Enhancing Symbolic Execution with Machine-Checked Safety Proofs

David Trabish
Technion
Haifa, Israel
davidtr1037@gmail.com

Shachar Itzhaky
Technion
Haifa, Israel
shachari@cs.technion.ac.il

Abstract

Symbolic execution (SE) is a program analysis technique that executes the program with symbolic inputs. In modern SE engines, when the analysis of a given program is exhaustive, the analyzed program is typically considered *safe*, i.e., free of bugs, but no formal guarantees are provided to support this. Rather than aiming for a formally verified SE engine that will provide such guarantees, which is challenging, we propose a systematic approach where each individual analysis additionally generates a formal safety proof that validates the symbolic computations that were carried out. Our approach consists of two main components: A formal framework connecting concrete and symbolic semantics, and an instrumentation of the SE engine which generates formal safety proofs based on this framework. We showcase our approach by implementing a KLEE-based prototype that operates on a subset of LLVM IR with integers and generates proofs in Rocq. Our preliminary experiments show that our approach generates proofs that have reasonable validation times, while the instrumentation incurs only a minor overhead on the SE engine. In addition, during the implementation of our prototype, we found previously unknown semantic implementation issues in KLEE.

CCS Concepts: • **Software and its engineering**;

Keywords: Symbolic Execution, Proof Assistants, Program Verification

ACM Reference Format:
David Trabish and Shachar Itzhaky. 2026. Enhancing Symbolic Execution with Machine-Checked Safety Proofs. In *Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '26), January 12–13, 2026, Rennes, France.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3779031.3779089

CPP '26, Rennes, France

ACM ISBN 979-8-4007-2341-4/2026/01
https://doi.org/10.1145/3779031.3779089

1 Introduction

Symbolic execution (SE) is a well-established program analysis technique with applications in many areas, including automated test generation, bug finding, program repair, and verification [3–5, 11, 18–20]. In symbolic execution, the program is run with a *symbolic* input. Whenever the execution reaches a branch that depends on a symbolic input, an SMT solver [8] is used to determine the feasibility of each branch side, and the feasible paths are further explored, with their path constraints updated accordingly. Once the execution of a given path is completed, the SMT solver uses the accumulated path constraints to compute a satisfying model, which is then translated into a concrete test case that can be used to cover that path.

When modern SE engines (KLEE [3], ANGR [24], etc.) analyze programs, they strive to explore as many paths as possible, and if all the paths are explored, then the analysis is considered as *exhaustive*. If the analysis is exhaustive and finds no bugs, we usually conclude that the program is *safe*, i.e., free of bugs. In contrast to the case when the analysis finds bugs, where one can examine the generated bug-triggering test cases in a realistic setting (e.g., by executing the program natively), in the case where the analysis is exhaustive and finds no bugs, neither the generated test cases nor any other information produced during the analysis can ascertain that the program is indeed safe. The analysis can be incomplete due to various forms of under-approximation such as state pruning and concretizations. Moreover, there is the risk of implementation issues within the SE engine: incorrect encoding of the program semantics, unsound SMT optimizations, etc.

Standard testing techniques for developing SE engines cannot guarantee the absence of such implementation issues, as will be demonstrated next. Consider, for example, the C program at the top of Figure 1, which declares the symbolic coordinates of two points (line 1), and then computes the gradient between these points (line 3) if they do not share the same x-coordinate. The corresponding LLVM IR [14] is partially shown at the bottom of Figure 1. Since the division operation is performed on signed integers, the `sdiv` instruction is used (line 8). According to the LLVM semantics, the `sdiv` instruction triggers *undefined behavior* in two cases: division by zero and overflow. Here, when we execute the `sdiv` instruction, the divisor cannot

```
int x1, y1, x2, y2; // symbolic
if (x1 != x2)
  int gradient = (y1 - y2) / (x1 - x2);
```

```
entry:
  ...
  %cmp = icmp ne i32 %x1, %x2
  br i1 %cmp, label %if.then, label %if.end
if.then:
  %sub1 = sub i32 %y1, %y2
  %sub2 = sub i32 %x1, %x2
  %gradient = sdiv i32 %sub1, %sub2
```

Figure 1. A simple C program with gradient computation and its translation to LLVM IR.

be zero since we make sure that x1 and x2 are distinct. However, if Δy (i.e., y1 - y2) and Δx (i.e., x1 - x2) are evaluated to *-2147483648* (the minimum value of a 32-bit signed integer) and *-1*, respectively, then an overflow occurs. Therefore, this program is not safe w.r.t. the LLVM semantics.[1] However, when we analyzed this program with KLEE [3], a *state-of-the-art* SE engine, it explored exactly two paths without reporting any errors. The reason for this is that the implementation of the semantics of sdiv in KLEE turned out to overlook the case of overflow.[2] So in this case, even though the analysis was exhaustive and found no bugs, concluding that the program is safe from that result alone would be a mistake. While this may seem like extreme nitpicking, this scenario exemplifies the inherent incompleteness of relying on the output of SE tools at face value.

One way to address this problem is to build a formally verified SE engine. Approaches such as [7, 12] formalize the concrete and symbolic semantics using a proof assistant, and then use extraction to obtain an executable implementation of the SE engine. So far, such approaches were demonstrated in rather simplified settings (e.g., prototype languages), and have yet to be applied in the context of real-world SE engines, which typically include advanced features such as: search heuristics, constraint solving optimizations, and other performance optimizations. For example, KLEE [3] consists of thousands lines of unverified C++ code and has unverified dependencies (C++ standard library, LLVM library, etc.), so its verification is an extremely challenging task.

Our approach for this problem takes inspiration from *translation validation* [21, 23]. Rather than verifying the SE engine in advance, we instrument the SE engine such that during the analysis of the given program, it generates a Rocq [25] proof that shows the safety of that program, if such a proof can be derived. Then, if a proof is successfully generated, we validate it using Rocq. We note that the seminal paper introducing symbolic execution [13] already suggested that if the execution tree of a program is finite, then the output of an exhaustive symbolic execution can be translated into a formal safety proof (Section 8 in [13]). However, this idea received little attention so far.

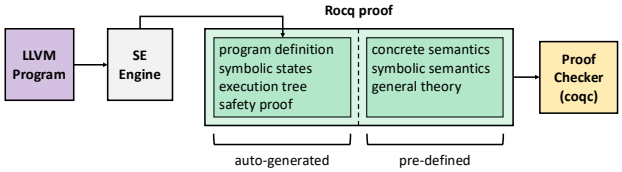

Figure 2. High-Level Architecture.

The high-level architecture of our approach is given in Figure 2. First, we formalize in Rocq the concrete and symbolic semantics of LLVM IR. To prove that a program is safe, i.e., the execution of the program according to the concrete LLVM semantics cannot result in errors (assertion failures, etc.), we rely on *execution trees*, which record the paths explored during the analysis. More specifically, we prove a theorem showing that if one provides an execution tree that is *error-free*, i.e., does not contain error symbolic states, and *complete*, i.e., all the feasible symbolic states derivable by the symbolic LLVM semantics are represented by equivalent symbolic states in the execution tree, then the program is safe. To apply this theorem, one needs to provide an execution tree, so to infer program safety, we instrument the SE engine to generate an execution tree along with a proof showing that it satisfies the required properties. Hence, the final output is a Rocq proof consisting of two sections: The first one is written manually and contains the formalized semantics and other general theory, and the second one is generated by the SE engine during the analysis of the input program. To be confident in the generated proof, you need to trust the kernel of Rocq and the SMT solver, and to be convinced in our specification of the concrete LLVM semantics, thus reducing the trusted computing base. Our approach provides flexibility in the development of the SE engine, as our formalization does not require updates to accommodate modifications made in the SE engine, unless core semantic features are modified. This provides separation of concerns and decoupling for aspects such as search heuristics and performance optimizations, and allows to benefit from previous engineering effort that was invested in existing tools.

We observed that a naive approach for proof generation results in large proofs, leading to slow validation or even validation failure due to memory exhaustion. To address this, we devise several optimizations that reduce the size of the generated proofs and help accelerate their validation.

The main contributions of this paper are:

[1] Running the corresponding program natively results in an error as well (floating point exception).

[2] We discovered this issue in the context of this work, and it was confirmed by the maintainers of KLEE (https://github.com/klee/klee/issues/1755).

$$
\begin{array}{lcl}
T \triangleq \{\mathsf{i}w \mid w \in \mathbb{N}\} & & \\
\mathit{binop} & ::= & \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{urem} \mid \ldots \\
\mathit{cmpop} & ::= & \mathsf{eq} \mid \mathsf{ne} \mid \mathsf{ugt} \mid \mathsf{uge} \mid \mathsf{ult} \mid \mathsf{ule} \mid \ldots \\
\mathit{op} & ::= & \mathit{binop} \mid \mathsf{icmp}_{\mathit{cmpop}} \\
\mathit{cast} & ::= & \mathsf{zext} \mid \mathsf{sext} \mid \mathsf{trunc} \mid \mathsf{bitcast} \\
e & ::= & n \mid x \mid \mathit{op}^{\tau}\ e_1\ e_2 \mid \mathit{cast}^{\tau \to \tau'}\ e \\
\mathit{instr} & ::= & x = e \mid x = \mathsf{phi}^{\tau}\ \{\ \overline{b_i \mapsto e_i}\ \} \mid \mathsf{br}\ b \mid \mathsf{br}\ e\ b_1\ b_2 \mid \\
 & & \mathsf{call}\ \mathit{id}\ (\overline{\tau_i\ e_i}) \mid x = \mathsf{call}^{\tau}\ \mathit{id}\ (\overline{\tau_i\ e_i}) \mid \mathsf{ret} \mid \mathsf{ret}^{\tau}\ e \mid \\
 & & \mathsf{unreachable} \\
\mathit{blk} & ::= & \mathit{id}\colon \overline{\mathit{instr}} \\
\mathit{func} & ::= & \mathsf{def}\ \tau\ \mathit{id}(\overline{\tau_i\ x_i})\ \{\ \overline{\mathit{blk}}\ \} \\
\mathit{module} & ::= & \overline{\mathit{func}}
\end{array}
$$

Figure 3. A simplified syntax of LLVM IR.

$$
\begin{array}{ll}
V^{\tau} \triangleq \{n@\tau \mid n \in \mathbb{N}\} \cup \{\mathit{undef}, \mathit{poison}\} & \\
V \triangleq \bigcup_{\tau \in T} V^{\tau},\ \Sigma \triangleq \mathit{Id} \mapsto V & \\
[\![n]\!]^{\tau}\sigma = n@\tau & \\
[\![x]\!]^{\tau}\sigma = \sigma(x) & \text{if } \sigma(x) \in V^{\tau} \\
[\![\mathit{op}^{\tau}\ e_1\ e_2]\!]^{\tau'}\sigma = & \text{if } [\![\mathit{op}^{\tau}]\!] : V^{\tau} \times V^{\tau} \to V^{\tau'} \\
\quad [\![\mathit{op}^{\tau}]\!]([\![e_1]\!]^{\tau}\sigma, [\![e_2]\!]^{\tau}\sigma) & \\
[\![\mathit{cast}^{\tau \to \tau'}\ e]\!]^{\tau'}\sigma = & \text{if } [\![\mathit{cast}^{\tau \to \tau'}]\!] : V^{\tau} \to V^{\tau'} \\
\quad [\![\mathit{cast}^{\tau \to \tau'}]\!]([\![e]\!]^{\tau}\sigma) & \\
[\![e]\!]^{\tau}\sigma = \bot & \text{in all other cases}
\end{array}
$$

Figure 4. Evaluation semantics of LLVM expressions.

1. We present a formal framework for symbolic semantics of an LLVM IR subset with integers, including properties that relate it to the concrete LLVM IR semantics. We realized this framework in Rocq.
2. Based on this framework, we present an approach for generating safety proofs in Rocq, which is designed as an instrumentation of the SE engine and incorporates optimizations that make proof validation tractable.
3. We implemented a KLEE-based prototype and evaluated it on programs that manipulate integers. Moreover, we found previously unknown semantic implementation issues in KLEE.

2 Preliminaries

In this section, we provide background on LLVM IR [14], an intermediate representation language used in LLVM, and on satisfiability modulo theories (SMT).

2.1 Concrete LLVM Semantics

The semantics presented here are largely based on *Vellvm* [28], focusing on core definitions to support the presentation of our symbolic semantics in Section 3.1.

2.1.1 Syntax. In this paper, we focus in a subset of LLVM IR with integers, whose simplified syntax is given in Figure 3. Integral values are represented as *bit-vectors*. A bit-vector type is denoted by iw, where w is the bit width (commonly, one of: 1, 8, 16, 32, 64). A bit-vector value is denoted by n@iw, where $0 \le n < 2^w$. Over this value domain, various operators are defined: arithmetic, comparison, etc. We define the set of bit-vector types T (Figure 3), and the set of values V^{τ} for each type $\tau \in T$ (Figure 4), which contains also the special values *undef* and *poison*, which are defined in the LLVM specification. The *undef* value indicates that the user of the value may receive an unspecified bit-pattern. Such values are useful as they indicate to the compiler that the program is well-defined no matter what value is used, which gives the compiler more freedom to optimize. To enable *speculative execution*, some instructions do not immediately trigger undefined behavior when provided with illegal operands, and return the *poison* value instead. For example, a shl operation returns *poison* if the bit-width operand is too large. Most instructions return *poison* if one of their operands is *poison*.

2.1.2 Semantics. We now describe the concrete state and the semantics over the LLVM IR syntax given in Figure 3.

Definition 2.1. A location $\ell \in L$ is specified by a function identifier, block identifier, instruction index, and an optional previous-block identifier. A *concrete state* is a tuple $\langle \ell, \sigma, \kappa \rangle$, where $\ell \in L$ is a location, $\sigma \in \Sigma$ (Figure 4) is a *local store* that maps variables to values in V, and $\kappa \in (L \times \Sigma \times \mathit{Id})^*$ is a call stack represented as a sequence of frames.

Figure 4 defines the evaluation semantics of LLVM expressions. The typed semantics of an expression $[\![e]\!]^{\tau}\sigma$ is defined as the value of e when the local store is $\sigma \in \Sigma$, in a context where a value of type τ is expected. When expressions do not adhere to the expected types, i.e., they are *ill-typed*, the semantics is undefined (marked by $\bot$). Operators (*op* and *cast* in Figure 3) are parameterized by bit-vector types, and their semantics is a function whose domains (and range) depend on those types (e.g., $[\![\mathsf{add}^{\tau}]\!] : V^{\tau} \times V^{\tau} \to V^{\tau}$).

Figure 5 defines the operational small-step semantics of LLVM IR based on the syntax in Figure 3. Location updates $\ell{+}1$ (next instruction) and $\ell \triangleright b$ (jump target) encapsulate the fine-grained control flow. The module Δ is assumed globally in order to simplify the notations in the semantic rules, and it consists of a set of functions, denoted by $\Delta.\mathit{funcs}$. A function f has the following attributes: $f.\mathit{id}$ is the identifier of the function, $f.\mathit{entry} \in L$ is the location of the first instruction, and $f.\mathit{args}$ is a vector of parameter identifiers. The semantic rules given in Figure 5 are largely standard, and we omit their detailed discussion for brevity. The only exceptions are the rules Make-Symbolic and Assume, which will be discussed in Section 3.1.

2.1.3 Program Safety. As we focus on a subset of LLVM IR with integers, we aim to detect two categories of errors: assertion failures and numerical errors.

In the LLVM IR produced by the compiler, assertions are typically translated into inline code that performs a conditional invocation of an external call, followed by the

L	$\{func : blk : i [: pblk] \mid func, blk, pblk \in Id, i \in \mathbb{N}\}$
Δ	top-level module
$\Delta[\ell]$	instruction at location ℓ
ℓ+1	next instruction location
$\ell \triangleright b$	jump target location

$$\frac{\Delta[\ell] = \text{`}x = e\text{'} \quad e = \text{`}op^{\tau}\ldots\text{'} / \text{`}cast^{-\rightarrow\tau}\ldots\text{'} \quad [\![e]\!]^{\tau}\sigma \neq \bot}{\langle \ell, \sigma, \kappa\rangle \rightarrow \langle \ell{+}1, \sigma[x \mapsto [\![e]\!]^{\tau}\sigma], \kappa\rangle} \text{ (Assign)}$$

$$\frac{\Delta[\ell] = \text{`}x = \text{phi}^{\tau}\ \{\overline{b_i \mapsto e_i}\}\text{'} \quad \ell = \_ : b_j \quad [\![e_i]\!]^{\tau}\sigma \neq \bot}{\langle \ell, \sigma, \kappa\rangle \rightarrow \langle \ell{+}1, \sigma[x \mapsto [\![e_j]\!]^{\tau}\sigma], \kappa\rangle} \text{ (Phi)}$$

$$\frac{\Delta[\ell] = \text{`br } b\text{'} \quad \ell \triangleright b \neq \bot}{\langle \ell, \sigma, \kappa\rangle \rightarrow \langle \ell \triangleright b, \sigma, \kappa\rangle} \text{ (Branch)}$$

$$\frac{\Delta[\ell] = \text{`br } e\ b_1\ b_2\text{'} \quad [\![e]\!]^{\text{i1}}\sigma = 1@\text{i1} \quad \ell \triangleright b_i \neq \bot}{\langle \ell, \sigma, \kappa\rangle \rightarrow \langle \ell \triangleright b_1, \sigma, \kappa\rangle} \text{ (Branch-True)}$$

$$\frac{\Delta[\ell] = \text{`br } e\ b_1\ b_2\text{'} \quad [\![e]\!]^{\text{i1}}\sigma = 0@\text{i1} \quad \ell \triangleright b_i \neq \bot}{\langle \ell, \sigma, \kappa\rangle \rightarrow \langle \ell \triangleright b_2, \sigma, \kappa\rangle} \text{ (Branch-True)}$$

$$\frac{\begin{array}{c}\Delta[\ell] = \text{`call } f.id\ (\overline{\tau_i\ e_i})\text{'} \quad |\overline{a}| = |\overline{e}| \\ f \in \Delta.funcs \quad \overline{a} = f.args \quad [\![e_i]\!]^{\tau_i}\sigma \neq \bot\end{array}}{\langle \ell, \sigma, \kappa\rangle \rightarrow \left\langle f.entry, \{\overline{a_i \mapsto [\![e_i]\!]^{\tau_i}\sigma}\}, \langle \ell{+}1, \sigma, \_\rangle \cdot \kappa\right\rangle} \text{ (Call-Void)}$$

$$\frac{\begin{array}{c}\Delta[\ell] = \text{`}x = \text{call}^{\tau}\ f.id\ (\overline{\tau_i\ e_i})\text{'} \quad |\overline{a}| = |\overline{e}| \\ f \in \Delta.funcs \quad \overline{a} = f.args \quad [\![e_i]\!]^{\tau_i}\sigma \neq \bot\end{array}}{\langle \ell, \sigma, \kappa\rangle \rightarrow \left\langle f.entry, \{\overline{a_i \mapsto [\![e_i]\!]^{\tau_i}\sigma}\}, \langle \ell{+}1, \sigma, x\rangle \cdot \kappa\right\rangle} \text{ (Call)}$$

$$\frac{\Delta[\ell] = \text{`ret}\text{'}}{\left\langle \ell, \sigma, \langle \ell', \sigma', \_\rangle \cdot \kappa\right\rangle \rightarrow \langle \ell', \sigma', \kappa\rangle} \text{ (Return-Void)}$$

$$\frac{\Delta[\ell] = \text{`ret}^{\tau}\ e\text{'} \quad [\![e]\!]^{\tau}\sigma \neq \bot}{\left\langle \ell, \sigma, \langle \ell', \sigma', x\rangle \cdot \kappa\right\rangle \rightarrow \langle \ell', \sigma'[x \mapsto [\![e]\!]^{\tau}\sigma], \kappa\rangle} \text{ (Return)}$$

$$\frac{\Delta[\ell] = \text{`}x = \text{call}^{\text{i32}}\ make\_symbolic()\text{'} \quad n \in \mathbb{N}}{\langle \ell, \sigma, \kappa\rangle \rightarrow \langle \ell{+}1, \sigma[x \mapsto n@\text{i32}], \kappa\rangle} \text{ (Make-Symbolic)}$$

$$\frac{\Delta[\ell] = \text{`call } assume(e)\text{'} \quad [\![e]\!]^{\text{i1}}\sigma = 1@\text{i1}}{\langle \ell, \sigma, \kappa\rangle \rightarrow \langle \ell{+}1, \sigma, \kappa\rangle} \text{ (Assume)}$$

Figure 5. Operational small-step semantics of LLVM IR.

unreachable instruction. Therefore, although the semantics of unreachable is unspecified, we found it convenient to interpret it as an assertion failure.

Regarding numerical errors, we focus on the following operations: division (udiv, sdiv, urem, and srem) and shift (shl, lshr, and ashr). According to the concrete LLVM semantics, when an invalid division operation occurs, i.e., division by zero or overflow, the result is undefined behavior. In addition, when an invalid shift operation occurs, i.e., shifting with an invalid bit-width, the result of the operation is *poison*. In symbolic execution, invalid shift operations are typically interpreted as errors, so we take this approach and interpret such cases as errors as well.

Based on this, we now formally specify when a concrete state is considered as an error state:

Definition 2.2. Let $c \triangleq \langle \ell, \sigma, \kappa\rangle$ be a concrete state. Then c is an *error* state, denoted by $error(c)$, if one of the following holds:

1. $\Delta[\ell] =$ 'unreachable'
2. $\Delta[\ell] =$ '$x = op\ \text{i}w\ e_1\ e_2$', and one of the following holds, where $v_k \triangleq [\![e_k]\!]^{\text{i}w}\sigma$:
   a. $op \in \{\text{udiv}, \text{sdiv}, \text{urem}, \text{srem}\}$ and $v_2 = 0@\text{i}w$
   b. $op = \text{sdiv}$, $v_1 = (-2^{w-1})@\text{i}w$, and $v_2 = (-1)@\text{i}w$
   c. $op \in \{\text{shl}, \text{lshr}, \text{ashr}\}$ and $v_2 = n@\text{i}w$, and $n \geq w$

Based on the defined semantics (Figure 5) and the previous definition, we now specify when a program, i.e., an LLVM module, is considered as safe:

Definition 2.3. Let f be the entry function in the module Δ. The *initial state* is defined as:

$$init(\Delta) \triangleq \langle f.entry, \{\}, []\rangle$$

Now, let $c_0 \triangleq init(\Delta)$. Then Δ is *safe*, denoted by $safe(\Delta)$, if:

$$\forall c.\ c_0 \rightarrow^* c \implies \neg error(c)$$

(The relation $\rightarrow^*$ is the reflexive-transitive closure of the relation $\rightarrow$.)

2.2 First-Order Terms and SMT

We focus on first-order logic modulo the theory of bit-vectors (BV), which is widely used in symbolic execution tools. In BV, there are infinitely many sorts $\{\text{BV}[w] \mid w \in \mathbb{N}, w > 0\}$, and each sort $\text{BV}[w]$ is interpreted as the set $\{0,1\}^w$. It introduces *interpreted* function symbols for operations over values of bit-vector sorts (e.g., *bvadd*, *bvextract*$[i, j]$). A BV *structure* m must map interpreted function symbols to their accepted functions over $\{0,1\}^w$. As is typically done in the context of symbolic execution, we identify the *Bool* sort with $\text{BV}[1]$, such that *true* and *false* correspond to 1 and 0, respectively.

Notations. We denote by $m[t]$ the interpretation of the term t in a structure m. We denote by $t_1 \equiv t_2$ the assertion that $m[t_1] = m[t_2]$ for any structure m.

3 Program Safety via Symbolic Semantics

The main result of this section is Theorem 3.20, which can be informally stated as follows: The safety of a program w.r.t. to the concrete LLVM semantics (Definition 2.3) can be guaranteed by the existence of an execution tree, a data-structure that records the explored paths during the analysis of the program, that satisfies certain properties. First, the execution tree must be *error-free*, i.e., it must not contain error symbolic states. Second, the execution tree must be *complete*, i.e., every feasible symbolic state derivable by the symbolic LLVM semantics is represented by an equivalent

symbolic state in the execution tree. In this section, we provide the theoretical foundation, and in Section 4, we show how to use the SE engine to generate such an execution tree along with a proof showing that it satisfies the required properties.

In Section 3.1, we present our symbolic semantics of LLVM IR. In Section 3.2, we discuss our assumptions about the LLVM module which later, in Section 3.3, enable us to establish the reachability completeness of the symbolic semantics w.r.t. a restricted subset of the concrete semantics. In Section 3.4, we show that if there is an execution tree which is error-free and complete, then the program is safe w.r.t. the restricted concrete semantics, which is then lifted to program safety w.r.t. the (complete) concrete semantics.

3.1 Symbolic LLVM Semantics

In symbolic execution, the program is run with a symbolic input, which is represented using *symbolic variables*. When a branch instruction is encountered, the SE engine performs a *fork*, which generates new symbolic states representing the feasible branches. We now describe the symbolic state and the symbolic semantics over the LLVM IR syntax given in Figure 3.

Definition 3.1. A *symbolic state* is a tuple $\langle \ell, \tilde{\sigma}, \tilde{\kappa}, \varphi \rangle$. The location $\ell \in L$ is defined as before. The concrete store $\sigma \in \Sigma$ is replaced with a symbolic store $\tilde{\sigma} \in \tilde{\Sigma}$, where $\tilde{\Sigma} \triangleq Id \longrightarrow E$ and E is a set of *SMT expressions*, i.e., first-order terms. The symbolic stack $\tilde{\kappa} \in (L \times \tilde{\Sigma} \times Id)^*$ is modified accordingly to hold symbolic stores instead of concrete ones. The *path constraint* $\varphi \in E$ is a formula of sort *Bool* which accumulates assumptions along the execution path.

The symbolic operational small-step semantics of LLVM IR is defined in Figure 6. The symbolic evaluation of an expression $[\![e]\!]^\tau \tilde{\sigma}$ is similar to the concrete case (Figure 4), except that $[\![op^\tau]\!]$ and $[\![cast^{\tau \to \tau'}]\!]$ are now *function symbols* whose application results in newly constructed terms. The symbolic evaluation is used, for example, in the Assign rule, derived from the analogous concrete rule by replacing the concrete store and stack with their symbolic counterparts. The rules Branch-True and Branch-False express the forking semantics: A symbolic state where the instruction is a conditional branch has two successors, and the appropriate condition is appended to the path constraint of the successors based on the jump target. We skip the discussion of the rules for Phi, Branch, Branch-True, Branch-False, Call-Void, Call, Return-Void, and Return, which are similarly analogous.

The symbolic execution environment provides additional intrinsic functions: *make_symbolic* and *assume*. The former creates a new symbolic variable of sort BV[32][3]. The latter is used to induce constraints on symbolic variables by adding a

[3]The width of 32 bits was chosen rather arbitrarily, and is not consequential.

$$\frac{\Delta[\ell] = \text{`}x = e\text{'} \quad e = \text{`}op^\tau \ldots\text{'} / \text{`}cast^{-\to\tau} \ldots\text{'} \quad [\![e]\!]^\tau \tilde{\sigma} \neq \bot}{\langle \ell, \tilde{\sigma}, \tilde{\kappa}, \varphi \rangle \rightsquigarrow \langle \ell{+}1, \tilde{\sigma}[x \mapsto [\![e]\!]^\tau \tilde{\sigma}], \tilde{\kappa}, \varphi \rangle} \text{ (Assign)}$$

$$\frac{\Delta[\ell] = \text{`br } e\ b_1\ b_2\text{'} \quad \ell \triangleright b_i \neq \bot \quad [\![e]\!]^{i1} \tilde{\sigma} \neq \bot}{\langle \ell, \tilde{\sigma}, \tilde{\kappa}, \varphi \rangle \rightsquigarrow \langle \ell \triangleright b_1, \tilde{\sigma}, \tilde{\kappa}, \varphi \wedge [\![e]\!]^{i1} \tilde{\sigma} \rangle} \text{ (Branch-True)}$$

$$\frac{\Delta[\ell] = \text{`br } e\ b_1\ b_2\text{'} \quad \ell \triangleright b_i \neq \bot \quad [\![e]\!]^{i1} \tilde{\sigma} \neq \bot}{\langle \ell, \tilde{\sigma}, \tilde{\kappa}, \varphi \rangle \rightsquigarrow \langle \ell \triangleright b_2, \tilde{\sigma}, \tilde{\kappa}, \varphi \wedge \neg [\![e]\!]^{i1} \tilde{\sigma} \rangle} \text{ (Branch-False)}$$

$$\frac{\Delta[\ell] = \text{`}x = \text{call}^{i32}\ make\_symbolic()\text{'} \quad \alpha \notin FV(\tilde{\sigma}) \cup FV(\tilde{\kappa})}{\langle \ell, \tilde{\sigma}, \tilde{\kappa}, \varphi \rangle \rightsquigarrow \langle \ell{+}1, \tilde{\sigma}[x \mapsto \alpha], \tilde{\kappa}, \varphi \rangle} \text{ (Make-Symbolic)}$$

$$\frac{\Delta[\ell] = \text{`call } assume(e)\text{'} \quad [\![e]\!]^{i1} \tilde{\sigma} \neq \bot}{\langle \ell, \tilde{\sigma}, \tilde{\kappa}, \varphi \rangle \rightsquigarrow \langle \ell{+}1, \tilde{\sigma}, \tilde{\kappa}, \varphi \wedge [\![e]\!]^{i1} \tilde{\sigma} \rangle} \text{ (Assume)}$$

Figure 6. Symbolic operational small-step semantics of LLVM IR.

conjunct to the path constraints. As for the Make-Symbolic rule, note that we need to make sure that newly created symbolic variables do not appear in the original symbolic state, otherwise, they might be restricted by existing path constraints, which would lead to incompleteness. Although introduced to support symbolic semantics, these functions must also have a concrete semantics in order to reason about them. The respective concrete semantics of Make-Symbolic and Assume is given in Figure 5.

The following two definitions describe symbolic errors states and initial symbolic states, which are analogous to those given in Section 2.1 (Definition 2.2 and Definition 2.3).

Definition 3.2. Let $s \triangleq \langle \ell, \tilde{\sigma}, \tilde{\kappa}, \varphi \rangle$ be a symbolic state. Then s is an *error* state, denoted by $error(c)$, if one of the following holds:

1. $\Delta[\ell] =$ `unreachable`
2. $\Delta[\ell] =$ `x = op iw e1 e2` and one of the following holds ($t_j \triangleq [\![e_j]\!]^{iw} \tilde{\sigma}$):
   a. $op \in \{\text{udiv}, \text{sdiv}, \text{urem}, \text{srem}\}$ and $\varphi \wedge (t_2 = 0)$ is satisfiable
   b. $op = \text{sdiv}$ and $\varphi \wedge (t_1 = -2^{w-1}) \wedge (t_2 = -1)$ is satisfiable
   c. $op \in \{\text{shl}, \text{lshr}, \text{ashr}\}$ and $\varphi \wedge (t_2 \geq w)$ is satisfiable

Definition 3.3. Let f be the entry function in Δ. The *initial symbolic state* is defined as:

$$\widetilde{init}(\Delta) \triangleq \langle f.entry, \{\}, [], true \rangle$$

Relating Symbolic and Concrete States. Informally, we say that a symbolic state *represents* a concrete state if the latter can be obtained from the former by concretizing the symbolic expressions using a structure that satisfies the path constraints. This relation will be needed later to establish reachability completeness (Section 3.3). The formal definition is outlined below in a bottom-up manner.

Definition 3.4. Let m be a structure. An SMT expression e *represents* a value $v \in V$ via m, denoted by $e \approx_m v$, if $m[e] = v$. A symbolic store $\tilde{\sigma}$ *represents* a concrete store σ via m, denoted by $\tilde{\sigma} \approx_m \sigma$, if both stores are defined over the same variables and for each such variable x, $\tilde{\sigma}(x) \approx_m \sigma(x)$. A symbolic frame $\langle \ell, \tilde{\sigma}, x \rangle$ *represents* a concrete frame $\langle \ell, \sigma, x \rangle$ via m if $\tilde{\sigma} \approx_m \sigma$. A symbolic stack $\tilde{\kappa}$ *represents* a concrete stack κ via a structure m, denoted by $\tilde{\kappa} \approx_m \kappa$, if they are of equal size, and the frames of $\tilde{\kappa}$ represent the frames of κ via m, pointwise. A symbolic state $s \triangleq \langle \ell, \tilde{\sigma}, \tilde{\kappa}, \varphi \rangle$ *represents* a concrete state $c \triangleq \langle \ell, \sigma, \kappa \rangle$ via m, denoted by $s \approx_m c$, if $\tilde{\sigma} \approx_m \sigma$, $\tilde{\kappa} \approx_m \kappa$, and $m[\varphi] = \mathit{true}$. Furthermore, s *represents* c, denoted by $s \approx c$, if $s \approx_m c$ for some structure m.

3.2 Module Assumptions

As mentioned in Section 2.1, some of the operations are *potentially erroneous*. For example, if a shl operation is performed with an invalid bit-width, the resulting value is *poison*. Rather than immediately invoking undefined behavior, the concrete LLVM semantics allows speculative execution with *poison* values. In symbolic execution, however, such erroneous operations are typically interpreted as errors [3].

Recall from Definition 2.2 that in the cases where the error is caused by an erroneous operation (division or shift), the erroneous operation itself is implicitly assumed to appear as the top-level expression. Note that the abstract syntax defined in Figure 3 allows instructions such as:

$$v = \mathsf{add\ i32}\ x\ (\mathsf{shl\ i32}\ y\ z)$$

where the potentially erroneous operation (shl) appears as the second operand and not as the top-level expression. If an erroneous shift operation indeed occurs in such case, then it will not be captured as an error by Definition 2.2.

To address this, one could generalize the definition of an error state, but this would unnecessarily complicate the formalization. Instead, we assume that a potentially erroneous operation can appear only as a top-level expression in an assignment instruction. In particular, this implies that every operand of an operation (or instruction) in the module is a *safe expression*, i.e., it does not contain potentially erroneous operations and it is not *poison* by itself. From our experience, in the LLVM IR produced by the compiler, the operands are either variables or constants. Therefore, the assumptions presented here are not restrictive in practice.

In addition, we assume that the module does not contain *undef* constants, since the SE engine replaces those with pre-defined constants before starting the analysis. We note that this pre-processing phase is valid, as the concrete LLVM semantics allows to replace *undef* with any integral value.

In the sequel, we assume that the module satisfies the assumptions mentioned above. For clarity, we say that a module is *supported* if it satisfies those assumptions.

3.3 Reachability Completeness

Ideally, we would like to establish the reachability completeness of the symbolic semantics w.r.t. the concrete semantics, i.e., prove that a sequence of steps in the concrete semantics can be simulated by a sequence of steps in the symbolic semantics. However, this is not possible since the concrete semantics allows speculative execution with *poison* values. According to Definition 3.4, a concrete state that contains a *poison* value cannot be represented by any symbolic state.

For example, consider the execution of the instruction 'x = shl i32 y 100'. On one side, by making a concrete step, x will be mapped to *poison*. On the other side, by making a symbolic step, x will be mapped to an SMT expression. Since an SMT expression can represent only integral values, this concrete step cannot be simulated by the symbolic one.

Therefore, in this section, rather than attempting to prove the reachability completeness of the symbolic semantics w.r.t. the (complete) concrete semantics (Section 2.1), we prove it w.r.t. a subset of the concrete semantics, which we refer to as the *non-speculative* concrete semantics. At first glance, such a result may seem too weak. However, later we will see that if a sequence of steps in the concrete semantics contains no error states, then the same sequence of steps is valid in the non-speculative concrete semantics as well. This will eventually allow us to guarantee safety w.r.t. the concrete semantics. Informally, the non-speculative concrete semantics allows only steps that do not introduce *poison* values. This restriction allows us to properly relate between symbolic and concrete states based on Definition 3.4, which, in turn, will allow us to establish reachability completeness. The formal definition is given below.

Definition 3.5. A concrete state $c \triangleq \langle \ell, \sigma, \kappa \rangle$ is *poison-free* if σ and κ do not contain *poison* values. Let c and c' be two concrete states. We say that there is a *non-speculative* step between c and c', denoted by $c \xrightarrow{ns} c'$, if:

1. $c \longrightarrow c'$, i.e., c' is derived from c according to the concrete semantics
2. c and c' are poison-free

Note that the absence of *poison* values alone is not enough to infer program safety, as error states can still be derived (assertion failures and division errors). As in Definition 2.3, we now specify when a program is considered as safe w.r.t. the non-speculative semantics.

Definition 3.6. Let $c_0 \triangleq \mathit{init}(\Delta)$, then Δ is *safe* w.r.t. the non-speculative concrete semantics, denoted by $\mathit{safe}_{ns}(\Delta)$, if:

$$\forall c.\ c_0 \xrightarrow{ns}{}^{*} c \implies \neg \mathit{error}(c)$$

Being a proper subset, the non-speculative concrete semantics is trivially *sound* w.r.t. the concrete semantics, but not *complete* w.r.t. to it. That is, a sequence of steps derived

by the concrete semantics might not be derivable in the non-speculative concrete semantics. However, we claim that if a sequence of steps does not contain error states, then such completeness property can be established. This is formulated in Lemmas 3.7 and 3.8.

Lemma 3.7. Let c and c' be concrete states. If Δ is supported, c is poison-free and not an error state, and $c \longrightarrow c'$, then $c \xrightarrow{ns} c'$.

Proof. The proof is done by enumerating over all the possible rules that derive c'. In our context, the only operation that can produce *poison* is a shift operation (e.g., shl). According to the module assumptions (Section 3.2), a shift operation can appear only in an assignment instruction, so *poison* can be introduced in c' only by the Assign rule. Therefore, the only relevant case is when c executes an instruction of the form:

$$x = op\ \mathsf{i}w\ e_1\ e_2$$

where op is a shift operation. Now, assume that $c \triangleq \langle \ell, \sigma, \kappa \rangle$. According to the module assumptions, we also know that both e_1 and e_2 are safe expressions (Section 3.2), so both $[\![e_1]\!]^{\mathsf{i}w}\sigma$ and $[\![e_2]\!]^{\mathsf{i}w}\sigma$ cannot be *poison*. Then, if the result of the shift operation above is *poison*, then it must be the case that $[\![e_2]\!]^{\mathsf{i}w}\sigma = n@\mathsf{i}w$ and $n \geq w$. According to Definition 2.2, this means that c is an error state, which contradicts the assumptions. □

Lemma 3.8. Let c and c' be concrete states. If Δ is supported, c is poison-free, no error states can be derived from c using the $\xrightarrow{ns}{}^*$ relation, and $c \longrightarrow^* c'$, then $c \xrightarrow{ns}{}^* c'$.

Proof. The proof is done by induction on the reflexive-transitive closure $\longrightarrow^*$ with the help of Lemma 3.7. □

The following theorem establishes the reachability completeness of the symbolic semantics w.r.t. the non-speculative concrete semantics. The proof of this theorem relies on additional lemmas which are given afterward (Lemmas 3.10 and 3.11).

Theorem 3.9. Let $c_0 \triangleq init(\Delta)$, and let $s_0 \triangleq \widetilde{init}(\Delta)$. If Δ is supported and $c_0 \xrightarrow{ns}{}^* c$, then there exists a symbolic state s such that $s_0 \overset{\sim}{\longrightarrow}{}^* s$ and $s \approx c$.

Proof. First, note that $s_0 \approx c_0$. Then, with the help of Lemma 3.10, the proof is done by induction on the reflexive-transitive closure $\xrightarrow{ns}{}^*$. □

Lemma 3.10. Let c and c' be concrete states, and let s be a symbolic state. If Δ is supported, $c \xrightarrow{ns} c'$, and $s \approx c$, then there exists s' such that $s \overset{\sim}{\longrightarrow} s'$ and $s' \approx c'$.

Proof. The proof is done by enumerating over all the possible rules that derive c'. As an example, we show how to handle two of these cases.

Assign rule: We know from the assumptions that there exists a structure m such that $s \approx_m c$, so in particular, $\tilde{\sigma}$ represents σ via m. To prove that s' represents c' via m, it is enough to prove that $[\![e]\!]^\tau\tilde{\sigma}$ represents $[\![e]\!]^\tau\sigma$ via m, since then $\tilde{\sigma}[x \mapsto [\![e]\!]^\tau\tilde{\sigma}]$ represents $\sigma[x \mapsto [\![e]\!]^\tau\sigma]$ via m. If e is a safe expression, then the required result is implied from Lemma 3.11. Otherwise, e is a potentially erroneous operation. As an example, we show how to prove the case where e is one of the shift operations: $\mathsf{shl}\ \tau\ e_1\ e_2$. According to the module assumptions, we know that both e_1 and e_2 are safe expressions, so both $[\![e_1]\!]^\tau\sigma$ and $[\![e_2]\!]^\tau\sigma$ must be integral values that are represented by some SMT expressions via m. If the value of the bit-width operand is invalid, then $[\![e]\!]^\tau\sigma$ is evaluated to *poison*, which contradicts the assumption that c' is poison-free. Otherwise, $[\![e]\!]^\tau\tilde{\sigma}$ is evaluated to the appropriate SMT expression that represents $[\![e]\!]^\tau\sigma$ via m.

Make-Symbolic rule: From the assumptions, we know that there exists a structure m such that $s \approx_m c$. According to the concrete semantics, the value of the assigned variable in c' is an arbitrary integer n. According to the symbolic semantics, the value of the assigned variable in s' is a symbolic variable α that cannot appear in s, and this means that α is not constrained by the path constraints of s. Therefore, for any n, we can extend the structure m to a structure m' such that $m'[\alpha] = n$ and m' satisfies the path constraints of s'. □

Lemma 3.11. If e is a safe expression (Section 3.2) and $\tilde{\sigma}$ represents σ via a structure m, then $[\![e]\!]^\tau\tilde{\sigma}$ represents $[\![e]\!]^\tau\sigma$ via m.

Proof. The proof is done by induction on the structure of e. As an example, we show how to prove the case where e is a binary operation: $op\ \tau\ e_1\ e_2$. By the induction hypothesis, both $[\![e_1]\!]^\tau\sigma$ and $[\![e_2]\!]^\tau\sigma$ are represented by some SMT expressions via m, so they must be integral values (i.e., neither *undef* nor *poison*). We assumed that op is not a potentially erroneous operation (i.e., division or shift), so $[\![e]\!]^\tau\sigma$ is evaluated to an integral value, and $[\![e]\!]^\tau\tilde{\sigma}$ is evaluated to the appropriate SMT expression that represents that value via m. □

3.4 Program Safety via Execution Trees

During the analysis of the program, the SE engine can produce an execution tree, which records the explored paths. In this section, we show how execution trees can be used as a backbone for safety proofs.

An execution tree t is a standard n-ary tree in which nodes are labeled with symbolic states. We denote by $t.state$ the symbolic state at the root of t, and by $t.children$ the subtrees representing the successor states of $t.state$. We also denote by $tree(s, l)$ the execution tree t in which $t.state \triangleq s$ and $t.children \triangleq l$, and use $leaf(s)$ as shorthand for $tree(s, [])$.

The SE engine employs SMT transformations and other optimizations, so the resulting symbolic states in the execution tree may differ from those derived by the symbolic semantics (Section 3.1). To use execution trees as a backbone

for safety proofs, we will need to relate the symbolic states in the execution tree and the symbolic states derived by the symbolic semantics. For this, we define the notion of *equivalence* between symbolic states.

Definition 3.12. The symbolic stores $\tilde{\sigma}_1$ and $\tilde{\sigma}_2$ are *equivalent*, denoted by $\tilde{\sigma}_1 \equiv \tilde{\sigma}_2$, if they are defined over the same variables, and for each such variable x, $\tilde{\sigma}_1(x) \equiv \tilde{\sigma}_2(x)$. The symbolic frames $\langle \ell_1, \tilde{\sigma}_1, x_1 \rangle$ and $\langle \ell_2, \tilde{\sigma}_2, x_2 \rangle$ are *equivalent* if $\ell_1 = \ell_2$, $\tilde{\sigma}_1 \equiv \tilde{\sigma}_2$, and $x_1 = x_2$. The symbolic stacks $\tilde{\kappa}_1$ and $\tilde{\kappa}_2$ are *equivalent*, denoted by $\tilde{\kappa}_1 \equiv \tilde{\kappa}_2$, if their frames are pointwise equivalent. The symbolic states $s_1 \triangleq \langle \ell_1, \tilde{\sigma}_1, \tilde{\kappa}_1, \varphi_1 \rangle$ and $s_2 \triangleq \langle \ell_2, \tilde{\sigma}_2, \tilde{\kappa}_2, \varphi_2 \rangle$ are *equivalent*, denoted by $s_1 \equiv s_2$, if $\ell_1 = \ell_2$, $\tilde{\sigma}_1 \equiv \tilde{\sigma}_2$, $\tilde{\kappa}_1 \equiv \tilde{\kappa}_2$, and $\varphi_1 \equiv \varphi_2$.

Note that the existence of an execution tree alone is not a sufficient condition for program safety, it merely reflects the symbolic exploration. To be able to prove program safety, we require execution trees to satisfy certain properties, which are formally defined below using the *safe-et* predicate.

Definition 3.13. Let $s \triangleq \langle \ell, \tilde{\sigma}, \tilde{\kappa}, \varphi \rangle$ be a symbolic state. If φ is unsatisfiable, then s is *infeasible*, denoted by $\mathit{infeasible}(s)$.

Definition 3.14. The predicate *safe-et* is defined inductively over execution trees:

$$\frac{s = \langle \ell, \tilde{\sigma}, \tilde{\kappa}, \varphi \rangle,\ \Delta[\ell] \in \{\text{`ret'}, \text{`ret}^{\tau}\ e\text{'}\},\ \tilde{\kappa} = []}{\mathit{safe\text{-}et}(\mathit{leaf}(s))}\ (\textsc{Term})$$

$$\frac{\neg \mathit{error}(s),\ \forall s'.\ (s \mathrel{\tilde{\rightarrow}} s') \rightarrow \mathit{repr}(s', l)}{\mathit{safe\text{-}et}(\mathit{tree}(s, l))}\ (\textsc{Step})$$

where:

$$\mathit{repr}(s, l) \triangleq \neg \mathit{infeasible}(s) \rightarrow \left(\exists t \in l.\ \mathit{safe\text{-}et}(t) \wedge s \equiv t.\mathit{state}\right)$$

The Term rule makes sure that the state s is about to terminate successfully, i.e., return from the entry function. The Step rule makes sure that there are no error states in the tree, and furthermore, that every feasible successor s' of any state s occurring in the tree is also represented in the tree, as a successor of s. Intuitively, infeasible successors are exempt because they represent no concrete states, and are thus irrelevant for safety properties.

Now, we claim that if the execution tree of the whole program satisfies the property described above, then the program is safe w.r.t. the non-speculative concrete semantics. This is formulated in the following lemma. The proof of this theorem relies on additional lemmas which are given afterward (Lemmas 3.16 to 3.19).

Lemma 3.15. Let $s_0 \triangleq \widetilde{\mathit{init}}(\Delta)$, and l be a list of execution trees. If Δ is supported and $\mathit{safe\text{-}et}(\mathit{tree}(s_0, l))$, then $\mathit{safe}_{ns}(\Delta)$.

Proof. According to Definition 3.6, let c be a state such that $c_0 \xrightarrow{ns}{}^{*} c$. Based on Theorem 3.9, we know that there exists a symbolic state s such that $s_0 \mathrel{\tilde{\rightarrow}}^{*} s$ and $s \approx c$, where $s_0 \triangleq \widetilde{\mathit{init}}(\Delta)$. According to Lemma 3.16, there are three possible consequences.

1. If $\mathit{safe\text{-}et}(\mathit{leaf}(s))$, then s is not an error state (Definition 3.14), and since $s \approx c$, then c is not an error state as well.
2. If there exist s_{et} and l' such that $s \equiv s_{et}$ and $\mathit{safe\text{-}et}(\mathit{tree}(s_{et}, l'))$, then according to Definition 3.14, s_{et} is not an error state, and since $s \equiv s_{et}$, then s is not an error state as well.
3. If s is infeasible, then we get a contradiction, since $s \approx c$, which means that there is a structure that satisfies the path constraints of s.

□

Lemma 3.16. Let s and s' be symbolic states and let l be a list of execution trees. If Δ is supported, $\mathit{safe\text{-}et}(\mathit{tree}(s, l))$, and $s \mathrel{\tilde{\rightarrow}}^{*} s'$, then one of the following holds:

1. $\mathit{safe\text{-}et}(\mathit{leaf}(s'))$
2. $\exists s'_{et}, l'.\ \left(s' \equiv s'_{et} \ \wedge\ \mathit{safe\text{-}et}(\mathit{tree}(s'_{et}, l'))\right)$
3. s' is infeasible

Proof. The proof is done by induction on the relation $\mathrel{\tilde{\rightarrow}}^{*}$. In the base case, the second condition is trivially satisfied. In the induction step, we have the symbolic states s, s', and s'', such that $s \mathrel{\tilde{\rightarrow}}^{*} s'$ and $s' \mathrel{\tilde{\rightarrow}} s''$. By the induction hypothesis, there exists a state s'_{et} that satisfies one of the three conditions.

1. If $\mathit{safe\text{-}et}(\mathit{leaf}(s'))$ then s' has no derivations, which contradicts the fact that $s' \mathrel{\tilde{\rightarrow}} s''$.
2. Suppose that there exist s'_{et} and l' such that $s' \equiv s'_{et}$ and $\mathit{safe\text{-}et}(\mathit{tree}(s'_{et}, l'))$. According to Lemma 3.17, it is enough to show that $\mathit{safe\text{-}et}(\mathit{tree}(s', l'))$, which can obtained from Lemma 3.18, as we know that $s' \equiv s'_{et}$.
3. If s' is infeasible, then s'' is also infeasible.

□

Lemma 3.17. Let s and s' be symbolic states and let l be a list of execution trees. If Δ is supported, $\mathit{safe\text{-}et}(\mathit{tree}(s, l))$, and $s \mathrel{\tilde{\rightarrow}} s'$, then one of the following holds:

1. $\mathit{safe\text{-}et}(\mathit{leaf}(s'))$
2. $\exists s'_{et}, l'.\ \left(s' \equiv s'_{et} \ \wedge\ \mathit{safe\text{-}et}(\mathit{tree}(s'_{et}, l'))\right)$
3. s' is infeasible

Proof. This follows almost immediately from Definition 3.14. □

Lemma 3.18. Let s_1 and s_2 be symbolic states and let l be a list of execution trees. If Δ is supported, $s_1 \equiv s_2$, and $\mathit{safe\text{-}et}(\mathit{tree}(s_1, l))$, then $\mathit{safe\text{-}et}(\mathit{tree}(s_2, l))$.

Proof. Following the Step rule from Definition 3.14, we know that s_1 is not an error state, so s_2 is not an error state as well, since $s_1 \equiv s_2$. Then, suppose that $s_2 \mathrel{\tilde{\rightarrow}} s'_2$. By Lemma 3.19, there exists s'_1 such that $s_1 \mathrel{\tilde{\rightarrow}} s'_1$ and $s'_1 \equiv s'_2$. By assumption, we know that $\mathit{safe\text{-}et}(\mathit{tree}(s_1, l))$, so there are two cases. In the first case, there exists $t \in l$ such that $\mathit{safe\text{-}et}(t)$ and $t.\mathit{state} \equiv s'_1$. Therefore, the same t can be used to show that $t.\mathit{state} \equiv s'_2$. In the other case, s'_1 is infeasible, so s'_2 is infeasible as well, since $s'_1 \equiv s'_2$. □

Lemma 3.19. Let s_1 and s_2 be symbolic states such that $s_1 \equiv s_2$. If Δ is supported and there exists s_1' such that $s_1 \rightsquigarrow s_1'$, then there exists s_2' such that $s_2 \rightsquigarrow s_2'$ and $s_1' \equiv s_2'$.

Proof. The proof is done by enumerating over all the possible rules that derive s_1'. In most of the cases, we rely on the fact that if $\tilde{\sigma}_1$ and $\tilde{\sigma}_2$ are equivalent, and if $[\![e]\!]^\tau \tilde{\sigma}_1$ is defined, then $[\![e]\!]^\tau \tilde{\sigma}_1 \equiv [\![e]\!]^\tau \tilde{\sigma}_2$. □

Recall that the non-speculative concrete semantics covers only a subset of the behaviors that are allowed in the concrete semantics, so the guarantee of Lemma 3.15 is not strong enough. However, Lemma 3.15 can be leveraged to guarantee program safety w.r.t. the concrete semantics. This is formulated in the following theorem.

Theorem 3.20. Let $s_0 \triangleq \widetilde{init}(\Delta)$, and l be a list of execution trees. If Δ is supported and $\mathit{safe\text{-}et}(\mathit{tree}(s_0, l))$, then $\mathit{safe}(\Delta)$.

Proof. According to Lemma 3.15, we know that $\mathit{safe}_{ns}(\Delta)$. Now, we will prove that $\mathit{safe}(\Delta)$. According to Definition 2.3, let c be a concrete state such that $c_0 \rightarrow^* c$, where $c_0 \triangleq \mathit{init}(\Delta)$. Note that c_0 is not an error state, otherwise s_0 would be an error state as well, since $s_0 \approx c_0$ and $\mathit{safe\text{-}et}(s_0, l)$. Then, note that the premise of Lemma 3.8 holds, since c_0 is poison-free and $\mathit{safe}_{ns}(\Delta)$, so we can conclude that $c_0 \overset{ns}{\rightarrow}{}^* c$. Finally, since $\mathit{safe}_{ns}(\Delta)$, then by Definition 3.6, we can conclude that c is not an error state. □

4 Generating Safety Proofs using Symbolic Execution

The implication of Theorem 3.20 is that if there exists an execution tree that satisfies the premise of the theorem, then we can obtain a safety proof of the input program. However, that theorem does not provide the means to compute such an execution tree. In this section, we present our approach that can automatically compute such execution trees and generate safety proofs.

At a high-level, our approach is designed as an instrumentation of the SE engine, and its general architecture is shown in Figure 2. The proof generated by our approach consists of a *pre-defined* section and an *auto-generated* section. The pre-defined section is written manually and it contains the various definitions that were given in Sections 2.1 and 3: concrete semantics, symbolic semantics, etc. The auto-generated section is generated by the SE engine during the analysis of the given program, and it contains definitions that are specific to that program: module definitions, symbolic states, execution tree, etc. The final output is a Rocq proof that can be validated for correctness. Note that a valid proof cannot be obtained if the analyzed program is unsafe, or if the SE engine performs incorrect computations.

In Section 4.1, we describe the structure of the generated proofs, and in Section 4.2, we present several optimizations that make the validation of the generated proofs tractable.

```
1  define i32 @main() {
2  entry:
3    %x = call i32 @make_symbolic()
4    %cmp1 = icmp ult i32 %x, 100
5    call void @assume(i1 %cmp1)
6    %y = add i32 %x, 1
7    %cmp2 = icmp ugt i32 %y, 50
8    br i1 %cmp2, label %if.then, label %if.end
9  if.then:
10   %cmp3 = icmp ult i32 %y, 200
11   br i1 %cmp3, label %if.end, label %if.else
12 if.else:
13   unreachable
14 if.end:
15   ret i32 0
16 }
```

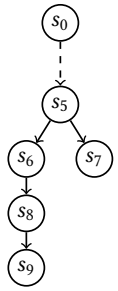

Figure 7. A simple LLVM IR program and its execution tree.

4.1 Proof Structure

The pre-defined section of the proof has been already discussed in detail in Sections 2.1 and 3, so now we give a further breakdown of the auto-generated section of the proof. First, we generate the module definitions and a proof of the module assumptions (Section 4.1.1). Then, during the symbolic exploration, we generate the definitions of the symbolic states (Section 4.1.2) and the execution tree (Section 4.1.3). Whenever the SE engine executes an instruction, we generate a corresponding lemma to prove the safety property (Definition 3.14) of the relevant part of the execution tree (Section 4.1.4). Once we generate the proof showing that the whole execution tree satisfies that safety property, we finalize the proof by applying Theorem 3.20 (Section 4.1.5).

To facilitate the reader's understanding, we use the LLVM program shown to the left of Figure 7. Its corresponding execution tree is shown to the right of Figure 7.

4.1.1 Module Assumptions. In this section, we generate the definition of the input program, i.e., the LLVM module. This includes the definitions of instructions, basic blocks, functions, etc. These are embedded in the generated proof using algebraic data types taken from *Vellvm* [28]. On top of this module encoding, we generate a proof showing that the module satisfies the assumptions discussed in Section 3.2. The generation of this proof is rather straightforward, so we do not discuss it here in detail.

4.1.2 Symbolic States. The definitions of the symbolic states are generated by the SE engine as tuples of the form $\langle \ell, \tilde{\sigma}, \tilde{\kappa}, \varphi \rangle$. The initial symbolic state is generated according to Definition 3.3. Then, whenever an instruction is executed by the SE engine, we generate the resulting successor states. Executing a branch instruction may result in two successor states if both branch sides are feasible. All other cases result in a single successor state.

In our example, the first two states are:

$$s_0 \triangleq \langle main{:}entry{:}0, \{\}, [], true\rangle$$
$$s_1 \triangleq \langle main{:}entry{:}1, \{x \mapsto \alpha\}, [], true\rangle$$

Here, the successor of s_0, which results from the execution of the instruction at line 3 from Figure 7, is s_1, where the local identifier x is associated with a fresh symbolic value α.

4.1.3 Execution Tree. The definition of the execution tree is generated bottom-up, such that if the execution of the symbolic state s results in the symbolic states $s_1, ..., s_n$, then the subtree of s is defined as $tree(s, [t_1, ..., t_n])$, where t_i is the definition of the subtree corresponding to s_i. In our example, the execution tree is defined in accordance to the right side of Figure 7 as follows:

$$t_9 \triangleq leaf(s_9),\ \ldots,\ t_5 \triangleq tree(s_5, [t_6, t_7]),\ \ldots,\ t_0 \triangleq tree(s_0, [t_1])$$

4.1.4 Execution Tree Safety. Once the execution tree is defined, we have to prove that it satisfies the *safe-et* relation (Definition 3.14). Similarly to the definition of the execution tree itself, we generate this proof in a bottom-up manner, showing that each subtree satisfies the required property. The safety of a given subtree can be proved by applying either the Term rule or the Step rule.

When the SE engine finishes to explore a symbolic state, we attempt to prove the safety of the corresponding subtree, which is a leaf node in this case. If the symbolic state was terminated due to an error (e.g., assertion failure), then it is impossible to apply any of the rules, as expected. Otherwise, the symbolic state was terminated safely, i.e., returned from the entry function, and then the proof is obtained by applying the Term rule.

For example, consider the subtree t_7, whose state s_7 is partially given below:

$$\langle main{:}if.end{:}9{:}entry, \{...\}, [],\ \ldots\ \rangle$$

Then, the safety of t_7 is proved using the Term rule.

When a symbolic state s executes an instruction, we attempt to prove the safety of the corresponding subtree t using the Step rule. To do so, we have to prove the following: First, s must not be an error state, and second, if s' can be derived from s according to the symbolic semantics, then either s' is equivalent to one of the children of t or s' is infeasible.

As mentioned above, the proof is generated bottom-up. In our example, we first generate the lemma of t_9, which is followed by the lemma of t_8 whose proof uses the lemma of t_9, and so on. The lemma of t_0 is generated only at the end.

Now, we will exemplify how we generate the proofs of these lemmas, using several instructions that are executed along the path that goes through lines 8, 11 and 15 in Figure 7.

Example 1: Assign. Consider the symbolic state s_3, which executes the instruction at line 6:

$$\langle main{:}entry{:}3, \{cmp1 \mapsto \alpha < 100, ...\}, [], ...\rangle$$

According to the Assign rule (Section 3.1), the only symbolic state derived from s_3 is:

$$\langle main{:}entry{:}4, \{y \mapsto \alpha + 1,\ cmp1 \mapsto \alpha < 100, ...\}, [], ...\rangle$$

According to the execution tree, the subtree t_3 corresponding to s_3 has a single child s_4, which differs from the symbolic state above only in the local store, which is given by:

$$\{y \mapsto 1 + \alpha,\ cmp1 \mapsto \alpha < 100, ...\}$$

The difference between those two symbolic states is a result of various SMT transformations employed by the SE engine: constant folding, normalization, etc. In our example, the expression $\alpha + 1$ is transformed to $1 + \alpha$, since the SE engine ensures that constants are placed on the left side of an *addition* operation. Nevertheless, we know that $\alpha + 1 \equiv 1 + \alpha$, so this transformation does not prevent us from proving the equivalence of those two symbolic states.

In general, we need the ability to prove such equivalences automatically. To do so, we define the function *transform* in Rocq, which mimics the SMT transformations performed by the SE engine, and then we manually prove that the resulting transformation preserves equivalence, i.e., $transform(e) \equiv e$. To apply this lemma, we apply the *transform* function on the expression generated according to the symbolic semantics. In the example above, $transform(\alpha + 1)$ evaluates to $1 + \alpha$, which gives us the required equivalence.

Example 2: Forking Branch. Consider the symbolic state s_5, which executes the instruction at line 8:

$$s_5 \triangleq \langle main{:}entry{:}5, \{...\}, [], \alpha < 100\rangle$$

Here, both branch sides are feasible, and according to the execution tree, the subtree t_5 corresponding to s_5 has two children: s_6 and s_7. According to the symbolic semantics, two symbolic states can be derived from s_5. The first one is derived by the Branch-True rule:

$$\langle main{:}if.then{:}6{:}entry, \{...\}, [], \alpha < 100 \wedge 50 < 1 + \alpha\rangle$$

and it is identical to the symbolic state s_6. The second one is derived by the Branch-False rule:

$$\langle main{:}if.end{:}9{:}entry, \{...\}, [], \alpha < 100 \wedge \neg(50 < 1 + \alpha)\rangle$$

and it differs from s_7 only in the path constraints, which are given by:

$$\alpha < 100 \wedge false = (50 < 1 + \alpha)$$

Here, as in the previous example, the difference is caused by SMT transformations. As before, applying the *transform* lemma gives us the required equivalence.

Example 3: Non-Forking Branch. Consider the symbolic state s_8, which executes the instruction at line 11:

$$\langle main{:}if.then{:}7{:}entry, \{...\}, [], \alpha < 100 \wedge 50 < 1 + \alpha\rangle$$

Here, only the true side of the branch is feasible. Based on the symbolic semantics, the two symbolic states derived from s_8

by the Branch-False and Branch-True rules are:

$$s_f \triangleq \langle main : if.else : 8 : if.then, \{...\}, [], \varphi_f \rangle$$
$$s_t \triangleq \langle main : if.end : 9 : if.then, \{...\}, [], \varphi_t \rangle$$

respectively, where

$$\varphi_f \triangleq \alpha < 100 \wedge 50 < 1 + \alpha \wedge \neg(1 + \alpha < 200)$$
$$\varphi_t \triangleq \alpha < 100 \wedge 50 < 1 + \alpha \wedge 1 + \alpha < 200$$

As for the Branch-False case, note that φ_f is unsatisfiable, so we use this to prove the infeasibility of s_f. In fact, we do not generate an explicit proof showing that this query is unsatisfiable. Instead, we assume the correctness of the underlying SMT solver, and define an *axiom* that states the unsatisfiability of this query. Proving unsatisfiability of SMT queries [2, 9] is outside the scope of this work.

As for the Branch-True case, according to the execution tree, the subtree t_8 corresponding to s_8 has a single child s_9. The definition of s_9 differs from s_t only in the path constraints, which are given by:

$$\alpha < 100 \wedge 50 < 1 + \alpha$$

This is a result of another optimization, in which the SE engine omits the last conjunct, since the other side of the branch is infeasible (w.r.t. the previous path constraints). Here, applying SMT transformations will not change the original SMT expression, so the *transform* lemma will not help us in this case. To handle such cases automatically, we use the following general lemma:

$$\forall pc, e.\ unsat(pc \wedge \neg e) \implies pc \wedge e \equiv pc$$

From the Branch-False case, we already have an axiom that states the unsatisfiability of the following query:

$$\alpha < 100 \wedge 50 < 1 + \alpha \wedge \neg(1 + \alpha < 200)$$

and using the lemma above we can automatically obtain the required equivalence.

4.1.5 Program Safety.

The last part of the generated proof is the application of Theorem 3.20. In our example, the proof till this point shows that: (1) the module Δ is supported (Section 4.1.1), (2) t_0 is safe (Section 4.1.4), and (3) $t_0.state$ matches the initial symbolic state $\widetilde{init}(\Delta)$. Therefore, we can conclude that the program from Figure 7 is safe w.r.t. to the concrete LLVM semantics.

4.2 Proof Compilation Optimizations

To check the validity of a Rocq proof, one has to compile it using *coqc*. Our proof-generation approach can generate valid proofs, however, in practice, their compilation is rather time-consuming. To explain the reasons for this, we further dive into the details of the generated proofs, and present two optimizations (Sections 4.2.1 and 4.2.2) that help accelerating proof compilation.

```
1  Lemma L_naive :
2    (safe_et t_3).
3  Proof.
4    apply SafeET.
5    - apply not_error_assign.
6    - intros s Hstep.
7      left.
8      exists (t_4).
9      split.
10     + apply in_eq.
11     + split.
12       { apply L_4. }
13       {
14         inversion Hstep.
15         ...
16       }
17 Qed.
```

```
1  Lemma L_optimized :
2    (safe_et t_3).
3  Proof.
4    apply (safe_tree_assign ...).
5    - apply (safe_expr ...).
6    - apply (equiv_smt_store ...).
7    - reflexivity.
8    - apply L_4.
9  Qed.
```

Figure 8. Two proof methods: naive and optimized.

4.2.1 Minimizing Proof Terms.

The proof that shows the safety of the execution tree (Section 4.1.4) is probably the most complex part of the auto-generated section. As mentioned before, this proof consists of lemmas, where each lemma proves the safety property of a given subtree. In Rocq, proofs are typically written using *Ltac*, a domain-specific language for writing custom tactics. These tactics eventually construct the proof term of a given lemma, which is then passed to the type checker.

To get an intuition about the specific structure of these proofs, consider the proof on the left of Figure 8, which shows that *safe-et*(t_3). First, it applies at line 4 the Step rule (Definition 3.14). Then, it shows at line 5 that s_3 is not an error state. Finally, at line 12 it shows that *safe-et*(t_4), and at lines 14 and 15 it shows that the symbolic state derived from s_3 is equivalent to s_4. The proofs for other instruction types are structured similarly.

Such a proof method, however, results in a rather slow compilation, even for simple programs. There are two main reasons for this: First, the proof uses time-consuming tactics such as: *inversion*, *subst*, etc. Second, the proof uses multiple tactics, some of which generate complex proof terms, which eventually sum up into a complex proof term. In this case, the size of the resulting proof term is roughly 53,000 bytes.

To address this, we aim to avoid expensive tactics and simplify the resulting proof terms as much as possible. We observed similarity between proofs that handle the same instruction type, so the idea is to create for each instruction type a generic lemma that can be used to infer execution tree safety. For example, for assignment instructions where the assigned expression is safe (Section 3.2), we define the following generic lemma:

Lemma 4.1. Let $s \triangleq \langle \ell, \tilde{\sigma}, \tilde{\kappa}, \varphi \rangle$ be a symbolic state, such that $\Delta[\ell] = inst(id, op(v, e))$. If the following holds:

1. e is a safe expression
2. $\tilde{\sigma}[v \mapsto [\![e]\!]^{\tau} \tilde{\sigma}] \equiv \tilde{\sigma}_{opt}$

3. $t.state = \langle l{+}1, \tilde{\sigma}_{opt}, \tilde{\kappa}, \varphi \rangle$
4. *safe-et*(t)

then *safe-et*$(tree(s, [t]))$.

Then, to prove that *safe-et*(t_3), we apply this lemma with the appropriate arguments, which are obtained with the help of the SE engine. The resulting proof is given on the right of Figure 8. This way, the proof does not use expensive tactics, which are shifted now to the proof of the generic lemma, which is compiled only once. In addition, the proof uses only simple tactics such as *apply* and *reflexivity*, which are generally fast and generates simple proof terms. Now, the resulting proof term consists of roughly 700 bytes only.

4.2.2 Reusing Terms. Another factor that impacts proof compilation is the complexity of explicit terms, those that appear in the definitions of the Rocq file. As mentioned in Section 4.1, the generated proof contains various terms, such as: SMT expressions, states, etc. As the complexity of the analysis grows, the complexity of the resulting terms typically grows as well.

For example, consider again the symbolic state s_5, which has two children in the execution tree: s_6 and s_7. Their path constraints are given by:

$$pc_6 \triangleq \alpha < 100 \wedge (50 < 1 + \alpha)$$
$$pc_7 \triangleq \alpha < 100 \wedge (\mathit{false} = (50 < 1 + \alpha))$$

As can be seen, the terms $\alpha < 100$ and $50 < 1 + \alpha$ appear in both of the definitions. This implies that each of these terms is type-checked twice, which is clearly undesirable.

To avoid this, we attempt to reuse terms that were already generated in the past. For example, when we translate an SMT expression given in the internal representation of the SE engine to a definition in Rocq, we decompose it into simpler definitions that correspond to sub-expressions of the original expression. If later we encounter an SMT expression which is already associated with a definition, then we simply use the variable associated with that definition, thus avoiding the re-generation of the full definition.

Going back to our example, the path constraints of s_6 and s_7 will be generated now as follows:

$$e_1 \triangleq \alpha < 100,\ e_2 \triangleq (50 < 1 + \alpha),\ e_3 \triangleq \mathit{false} = e_2$$
$$pc_6 \triangleq e_1 \wedge e_2,\ pc_7 \triangleq e_1 \wedge e_3$$

Now, the terms $\alpha < 100$ and $50 < 1 + \alpha$ are type-checked only once. A similar approach is applied to reuse terms in other definitions (stack frames, etc.).

4.3 Limitations

In this section, we discuss the main limitations of our proof-generation approach.

Handling SMT Transformations. Our proof-generation approach must be aware of the SMT transformations used by the SE engine. As mentioned in Section 4.1.4, we need an automatic way to prove the equivalence of symbolic states. To do so, we formalized the SMT transformations used by the SE engine, and developed an additional theory to prove that these transformations preserve equivalence. We note, however, that this additional theory can be adapted to other SE engines and it does not affect the formulation of the general theory presented in Section 3.

Handling SMT Queries. As mentioned in Section 4.1.4, we currently do not prove the unsatisfiability of SMT queries and assume the correctness of the underlying SMT solver instead, which increases our trusted computing base. Existing tools [2, 9] can be used to fill this gap.

5 Implementation

Our prototype implementation[4] consists of two components: A formalization of the concrete and symbolic semantics (Sections 2.1 and 3) and an extension of the SE engine. Our formalization is manually written in Rocq (8.19.1) and consists of roughly 18,000 lines of code, excluding third-party dependencies. We used CompCert [15] for arbitrary bit-width integers, and formalized the concrete LLVM semantics based on *Vellvm* [28]. Our supplementary material provides additional documentation which links between the proof sketches given in the paper and their corresponding definitions in our formalization. Our proof-generation approach (Section 4) is implemented on top of KLEE [3], a *state-of-the-art* SE engine that operates on LLVM bitcode [14]. Currently, our formalization does not support some of the SMT transformations used in KLEE, but we plan to implement them in future versions. As mentioned in Section 1, we found an issue in KLEE's implementation of the semantics of `sdiv`. This issue was confirmed by the maintainers of KLEE, and we fixed this issue in our implementation.

6 Evaluation

In our evaluation, we aim to provide preliminary evidence showing the applicability of our approach. We compare between several modes: PG_{opt}, PG_{ltac}, and PG_{none} are proof-generating modes, and the *Base* mode is vanilla KLEE. PG_{opt} uses the optimizations described in Sections 4.2.1 and 4.2.2, PG_{ltac} uses only the optimization described in Section 4.2.1, and PG_{none} uses none of these optimizations. As mentioned in Section 5, some of the SMT transformations are not supported by our formalization. Therefore, these transformations are disabled in the proof-generating modes.

The following research questions guide our evaluation:

(RQ1) Can our approach generate valid proofs that can be efficiently checked?
(RQ2) What is the overhead of proof generation during symbolic execution?

[4] https://github.com/davidtr1037/klee-rocq/tree/cpp-2026

(RQ3) What is the contribution of the proof compilation optimizations?

6.1 Setup

Each mode is run using the following configuration: The search heuristic is set to DFS,[5] the timeout is set to one hour, the memory limit is set to 4GB, and the SMT solver is set to STP 2.3.3 [10]. In all the modes, we measure the following metrics: analysis time, number of explored paths, and number of executed instructions. In the proof-generating modes (PG_{opt}, PG_{ltac}, and PG_{none}), we use *coqc* to validate the generated proofs, and we measure the *proof time*, i.e., the time required to compile the proof using *coqc*, and the *proof size*, i.e., the size of the compiled proof file (`.vo` file) created by *coqc*. We performed our experiments on Ubuntu 24.04, equipped with Intel Core i9-9900 and 32GB of RAM.

6.2 Benchmarks

Our prototype implementation supports a subset of LLVM IR with integers, so our evaluation is focused on programs that operate on integers. First, we selected a subset of programs from the *bit-vector* section of SVComp[6] that can be supported by our prototype implementation. Second, we selected a subset of bug-free programs from WiSE [7], and translated them to C from the prototype language IMP (Section 1 in [7]). Finally, we reused existing implementations of various algorithms and manually built test drivers that run those algorithms with symbolic inputs. The programs *jenkins*, *murmur*, and *fnv1a* implement hash algorithms, *is_prime* implements primality test algorithms, and *reverse* implements bit-wise reverse of integers. In some of the programs taken from SVComp and WiSE [7], the analysis with KLEE resulted in timeouts due to path explosion and complex SMT queries. To achieve exhaustive analysis in these cases, we used KLEE's intrinsic function `klee_assume` to impose additional constraints that limit the range of the symbolic inputs.

6.3 Results: Proof Generation

In this experiment, we focus on the validity and effectiveness of our approach. We run each program using the *Base* and PG_{opt} modes. The results are shown in Table 1.

In the *Base* section, the column *Time* shows the analysis time with vanilla KLEE. The *Proof-Generation* section shows the results of the proof-generating modes: The column *Analysis Time* shows the time required to complete the instrumented analysis when using PG_{opt}, and the columns *Proof Time* and *Proof Size* show the proof time and the proof size for each proof-generating mode, respectively. Our approach does not affect the symbolic exploration, so the number of explored paths (and executed instructions) is identical across all modes, as expected, and as confirmed by this experiment. The two metrics are shown under the columns *#Paths* and *#Inst.*

Our approach (PG_{opt}) was able to generate valid proofs for all the programs, which means that they are safe w.r.t. the formalized concrete LLVM semantics. The proof time was generally higher than the analysis time of *Base* (and PG_{opt}). However, in some cases (*jenkins*, *murmur*, and *fnv1a*), KLEE spent considerable time on constraint solving, so eventually the analysis time was much higher than the proof time. Typically, KLEE spends a significant amount of time on constraint solving, so in general, the ratio between the analysis time and the proof time depends on the complexity of the SMT queries. The proof size seems to be roughly linear w.r.t. the number of executed instructions, which makes sense, as we prove the safety of the subtrees corresponding to the executed instructions.

To assess the overhead of proof generation, we measured the slowdown of PG_{opt} compared to *Base*. The results indicate that the overhead is reasonable, with an average slowdown of 1.1× and a median of 1.0×. The maximum slowdown is obtained in *svcomp_byte_add* (2.75×), but in this case, the absolute difference in analysis time is less than one second.

Answering RQ1 and RQ2: Our approach generates valid safety proofs which can be checked in reasonable time, and the proof generation overhead is minor.

6.4 Results: Optimizations

In this experiment, we evaluate the significance of the optimizations presented in Section 4.2. To do so, we run each program in each of the proof-generating modes, and report the proof time and the proof size.

The results are shown in Table 1. When applying PG_{none} and PG_{ltac}, the size of the generated proof (`.v` file) in *svcomp_modulus* and *WiSE_gcd* was roughly 23GB and 2GB, respectively, so *coqc* could not compile the proof due to memory exhaustion. In terms of proof time, the average speedup of PG_{ltac} compared to PG_{none} is 2.1×, and the average speedup of PG_{opt} compared to PG_{none} is 4.8×. In terms of proof size, the average decrease achieved with PG_{ltac} relatively to PG_{none} is 53%, and the average decrease achieved with PG_{opt} relatively to PG_{none} is 77%.

Answering RQ3: The optimizations help reduce proof size significantly and accelerate proof compilation, to the point of making larger analyses feasible.

7 Related Work

Lindner et al. [16] propose an approach that operates on BIR, a low-level IR used for binary executables. A formal description of all the reachable symbolic states is given in the form of inference rules, and the SE engine operates strictly by applying them sequentially. Lundberg et al. [17] use symbolic

[5] Our approach is agnostic to the search heuristic. Similar results were obtained with KLEE's default search heuristic.

[6] https://github.com/sosy-lab/sv-benchmarks

Table 1. Comparison between the *Base* mode and the proof-generating modes.

	Base	Proof-Generation							#Paths	#Inst
	Analysis Time (seconds)	Analysis Time (seconds)	Proof Time (seconds)			Proof Size (KBs)				
			PG_{opt}	PG_{ltac}	PG_{none}	PG_{opt}	PG_{ltac}	PG_{none}		
svcomp_gcd1	0.1	0.1	1.4	1.9	4.5	419	476	1416	4	82
svcomp_gcd2	2.6	2.7	9.1	95.3	159.6	3242	27373	40743	42	1026
svcomp_gcd3	2.9	3.0	9.3	96.1	158.6	3269	27441	40897	45	1035
svcomp_sum	0.1	0.1	1.7	2.1	5.9	523	579	2329	10	158
svcomp_modulus	288.8	291.0	562.9	*OOM*	*OOM*	44517	-	-	398	15082
svcomp_num1	0.1	0.1	2.8	3.3	9.6	959	1064	3924	1	227
svcomp_num2	0.7	1.3	102.3	181.4	522.7	38983	42780	191914	256	11744
svcomp_bits	0.1	0.1	6.4	18.2	50.2	2358	7425	16529	1	650
svcomp_byte_add	0.4	1.1	142.2	263.8	482.6	70197	110753	192877	28	6239
WiSE_gcd	115.2	120.0	250.4	*OOM*	*OOM*	85845	-	-	1507	29228
WiSE_factorial	0.1	0.1	7.2	16.7	47.4	2967	3356	18903	13	1109
WiSE_sqrt	6.4	6.5	15.6	111.4	172.7	4656	10764	32428	91	1735
jenkins	30.9	31.1	1.0	1.4	2.9	231	340	702	2	36
murmur	35.4	35.3	1.0	1.3	2.7	237	294	631	2	36
fnv1a	2037.2	2049.3	0.9	0.9	1.5	171	175	327	2	18
is_prime	1.2	1.3	16.0	38.1	128.1	7075	8308	44624	30	2553
reverse	0.1	0.1	1.9	48.0	58.6	315	15274	15973	1	59

execution to produce functional correctness proofs for P4 programs using HOL4. They devise a symbolic execution algorithm that outputs a collection of theorems that can be used to prove properties about the result of the analysis. Our approach is less intrusive, as it allows the SE engine to operate independently from the formalized semantics, thus enabling more freedom in applying various optimizations.

Correnson and Steinhöfel [7] formalize a Rocq framework with concrete and symbolic semantics of a small prototype language. Based on this framework, they formalize a symbolic executor, which is then extracted into an executable OCaml implementation. Their formalization of the symbolic executor is coupled with a specific search heuristic (BFS), while our approach is agnostic to the underlying search heuristic. In contrast to our approach, their implementation does not include important optimizations that are crucial for the competitiveness of SE engines: advanced search heuristics, SMT optimizations, and other performance optimizations. The approach proposed by Keuchel et al. [12] is similar to the work mentioned above, but based on axiomatic semantics instead.

Porncharoenwase et al. [22] propose two implementations of symbolic evaluators: The first one is built on top of the reusable evaluator of Rosette [27], and the second is written in Lean. As mentioned in [22], the former is optimized but unverified, while the latter is verified but unoptimized. Our approach addresses this gap, since it provides formal guarantees, i.e., machine-checked safety proofs, while still allowing standard features and optimizations that are used in SE engines.

Chen et al. [6] envision a framework where analysis tools are automatically generated according to a user-specified formal language specification. To avoid formally verifying the entire framework, they propose an approach that generates a machine-checked proof as a correctness certificate for each individual task performed by the analysis tools. Their approach was demonstrated on a simple prototype language, and the use case of symbolic execution is not the focus of their work.

BinSym [26] is implemented by lifting executable formal specifications of ISA instructions to symbolic semantics. This approach helps to avoid errors in the implementation of the translation from ISA instructions to SMT expressions, but the implementation of the whole symbolic execution algorithm is still unverified. In addition, this approach requires the SE engine to be written in a particular way, which is not required by our approach.

8 Conclusion and Future Work

We address the challenge of providing a safety proof when the symbolic execution of a given program is exhaustive. We propose a proof-generating approach, which is based on a formal semantics framework and an instrumentation of the SE engine. Preliminary experiments with our prototype demonstrate the viability and applicability of our approach.

We plan to further extend our approach to support more advanced features of LLVM that require memory modeling: pointers, arrays, structs, global variables, dynamic allocations, etc. We presented optimizations that accelerate proof compilation, and we believe that there is room for further improvements.

Data Availability

Our supplementary material[7] contains a replication package for running our experiments and additional documentation for our Rocq formalization.

Acknowledgements

The research leading to these results has received funding from the Israel Science Foundation (ISF) grant No. 2117/23.

[7] https://doi.org/10.6084/m9.figshare.30689198

References

[1] [n. d.]. *Communications of the Association for Computing Machinery (CACM)* ([n. d.]).

[2] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. 2011. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In *International Conference on Certified Programs and Proofs*. Springer, 135–150.

[3] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)* (San Diego, CA, USA).

[4] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. 2006. EXE: Automatically Generating Inputs of Death. In *Proc. of the 13th ACM Conference on Computer and Communications Security (CCS'06)* (Alexandria, VA, USA). doi:10.1145/1455518.1455522

[5] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90. doi:10.1145/2408776.2408795

[6] Xiaohong Chen, Zhengyao Lin, Minh-Thai Trinh, and Grigore Roşu. 2021. Towards a trustworthy semantics-based language framework via proof generation. In *International Conference on Computer Aided Verification*. Springer, 477–499.

[7] Arthur Correnson and Dominic Steinhöfel. 2023. Engineering a formally verified automated bug finder. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1165–1176.

[8] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: introduction and applications. *Commun. ACM* 54, 9 (2011), 69–77. doi:10.1145/1995376.1995394

[9] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark Barrett. 2017. 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*. Springer, 126–133.

[10] Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-vectors and Arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification* (Berlin, Germany) *(CAV'07)*. Springer-Verlag, Berlin, Heidelberg, 519–531. http://dl.acm.org/citation.cfm?id=1770351.1770421

[11] Wei Jin and Alessandro Orso. 2012. BugRedux: Reproducing Field Failures for In-house Debugging. In *Proc. of the 34th International Conference on Software Engineering (ICSE'12)* (Zurich, Switzerland). doi:10.1109/ICSE.2012.6227168

[12] Steven Keuchel, Sander Huyghebaert, Georgy Lukyanov, and Dominique Devriese. 2022. Verified symbolic execution with Kripke specification monads (and no meta-programming). *Proc. ACM Program. Lang.* 6, ICFP, Article 97 (Aug. 2022), 31 pages. doi:10.1145/3547628

[13] James C. King. 1976. Symbolic execution and program testing. See[1], 385–394.

[14] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of the 2nd International Symposium on Code Generation and Optimization (CGO'04)* (Palo Alto, CA, USA). doi:10.1109/CGO.2004.1281665

[15] Xavier Leroy. 2009. Formal verification of a realistic compiler. See[1], 107–115.

[16] Andreas Lindner, Roberto Guanciale, and Mads Dam. 2023. Proof-Producing Symbolic Execution for Binary Code Verification. *CoRR* abs/2304.08848 (2023). arXiv:2304.08848 doi:10.48550/ARXIV.2304.08848

[17] Didrik Lundberg, Roberto Guanciale, and Mads Dam. 2024. Proof-Producing Symbolic Execution for P4. In *International Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 70–83.

[18] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*. ACM, 691–701. doi:10.1145/2884781.2884807

[19] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proc. of the 35th International Conference on Software Engineering (ICSE'13)* (San Francisco, CA, USA). doi:10.1109/ICSE.2013.6606623

[20] Corina S. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. 2013. Symbolic PathFinder: Integrating Symbolic Execution with Model Checking for Java Bytecode Analysis. In *Proc. of the 28th IEEE International Conference on Automated Software Engineering (ASE'13)* (Palo Alto, CA, USA).

[21] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems: 4th International Conference, TACAS'98 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'98 Lisbon, Portugal, March 28–April 4, 1998 Proceedings 4*. Springer, 151–166.

[22] Sorawee Porncharoenwase, Luke Nelson, Xi Wang, and Emina Torlak. 2022. A formal foundation for symbolic evaluation with merging. *Proc. ACM Program. Lang.* 6, POPL, Article 47 (Jan. 2022), 28 pages. doi:10.1145/3498709

[23] Hanan Samet. 1975. *Automatically proving the correctness of translations involving optimized code*. Ph. D. Dissertation. Stanford University.

[24] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'16)* (San Jose, CA, USA). doi:10.1109/SP.2016.17

[25] The Coq Development Team. 2023. *The Coq Proof Assistant Reference Manual*. https://coq.inria.fr Version 8.17.

[26] Sören Tempel, Tobias Brandt, Christoph Lüth, Christian Dietrich, and Rolf Drechsler. 2024. Accurate and Extensible Symbolic Execution of Binary Code based on Formal ISA Semantics. *arXiv preprint arXiv:2404.04132* (2024).

[27] Emina Torlak and Rastislav Bodik. 2014. A lightweight symbolic virtual machine for solver-aided host languages. *ACM SIGPLAN Notices* 49, 6 (2014), 530–541.

[28] Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 427–440.

Received 2025-09-12; accepted 2025-11-13