A Recursive Strategy for Symbolic Execution Expressed in Coq

A. Byrnes and C. Sturton

University of North Carolina at Chapel Hill Department of Computer Science, 201 S Columbia St, Chapel Hill, NC 27599
{abyrnes1,csturton}@cs.unc.edu

Abstract. Prior work has proposed the use of symbolic execution for the security validation of a processor design. The approach uses a recursive search strategy that is designed to counter the combinatorial explosion of paths inherent in multi-cycle processor execution. In this work, we examine the search strategy in order to prove its correctness. We first formulate an abstract model of symbolic execution in Coq. We then implement the recursive search strategy, and we find that the strategy as proposed is not guaranteed to be sound—that is, the search may find a symbolic path for which no concrete path from the processor's initial state to the searched-for error state exists. We tighten one of the requirements of the search strategy and then prove the modified strategy correct.

Keywords: Symbolic Execution · Coq · Hardware Validation.

1 Introduction

Researchers have recently begun exploring the use of software-style symbolic execution for the verification of hardware designs [17,14]. Symbolic execution has a proven track record in the software community as a bug-finding tool [5,6,12] and as an aid in formal verification [9,10]. However, bringing these benefits to bear on hardware designs has been a challenge—the complexity of the search space of relatively simple hardware designs more closely resembles that of large, continuously interactive software systems than that of the stand-alone software programs that are the classic targets of symbolic execution. In response to this challenge a recent paper proposed a recursive search strategy for symbolic execution [23]. Zhang et al. showed the strategy to be practical [22], but the soundness of the approach has not been demonstrated. A proof of correctness is needed.

In this paper we prove that a recursive search strategy for the symbolic execution of hardware designs is sound: a list of constraints returned by a successful search defines a set of concrete input sequences, each of which will take the processor from its initial reset state to an error state.

Our goal is to validate the search strategy itself rather than any one implementation of it. Therefore we decouple our model of symbolic execution from the particular programming language to be symbolically executed. We formalize an abstract model of symbolic execution in Gallina, the specification language of the Coq proof assistant [21]; the structure of the model is built around the three fundamental properties of symbolic execution as first laid out by King in 1976 [13]. This has the advantage of providing soundness guarantees for the recursive search strategy when implemented by any symbolic execution engine which abides by the three King properties.

The symbolic exploration of a program produces a rooted, binary tree. The root node represents the entry point of the program, and each path from root to a leaf node represents one possible path of the execution through the program. Although the mechanics are similar, the symbolic exploration of a hardware design differs from this model conceptually: The tree produced by the symbolic exploration of a hardware design represents the state transitions possible in a single clock cycle from a given state (the root node). A sequence of state transitions, for example from the hardware's initial, reset state to an error state is represented by a sequence of trees. The set of all possible n transitions from a given state requires a tree (of depth n) of trees.

The strategy proposed by Zhang and Sturton tackles this search complexity with a recursive search. First the n^{th} tree in the desired sequence of trees is found, then the $n-1^{\text{st}}$ tree, and so on, until the first tree in the sequence, whose root node starts at the reset state, is found.[1] The strategy lays out three properties that, if satisfied at each iteration of the search, are sufficient to ensure the final sequence of paths through trees represents a possible, multi-cycle sequence of state transitions from the hardware's initial state to the desired (error) state.

The bulk of our proof is a proof by induction over the sequence of trees to show that the trees are correctly stitched together. In the base case we show that symbolic execution, as defined by King, implies the correct *concretization* of a path through a single symbolic execution tree as defined by Zhang and Sturton. In the inductive step we show that a path through a sequence of trees stitched together according to the Zhang and Sturton requirements will yield a correct concretization of a path through the sequence of trees. Finally, we prove that the sequence begins in a reset state and ends in the desired error state.

We find that the recursive search strategy as originally proposed is not sound—that is, the search may find a symbolic path for which no concrete path from the processor's initial state to the given error state exists. We tighten one of the three requirements of the strategy and then prove the modified strategy is sound.

We present two contributions of this work:

– An abstract model of symbolic execution expressed in the Coq framework. This model can form the starting point for building a provably correct implementation of a symbolic execution engine for a particular language. We make our model available online at `https://github.com/AlyssaByrnes/recursive-strategy-symbolic-execution-proof`.

[1] The problem is undecidable in the general case, and Zhang et al. introduce a set of heuristics to make it work in practice [22].

- We verify the soundness of the recursive search strategy for symbolic execution. We find and fix a flaw in the original formulation of the strategy. The recursive search strategy was originally developed for the verification of hardware designs; however, any symbolic execution engine that implements the proven strategy will be assured the same guarantee of soundness.

2 Prior Work

Symbolic execution was first formally defined by King in 1976 [13] and has been widely adopted by the software engineering and software security communities. (See Schwartz et al. [20] for a recent survey.)

Symbolic execution has since shown practical value through implementations. Anand et al. introduced a tool JPF-SE that works as an extension of the Java PathFinder model checker to allow for symbolic execution of Java programs [2]. Păsăreanu et al. later introduced the tool Symbolic PathFinder that also utilizes Java Path Finder to symbolically execute Java bytecode [19]. Noller et al. recently released a tool that utilizes Java PathFinder to perform "shadow symbolic execution" on Java bytecode [18].

2.1 Use of Symbolic Execution in Hardware

In addition to the paper by Zhang and Sturton that lays out the three-part strategy we prove here [23], there are a handful of papers that examine the use of symbolic execution in hardware. In a subsequent paper, Zhang et al. [23] make use of their recursive strategy to find new security vulnerabilities in processor designs. Their work demonstrates that the strategy is useful in practice.

Prior work first explored the use of software-style symbolic execution for hardware designs. The STAR tool combines symbolic and concrete simulation of hardware designs to provide high statement and branch coverage [14]; PATH-SYMEX uses forward symbolic execution applied to an ANSI-C model of the hardware design [17]. Both tools use a forward search strategy and have limits on how deep or broadly they can search.

2.2 Formalization of Symbolic Execution

Arusoaie et al. provide a formal framework for a symbolic execution tool that is language-independent [3,4,15]. However, they do not formally verify their tool or its outputs. Nor does the framework represent King's branching properties [13].

2.3 Backward Search Strategies in Symbolic Execution

A backward search strategy for symbolic execution of software has been studied [16,7,11,8]. There the approach is to search backward through function call chains from a goal line of code to the program's entry point. The strategy is hindered by complications that arise in software, such as floating point calculations and external method calls. To the best of our knowledge, the strategy has not been formalized in the software community, nor has its validity been proven.

3 Background

We provide a brief review of symbolic execution and describe the key features of a hardware design that necessitate a new approach for symbolic execution. We explain the recursive strategy for symbolic execution and provide some background on the Coq proof assistant.

3.1 Symbolic Execution

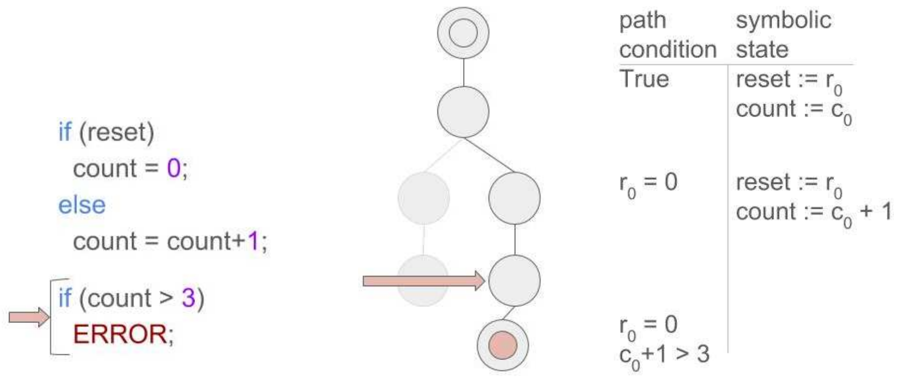
In symbolic execution the parameters to a program's entry point are assigned symbolic values. The program is then simulated by a symbolic execution engine that uses the symbolic values in place of concrete literals. As execution continues the resulting symbolic expressions propagate throughout the program's state. A symbolic expression e contains at least one symbol and may contain zero or more concrete literals, arithmetic operators, and logical operators. Examples of symbolic expressions include 'α' and '$\alpha + 1$,' where α is a symbol used by the symbolic execution engine.

In addition to the symbolic state, the symbolic execution engine keeps track of the *path condition* (π) for the current path of execution. When execution begins the path condition is initialized to TRUE. At each conditional branch in the program, the branching condition b is evaluated. If $\pi \rightarrow b$ is valid, the `then` branch is taken and the path condition is updated, $\pi := \pi \wedge b$. If $\pi \rightarrow \neg b$ is valid, the `else` branch is taken and the path condition is updated, $\pi := \pi \wedge \neg b$. If neither $\pi \rightarrow b$ nor $\pi \rightarrow \neg b$ hold, then both branches are possible. Execution forks and each branch is explored in turn with the path condition updated appropriately for each branch.

In general, a path condition π is a conjunction of propositions involving symbolic expressions. For example '$\pi := \alpha + 1 \geq 0 \wedge \alpha < 1$' is a possible path condition.

All the paths explored through the program form a logical tree T. The root represents the program's entry point and each node of the tree represents a line of code in the program. Associated with each node is the (partially) symbolic state of the program and the path condition at that point of execution. A path in the tree from root to leaf represents a path of execution through the program.

Figure 1 demonstrates the idea. Given the code in Figure 1a, `reset` and `count` are initialized with the symbolic values r_0 and c_0, respectively, and after symbolically executing lines 1, 3, and 4 `count` may be set to the symbolic expression $c_0 + 1$ as shown in Figure 1c. The path condition for the path through lines 1, 3, 4, 5, 6 is also shown. When execution reaches the `ERROR` at line 6 the path condition is $\pi := r_0 = 0 \wedge c_0 + 1 > 3$. This expression can then be passed to a standard, off-the-shelf SMT solver to find a satisfying solution, say $r_0 := 0$ and $c_0 := 3$. Substituting these values for `reset` and `count`, respectively, and executing the code from the beginning using these concrete values would cause execution to follow the same path as was followed symbolically. Figure 1b illustrates the tree of paths explored in the complete symbolic execution of the code fragment.



```
if (reset)
  count = 0;
else
  count = count+1;

if (count > 3)
  ERROR;
```

path condition	symbolic state

True	reset := r_0
	count := c_0

$r_0 = 0$	reset := r_0
	count := $c_0 + 1$

$r_0 = 0$ $c_0+1 > 3$	

(a) Code snippet (b) Tree of program paths (c) π and symbolic state

Fig. 1: Symbolic Execution

3.2 Register Transfer Level Hardware Designs

For their recursive strategy, Zhang et al. are targeting the symbolic execution of a synchronous (i.e., clocked) hardware design described at the register transfer level (RTL) of abstraction. The RTL specifies the stateful registers in the design (implemented as flip-flops or latches) and the combinational logic connecting those registers. A complete simulation of the design corresponds to a single clock cycle; the symbolic execution of the design produces a tree where the root node represents the design in a given state and each leaf node represents a possible next-state of the design at the next clock cycle.

The hardware design differs from software code in that it is not meant to be executed sequentially, rather it describes a design in which updates to different registers are all done in parallel. When run in simulation—during which the design code must be executed sequentially—an order of execution is imposed and temporary variables may be introduced in order to achieve the correct next state. The same strategy is used during symbolic execution and therefore, while the root and leaf nodes of the resulting symbolic execution tree correspond to valid states of the hardware design, the intermediate nodes of the tree do not and are largely ignored.

3.3 Recursive Search Strategy

The symbolic execution tree represents an exploration of the design for a single clock cycle. But it is often desirable to find states that arise only after many steps of execution. Because of this the complexity of the search space quickly becomes untenable. Exploring a design for two clock cycles would require first producing the symbolic execution tree of the design when started at the initial, reset state, and then, for each leaf in that tree, a new symbolic execution of the design, this time starting at the state described by the leaf node. It is to counter this combinatorial explosion that Zhang et al. developed their recursive search strategy.

The gist of the recursive search strategy is to search backward from the error state as follows. First use (forward) symbolic execution to find a state s_i that has the error state as one of its possible next-state transitions. Then use (forward) symbolic execution to find a state s_{i-1} that has s_i as one of its possible next-state transitions. This search continues until one of the found s's is an initial state of the processor. The result is a sequence of symbolic execution trees that can be used to produce a sequence of concrete input values that take the processor from its initial state to the error state.

3.4 Using Coq as a Verification Tool

Coq is a formal, interactive proof assistant that allows for machine-checked proofs of systems [1]. It implements the inductive language Gallina, which is based on the *Calculus of Inductive Constructions*, a typed λ-calculus. It allows definitions of methods that can be evaluated, the expression of theorems and software specifications, and assists the user in building machine-checkable proofs. If the proof is complete, it will compile.

Additionally, we make use of Coq's module system, representing different parts of our system as modules. There is a module defining concrete execution, a module defining symbolic execution that includes the King properties as axioms, and a module defining the recursive symbolic execution strategy that includes the requirements laid out by Zhang and Sturton as axioms and the correctness property expressed as a theorem.

Our definitions and fixpoints are the methods that are implemented in the system, such as execution of the list of trees provided by the recursive symbolic execution tool. Our variables are system-specific variables, such as the initial state and the error states, and they are contained in the recursive symbolic execution module as well.

We use the built-in Logic library for assistance in small proofs and the Ensembles library for assistance reasoning about finite sets. Additionally, we use the built-in List structure to represent lists of symbolic execution trees.

4 Definitions and Notation

4.1 Processor Model

In the formalization of their search strategy [23], Zhang et al. model a processor as a tuple $M = (S, s_0, I, \delta, O, \omega)$, where

- $S = \{s_0, s_1, \ldots, s_k\}$ is the finite set of states of the processor,
- $s_0 \in S$ is the initial state of the processor,
- $I = \{0,1\}^n$ is the finite set of input strings to the processor,
- $\delta : S \times I \to S$ is the transition function of the processor,
- $O = \{0,1\}^m$ is the finite set of output strings of the processor, and

– $\omega : S \rightarrow O$ is the output function of the processor.

A state $s \in S$ is determined by the values of the *stateful* registers of the design. These include both architectural registers, which are visible to software, and microarchitectural registers, which are not visible to software. Examples of the former include general purpose registers, the stack pointer, the instruction pointer, and some control registers. Examples of the latter include the buffers between stages of a pipeline, branch prediction registers, reorder buffers, and many control signals. These are all stateful in that they hold their current value up to and until getting updated at the next clock cycle. As such, their current-state value can be an input to the calculation of their next-state value.

The initial state s_0 represents the starting state of the processor immediately after a power-on or reset cycle. Many registers have a value of 0 in this state.

An input string $i \in I$ represents the concatenation of the several input signals to the processor. The string has a fixed length n. The list of input signals includes the instructions and data fetched from memory (or from a cache), plus control, error, debug, and interrupt signals.

The transition function δ defines how state is updated in a single clock cycle. It is left-total: for every $s \in S$ and every $i \in I$, δ is defined.

An output string $o \in O$ represents the concatenation of the several output signals of the processor. The string has a fixed length. The list of output signals includes addresses and data values to be written to memory, control signals, and error signals.

The output function ω determines the value of o in each clock cycle. It is typically the identity function taken over a subset of the stateful registers in the design.

4.2 Symbolic Execution

The symbolic exploration of a design is modeled by a tuple $T = (N, E)$ representing a rooted, binary tree, where N is the set of nodes of the tree and E is the set of directed edges connecting two nodes. Each node $n \in N$ of the tree represents a conditional branch point in the design; we elide the non-branching nodes—those that represent straight-line code and would have only a single child node. The two edges connecting n to its two children nodes represent the two possible paths of forward execution at that branch point.

Each node is a tuple $n = (\sigma, \pi)$, where

– σ is the symbolic state of the module at the current point of execution (later in this paper, we will often refer to it as the symbolic assignment), and
– π is the path condition associated with the current symbolic state.

As with standard execution, in symbolic execution, a symbolic state σ is determined by the values of the stateful registers of the design. However, the values may be symbolic, or concrete literals as in standard execution, or a combination of the two.

The path condition π is a Boolean formula over a subset of the input signals and stateful registers of the design. The path condition of the root node of the tree is always initialized to true: $n_r = (\sigma, \text{TRUE})$.

4.3 Coq Model of a Processor Design

We build a model of a hardware design that represents the stateful registers, their values at clock cycle boundaries, and the transition function. The model is abstract and suitable to the representation of any design that is synchronous (i.e., clocked) and includes sequential logic (i.e., stateful registers).

We model the design as a Coq program with the following types.

- Register is an abstract type that is used to represent the stateful registers in the RTL design.
- InputSignal is an abstract type that is used to represent the input signals in the RTL design.
- Value is an abstract type that is used to represent the literals that can be assigned to Registers or InputSignals.
- Assignment : $(r : \mathsf{Register},\ v : \mathsf{Value})$ is a tuple type that represents the assignment of a value $v \in \mathsf{Value}$ to a register $r \in \mathsf{Register}$.
- InputAssignment : $(i : \mathsf{InputSignal},\ v : \mathsf{Value})$ is a tuple type that represents the assignment of a value $v \in \mathsf{Value}$ to an input signal $i \in \mathsf{InputSignal}$.

A state of the processor is represented by a list of Assignment elements, $[a_0, a_1, a_2, \ldots]$, where each element is a tuple as defined by the type. The list is defined inductively and has unbounded length in the Coq program. The initial state of the processor is represented by a particular list of Assignment elements, denoted as *init*. We will use vector notation to represent a list of a given type (e.g., $\overrightarrow{\mathsf{Assignment}} \doteq [a_0, a_1, a_2, \ldots]$).

Execution is modeled by a function, `exec`(), that maps a list of Assignment elements and a list of InputAssignment elements to a list of Assignment elements. In other words, `exec`() takes a processor state and a set of inputs and returns a new processor state. As in our abstract model, it is deterministic and left-total:

- $\mathtt{exec}(c : \overrightarrow{\mathsf{Assignment}},\ i : \overrightarrow{\mathsf{InputAssignment}}) : \overrightarrow{\mathsf{Assignment}}$

We use c as the name for the first parameter to `exec` to emphasize that $\overrightarrow{\mathsf{Assignment}}$ represents a concrete state.

We do not model the output signals of the hardware design explicitly as they not needed for our proof. If needed, they could be modeled as a subset of the Register type.

4.4 Coq Model of Symbolic Execution

To model the symbolic execution of the processor in our Coq program we introduce types representing symbolic state, symbolic execution, and path conditions. The first three types are used in our representation of symbolic state.

- $\mathsf{SymbolicExpression}$ is an abstract type that is used to represent the symbolic expression that can be assigned to $\mathsf{Registers}$ or $\mathsf{InputSignals}$.
- $\mathsf{SymbolicAssignment} : (r : \mathsf{Register},\ e : \mathsf{SymbolicExpression})$ is a tuple type that represents the assignment of an expression $e \in \mathsf{SymbolicExpression}$ to a register $r \in \mathsf{Register}$.
- $\mathsf{SymbolicInputAssignment} : (i : \mathsf{InputSignal},\ e : \mathsf{SymbolicExpression})$ is a tuple type that represents the assignment of a symbolic expression $e \in \mathsf{SymbolicExpression}$ to an input signal $i \in \mathsf{InputSignal}$.

A symbolic state of the processor is represented by a list of $\mathsf{SymbolicAssignment}$ elements, $\overrightarrow{\mathsf{SymbolicAssignment}}$.

Symbolic execution is modeled by a function, `symExec`(), that maps a list of $\mathsf{SymbolicAssignment}$ elements and a list of $\mathsf{SymbolicInputAssignment}$ elements to a binary tree. In other words, it takes a symbolic state and a set of symbolic inputs and returns a tree that represents the complete symbolic exploration of the design starting from the given symbolic state. The function is defined as follows. (We define the $\mathsf{SymExecTree}$ type in the following paragraph.)

- $\texttt{symExec}(\sigma : \overrightarrow{\mathsf{SymbolicAssignment}},\ \iota : \overrightarrow{\mathsf{SymbolicInputAssignment}}) : \mathsf{SymExecTree}$

The tree returned by `symExec`() is a binary tree of nodes, where each node contains a symbolic state and a path condition. We introduce the following types in our Coq program.

- $\mathsf{PathCondition}$ is an abstract type that represents a propositional formula over the symbols used in symbolic expressions.
- $\mathsf{Node} : (\sigma : \overrightarrow{\mathsf{SymbolicAssignment}},\ \pi : \mathsf{PathCondition})$ is a tuple type that represents a node in the binary tree, containing a symbolic state (the list of $\mathsf{SymbolicAssignment}$ elements) and a path condition.

A $\mathsf{SymExecTree}$ is defined inductively as a binary tree of Nodes.

4.5 Relating Symbolic to Concrete Execution

The nodes of a symbolic execution tree are abstractions, each representing a set of possible concrete states. Given a node $n = (\sigma, \pi)$, the corresponding concrete states can be produced by replacing the symbols used in the symbolic state, σ, and the path condition, π, with literals in the following way. Let Σ be the alphabet of symbols appearing in σ or π. Let $\mathcal{L}$ be the domain of values for the program. For each $\alpha \in \Sigma$, choose a literal $l \in \mathcal{L}$. Replace the symbols appearing in the path condition π with their mapped-to literal and evaluate the resulting expression. If it evaluates to TRUE, then use the same mapping to replace the symbols appearing in the symbolic state σ with literals. Evaluating the resulting expressions will produce a concrete state.

As an example, consider the node $n = ([(r_0, \alpha), (r_1, \alpha + 1)], \alpha < 2)$, which assigns the symbolic expression "α" to register r_0, the symbolic expression "α+1" to register r_1, and has the path condition that constrains α to be less than 2.

Mapping the symbol "α" to the literal "1" would produce the path condition $1 < 2$, which is valid. Replacing symbols with their mapped-to literals in the symbolic state would produce the concrete state $[(r_0, 1), (r_1, 2)]$. A second concrete state, $[(r_0, 0), (r_1, 1)]$, is also in the set of concrete states represented by node n.

We define the following type and functions in our Coq model to relate symbolic state to concrete state.

- SymbolicMapping is an abstract type. Members of this type can be viewed as functions mapping symbols appearing in symbolic expressions to literals in the domain of the program.
- `instantiateState`$(\sigma : \overrightarrow{\mathsf{SymbolicAssignment}},\ m : \mathsf{SymbolicMapping}) : \overrightarrow{\mathsf{Assignment}}$ is a function that takes a symbolic state and a mapping from symbols to literals, and returns the concrete state produced by replacing all symbols with mapped-to literals and evaluating the resulting expression.
- `evaluatePC`$(\pi : \mathsf{PathCondition},\ m : \mathsf{SymbolicMapping}) : \mathsf{Prop}$ is a function that takes a path condition and a mapping from symbols to literals and returns the proposition (TRUE or FALSE) produced by replacing all symbols with mapped-to literals and evaluating the resulting Boolean formula.

4.6 Properties of Symbolic Execution

King formalized the use of symbolic execution [13] and described three fundamental properties provided by a symbolic execution engine. We name and summarize the properties and for each, give our formal expression of the property.

Property K.1 (Sound Paths). The path condition π never becomes unsatisfiable. For each leaf node l of a symbolic execution tree, there exists, for the path condition associated with l, at least one satisfying concrete valuation to the symbols of the path condition; that is, one mapping of symbols to concrete values that would make the path condition evaluate to TRUE. If this mapping of satisfying concrete values were applied to the initial symbolic state and symbolic inputs, the resulting concrete execution would follow the same path through the program as was taken by the symbolic execution engine to arrive at the leaf node l. In other words, all paths taken by the symbolic execution engine correspond to feasible paths through the program. We express this in the following.

$$\begin{aligned}\forall\ \sigma \in \Sigma,\ \alpha \in \Sigma_i,\ & n \in T = \texttt{symExec}(\sigma, \pi_0, \alpha),\\ & \exists m \in \mathsf{SymbolicMapping},\\ & \texttt{evaluatePC}(n.\pi, m) = \textsc{true}.\end{aligned}$$

Where Σ is the set of all possible symbolic states, Σ_i is the set of symbolic input values, and n is a node in the tree T produced by a call to `symExec`.

Property K.2 (Unique Paths). The path conditions π_1 and π_2 associated with any two paths of the tree are mutually unsatisfiable. In other words, there exists no concrete valuation that could drive execution down two distinct paths of the symbolic execution tree. We express this in the following.

$$\begin{aligned}\forall m \in \mathsf{SymbolicMapping},\ & \sigma \in \Sigma,\ \alpha \in \Sigma_i,\ n_1,\ n_2 \in T = \texttt{symExec}(\sigma, \pi_0, \alpha),\\ & n_1 \neq n_2\\ & \wedge (\texttt{ischildof}(n_1, n_2) \vee \texttt{ischildof}(n_2, n_1)) = \text{FALSE}\\ & \rightarrow \texttt{evaluatePC}(n_1.\pi \wedge n_2.\pi, m) = \text{FALSE}.\end{aligned}$$

Property K.3 (Commutativity). The actions of symbolically executing a program and instantiating symbols with concrete values are commutative. For any mapping of symbols to literals, let s be the end state produced by first applying the mapping and then executing the program concretely. The same state s will be found by first executing the program symbolically and then applying the mapping to every leaf node and choosing the leaf node whose path condition evaluates to TRUE. We express this in the following.

$$\begin{aligned}\forall m \in \mathsf{SymbolicMapping},\ & \sigma \in \Sigma,\ \alpha \in \Sigma_i,\ l \in T = \texttt{symExec}(\sigma, \pi_0, \alpha),\\ & \texttt{isleaf}(l, T) = \text{TRUE} \wedge \texttt{evaluatePC}(l.\pi, m) = \text{TRUE}\\ & \rightarrow \texttt{instantiateState}(l.\sigma, m) =\\ & \quad \texttt{exec}(\texttt{instantiateState}(\sigma, m), \texttt{instantiateState}(\alpha, m)).\end{aligned}$$

5 Recursive Search Strategy

The goal of the symbolic execution of a hardware design is to find a sequence of n inputs that will take the hardware module from an initial reset state to an error state in n clock cycles. It starts with a node s_i that contains an error state, and uses symbolic execution to find a state s_{i-1} that has s_i as one of its possible next-state transitions. This search continues until one of the found s's is a reset state. Each application of symbolic execution produces a tree with a particular leaf node of interest—the desired next-state s_i. The path conditions of each of these leaf nodes can be used to find an input leading to the next leaf node, producing a string of inputs leading to an error state.

We express the three original requirements of the recursive strategy as laid out by Zhang et al. [23]. We start by formally defining two instantiation operations that are only informally described by Zhang et al. but that are used by the strategy requirements.

5.1 Instantiation Operations

The root node of a symbolic execution tree T contains a symbolic state ($\texttt{getroot}(T).\vec{\sigma}$), which can be viewed as a representation of a set of possible concrete states. If $\vec{\sigma}$ is fully symbolic, meaning that all registers in the design are assigned symbolic expressions ($[(r_0, e_0), (r_1, e_1), \ldots]$), then it represents the set of all possible concrete states C (including states that may be unreachable from the initial state). From the set of all possible concrete states, all possible next states are reachable, and these are represented by the leaves of the symbolic execution tree. The symbolic state of each leaf node of the tree represents a subset of C.

There are two instantiation operations: *concretizeRoot* and *concretizeLeaf*. The effect of the first is to find a particular subset of the concrete states represented by a root node of a symbolic execution tree. The effect of the second is to find the set of concrete states represented by a particular leaf node of a symbolic execution tree.

Definition 1 (`concretizeRoot`)**.**
For any given leaf node of the tree, the set of concrete states represented by that node is reachable, with a given input, from a subset of the concrete states represented by the root node. The `concretizeRoot` instantiation operation finds that subset of concrete states for the particular leaf node of interest of a tree. The operation takes as input a tree T and a particular leaf node of interest l and returns a set of concrete states. We define it formally here.

$$\begin{aligned}\forall\ T \in \mathcal{T},\ n \in N,\ c \in C,\ & c \in \texttt{concretizeRoot}(T, n) \leftrightarrow \\ & \exists\ r, l \in T,\ m \in \mathsf{SymbolicMap} \text{ such that} \\ & r = \texttt{getroot}(T),\ l = \texttt{isleaf}(T),\ l = n, \\ & \texttt{evaluatePC}(l.\pi, m) = \text{TRUE and} \\ & c = \texttt{instantiateState}(r.\sigma, m).\end{aligned}$$

Where $\mathcal{T}$ is the set of all possible symbolic execution trees, N is the set of all possible tree nodes, and C is the set of all possible concrete states.

The definition says that a concrete state c is in the set returned by `concretizeRoot` if and only if there exists a mapping from symbols to literals that makes the path condition of the leaf node evaluate to true and, when applied to the symbolic state of the root node, produces c.

Definition 2 (`concretizeLeaf`)**.**
This instantiation operation finds the set of concrete states represented by the particular leaf node of interest of a tree.

$$\begin{aligned}\forall\ T \in \mathcal{T},\ n \in N,\ c \in C,\ & c \in \texttt{concretizeLeaf}(T, n) \leftrightarrow \\ & \exists\ l \in T,\ m \in \mathsf{SymbolicMap} \text{ such that} \\ & l = \texttt{isleaf}(T), l = n, \\ & \texttt{evaluatePC}(l.\pi, m) = \text{TRUE and} \\ & c = \texttt{instantiateState}(l.\sigma, m).\end{aligned}$$

Where $\mathcal{T}$ is the set of all possible symbolic execution trees, N is the set of all possible tree nodes, and C is the set of all possible concrete states.

The definition says that a concrete state c is in the set returned by `concretizeLeaf` if and only if there exists a mapping from symbols to literals that makes the path condition of the leaf node evaluate to true and, when applied to the symbolic state of the leaf node, produces c.

Next, we express and prove the following lemma, stating that for root r and selected leaf l in tree $T \in \mathcal{T}$, the execution of $\texttt{concretizeRoot}(r, l)$ for all valid inputs will result in a state $\in \texttt{concretizeLeaf}(T)$

Lemma 1. *$\forall s \in \textsf{SymbolicAssignment}, i \in \textsf{SymbolicInputAssignment}, c \in \overrightarrow{\textsf{Assignment}}$, and $m \in \textsf{SymbolicMapping}$, if $c \in \texttt{concretizeRoot}(t)$, then $\texttt{exec}(c, \texttt{getInput}(i, m)) \in \texttt{concretizeLeaf}(T)$, where l is a leaf of $T = \texttt{symExec}(s, i)$.*

In order to prove this, we utilize the commutativity property expressed earlier.

5.2 Three Original Requirements

This sequence of trees (*tree_list*) must satisfy these three requirements as laid out by Zhang et al.

Property Z.1 (Start in initial state). The leaf node of interest in the first tree in the sequence must be reachable from the initial, reset state.

$$c_0 \in \texttt{concretizeRoot}(\textit{tree\_list}[0]).$$

Property Z.2 (End in error state). The leaf node of interest in the last tree in the sequence must include, in the set of concrete states it represents, one of the desired error states.

$$\texttt{concretizeLeaf}(\textit{tree\_list}[n]) \cap \textit{ER} \neq \emptyset,$$

where $\textit{ER}$ represents the set of desired error states.

Property Z.3 (Stitch trees together). Consecutive trees in the sequence must form a continuous path of execution. That is, the leaf node of one tree must represent a subset of the states represented by the root node of the subsequent tree in the sequence.

$$\forall i, 0 \leq i < n,$$
$$\texttt{concretizeLeaf}(\textit{tree\_list}[i]) \subseteq \texttt{concretizeRoot}(\textit{tree\_list}[i+1]).$$

5.3 Modified Requirement

We find the above properties are not sufficient to prove correctness of the recursive strategy. We strengthen Property Z.2 by replacing it with the following.

Property Z.2' (Property Z.2 correction). The leaf node of interest in the last tree in the sequence must represent a subset of the desired error states. The leaf node can not represent any concrete state that is not an error state.

$$\texttt{concretizeLeaf}(\textit{tree\_list}[n]) \subseteq \textit{ER}.$$

5.4 Coq Representation of Recursive Strategy

Our model of the recursive strategy in Coq requires three global variables and two functions. The three global variables are defined as follows.

- $\mathit{tree\_list} : \overrightarrow{\mathsf{SymExecTree}}$ is a list of symbolic execution trees. It represents the list returned by the recursive strategy.
- $\mathit{init\_conc\_state} : \overrightarrow{\mathsf{Assignment}}$ is a concrete state. It represents the initial state of the processor.
- $\mathit{error\_states} \subseteq \{c \mid \overrightarrow{\mathsf{Assignment}}\}$ is a set of concrete states. It represents the error states of the processor.

The two functions are defined as follows.

- $\texttt{executeTreeList}(t : \overrightarrow{\mathsf{SymExecTree}}) : \overrightarrow{\mathsf{Assignment}}$ takes a list of symbolic execution trees and returns a concrete state.
- $\texttt{getInput}(\pi : \mathsf{PathCondition}) : \overrightarrow{\mathsf{InputAssignment}}$

The function `executeTreeList` uses the path conditions in the leaf nodes of interest of the sequence of trees to find a sequence of concrete input values, and then executes the model concretely to arrive at a final concrete state.

Additionally, we define the abstract method, `getInput` which finds a mapping that does not violate a given symbolic state's path constraint, applies that mapping, and returns a concrete input.

6 Proof Strategy

The claim made by Zhang and Sturton is that a sequence of symbolic execution trees that abides by the three requirements will yield a set of sequences of concrete input values, each of which will take the hardware module from the reset state to an error state.

We prove that if a sequence of trees produced by our abstract Coq model of symbolic execution, which is based on the three King properties (Properties K.1, K.2, K.3), abides by Properties Z.1, Z.2', and Z.3, then it does yield a set of sequences of concrete inputs that take the abstract Coq model of concrete execution from the reset state to an error state.

Theorem 1 (Correctness of Recursive Strategy). *Consider a sequence of n trees $\mathit{tree\_list} = T_0, T_1, \ldots, T_{n-1}$, where each tree satisfies King Properties K.1-K.3. If tree_list satisfies the requirements Z.1, Z.2', Z.3, then* $\texttt{executeTreeList}(\mathit{tree\_list}) \in \mathit{error\_states}$.

In order to prove this, we first prove the following theorem:

Theorem 2 (Execution ends in leaf of the last tree). $\texttt{executeTreeList}(\mathit{tree\_list}) \in \texttt{concretizeLeaf}(T)$*, where T is the last element of tree_list.*

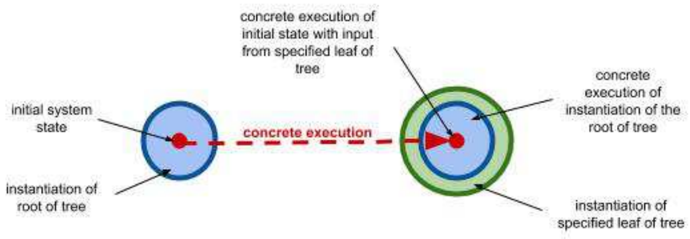

concrete execution of initial state with input from specified leaf of tree
initial system state
concrete execution
concrete execution of instantiation of the root of tree
instantiation of root of tree
instantiation of specified leaf of tree

Fig. 2: Visual depiction of the base case of the proof.

We then show that `concretizeLeaf`$(T) \in \mathit{error\_states}$, giving us our result.

We prove Theorem 2 by induction. For our base case, we show that if the list only contains one tree, execution of that tree's root node with input specified by a selected leaf will result in an element of `concretizeLeaf`(T).

In other words, as depicted in Figure 2, we show that the initial state is an element of `concretizeRoot`(T) of the tree T, and that concretely executing any element of `concretizeRoot`(T) will result n an element inside `concretizeLeaf`(T).

For our inductive step, as depicted in Figures 3 and 4, we show that execution of each root with inputs from each specified leaf in a tree list of size n will result in an element of `concretizeLeaf`(T_n).

Our inductive hypothesis is that `executeTreeList`$(\mathit{tree\_list}') \in$ `concretizeLeaf`(T_{n-1}), where $\mathit{tree\_list}'$ is a $\mathit{tree\_list}$ with the last element removed. We then show that `concretizeLeaf`$(T_{n-1}) \subseteq$ `concretizeRoot`(T_n), and therefore the concrete execution of any element in `concretizeLeaf`(T_{n-1}) is in the set of of the concrete execution of any element in `concretizeRoot`(T_n). Next, we show that `concretizeRoot`$(T_n) \subseteq$ `concretizeLeaf`(T_n), giving us our result.

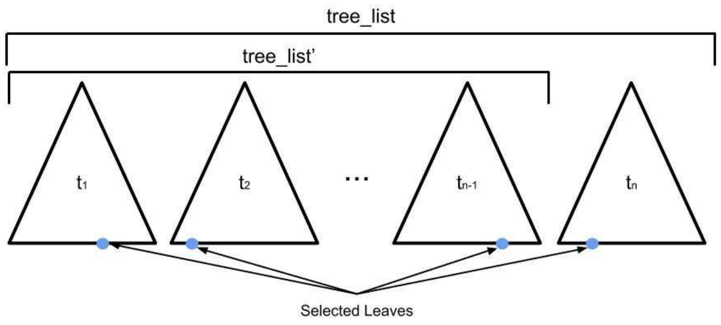


Fig. 3: List of trees of length n.

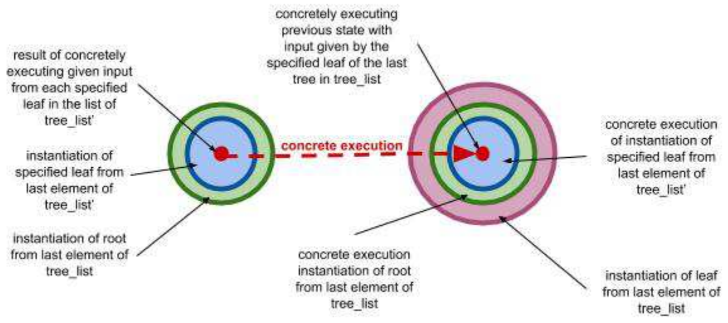

result of concretely executing given input from each specified leaf in the list of tree_list'
instantiation of specified leaf from last element of tree_list'
instantiation of root from last element of tree_list
concretely executing previous state with input given by the specified leaf of the last tree in tree_list
concrete execution
concrete execution of instantiation of specified leaf from last element of tree_list'
concrete execution instantiation of root from last element of tree_list
instantiation of leaf from last element of tree_list

Fig. 4: Visual depiction of the inductive step of the proof.

The reason we need Z.2' is because Z.2 is not sufficient. This is because if $\texttt{executeTreeList}(tree\_list) \in \texttt{concretizeLeaf}(\texttt{lastElement}(tree\_list))$ and $\texttt{concretizeLeaf}(\texttt{lastElement}(tree\_list)) \cap error\_states \neq \{\}$, we could get the case where $\texttt{executeTreeList}(tree\_list) \notin error\_states$, as shown in Figure 5.

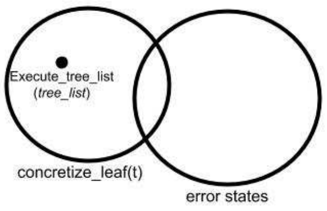


Fig. 5: Example of Property 2 not being sufficient to show $\texttt{executeTreeList}(tree\_list) \in error\_states$.

7 Conclusion

In conclusion, we prove the correctness of a recursive symbolic execution strategy (with one alteration) while also demonstrating the usefulness of formal verification by finding a flaw in the strategy's requirements. We also provide a formal framework of symbolic execution in Coq, allowing for future verification of other symbolic execution tools and strategies.

References

1. The Coq proof assistant. https://coq.inria.fr/, INRIA.

2. Anand, S., Păsăreanu, C.S., Visser, W.: Jpf–se: A symbolic execution extension to java pathfinder. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 134–138. Springer (2007)
3. Arusoaie, A., Lucanu, D., Rusu, V.: A generic framework for symbolic execution. Tech. Rep. RR-8189, Inria (2014)
4. Arusoaie, A., Lucanu, D., Rusu, V.: Symbolic execution based on language transformation. Computer Languages, Systems & Structures **44**, 48–71 (2015)
5. Cadar, C., Dunbar, D., Engler, D.R., et al.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. vol. 8, pp. 209–224 (2008)
6. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing mayhem on binary code. In: Security and Privacy (SP), 2012 IEEE Symposium on. pp. 380–394. IEEE (2012)
7. Chandra, S., Fink, S.J., Sridharan, M.: Snugglebug: a powerful approach to weakest preconditions. In: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation. pp. 363–374. ACM (2009)
8. Charreteur, F., Gotlieb, A.: Constraint-based test input generation for java bytecode. In: Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering. pp. 131–140. ACM (2010)
9. Chi, A., Cochran, R., Nesfield, M., Reiter, M.K., Sturton, C.: Server-side verification of client behavior in cryptographic protocols. arXiv preprint arXiv:1603.04085 (2016)
10. Davidson, D., Moench, B., Ristenpart, T., Jha, S.: Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In: USENIX Security Symposium. pp. 463–478 (2013)
11. Dinges, P., Agha, G.: Targeted test input generation using symbolic-concrete backward execution. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering. pp. 31–36. ACM (2004)
12. Do, Q.H., Kamburjan, E., Wasser, N.: Towards fully automatic logic-based information flow analysis: An electronic-voting case study. In: International Conference on Principles of Security and Trust. pp. 97–115. Springer (2016)
13. King, J.C.: Symbolic Execution and Program Testing. Communications of the ACM **19**(7), 385–394 (Jul 1976), http://doi.acm.org/10.1145/360248.360252
14. Liu, L., Vasudevan, S.: STAR: Generating input vectors for design validation by static analysis of RTL. In: IEEE International Workshop on High Level Design Validation and Test Workshop. pp. 32–37. IEEE (2009)
15. Lucanu, D., Rusu, V., Arusoaie, A.: A generic framework for symbolic execution: A coinductive approach. Journal of Symbolic Computation **80**, 125–163 (2017)
16. Ma, K.K., Yit Phang, K., Foster, J.S., Hicks, M.: Directed symbolic execution. In: Yahav, E. (ed.) Proceedings of the 18th International Static Analysis Symposium. pp. 95–111. Springer Berlin Heidelberg, Berlin, Heidelberg (2011), https://doi.org/10.1007/978-3-642-23702-7_11
17. Mukherjee, R., Kroening, D., Melham, T.: Hardware Verification using Software Analyzers. In: Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI). IEEE (2015)
18. Noller, Y., Nguyen, H.L., Tang, M., Kehrer, T.: Shadow symbolic execution with java pathfinder. SIGSOFT Softw. Eng. Notes **42**(4), 1–5 (Jan 2018). https://doi.org/10.1145/3149485.3149492, http://doi.acm.org/10.1145/3149485.3149492

19. Păsăreanu, C.S., Rungta, N.: Symbolic pathfinder: symbolic execution of java bytecode. In: Proceedings of the IEEE/ACM international conference on Automated software engineering. pp. 179–180. ACM (2010)
20. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: Security and privacy (SP), 2010 IEEE symposium on. pp. 317–331. IEEE (2010)
21. The Coq Development Team: The Coq reference manual. INRIA (8.8.2) (2018)
22. Zhang, R., Deutschbein, C., Huang, P., Sturton, C.: End-to-end automated exploit generation for validating the security of processor designs. In: Proceedings of the International Symposium on Microarchitecture (MICRO). IEEE/ACM (2018)
23. Zhang, R., Sturton, C.: A recursive strategy for symbolic execution to find exploits in hardware designs. In: Proceedings of the International Workshop on Formal Methods and Security (FMS). ACM (2018)