that verify the execution of LLVM programs. Diet Mac'n'cheese is an interactive VOLE based zero-knowledge proof backend, capable of verifying ZK statements. We developed a clear-text C++ version of ZKSMT that verifies SMT statements, and we used Cheesecloth and Diet Mac'n'cheese to verify the shortest Boogie benchmark (with only 6 steps) in ZK. The results in Table 4 demonstrate that ZKSMT is significantly faster than the baseline, taking seconds instead of hours to verify. With a 3,330× improvement in runtime, it is clear that ZKSMT provides a significant improvement over the zkVM approach in enabling SMT validation for program verification in ZK.

8 Conclusion

This paper introduces ZKSMT, an efficient protocol for validating SMT formulas in ZK. This work sets up exciting future work in multiple directions. First, protocols can be developed for other theories that model practical verification problems but are not currently supported, including the theory of arrays and the theory of bit-vectors [GD07]. Arrays and bit-vectors are commonly used by symbolic execution engines that execute low-level code [CDE+08]. Second, the core logic itself can be extended to validate formulas that contain universal and existential quantifiers. Prominent program verification toolchains often produce quantified formulas as output [BCD+06, Lei10].

Acknowledgments

The authors would like to thank Chantal Keller for early discussions on SMTCoq, Stuart Pernsteiner for helping run Cheesecloth, and Tanja Schindler and Jochen Hoenicke for helping with SMTInterpol. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001120C0085 and HR001120C0087. Timos Antonopoulos was partially supported by the NSF awards CCF-2131476, CCF-2106845, CCF-2318974 and CCF-2219995. Ruzica Piskac and John Kolesar were supported by CCF-2131476 and CCF-2219995. Work of Xiao Wang is also supported by NSF awards #2236819, #2318974, and research awards from Meta and Google. Work by Daniel Luick is supported in part by NSF awards #1763399 and #2019285. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA). Approved for Public Release, Distribution Unlimited.

References

[14001] FIPS PUB 140-2. Security requirements for cryptographic modules. *National Institute of Standards and Technology*, 2001.

[BBB+22] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. cvc5: A versatile and industrial-strength smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442. Springer, 2022.

[BBD+17] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, K. Rustan M. Leino, Jay R. Lorch, Kenji Maillard, Jianyang Pan, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella Béguelin, and Jean Karim Zinzindohoue. Everest: Towards a verified, drop-in replacement of HTTPS. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*, volume 71 of *LIPIcs*, pages 1:1–1:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[BCD+06] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures 4*, pages 364–387. Springer, 2006.

[BCG+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 90–108. Springer, Heidelberg, August 2013.

[BCJ+07] David Brumley, Tzi-cker Chiueh, Robert Johnson, Huijia Lin, and Dawn Song. Rich: Automatically protecting against integer-based vulnerabilities. In *Proceedings of the 14th Network and Distributed System Security Symposium (NDSS)*. Carnegie Mellon University, 2007.

[BCTV14] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security 2014*, pages 781–796. USENIX Association, August 2014.

[BEG+91] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. In *32nd FOCS*, pages 90–99. IEEE Computer Society Press, October 1991.

[BMRS21] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 92–122, Virtual Event, August 2021. Springer, Heidelberg.

[CDE+08] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[CHN12] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. Smtinterpol: An interpolating smt solver. In *International SPIN Workshop on Model Checking of Software*, pages 248–254. Springer, 2012.

[CHP+23] Santiago Cuéllar, Bill Harris, James Parker, Stuart Pernsteiner, and Eran Tromer. Cheesecloth: Zero-Knowledge proofs of real world vulnerabilities. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6525–6540, Anaheim, CA, August 2023. USENIX Association.

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *34th ACM STOC*, pages 494–503. ACM Press, May 2002.

[DdSGOTV22] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, Titouan Tanguy, and Michiel Verbauwhede. Efficient proof of ram programs from any public-coin zero-knowledge system. In *International Conference on Security and Cryptography for Networks*, pages 615–638. Springer, 2022.

[DIO21] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. Line-Point Zero Knowledge and Its Applications. In *2nd Conference on Information-Theoretic Cryptography (ITC 2021)*, Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[EMT+17] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark Barrett. Smtcoq: A plug-in for integrating smt solvers into coq. In *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II 30*, pages 126–133. Springer, 2017.

[FKL+21] Nicholas Franzese, Jonathan Katz, Steve Lu, Rafail Ostrovsky, Xiao Wang, and Chenkai Weng. Constant-overhead zero-knowledge for RAM programs. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 178–191. ACM Press, November 2021.

[Gal19] Galois, Inc. swanky: A suite of rust libraries for secure computation. https://github.com/GaloisInc/swanky, 2019.

[GD07] Vijay Ganesh and David L Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification: 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007. Proceedings 19*, pages 519–531. Springer, 2007.

[GHAH+23] Matthew Green, Mathias Hall-Andersen, Eric Hennenfent, Gabriel Kaptchuk, Benjamin Perez, and Gijs Van Laer. Efficient proofs of software exploitability for real-world processors. *Privacy Enhancing Technologies Symposium*, 2023.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In Richard E. Ladner and Cynthia Dwork, editors, *40th ACM STOC*, pages 113–122. ACM Press, May 2008.

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *17th ACM STOC*, pages 291–304. ACM Press, May 1985.

[GMW91] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *Journal of the ACM*, 38(3):691–729, 1991.

[Gro10] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 321–340. Springer, Heidelberg, December 2010.

[GSC+16] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent os kernels. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 653–669. USENIX Association, 2016.

[GW20] Ariel Gabizon and Zachary J Williamson. plookup: A simplified polynomial protocol for lookup tables. *Cryptology ePrint Archive*, 2020.

[HK20] David Heath and Vladimir Kolesnikov. A 2.1 KHz zero-knowledge processor with BubbleRAM. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 2055–2074. ACM Press, November 2020.

[HMR15] Zhangxiang Hu, Payman Mohassel, and Mike Rosulek. Efficient zero-knowledge proofs of non-algebraic statements with sublinear amortized cost. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 150–169. Springer, Heidelberg, August 2015.

[HS22] Jochen Hoenicke and Tanja Schindler. A simple proof format for smt. In *International Workshop on Satisfiability Modulo Theories (SMT)*, volume 3185, pages 54–70, 2022.

[HYDK21] David Heath, Yibin Yang, David Devecsery, and Vladimir Kolesnikov. Zero knowledge for everything and everyone: Fast ZK processor with cached ORAM for ANSI C programs. In *2021 IEEE Symposium on Security and Privacy*, pages 1538–1556. IEEE Computer Society Press, May 2021.

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *39th ACM STOC*, pages 21–30. ACM Press, June 2007.

[JKO13] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 955–966. ACM Press, November 2013.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an os kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 207–220. ACM, 2009.

[LAH+22] Ning Luo, Timos Antonopoulos, William R. Harris, Ruzica Piskac, Eran Tromer, and Xiao Wang. Proving UNSAT in zero knowledge. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 2203–2217. ACM Press, November 2022.

[Lei10] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*, pages 348–370. Springer, 2010.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, jul 2009.

[LV11] Shuvendu K Lahiri and Julien Vanegue. Explainhoudini: making houdini inference transparent. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 309–323. Springer, 2011.

[MM69] Zohar Manna and John McCarthy. *Properties of programs and partial function logic*. Stanford University, 1969.

[MRS17] Payman Mohassel, Mike Rosulek, and Alessandra Scafuro. Sublinear zero-knowledge arguments for RAM programs. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 501–531. Springer, Heidelberg, April / May 2017.

[ORS09] Duckki Oe, Andrew Reynolds, and Aaron Stump. Fast and flexible proof checking for smt. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, pages 6–13, 2009.

[Set20] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 704–737. Springer, Heidelberg, August 2020.

[smt] SMT-LIB: The Satisfiability Modulo Theories Library - Benchmarks. http://smtlib.cs.uiowa.edu/benchmarks.shtml.

[WiS] WiSA: Wisconsin Safety Analyzer. https://research.cs.wisc.edu/wisa/.

[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. Emp-toolkit: Efficient multiparty computation toolkit. https://github.com/emp-toolkit, 2016.

[WSR+15] Riad S. Wahby, Srinath T. V. Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS 2015*. The Internet Society, February 2015.

[WYKW21] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *2021 IEEE Symposium on Security and Privacy*, pages 1074–1091. IEEE Computer Society Press, May 2021.

[WYY+22] Chenkai Weng, Kang Yang, Zhaomin Yang, Xiang Xie, and Xiao Wang. AntMan: Interactive zero-knowledge proofs with sublinear communication. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 2901–2914. ACM Press, November 2022.

[YSWW21] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2986–3001. ACM Press, November 2021.

A Trusting Proof Systems

Formula Interpretations An interpretation ι of a vocabulary defines a domain of values D and assigns each function and predicate symbol to functions and predicates in D. Specifically, the interpretation of each function symbol $f \in \mathcal{F}$, denoted $\iota_f : D^{|f|} \to D$, maps $|f|$-tuples in D to D. ι defines a natural interpretation of each term t as a function from variable assignments to D, denoted $\iota[t] : (\mathcal{V} \to D) \to D$, by composing the interpretations ι_f of all function symbols $f \in \mathcal{F}$ occurring in t. Similarly, an interpretation of each *predicate symbol* $p \in \mathcal{P}$ is a set of $|p|$-tuples in D, denoted $\iota[p] \subseteq D^{|p|}$. Combined, ι^F and ι^P interpret each atom $p(t_0, \ldots, t_n)$ over $\mathcal{F}$ and $\mathcal{P}$ as a set of $\mathcal{V}$-assignments $\iota[p(t_0, \ldots, t_n)]$ that *satisfy* them: assignment $X : \mathcal{V} \to D$ satisfies the atom's interpretation if $(\iota^{\mathcal{F}[t_0]}, \ldots, \iota^{\mathcal{F}[t_n]}) \in \iota^{\mathcal{P}(p)}$. ι^F and ι^P define an interpretation of each formula φ as a set of assignments that satisfy it, by combining the interpretation of atoms with the standard interpretations of Boolean connectives.

For example, under an interpretation ι with domain $\mathbb{N}$ that interprets $\mathtt{0}$ as $0 \in \mathbb{N}$, interprets $+$ as addition of naturals, and interprets as $\mathtt{succ}$ as the $\mathbb{N}$-valued function $s(x) = 1 + x$, the logical term $\mathtt{X} + \mathtt{succ}(\mathtt{succ}(\mathtt{0}))$ denotes the $\mathbb{N}$-valued function $t(x) = x + 2$ from X-assignments to $\mathbb{N}$. Under an interpretation of predicate symbols $=$ and $\leq$ as the equality and less-than-or-equal relations of $\mathbb{N}$, the assignment $x \mapsto 2$ satisfies the atom $4 \leq x + 3$.

Theory Axiomatizations Typically, we are interested in determining if a formula is unsatisfiable not under arbitrary interpretations, but specifically ones that satisfy laws of a mathematical domain of interest. Such laws are represented as the theory's *axioms*, which are simply an automatically recognizable set of formulas. A formula is *valid* under a set of axioms if it *accepts* every interpretation and assignment that satisfies all axioms; it is *unsatisfiable* under the axioms if it *rejects* every such interpretation and assignment.

As an example, the standard axiomatization of linear arithmetic contains the set of formulas $x + \mathtt{succ}(\mathtt{y}) = \mathtt{succ}(x) + y$, for all variables x and y (among others). These axioms are satisfied by the standard model of natural numbers (given above), but not, for instance, by a non-standard interpretation that interprets $\mathtt{succ}$ as the function $s(z) = 0$.

In general, a proof system may contain proofs of arbitrary formulas, even False, but such *inconsistent* systems typically are not of much practical use. Instead, we are typically interested in proof systems that, for a given set of axioms $\mathcal{A}$, are *sound*: they only contain proofs of formulas that are valid under $\mathcal{A}$. Designing and certifying a sound proof system can be highly non-trivial: depending on the theories of interest, it may achieved either by mechanical proof in an even more expressive logic or by social processes. Our work considers settings in which the Prover and Verifier have agreed upon a proof system for refuting formulas: typically, these will be systems that have been argued as sound for some axiomatization of interest, which is indeed the case for all theories to which we have applied our framework.

RuleID	Conclusion
Boolean	
TruePos	$\bigvee\{\mathsf{True}\}$
FalseNeg	$\bigvee\{\neg\mathsf{False}\}$
ExclMid	$\bigvee\{\neg a, a\}$
ImplPos1	$\bigvee\{a \to b, a\}$
ImplPos2	$\bigvee\{a \to b, \neg b\}$
ImplNeg	$\bigvee\{\neg(a \to b), \neg a, b\}$
EquivPos1	$\bigvee\{a = b, a, b\}$
EquivPos2	$\bigvee\{a = b, \neg a, \neg b\}$
EquivNeg1	$\bigvee\{\neg(a = b), a, \neg b\}$
EquivNeg2	$\bigvee\{\neg(a = b), \neg a, b\}$
EUF	
Refl	$\bigvee\{a = a\}$
Symm	$\bigvee\{a = b, \neg(b = a)\}$
Trans	$\bigvee\{a = c, \neg(a = b), \neg(b = c)\}$
LIA	
Total	$\bigvee\{a \leq b, b < a\}$
Trichotomy	$\bigvee\{a < b, a = b, b < a\}$
AddSingle	$\sum\{a\} = a$
MulSingle	$1 * a = a$
MulDist	$c * (\sum_{i=0}^{n} d_i * x_i) = \sum_{i=0}^{n} cd_i * x_i$

Table 5: ZKSMT's rules that have no premises or side conditions, grouped by theory.

B Proof Rule Tables

Tables 5 and 6 show our full set of proof rules for Boolean logic, EUF, and LIA. Table 5 contains the simple rules that have no premises or side conditions, and Table 6 contains the more complex rules.

C Proofs

We prove that ZKSMT is sound and complete by demonstrating that ZKSMT and SMTInterpol are equipotent when operating on Boolean logic, EUF, and LIA. A refutation proof of a formula exists in ZKSMT's format if and only if a corresponding proof exists in SMTInterpol's format based on only Boolean logic, EUF, and LIA. We restrict our attention to only these three theories because they are the ones supported by the implementation.

C.1 Proof of VM Soundness

We will start with soundness: if a refutation proof exists in ZKSMT's format, a corresponding refutation proof exists in SMTInterpol's format. Let Π be a proof in ZKSMT's format that derives False from φ. Our goal is to convert Π into a new proof Π' in SMTInterpol's format that derives an empty disjunction from φ. (SMTInterpol uses an empty disjunction as the end goal of refutation proofs rather than False.) If we can construct Π', then we know that φ is boundedly verifiable because proofs in SMTInterpol's format must be finite. We can construct Π' by induction. In both formats, a proof is a tree of derivations, so we can convert Π into Π' by providing a conversion process for every individual proof rule that Π could contain. For most proof rules, the conversion

RuleID	Side Condition	Premises	Conclusion
Boolean			
Resolution	$\exists p.p \in \langle A \rangle$, $\neg p \in \langle B \rangle$, $\langle A \rangle \subseteq \langle C \rangle \uplus \langle p \rangle$, $\langle B \rangle \subseteq \langle C \rangle \uplus \langle \neg p \rangle$	$\bigvee A$, $\bigvee B$	$\bigvee C$
DeDup	$\forall a \in \langle A \rangle.\ a \in \langle B \rangle$	$\bigvee A$	$\bigvee B$
OrNil		$\bigvee \{\}$	False
OrSingle		a	$\bigvee \{a\}$
OrSingleRev		$\bigvee \{a\}$	a
AndPos	$\exists A, B.\langle \bigwedge A \rangle \uplus \langle B \rangle = \langle C \rangle$, $\bigwedge A = \bigwedge_{i=0}^{n} a_i, \bigvee B = \bigvee_{i=0}^{n} \neg a_i$		$\bigvee C$
AndNeg	$a \in \langle A \rangle$		$\bigvee\{\neg \bigwedge A, a\}$
OrPos	$a \in \langle A \rangle$		$\bigvee\{\bigvee A, \neg a\}$
OrNeg	$\exists A.\langle \neg \bigvee A \rangle \uplus \langle A \rangle = \langle B \rangle$		$\bigvee B$
EUF			
Congruence	$\exists A, B, f.(fA = fB) \in C$, $\lvert A \rvert = \lvert B \rvert$, $\forall i \in \{0, \ldots, \lvert A \rvert - 1\}.\neg(A_i = B_i) \in C$		$\bigvee C$
LIA			
TotalInt	$i_0 = m * \texttt{ONE}$, $i_1 = (m+1) * \texttt{ONE}$		$\bigvee\{a \leq i_0, i_1 \leq a\}$
Consolidate	$\exists a, A_a, B_a, C.\ \langle A_a \rangle \uplus \langle C \rangle = \langle A \rangle, \langle B_a \rangle \uplus \langle C \rangle = \langle B \rangle$, $A_a = \{\alpha_0 * a, \ldots, \alpha_{t-1} * a\}, B_a = \{\beta_0 * a, \ldots, \beta_{t'-1} * a\}$, $\alpha_0 + \cdots + \alpha_{t-1} = \beta_0 + \cdots + \beta_{t'-1}$		$\sum A = \sum B$
Flatten	$\exists \langle C \rangle, \langle D \rangle.\ \langle C \rangle \uplus \langle \sum D \rangle = \langle A \rangle$, $\langle C \rangle \uplus \langle D \rangle = \langle B \rangle$		$\sum A = \sum B$
Farkas	$\forall i \in \{0, \ldots, n\}.\ m_i \geq 0$ either $c > 0$, or $c = 0$ and $\exists j.\ \leq_j$ is $<$	$\sum_{i=0}^{n}(m_i * a_i)+$ $(-m_i * b_i) = c$	$\bigvee_{i=0}^{n}\{\neg(a_i \leq_i b_i)\}$

Table 6: ZKSMT's rules that have premises or side conditions, grouped by theory. Capital letters represent argument lists for n-ary operations, and lowercase letters represent individual expressions.

is trivial because SMTInterpol supports a functionally identical rule. We will show only the cases for proof rules in our format that do not have exact analogues among SMTInterpol's rules.

DeDup DeDup has no analogue in SMTInterpol because SMTInterpol's disjunctions cannot contain duplicate elements. To convert a proof in our format into SMTInterpol's format, we can discard occurrences of DeDup.

OrNil OrNil simply converts an empty disjunction into False. Our refutation proofs never contain more than one occurrence of OrNil because we are always finished when we reach False. To convert a proof in ZKSMT's format into an SMTInterpol proof, we can simply discard occurrences of OrNil and treat the empty disjunction that OrNil uses as a premise as the end of the proof.

OrSingle and OrSingleRev We can discard all steps that use OrSingle and OrSingleRev. In ZKSMT's format, OrSingle and OrSingleRev interchange singleton disjunctions with the formulas inside them. In SMTInterpol's format, every formula is implicitly a disjunction, so the conversions that OrSingle and OrSingleRev perform in Π are unnecessary in Π'.

Congruence We omit the details in Section 4, but ZKSMT has multiple distinct congruence rules: one for uninterpreted functions, one for n-ary Boolean and LIA connectives, one for binary connectives, and so on. On the other hand, SMTInterpol has only one `cong` rule. All of our congruence rules map onto `cong` because SMTInterpol's `cong` works for Boolean connectives and arithmetic operations along with uninterpreted functions.

Addition Rules AddSingle, Consolidate, and Flatten are all restricted versions of `poly+`, SMT-Interpol's general-purpose polynomial addition rule. Anything that can be proven with AddSingle, Consolidate, or Flatten can also be proven with `poly+` because `poly+` proves equalities for arbitrary sums of polynomials. If a proof in our format uses AddSingle to derive $\sum_{i=0}^{0} a = a$, then we can use `poly+` to derive `(= (+ a) a)` in SMTInterpol's format. Likewise, if we prove $\sum A = \sum B$ with Consolidate or Flatten, we can derive the same conclusion with `poly+`.

MulDist ZKSMT's MulDist rule is a restricted version of `poly*`, SMTInterpol's polynomial multiplication rule. ZKSMT's MulDist rule allows only multiplications of linear sums by constants, but `poly*` supports arbitrary polynomial multiplications. Therefore, any use of MulDist in Π can be translated into a corresponding use of `poly*` in Π'.

Farkas' Lemma Both ZKSMT and SMTInterpol have rules for Farkas' lemma, but one significant difference exists between the two tools' rules. ZKSMT's version takes an equation as a premise, but SMTInterpol's version treats the same equation as a side condition. If Π contains an application of Farkas' lemma along with a series of steps proving the premise, we can discard the steps used for the premise and convert the application of Farkas' lemma into SMTInterpol's `farkas` rule for Π'.

C.2 Proof of VM Completeness

For the reverse direction, we will start with a proof Π' in SMTInterpol's format and produce a new proof Π in ZKSMT's format. Again, we will cover only the rules that require non-trivial conversions. We convert every step in Π' into a finite number of steps in Π, and Π' itself must be finite, so Π will be finite as well. Because Π must be finite, we can always place upper limits on π, χ, μ, α, and ρ once the conversion is finished.

Congruence SMTInterpol's `cong` rule can be applied not only to uninterpreted functions but also to arithmetic and Boolean operations. ZKSMT has multiple distinct congruence rules, but all of SMTInterpol's uses of `cong` within Boolean logic, EUF, and LIA map exactly to one of them.

Resolution SMTInterpol's `res` rule does not include duplicate entries in the conclusion, but ZKSMT's Resolution rule does. When we convert an application of `res` from Π' into ZKSMT's format, we may need to add duplicates to the conclusion. We can eliminate the duplicates immediately afterward with DeDup.

Polynomial Addition SMTInterpol's `poly+` rule supports arbitrary polynomial additions. We do not allow arbitrary additions to be performed atomically, but our LIA rules allow us to achieve the same end result over the course of multiple steps.

If we need to prove that $\sum A = \sum B$ for two arbitrary sums, then we can start by proving that $\sum A = \sum A'$, where A' is a normalized version of A that is a flat sum of Mul nodes, where no two distinct nodes have the same child. Flatten can eliminate nested sums, MulSingle can convert every entry in the sum into a Mul node that is not one already, MulDist can eliminate nested products, and Consolidate can combine Mul nodes with the same child. We can take the same approach to prove that $\sum B = \sum B'$, where B' is a normalized version of B. Because $\sum A$ and $\sum B$ are equal, A' and B' should be identical apart from the ordering of their elements. This means that a single application of Consolidate can prove $\sum A' = \sum B'$. At that point, we can use our EUF rules to chain all of the equalities together, producing $\sum A = \sum B$ as the end result. Overall, the conversion requires only finitely many proof steps in Π because A and B can contain only finitely many nested sums, non-Mul entries, nested products, and pairs of Mul nodes with the same child.

Polynomial Multiplication SMTInterpol's `poly*` rule supports arbitrary polynomial multiplications, but ZKSMT's MulDist rule allows only multiplications of sums by constants. SMTInterpol will not include non-linear multiplications in a proof unless the input formula itself includes non-linear multiplications. We are restricting our attention to Boolean logic, EUF, and LIA, so we do not need to take non-linear multiplications into consideration.

Any use of `poly*` that we receive from SMTInterpol for LIA can be translated easily into a use of our MulDist rule. At least one of the factors in a multiplication must be a constant, so we always have a value to use as the scaling factor. We use the other factor as the child of the multiplication node.

The requirement for every entry in the sum to be a Mul node does not have an analogue in SMTInterpol's format. If a sum that we receive from SMTInterpol contains entries that are not products, we can always use MulSingle and Congruence to normalize every entry in the sum.

Farkas' Lemma The difference between ZKSMT's rule for Farkas' lemma and SMTInterpol's `farkas` rule is that ZKSMT's version takes a premise. For the side condition in SMTInterpol's version of Farkas' lemma, the weighted sum of the polynomials provided as arguments must equal a nonnegative integer constant. For our own version, we take a premise that asserts the same condition. To convert SMTInterpol's version into our version, we need to construct the premise and the steps required to prove it.

Let c be the nonnegative constant used for the original application of `farkas`. We can start by applying Refl to get that $c = c$. We can then construct the sum that we need for the premise gradually, applying Consolidate to introduce expressions that cancel each other and using our EUF rules to chain all of the equations that we produce. Finally, we group the terms into the configuration that the premise requires by using MulDist and Flatten. This conversion results in only a finite number of new steps in Π for the same reason that the conversion for `poly+` does.

Singleton Disjunctions In SMTInterpol's representation, every formula is implicitly a disjunction. Just as we can discard uses of OrSingle and OrSingleRev when converting one of our own proofs into an SMTInterpol proof, we can add uses of OrSingle and OrSingleRev when converting a proof from SMTInterpol into ZKSMT's format.

Empty Disjunctions SMTInterpol's refutation proofs always end by deriving an empty disjunction, but ZKSMT's refutation proofs end with False. At the point where Π' derives an empty disjunction, we can add an application of OrNil in Π to complete the proof.