Yippelia: Triggering Deep Property Violations in Hardware Designs through Symbolic Execution

By
Martin Meng
Senior Honors Thesis
Department of Computer Science
The University of North Carolina at Chapel Hill
Advisor: Cynthia Sturton
Second Reader: Parasara Sridhar Duggirala

***Abstract*—We in Yippelia attempt to automatically identify deep bugs in hardware designs by symbolically exploring hardware designs for one clock cycle and then stitching the generated simple paths to form a multi-cycle path from the reset state to the buggy state. Compared to a state-of-the-art symbolic execution engine, Yippelia has an average speedup of at least four orders of magnitude on finding deep bugs on the up-down counter hardware design.**

I. Introduction

This thesis presents Yippelia, a hardware-oriented symbolic execution framework that automatically finds property-violations in hardware designs. Given a hardware design and a property, Yippelia can find a property-violation and generate a sequence of instructions which triggers the property-violation by taking the hardware design from the reset state to the property-violated state.

Hardware is the fundamental trust of computer systems. Recent attacks such as Spectre [1] and Meltdown attacks [2] have demonstrated that attackers can utilize vulnerabilities in hardware to compromise the security of the overall systems even though softwares and systems are proven to be safe. Attackers can exploit the hardware security vulnerabilities to gain the credential information of the users and even threaten people's physical safety in the case of embedded and cyber-physical systems. Moreover, while most software bugs can be resolved by software patches or the release of a new version of the software, fixing some hardware bugs requires recalling and replacing all products, causing a great amount of loss of money – the Pentium FDIV bug [3] which is rooted in the floating-point unit of Intel's Pentium processor caused Intel half a billion dollars to recall and replace all shipped products. Therefore, it is important to eliminate hardware bugs in the early stages of the hardware development.

The research community responds to this challenge with various approaches, one of which is assertion-based verification. Assertion-based verifications formally prove that a design satisfies some security-critical properties. In the hardware security community, works have been done to automatically generate security properties from design specifications [4], [5], [6], [7]. However, state-of-the-art verifiers that are designed to prove these properties are limited by their poor scalability to large and complex real-world designs. We respond to this challenge by presenting Yippelia, a light-weight verifier that can evaluate complex hardware designs efficiently.

Yippelia uses symbolic execution to systematically explore hardware designs and identify property violations. Compared to model checking, symbolic execution sacrifices rigorousness for scalability. Similar to other hardware-oriented symbolic execution engines such as Coppelia [8], Yippelia is sound but not complete. If Yippelia finds a property violation, then the violation indeed exists. On the other hand, Yippelia cannot give any guarantee that the design is bug-free even if Yippelia does not find any property violation. In other words, Yippelia can trigger, but cannot prove the absence of, property violations in hardware designs. The sacrifice in completeness provides, with symbolic execution, good scalability on covering complex designs in return.

Even though symbolic execution is one of the most scalable formal methods, the application of symbolic execution is still limited by the complexity of the hardware designs. This limitation is due to the severe path explosion problem in the context of hardware. Because a processor can run in an infinite number of clock cycles, akin to an infinite while loop in software, the number of paths grows exponentially as the number of clock cycles increases towards infinity. Therefore, the application range of normal symbolic execution engines is limited to small-size hardware designs [9].

To handle symbolic execution's scalability problem in hardware, Zhang et al. presented Coppelia [8], an end-to-end tool that automatically generates exploit programs for processor designs utilizing a backward search strategy that was proposed in [10]. However, Coppelia performs symbolic execution on every clock cycle, so Coppelia's scalability is still limited by the number of clock cycles a trigger program consists of. We in Yippelia attempt to improve the scalability of hardware-oriented symbolic execution engines by performing symbolic execution on hardware designs only once and for one clock cycle. Then, Yippelia reuses the results of the symbolic execution to build a path from a reset state to a property-violation state in the hardware design. The key contributions of this thesis are as follows:

- We propose a framework to apply symbolic execution

to hardware designs. This framework saves execution time on finding deep property violations in hardware by running symbolic execution only once on the design for one clock cycle. The framework incorporates techniques from the AI community to stitch a multi-cycle path from the reset state of the hardware design to a property-violated state. Combining these two aspects, our framework can increase the efficacy and scalability of finding deep property-violations in hardware.
- We develop Yippelia, a hardware-oriented symbolic execution tool that implements the framework described above. Compared to KLEE [11], a state-of-the-art symbolic execution engine, Yippelia has an average speedup of at least four orders of magnitude on finding deep property-violations on an up-down counter design.

The rest of this thesis is organized as follows: we first introduce an up-down counter design as the motivating example in Section II. Section III presents Yippelia's design. Section IV describes Yippelia's implementation. We evaluate Yippelia's performance in Section V. We then discuss threats to Yippelia's validity as well as the future work from Yippelia in Section VI. Section VII presents a review of research topics that are relevant to Yippelia in Section VII. Finally, Section VIII concludes the thesis.

II. Example

In this section, we introduce the up-down counter, a type of hardware designs that can either counts up or counts down in one clock cycle. Throughout this paper, we use the up-down counter as an example to demonstrate the key ideas of Yippelia.

An up-down counter is a type of counter in which a register called *value* either increments or decrements based on an input signal called *inst*. The range of the register *value* is between 0 and a pre-defined maximum value. When the counter reaches the maximum value and still counts up, it overflows to 0; similarly, when the counter reaches 0 and still counts down, it overflows to the maximum value.

Figure 1 shows the schematic design of the up-down counter and Figure 2 shows the corresponding RTL design implemented in SystemVerilog [12]. The up-down counter takes as inputs a *clock* signal, a *reset* signal, and an *inst* signal. It displays a 32-bit *value* signal as the output. Internally, the counter maintains a 32-bit register called *internalvalue*. Whenever *reset* is 1, *internalvalue* is reset to be 0. Otherwise, at each clock cycle, *internalvalue* increments if *inst* is 0 and decrements if *inst* is 1. The output *value* always displays the current value of *internalvalue*.

We use the up-down counter as the example to showcase our ideas because the design is both simple and nontrivial. The design is simple in that it only has two program paths within one clock cycle: the counter either increments or decrements. On the other hand, the design is nontrivial because 1) the design consists of sequential logic in that the register *value* does not only depends on the current input but also depends on its value in the previous clock cycle, and 2) the design consists

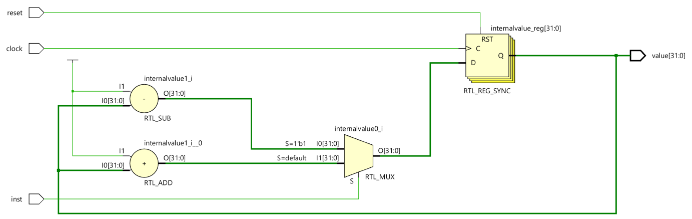

Fig. 1. up-down Counter's Schematic Design

```
module updowncounter(
    input wire clock,
    input wire reset,
    input wire inst,     // 1-bit instruction.
                         // 0 <-> count up
                         // 1 <-> count down
    output wire [31:0] value
    );

    logic [31:0] internalvalue;

    always_ff @ (posedge clock) begin
        internalvalue <= reset ? 32'b 0:
                                    inst ? internalvalue - 1 :
                                           internalvalue + 1;
    end

    assign value = internalvalue;


endmodule
```

Fig. 2. up-down Counter's RTL Design

of more than one path in one clock cycle, so the number of program paths grows exponentially with respect to the number of clock cycles. The fact that the design has a stateful register, i.e. the register *value*, enables us to use symbolic execution to check properties across multiple clock cycles. The exponential grow of the program paths with respect to the number of clock cycles reveals the main challenge for symbolic execution in both hardware and software domain, the path explosion problem [13]. Therefore, the simplicity and the non-trivialness of the up-down counter allows us to highlight the key ideas in Yippelia.

III. Design

A. Overview

Yippelia takes an input a hardware design at the register transfer level (RTL) and a set of assertions representing security-critical properties. If Yippelia does not find a property-violation in the design, Yippelia issues a certificate. Otherwise, Yippelia outputs a sequence of instructions that trigger the assertion violations in the hardware design. Figure 3 shows Yippelia's high-level input-output workflow.

Yippelia consists of a 4-stage pipeline: *preprocessing*, *symbolic execution*, *optimization*, and *search*, as shown in Figure 4. Yippelia first translates the RTL hardware designs to C++ in the *preprocessing* stage. Then, Yippelia performs a one-cycle symbolic exploration to gather all possible atomic

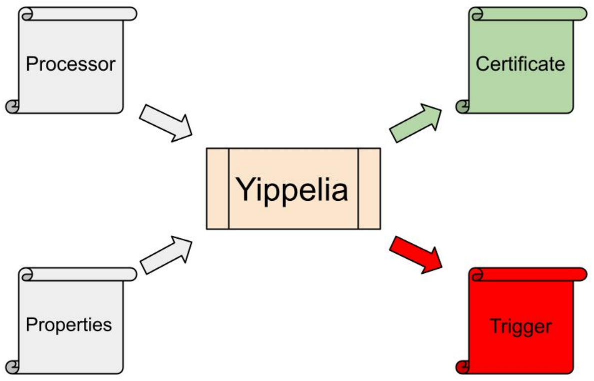

Fig. 3. The Input-output Workflow of Yippelia

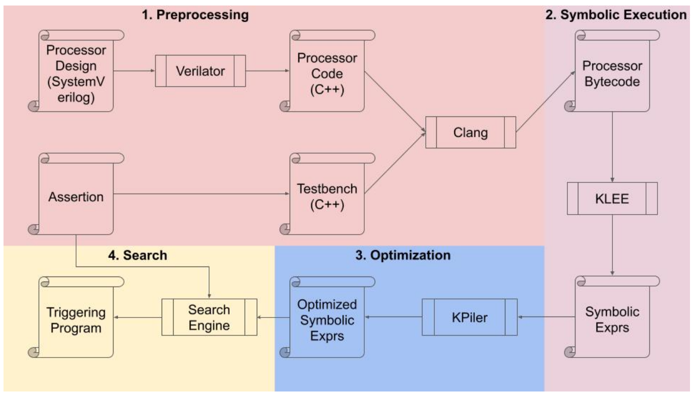

Fig. 4. The Architecture of Yippelia

paths of the designs within one clock cycle in the *symbolic execution* stage. To handle the case in which the symbolic paths are too long for SMT solvers to handle efficiently, we in the *optimization* stage constructs a compiler called *KPiler* to shorten the symbolic paths with equivalent semantic meanings. Lastly, Yippelia formulates the problem of finding a multi-cycle property violation as a search problem and uses our search engine to solve it in the *search* stage. In the following sections, we describe each stage of the pipeline in detail.

B. Preprocessing

To begin with, Yippelia translates the RTL hardware design implemented in SystemVerilog [12] to a piece of code in C++ [14] that is semantically equivalent to the RTL hardware design. The reason to translate the hardware design to C++ is that we can utilize KLEE [11], a popular software symbolic execution engine, to run symbolic simulation on the C++ code (Section III-C).

To translate the hardware design to C++, we utilize Verilator [15], an open-source, matured hardware simulator that translate hardware designs specified in hardware description language (HDL) such as Verilog and SystemVerilog to C++ or SystemC. However, Verilator does not perform a direct translation because of the intrinsic difference of hardware and software: while the hardware is parallel, the software is sequential by default. Because blocking statements in HDL are originally sequential, Verilator is able to perform a direct translation. However, for non-blocking statements which perform parallel assignments, because instructions in C++ have a temporal order, Verilator needs to introduce temporary variables in C++ so that assignments in the same clock cycle will not influence one another. Algorithm 1 illustrates this strategy. Verilator also performs other optimizations such as eliminating dead codes and matching width lengths during the translation.

Algorithm 1 Parallel assignments

```
1: procedure PARALLEL-ASSIGN($x, y$)
2:     $temp\_x \leftarrow y + 10$        ▷ assign temporary variables
3:     $temp\_y \leftarrow x + 10$
4:     $x \leftarrow temp\_x$        ▷ load the new values back to x, y
5:     $y \leftarrow temp\_y$
6:     return $x, y$
```

The resulting C++ code consists of one C++ class for each module in the RTL design. In addition, Verilator requires users to attach a C++ wrapper file to instantiate the top level module of the design during the simulation. This wrapper file supports linking to external libraries, so we use this file as a chance to run symbolic execution engine and perform a symbolic simulation on the design (Section III-C). In the wrapper file, we interact with the top level module via a function `eval()`. `eval()` propagates inputs and calls necessary functions to simulation the clock transition at the boundary. Therefore, two calls to `eval()` with opposite *clock* values correspond to the simulation of the design for one clock cycle.

Threats to Validity. Although Yippelia is sound, Yippelia is not complete with respect to finding vulnerability by adopting this preprocessing scheme. As indicated by Zhang et al. [10], Verilator may eliminate bugs during the compilation of the hardware design, because the compilation is not always faithful due to compiler optimization. As a result, even though Yippelia is sound in that every bug that is found by Yippelia is a true property violation, Yippelia is incomplete in that it may miss some bugs in the hardware designs. To mitigate this threat, our group is developing native symbolic execution methods on hardware that apply symbolic execution directly to the RTL hardware designs. We encourage the community to further explore this direction to circumvent the incompleteness problem.

C. Symbolic Execution

In the symbolic execution stage, Yippelia takes the output code from Verilator and performs symbolic execution on the hardware design for one clock cycle. We use KLEE [11], a matured symbolic execution engine that is widely used in the software domain, to perform the one-cycle symbolic exploration of the hardware design. Prior to running symbolic simulation, Yippelia concretely simulates the hardware design in reset state for one cycle to obtain the root state of the hardware design. Running the hardware code in reset state also ensures that the symbolic execution starts with all registers initialized to their default values. In the case of the up-down counter, Yippelia sets the *reset* input to be 1 and simulates the design for one clock cycle.

With all registers initialized to their reset values, Yippelia runs KLEE to perform a one-cycle symbolic execution on the hardware design. After finishing each path during the symbolic executions, Yippelia asks KLEE to print out the symbolic expressions of a set of predefined signals that includes all register signals in the target property. Each set of symbolic expressions represents one symbolic atomic path from one symbolic state of the hardware design to another in one clock cycle. In the case of the up-down counter, KLEE first makes the register *value* in the parent state symbolic, e.g. $\textit{value} = \alpha$. Then, after KLEE finishes the one-cycle symbolic execution, Yippelia gathers one symbolic expression for each path: $\textit{value} = \alpha + 1$ for the path where *value* increments and $\textit{value} = \alpha - 1$ for the path where *value* decrements. Figure 5 shows the symbolic tree representing the one-cycle symbolic exploration on the up-down counter. Table I shows the two symbolic expressions representing two symbolic atomic paths of the up-down counter. The symbolic atomic paths, which are represented by the sets of symbolic expressions, are the building blocks for all longer paths spanning multiple clock cycles. We describe how Yippelia utilizes these building blocks to form a path from the reset state to the property-violated state of a hardware design in Section III-E.

Within the symbolic execution stage, Yippelia performs a sub-stage called *fast validation* to make sure that property is not violated within one clock cycle. If the property is violated, then Yippelia will terminate and return as result the single instruction corresponding to the atomic path that leads to the assertion violation. Otherwise, Yippelia continues by sending the collection of symbolic atomic paths to the optimization stage (Section III-D) and then to the search engine that is described in Section III-E to form a path that spans several clock cycles and ends at an assertion-violated state. The reason behind this sanity check is that one-cycle symbolic execution is able to find those one-cycle property violations, so Yippelia does not need to spend time on the later stages. Also, the sanity check ensures that the search engine (Section III-E) only searches for multi-cycle property-violations, so our search engine does not need to handle corner cases caused by one-cycle property violation.

Threats to Validity: Yippelia relies on KLEE to perform the symbolic exploration of hardware designs within one clock cycle. Although KLEE is a matured, well-maintained symbolic execution engine on the software domain, the symbolic expressions that are printed out by KLEE are lengthy. As a result, the query formed by the symbolic expressions often require a nontrivial amount of time to be solved by an SMT solver. To mitigate this threat, we performed optimizations to shorten the symbolic expressions (Section III-D). We leave as future work to explore other open-source symbolic execution frameworks such as angr [16], [17], [18].

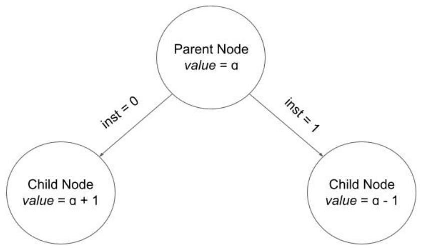

Fig. 5. Symbolic Tree for the Up-down Counter in One Cycle

TABLE I
TWO SYMBOLIC EXPRESSIONS REPRESENTING TWO ATOMIC PATHS OF THE UP-DOWN COUNTER.

Instr	Atomic path	Symbolic expression
0	value $= \alpha + 1$	(Eq leaf_value (Add w32 1 (ReadLSB w32 0 value)))
1	value $= \alpha - 1$	(Eq leaf_value (Add w32 4294967295 (ReadLSB w32 0 value)))

D. Optimization

In this section, we describe our optimization methods to shorten the length of symbolic expressions that are produced by the symbolic execution stage (Section III-C). While the symbolic expressions produced by KLEE for the up-down counter example has reasonable sizes (231 bytes on average), the symbolic expressions for the complex real-world hardware designs are significantly large: for a single-cycle MIPS processor, the average size of a set of symbolic expressions is 14187 bytes. After we formed queries based on the symbolic expressions (see Section III-E3 for the details of forming queries), the longest query takes up to 48 minutes to be solved by the query solver (Table X).

The aim of this stage is to shorten the length of the symbolic expressions produced by the previous stage so that Yippelia can scale to larger hardware designs. We perform two optimizations to achieve this goal.

1) Optimization 1: The first optimization is on the query level: we delete unused symbolic expressions in a query to make the query shorter. We observed that over all sets of symbolic expressions, the expressions of the input signals do not change semantically. The reason is that the symbolic execution stage (Section III-C) only runs symbolic execution for one clock cycle, and the input signals stay stable within one clock cycle. Because the input signals do not change, the symbolic expressions corresponding to the input signals do not semantically impose any additional constraints on the query. Therefore, we can safely delete in each query the symbolic expressions corresponding to the input signals. The new query without the symbolic expressions with respect to the input signals is semantically equivalent to the original query, i.e. the new query is valid if and only if the original query is valid.

2) Optimization 2: While the first optimization (Section III-D1) is on the query level to delete unused symbolic expressions, the second optimization is on the expression level: it aims to shorten a single symbolic expression by deleting unused parts of a symbolic expression.

This optimization is based on a common pattern in the symbolic expressions: a significantly large portion of array updates are unused. For example, let us consider the following symbolic expression: (Read w8 10 [10=0, 5=0, 0=0]@ const_arr7). Because the expression only reads the element at index 10 of the updated version of a constant array, we can safely delete the updates to index 5 and index 0. In practice, these unused updated statements are often very lengthy, so the query can be shortened significantly if all unused update statements are removed. As a result, the execution time to solve these optimized queries can be significantly reduced.

To automate the step of removing unused update statements for symbolic expressions, we construct a compiler called KPiler for expressions in the KQuery language, a language that is used by KLEE to represent constraint expressions [19]. KPiler takes in a query in the format of KQuery language and produces an optimized query that is semantically equivalent to the original query. The optimized query is equivalent to the original query in that the optimized query is valid if and only if the original query is valid. The length of the optimized query is the same or shorter than the original query so the time for the query solver to solve the optimized query is expected to be shorter.

The workflow of KPiler is shown in Figure 6. KPiler consists of three parts: the front-end, the analyzer, and the back-end.

a) Front-end: In the front-end, we first developed a program called KLexer to tokenize the expressions. Then, we collected the grammar of KQuery both from the partial specification in [19] and from the symbolic expressions that are generated by KLEE. Figure 7 summarizes the grammar for KQuery. Based on this grammar, we created a parser called KParser. KParser generates an abstract-syntax tree (AST) for each expression in the query. Figure 8 shows our design of the AST for KQuery. In addition, KParser also generates a set of ASTs for declarations that are appeared in the expression. With this set of declaration nodes, we are free to delete any unused update statements without worrying about deleting any declaration within the unused update statements.

b) Analyzer: In the analyzer part, we developed three analysis tools for the purposes of debugging and experimenting: KAST_Display, KAST_Depth, and KAST_Size. KAST_Display visualizes an AST node by printing out the current node and recursively printing out its children nodes with proper indentations. KAST_Depth and KAST_Size calculates the depth and the number of nodes of the AST, respectively. All three analysis tools implement a common interface called KVisitor. KVisitor systematically visits the

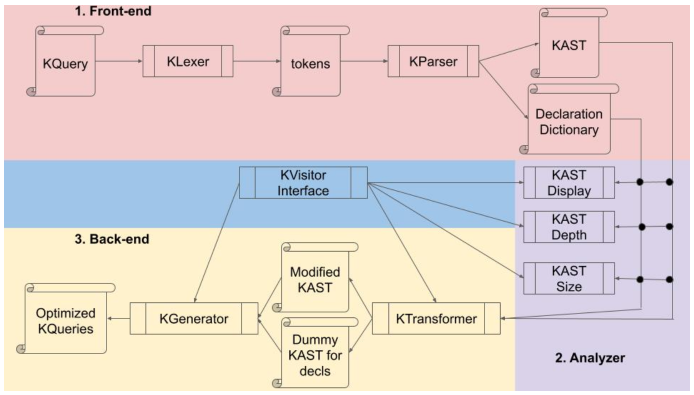

Fig. 6. The architecture of KPiler, a compiler for KQuery that is used in KLEE

whole AST through a visiting method for each type of the AST node.

c) Back-end: In the back-end, we developed a program called KTransformer which performs the actual optimization step of deleting unused update statements. KTransformer takes as inputs an AST node corresponding to the target symbolic expression and a set of AST nodes corresponding to the declarations that are appeared in the target symbolic expression. KTransformer recursively analyzes each AST node (the AST node for the symbolic expression and the AST nodes for the declarations) and deletes the unused update statements within them. KTransformer also *transforms* the modified AST nodes for the declarations (i.e. DeclExpr in Figure 8) to comparison AST nodes (i.e. CompExpr in Figure 8) so that these declarations can be included in the optimized query.

Lastly, we developed a program called KGenerator which takes as input an AST node and *generates* the corresponding symbolic expressions in the format of KQuery. KGenerator follows the grammar of KQuery (Figure 7) strictly to faithfully translate the AST nodes to expressions in KQuery. Same as the analysis tools in Section III-D2b, KTransformer and KGenerator also implement the interface KVisitor to recursively, systematically visit all parts of an AST node.

3) Threats to Validity: We consider potential threats to the validity for our optimization methods. First, our modified version of KQuery grammar is complete but not sound. We sacrifice soundness for the ease of implementation. As a result, our KPiler tool can accept all correct KQuery expressions, but it may also accept some incorrect expressions. We mitigate this threat by assuming that the symbolic expressions produced by KLEE are correct with respect to the grammar. Within each step of KPiler's workflow, we do not perform any operation that makes a correct expression to be incorrect. Therefore, the output of KPiler should still be correct KQuery expressions even though our grammar is not sound. Second, our KPiler tool may consist of implementation bugs. We mitigated this threat by testing our tool against 162 long queries with an average query size of 14187 bytes. We also performed thorough code review for our tool.

KQuery Grammar

⟨*expression*⟩ ::= ⟨*identifier*⟩ ':' ⟨*expression*⟩
 | ⟨*identifier*⟩
 | (⟨*number*⟩ | '(' ⟨*type*⟩ ⟨*number*⟩ ')')
 | '(' ⟨*arithmetic-expr-kind*⟩ ⟨*type*⟩ ⟨*expression*⟩ ⟨*expression*⟩ ')'
 | '(' 'Not' [⟨*type*⟩] ⟨*expression*⟩ ')'
 | '(' ⟨*bitwise-expr-kind*⟩ ⟨*type*⟩ ⟨*expression*⟩ ⟨*expression*⟩ ')'
 | '(' ⟨*comparison-expr-kind*⟩ [⟨*type*⟩] ⟨*expression*⟩ ⟨*expression*⟩ ')'
 | '(' 'Concat' [⟨*type*⟩] ⟨*expression*⟩ ⟨*expression*⟩ ')'
 | '(' 'Extract' ⟨*type*⟩ ⟨*number*⟩ ⟨*expression*⟩ ')'
 | '(' 'ZExt' ⟨*type*⟩ ⟨*expression*⟩ ')'
 | '(' 'SExt' ⟨*type*⟩ ⟨*expression*⟩ ')'
 | '(' 'Read> <type> <expression> <version> ')'
 | '(' 'Select' ⟨*type*⟩ ⟨*expression*⟩ ⟨*expression*⟩ ⟨*expression*⟩ ')'
 | '(' 'Neg' [⟨*type*⟩] ⟨*expression*⟩ ')'
 | '(' 'ReadLSB' ⟨*type*⟩ ⟨*expression*⟩ ⟨*version*⟩ ')'
 | '(' 'ReadMSB' ⟨*type*⟩ ⟨*expression*⟩ ⟨*version*⟩ ')'

⟨*identifier*⟩ ::= '[a-zA-Z_][a-zA-Z0-9._]*'

⟨*number*⟩ ::= 'true' | 'false' | ⟨*signed-constant*⟩

⟨*signed-constant*⟩ ::= ['+' | '-'] (⟨*dec-constant*⟩ | ⟨*bin-constant*⟩ | ⟨*oct-constant*⟩
 | ⟨*hex-constant*⟩)

⟨*dec-constant*⟩ ::= '[0-9_]+'

⟨*bin-constant*⟩ ::= '0b[01_]+'

⟨*oct-constant*⟩ ::= '0o[0-7_]+'

⟨*hex-constant*⟩ ::= '0x[0-9a-fA-F_]+'

⟨*type*⟩ ::= 'w[0-9]+'

⟨*arithmetic-expr-kind*⟩ ::= ('Add' | 'Sub' | 'Mul' | 'UDiv' | 'URem' | 'SDiv' | 'SRem')

⟨*bitwise-expr-kind*⟩ ::= ('And' | 'Or' | 'Xor' | 'Shl' | 'LShr' | 'AShr')

⟨*comparison-expr-kind*⟩ ::= ('Eq' | 'Ne' | 'Ult' | 'Ule' | 'Ugt' | 'Uge' | 'Slt' |
 'Sle' | 'Sgt' | 'Sge')

⟨*version*⟩ ::= ⟨*identifier*⟩ ':' ⟨*version*⟩
 | ⟨*identifier*⟩
 | '[' [⟨*update-list*⟩] ']' '@' ⟨*version*⟩

⟨*update-list*⟩ ::= ⟨*update-stmt*⟩ [',' ⟨*update-list*⟩]

⟨*update-stmt*⟩ ::= ⟨*expression*⟩ '=' ⟨*expression*⟩

Fig. 7. The modified grammar of KQuery

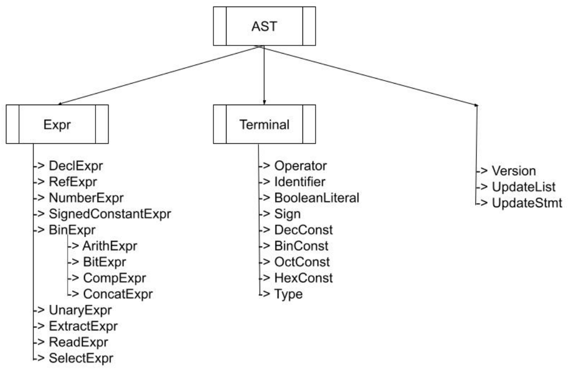

Fig. 8. The inherence hierarchy of AST classes. An arrow from class A to class B indicates that class A is the parent class of class B; i.e. class B inherits from class A.

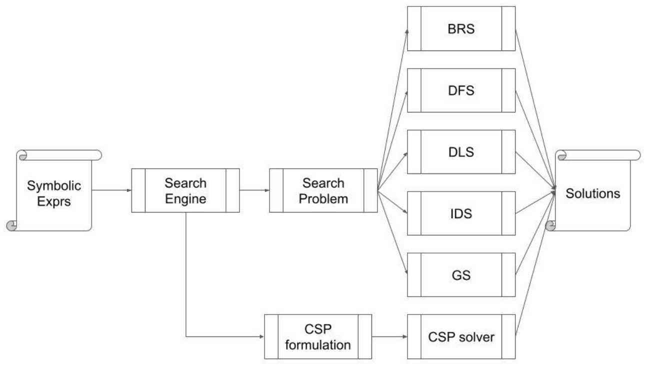

Fig. 9. The Workflow of the Search Engine.

E. Search

1) Overview: In the search stage, we use the symbolic atomic paths as the building blocks to form a multi-cycle path that brings the hardware design from a reset state to a property-violated state. The symbolic atomic paths are represented by queries that are formed by the symbolic expressions which are produced in the symbolic execution stage (Section III-C) and then optimized in the optimization stage (Section III-E3 for details in query formulation).

To form a path from the reset state to the property-violated state, we explore various techniques from the artificial intelligence research community. We first formulate the problem as a search problem. We then explore various search algorithms to solve this search problem: Depth-first Search (DFS), Depth-limited Search (DLS), Iterative Deepening Search (IDS), and Greedy Search (see Chapter 3 in [20]). We also formulate this problem as a Constraint Satisfaction Problem (CSP, see Chapter 4 in [20]), but this formulation is not yet scalable for complex hardware designs. Figure 9 presents the high-level overview of the search engine.

2) Search Problem Formulation: We formulate the problem of finding a property violation in a hardware design as a search problem. In our formulation, the states are the sets of values that are assigned to the registers in the processor design. The initial state of the search problem is the set of register values in the reset state of the processor design. We retrieve the initial state by setting the reset signal to be 1 and concretely simulates the hardware design for one clock cycle. The initial state of the up-down counter is the state such that the register *value* $= 0$. The goal state of the search problem is a state in which a property that is provided by the user is violated. If there is a property for the up-down counter saying that the register *value* $\neq 10$, then the state in which *value* $= 10$ is the goal state. In more complex hardware designs, we state the goal state implicitly with a goal test which checks if the current state satisfies the property.

The operators in our search problem formulation are the atomic paths that brings the design from one state to another immediate state, i.e., another state that is reachable from the original state within one clock cycle. Each atomic path cor-

TABLE II
SEARCH PROBLEM FORMULATION

States	Sets of register values
Initial states	Reset states of the hardware designs
Goal states	Property-violated states
Operators	Atomic paths
Path cost function	1 (uniform cost function)
Solutions	Multi-cycle paths from reset states to property-violated states
Search space	∞

TABLE III
QUERY CONSTRAINTS

Description	Constraint
Current State	(Eq w32 0 (ReadLSB w32 0 value))
Symbolic Expression	(Eq w32 (ReadLSB w32 0 leaf_value) (Add w32 1 (ReadLSB w32 0 value)))
Path Constraint	(Eq 0 (ReadLSB w32 0 inst)

responds to a set of instructions that satisfy a path constraint. We obtain these atomic paths from the symbolic execution stage (Section III-C) by combining the current states, the symbolic expressions, and the path constraints to form queries in the KQuery language [19]. We solve these queries by Kleaver [21], a query solver that is used by KLEE [11]. We also use Z3 [22] as the back-end SMT solver of Kleaver. The solution of the query is a new state that is the result of applying the atomic path to the current state.

We also consider a path cost function for each path. Currently, we assign a uniform path cost for each atomic path because we do not want to have preference over each atomic path, but in the future we will assign the path cost of each atomic path to be proportional to the time an SMT solver takes to solve the query corresponding to the atomic path. In this way, our search engine will prefer to explore paths with shorter query.

Moreover, the solution of our search problem is a multiple-clock-cycle path from one of the reset states of the hardware designs to a property-violated state. We don't consider single-clock-cycle paths (i.e. the atomic paths) as our solutions because they are handled by KLEE [11] in the symbolic execution stage (Section III-C). Lastly, the state space of our search problem is infinite because hardware designs can run an infinite number of clock cycles. Table II summarizes our search problem formulation.

3) Query Formulation: After describing our search problem formulation, we describe how to transit from one state to another state in one clock cycle. To perform the transition, the search engine first forms a query in the format of KQuery [19].

The query consists of constraints over a concrete state, a set of symbolic expressions produced and optimized by previous stages, and the path constraints corresponding to the set of symbolic expressions. The symbolic expressions and the path constraints correspond to the same atomic path, which is the operator in our search problem formulation. The solution of this query is a new state that is the result of applying the atomic path to the original state.

Let us consider an example query on the up-down counter. Suppose the search engine starts with the reset state where the register *value* is 0. Moreover, the search engine follows the atomic path in which *value* is incremented, i.e. the input signal *inst* is 0. The query has two parts: the query constraints and the query body. The query solver checks whether the query body is valid or not assuming the query constraints. The query constraints consist of a KQuery expression regarding the current state ($value = 0$), and symbolic expressions as well as path constraints regarding the symbolic atomic path. We set the query body to be *false*. Therefore, by the principle of explosion (*ex falso quodlibet*: from falsehood anything follows), the query is valid if and only if the query constraints are unsatisfiable. On the other hand, if the query is invalid, then the query constraints are satisfiable. Then, the query solver will provide a counterexample as a reason to the invalidity of the query; this counterexample is a satisfying assignments to the query constraints. In this case, the query solver will give a counterexample indicating that the *value* register will be 1 in the next clock cycle. Table III shows the constraints in the query that is described in the scenario above. Listing 1 shows the complete query.

```
array inst[4] : w32 -> w8 = symbolic
array value[4] : w32 -> w8 = symbolic
array leaf_value[4] : w32 -> w8 = symbolic

(query [
  (Eq w32 0
        (ReadLSB w32 0 value))

  (Eq w32
        (ReadLSB w32 0 leaf_value)
        (Add w32 1 (ReadLSB w32 0 value)))

  (Eq 0
        (ReadLSB w32 0 inst))
] false [] [leaf_value])
```

Listing 1. A query that applies an atomic path the reset state of the up-down counter

The reason for us to form our queries in the format of KQuery instead of an SMT interface format (such as SMTLIB2) is that the symbolic expressions and the path constraints are in the KQuery format. Instead of translating the expressions to an SMT interface format, we formed the query in a way such that the query is invalid if and only if the constraints in the query is satisfiable. We used Kleaver [21], the built-in query solver for KLEE, to solve the query. Since we use Z3 [22] as the back-end SMT solver for Kleaver, the speed of Kleaver is comparable to the state-of-the-art SMT solver.

4) Backward Recursive Search: Having described how to transit from one state to another through query formulation (Section III-E3), we now propose solutions to the search problem that is formulated in Section III-E2.

We begin with a backward recursive search strategy that is proposed by Zhang et al. (see Algorithm 1 in [10]). This strategy starts with the property-violated state, i.e. the goal state in our problem formulation, and attempts to search for the reset state, i.e. our initial state. Figure 10 shows a symbolic execution tree in which the root node is the reset stare and one of the leaf node is the property-violation state.

From the concrete property-violated state, We find the parent node of the current node by solving the query representing an atomic path. Which atomic path to choose depends on the heuristic search policy we use (Section III-E8). We iterate this process to find the parent node of each current node until we reach the root node representing the reset state of the hardware design.

5) Depth-first Search: Under the Backward Recursive Search strategy, we adopted the Depth-first Search (DFS) algorithm (see Section 22.3 in [23]). In the symbolic execution tree (Figure 10), a general DFS starts with the root node and recursively explores each child node in sequence. Only when DFS fully explores the subtree rooted at the first child node will it move to the second child node. Therefore, DFS always explores the deepest nodes first whenever possible. Under the Backward Recursive Search strategy (Section III-E4), we started with the property-violated node instead of the root node. We recursively explored the neighbor nodes until we reached the root node.

Let us consider the time complexity of DFS. Let *b* be the number of branches and *m* be the maximal depth. Then the time complexity of DFS is $1 + b + b^2 + b^3 + b^4 + ... + b^m = O(b^m)$. Because hardware can run an infinite number of clock cycles, the depth is unbounded, i.e. $m = \infty$. As a result, DFS may get stuck in a subtree and never terminate if the target node is in another subtree.

6) Depth-limited Search: While DFS can explore a symbolic execution tree deeply, the drawback of DFS is that it will not stop and backtrack until it reaches a leaf node. Since in theory hardware designs can run an infinite number of clock cycles, DFS will *never* reaches a leaf node. As a result, DFS will never find a bug in the subtree rooted at the second child node because it will spend all time exploring the subtree rooted at the first child node. To overcome this drawback, we adopted Depth-limited Search (DLS) which allows the user to set a maximal depth limit for DFS to explore in any subtree. The condition for DLS to stop exploring and start to backtrack is that DLS reaches a leaf node *or* it exceeds the maximal depth.

In the search engine, we implemented DLS under the strategy of the Backward Recursive Search strategy. The drawback of DLS is that it requires the user to provide a maximal depth limit which is based on the user's empirical knowledge. While it is easy to obtain a reasonable maximal depth for simple hardware designs such as the up-down counter, it is difficult if not impossible to figure out the maximal depth for a complex property in a complex, real-world, open-source hardware designs.

7) Iterative Deepening Search algorithm: DLS's need for empirical experience to set a *good* depth limit can be satisfied by the Iterative Deepening Search algorithm (IDS). IDS sets the depth limit from 0 to ∞ and runs DLS in sequence. We leave as future work to include IDS into Yippelia's search engine.

8) Greedy Search: Greedy Search (GS) is a heuristic search algorithm that chooses to first explore the node that *appears* to be closest to the target node (see Chapter 16 in [23] for details). Specifically, GS uses a heuristic function *h* that takes as input a node *n* and outputs *h(n)*, the estimated cost from *n* to the goal node. Then, GS chooses the node *n* with the smallest *h(n)* to explore first. As a result, GS always chooses the node that appears to be the closest to the goal node.

Yippelia's search engine implemented GS along with DLS and the Backward Recursive Search strategy. Our heuristic function is the sum of normalized differences between registers in the current state and those in the reset state. Equation 1 is the formal definition of our heuristic function:

$$h(n) = \sum_{i=0}^{k} \frac{|register_i(n) - register_i(reset\_state)|}{size(register_i)} \quad (1)$$

In Equation 1, *n* is an arbitrary state, *k* is the number of registers in the concerning hardware design. $register_i(n)$ gives the value of the i^{th} register in the state *n*. $size(register_i)$ gives the size of the i^{th} register in the hardware design.

The reason behind Equation 1 is our observation that in practice, two states with similar register values are more likely to reach one another. Equation 1 is by no means the only heuristic. For example, we can use Hamming distance, which calculates the number of different bits in two registers, to replace the subtraction in Equation 1. We leave as future work to explore different heuristic functions.

9) Constraint Satisfaction Problem Formulation: Instead of using various search algorithms to solve the search problem, we formulated the problem as a Constraint Satisfaction Problem (CSP) and then used an industrial CSP solver in a push-button manner to find a solution to the search problem. A CSP consists of a set of variables *X*, a set of Domains *D* such that:

$$D = \{D_i = domain(X_i) | X_i \in X\}$$

and a set of constraints *C*. A problem is solved if for each variable X_i, there exists a value from the corresponding domain D_i such that all constraints in *C* are satisfied (see Chapter 6 in [20] for further details).

Let us consider the up-down counter example.The set of variables, *X*, consists of signals *inst* and *value*, each of which is represented by an array of variables whose element is the value of the signal at the clock cycle indicated by the index of the element. For instance, $inst[i]$ represents the value of the signal $inst$ at the i^{th} clock cycle. Table IV presents the set of domains *D* for *X* the variable set. Table V presents the set of constraints *C*.

10) Threats to Validity: The search engine may not be efficient because it spends the majority of the time on solving SMT queries. Although the queries are optimized by the optimization stage, the queries for complex, real-world processor

TABLE IV
CSP DOMAIN

Variable X_i	Domain D_i
insts[i]	[0,1]
values[i]	[-clock_cycle_number, clock_cycle_number]

TABLE V
CSP CONSTRAINTS

Constraints	Reasons
values[0] == 0	In the reset state, the initial value for register $value$ is 0.
$\neg$ insts[i] $\rightarrow$ values[i+1] == values[i] + 1	When $inst$ is 0, the counter counts up.
insts[i] $\rightarrow$ values[i+1] == values[i] - 1	When $inst$ is 1, the counter counts down.
values[last] == assert_value	This constraint represents the assertion.

design may still be lengthy and SMT solvers may still take a long time to solve them. To mitigate this threat, we use as our SMT solver Z3 [22] that is one of the state-of-the-art SMT solver. To further reduce the size of queries, we leave as future work to explore other symbolic execution in the software domain that can produce more compact symbolic expressions than KLEE.

Our current formulation of the CSP is not scalable to complex hardware designs because our formulation requires to abstractly implement the control flow of the hardware design in the CSP solver. We attempted to mitigate this threat by utilizing properties of hardware-oriented symbolic execution that are formulated by Zhang et al. (see Section 4.1 in [10] for details). However, the formulation of these properties require more powerful problem formulation such as Constraint Logic Programming. We leave as future work to explore general Constraint Logic Programming. formulation that does not depend on the knowledge of the design.

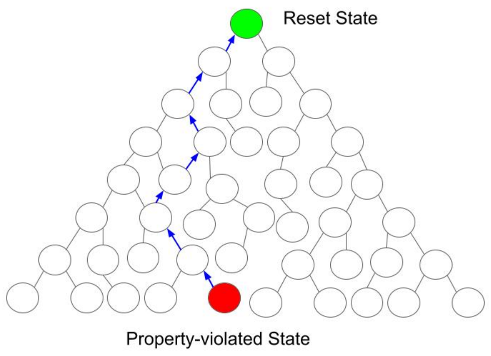

Fig. 10. The Symbolic Execution Tree as a result of the Backward Recursive Search Algorithm

TABLE VI
TIMES FOR Z3 TO SOLVE 162 YIPPELIA'S QUERIES.

time (minutes)	$[45, \infty)$	$[10, 45)$	$[1, 10)$	$[0, 1)$
number of queries	2	4	7	149
percentages	1.2%	2.5%	4.3%	92.0%

IV. IMPLEMENTATION

A. Overview

Yippelia is implemented in Python [24], [25] and C++ [14]. We consider RTL hardware designs in SystemVerilog [12]. We used Verilator [15] to translate the RTL hardware designs to C++. We also wrote C++ script to symbolically simulate the hardware designs for one clock cycle by KLEE [11]. The KPiler tool in the optimization stage and the search engine are implemented in Python.

B. Preprocessing

In the preprocessing stage, we used Verilator [15] to translate RTL hardware designs in SystemVerilog [12] to C++ [14]. Verilator has different levels of compiler optimizations. Verilator's optimization trades completeness for efficiency. Verilator's optimization simplifies the control logic, flattens the designs, and eliminates some signals. As a result, Verilator's optimization is possible to eliminate bugs in the RTL designs, making the symbolic execution more incomplete. On the other hand, the compiler optimization can shorten the length of the design in C++, so KLEE can finish the one-cycle symbolic execution in a shorter time. Zhang et al. used Verilator's compiler optimization in [8] because their tool requires to run symbolic execution multiple times, one time for each clock cycle. In Yippelia, we only run symbolic execution one time for one clock cycle, so Yippelia can afford a more expensive but more complete symbolic execution. We leave as a user input to let the users decide whether they would like to enable Verilator's compiler optimizations when running Yippelia.

C. Symbolic Execution

Yippelia uses KLEE [11] as the symbolic execution engine to symbolically explore a hardware design for one clock cycle. KLEE has four search modes: Depth-First Search (DFS), Random State Search, Random Path Selection, and Non Uniform Random Search (NURS). Yippelia uses the DFS heuristic, and the reasons are twofold. First, because we need to fully explore a hardware design for one clock cycle, in which order to explore the states will not make a difference in execution time; no matter which heuristic we use, the total number of states KLEE needs to visit is the same. Moreover, we prefer KLEE's exploration to be in-order and deterministic because this provides convenience for us to match the symbolic expressions with the path constraints. As a result, Yippelia uses DFS as KLEE's search mode.

D. Optimization

1) Optimization 1: In Yippelia's implementation, we moved the first optimization to the symbolic execution stage.

Before symbolic execution, we filtered out the input signals from the set of signals that is pre-calculated for KLEE to produce their symbolic expressions. By filtering out the input signals early, Yippelia saves time by not printing out the symbolic expressions for inputs signals which eventually will not be used to form the queries.

2) Optimization 2: KPiler is implemented in Python. Our implementation strictly follows the compiler workflow that is described in Chapter 4 (Syntactic Analysis) and Chapter 7 (Code Generation) in [26]. We skipped the Contextual Analysis section because we assume that KLEE produces correct KQuery expressions. The front-end part of KPiler implements the Syntactic Analysis part. It utilizes a Python package called ply which implements the lex and yacc parsing tool in Python [27]. The back-end part of KPiler implements the Code Generation part. Both tools in the back-end (KTransformer and KGenerator) implements a visitor interface that systematically and recursively explores each node in the AST. The analyzer of KPiler also implements the visitor interface to visualize and analyze the AST. We have not finished the implementation of KPiler since KTransformer has an unresolved implementation bug. We leave as future work to fix the bug.

E. Search Engine

The search engine, including the search algorithms and the constraint satisfaction problem formulation, is implemented in Python.

1) Query Formulation: We describe in Section III-E3 how we combine symbolic expressions to form queries. In this section, we describe our solution to a technical challenge in combining the symbolic expressions. The challenge is, two symbolic expressions may contain the same identifier name but they refer to two different identifier declarations. Therefore, when we put the two expressions into the same query, the query will have a *duplicate declarations* bug. We overcame this challenge by renaming the identifiers in each symbolic expression. Our solution is based on an observation that identifiers names in the symbolic expressions produced by KLEE are in sequence. For example, if an expression contains 10 identifiers, the identifiers are named by N1, N2, N3, ..., N10. We also utilized the idea of segmentation in the operating system community which translates a virtual address to a physical address by adding a base register to the virtual address. We implemented a similar translation process for identifier names in a Python script. We have a base register that is initialized to be 0. Every time we add a new symbolic expression to the query, we first add the base register to all occurrences of identifier names; we then increment the base register by the number of identifier declarations. As a result, within one query, there does not exist two expressions with the same identifier.

2) CSP Formulation: To formulate the search problem as a CSP and solve it, we utilized Google OR-Tools, an open source package for combinatorial optimization [28]. We used the interface in Python as a third-party library. We added our constraints (Table V) to the Constraint Programming Solver, and asked the solver to return a satisfying solution in a push-button manner.

V. Evaluation

A. Research Questions

In this section, we answer the following research question: How much time can Yippelia save in finding property-violations in hardware designs compared to the state-of-the-art symbolic execution engines?

B. Experimental Setup

Before we answer the research questions, we first describe our experimental setup.

1) Hardware Designs: We evaluated Yippelia on two hardware designs. The first design is the up-down counter that is detailedly described in Section II. The design of the up-down counter is simple in that it only has two program paths in one clock cycle, but the design is nontrivial in that it is stateful. Therefore, we used the up-down counter as a playground to explore and demonstrate the workflow of Yippelia.

The second design is our own implementation of a single-cycle processor which implements the MIPS architecture described in [29]. The design complexity of the MIPS processor is at the same level as other open-source processor designs such as the RISC-V processors and the OR1200 processor which implements the OpenRISC 1000 architecture. The MIPS processor is a single-cycle design in that all instructions are finished within one clock cycle. We leave as future work to perform experiments on a larger range of open-source processor designs that include multi-cycle and pipelined designs.

2) Properties: The properties we considered for the up-down counter simply assert that the register *value* does not equal to certain value *X*. The absolute value of *X* also determines the optimal number of clock cycles the search engine requires to find a property-violation. For example, the property $\neg reset \rightarrow value \neq -10$ asserts that the register *value* does not equal to -10. The violation to this property can be found within 10 clock cycles. Table VII presents example properties for deep bugs and shallow bugs for the up-down counter.

For the MIPS processor, we infer from the MIPS Instruction Set [30] a property about the *store-word* instruction. *store-word* stores a register's value to the memory. Because this instruction does not write to the register file, the signal *werf*, which indicates whether the register file is write-enabled, should be set to 0. Formally, the property is: $(\neg((instr\ \&\ `hFC000000) >> 26) == 43)\ ||\ (werf == 0)\ ||\ (reset == 1)$. This property says that when the processor is not in the reset state, if the instruction is *store-word*, then the signal *werf* is 0. We inserted a bug to the MIPS processor that violates this property whenever the processor runs a *store-word* instruction. Since the property violation can be found within one clock cycle, we disabled the *fast validation* sub-stage in the symbolic execution stage (Section III-C). As a result, Yippelia needs to find the property violation in the search engine instead of finding it early in the symbolic execution

TABLE VII
EXAMPLE SHALLOW AND DEEP PROPERTIES FOR THE UP-DOWN COUNTER

Shallow Property	$\neg reset \rightarrow value \neq -2$
Deep Property	$\neg reset \rightarrow value \neq 980$

TABLE VIII
COMPARISON ON EXECUTION TIME FOR YIPPELIA AND KLEE IN FINDING SHALLOW AND DEEP BUGS

	Shallow bugs	Deep bugs
Yippelia	6.74 seconds	18.31 seconds
KLEE	4.16 seconds	didn't finish after 121 hours

stage, so we are able to test the optimization stage and the search engine using this property.

3) Running Experiments: We wrote scripts to automatically run tests on both hardware designs and collect data that measure execution time. We performed the experiments on a machine with Intel Core i5-3337U 4-core 1.80GHz CPU and 3G available RAM.

C. Up-down Counter

We evaluated Yippelia on the up-down counter design (Section II) against KLEE [11], a matured, open source, and well-maintained symbolic execution engine in the software domain that represents the state-of-the-art symbolic execution engines. In our experiments, KLEE uses the default search heuristic which is the Random Path Selection interleaved with Non Uniform Random Search (NURS) with Coverage-New heuristic (see Section 3.4 in [11]). Same as Yippelia, KLEE uses Z3 [22] as its back-end SMT solver.

Table VIII presents a comparison on Yippelia and KLEE on finding shallow bugs and deep bugs. In Table VIII, we define the shallow bugs to be property-violations that can be found within 10 clock cycles, and deep bugs to be those that can be found from 980 to 990 clock cycles. Table VIII shows that while KLEE has comparable timing performance with Yippelia on shallow bugs, Yippelia has an average speedup of at least four orders of magnitude on deep bugs.

Figure 11 presents a comparison on execution time between Yippelia and KLEE on finding property violations from 1 to 1000 clock cycles. Figure 11 shows that while KLEE's execution time grows exponentially, the growth rate of Yippelia's execution time is close to linear with respect to the number of clock cycles. Figure 12 and Figure 13 are parts of Figure 11 in that Figure 12 focuses on Yippelia's graph while Figure 13 focuses on KLEE's graph.

D. MIPS processor

While Yippelia has efficient performance on the up-down counter design, Yippelia took a significantly large amount of time to finish execution when we evaluated Yippelia on the MIPS processor without the optimization stage (Section III-D). While Yippelia can find the property-violation, Yippelia took 198.9 minutes to finish the process. The reason for Yippelia to take such a long time is that, for a complex, industrial-level

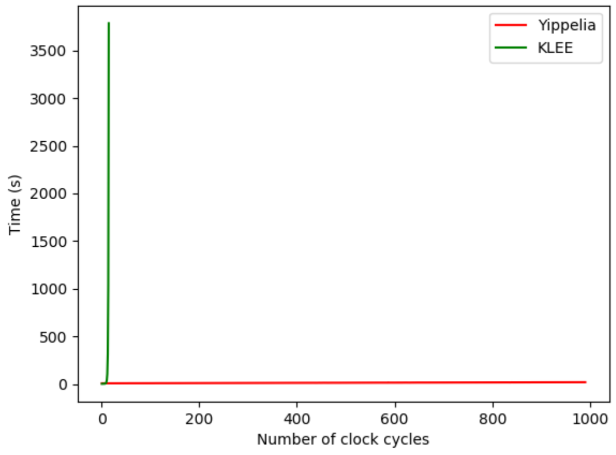

Fig. 11. Execution time comparison between Yippelia and KLEE

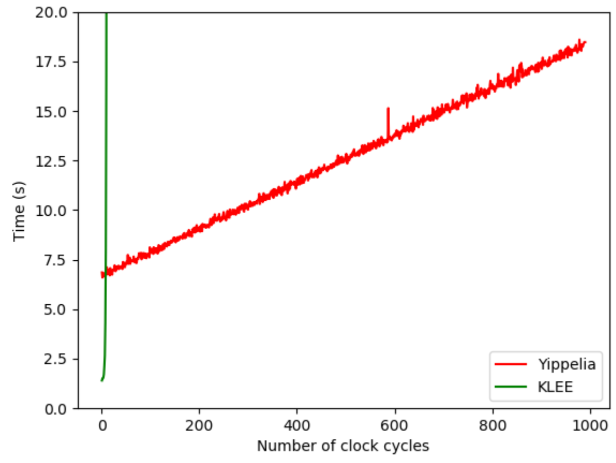

Fig. 12. Execution time comparison focusing on Yippelia

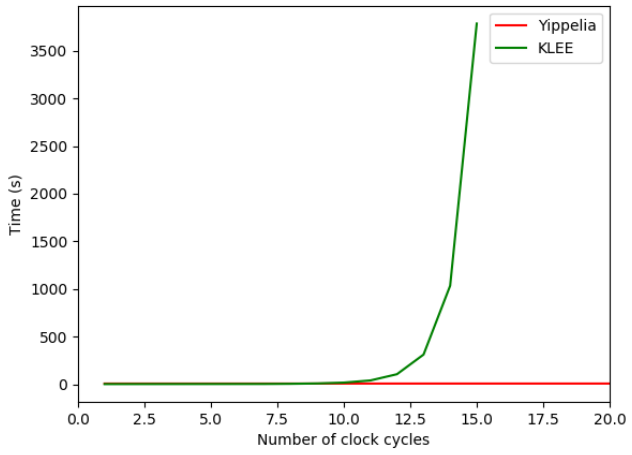

Fig. 13. Execution time comparison focusing on KLEE

processor design such as the MIPS processor, the symbolic expressions generated from KLEE are too lengthy. As a result, Yippelia's back-end SMT solver is not able to efficiently solve queries made up of these lengthy symbolic expressions. To provide a sense of the execution time for solving the queries, we performed experiments on using Z3 [22] to solve 162 queries formed by the symbolic expressions from the MIPS processor. The results are presented in Table X. Specifically, 3.7% of the queries take more than 10 minutes. Within them, 2 queries in particular need more than 45 minutes. This performance makes Yippelia without the optimization stage not practical in finding property-violations on the MIPS processor.

E. Optimizations

To reduce the execution time for our SMT solver to solve the queries for the MIPS processor, Yippelia performs 2 optimizations in the optimization stage (Section III-D). The first optimization removes from the queries the symbolic expressions constraining the input signals. Table IX shows that the first optimization reduces the overall execution time by 70%. In comparison with Table X, Table XI also shows the time for Z3 [22] to solve the 162 queries formed by the symbolic expressions from the MIPS processor, but with Optimization 1 enabled. Table XI shows that the percentage of the 162 queries which take more than 10 minutes shrinks from 3.7% to 0.6%. Moreover, the execution for the longest query (the bottleneck) shrinks by 61.3%. Therefore, the first optimization greatly improves the scalability of Yippelia on complex hardware designs. Because the implementation of the second optimization is still in progress, we leave as future work to evaluate the speedup of the second optimization.

TABLE IX
TIME COMPARISON FOR YIPPELIA WITH AND WITHOUT OPTIMIZATION 1 TO FIND THE PROPERTY-VIOLATION REGARDING THE STORE-WORD INSTRUCTION

	no Optimization 1	with Optimization 1
Time (min)	198.9	60.2
% regarding Yippelia without Optimization 1	100%	30%

TABLE X
TIMES FOR Z3 TO SOLVE 162 QUERIES FOR THE MIPS PROCESSOR WITH THE OPTIMIZATION STAGE DISABLED

time (minutes)	$[45, \infty)$	$[10, 45)$	$[1, 10)$	$[0, 1)$
number of queries	2	4	7	149
percentages	1.2%	2.5%	4.3%	92.0%

VI. DISCUSSION

A. Threats to Validity

1) Incompleteness: With respect to finding property violations, though Yippelia is sound in that every bug it reports is a true property violation, Yippelia is incomplete in that there may exist a property violation which Yippelia cannot find. The source of Yippelia's incompleteness is threefold. First, the number of clock cycles that is required to reach certain property violation is significantly large. In the case of the up-down counter, this type of property would be asserting that the register *value* equals to 2^{1000}. Yippelia will time-out in this case. The second type of incompleteness rises from the preprocessing stage (Section III-B). As indicated by Zhang et al. [10], Verilator may eliminate bugs during the compilation of the hardware design, because the compilation is not always faithful due to compiler optimization. To mitigate this threat, our research group is developing so called the native symbolic execution approach on hardware which runs symbolic execution directly on the RTL hardware designs (Section VI-B3). Lastly, Yippelia may miss a bug because our search heuristics (Section III-E) do not explore the correct path. We leave as future work to develop better search heuristics.

TABLE XI
TIMES FOR Z3 TO SOLVE 162 QUERIES FOR THE MIPS PROCESSOR WITH OPTIMIZATION 1 ENABLED

time (minutes)	$[10, 18)$	$[5, 10)$	$[1, 15)$	$[0, 1)$
number of queries	1	2	8	151
percentages	0.6%	1.2%	5.0%	93.2%

2) Symbolic Expressions: As described in Section III-C, Yippelia relies on KLEE to perform the symbolic exploration of hardware designs within one clock cycle. Although KLEE is a matured, well-maintained symbolic execution engine on the software domain, the symbolic expressions that are printed out by KLEE are lengthy. As a result, the query formed by the symbolic expressions often require a nontrivial amount of time to be solved by an SMT solver. As a result, the query formed by the symbolic expressions often require a nontrivial amount of time to be solved by an SMT solver. To mitigate this threat, we performed optimizations to shorten the symbolic expressions (Section III-D). We leave as future work to explore other open-source symbolic execution frameworks such as angr [16], [17], [18].

3) KPiler: We consider potential threats to the validity for our query optimization tool, KPiler (Section III-D2). First, our modified version of KQuery grammar is complete but not sound: our KPiler tool can accept all correct KQuery expressions, but it may also accept some incorrect expressions. We sacrifice soundness for the ease of implementation. We mitigate this threat by assuming that the symbolic expressions produced by KLEE are correct with respect to the grammar. Within each step of KPiler's workflow, we do not perform any operation that makes a correct expression to be incorrect. Therefore, the output of KPiler should still be correct KQuery expressions even though our grammar is not sound. Second, our KPiler tool may consist of implementation bugs. We mitigated this threat by testing our tool against 162 long queries with an average query size of 14187 bytes. We also performed thorough code review for our tool.

4) Search Engine: The search engine may not be efficient because it spends the majority of the time on solving SMT queries. Although the queries are optimized by the optimization stage, the queries for complex, real-world processor design may still be lengthy and SMT solvers may still take a long time to solve them. To mitigate this threat, we use Z3 [22], one of the state-of-the-art SMT solver, as Yippelia's back-end SMT solver. To further reduce the size of the queries, we leave as future work to explore other symbolic execution in the software domain that can produce more compact symbolic expressions than KLEE.

5) Constraint Satisfaction Problem: Our current formulation of the Constraint Satisfaction Problem is not scalable to complex hardware designs because our formulation requires to abstractly implement the control flow of the hardware design in the CSP solver. We attempted to mitigate this threat by utilizing properties of hardware-oriented symbolic execution that are formulated by Zhang et al. (see Section 4.1 in [10] for details). These three properties define a sequence of path constraints which takes the hardware design from a reset state to a property-violated state. Therefore, we can replace the constraints in our CSP formulation with these three properties. However, the properties involve set operations such as set membership, intersection, and sub-set relations. As a result, we cannot express the properties as constraints in CSP. Constraint Logic Programming (CLP) [31], [32], which combines logic programming with CSP, can represent the set relations with the knowledge representation in the logic programming. Therefore, CLP has the potential to express the three properties in logic programming, incorporate them as constraints, and solve the problem in a similar manner as CSP. Once we finish the problem formulation, we can utilize the industrial CLP solvers such as SAS-OR tool to solve the problem in a push-button manner. We leave as future work to explore general Constraint Logic Programming formulation that does not depend on the knowledge of the design.

6) Implementation: The search engine (Section III-E) will encounter a stack-overflow error after 1000 clock cycles for the up-down counter design. The reason is that the search algorithms in the search engine are implemented in a recursive manner. This problem can be solved by translating the recursive implementation to an iterative implementation. The translation is achievable because a recursive algorithm can always be represented by an iterative algorithm. However, because of the tremendous engineering work to perform the translation, we decided to leave it as a future work.

7) Evaluation: Our evaluation of Yippelia is not thorough. We attempted to evaluate Yippelia on the up-down counter (Section II) and the MIPS processor [29]. Because we did not finish the implementation of KPiler (Section III-D2), we were not able to complete the evaluation on the MIPS processor, so the efficacy of Yippelia on complex hardware designs is still doubtable. Because the MIPS processor follows the single-cycle architecture, we cannot evaluate Yippelia against complex properties whose violations cross multiple clock cycles. We leave as future work to evaluate Yippelia on designs with multi-cycle and pipeline architecture. Moreover, Yippelia is compared with KLEE [11]. Although KLEE is a state-of-the-art symbolic execution engine on the software domain, KLEE is not designed for hardware. In the future, we plan to compare Yippelia with the state-of-the-art hardware-oriented symbolic execution engine such as Coppelia [8]. We also plan to compare Yippelia's performance with other formal verification techniques such as model checking [9] and concolic execution [33].

B. Future Work

1) Massive-Properties Symbolic Execution on Hardware: In the real world, hardware designers often need to check multiple properties on a hardware design simultaneously. The current workflow of Yippelia only checks one property at a time, but Yippelia can be fine-tuned to efficiently check multiple properties. When checking multiple properties, Yippelia needs to run the preprocessing stage (Section III-B), the symbolic execution stage (Section III-C), and the optimization stage (Section III-D) only once. The set of registers that Yippelia uses in the symbolic execution stage is the *union* of the sets of registers for all properties. Although Yippelia needs to run the search engine (Section III-E) once for each property, Yippelia can save time on verifying massive properties by running the previous stages only once.

2) Evolutionary Hardware-oriented Symbolic Execution: Similar to regression testing in the software domain, hardware designers often need to run tests and check properties on multiple versions of a design. Inspired by the Evolutionary Runtime Verification approach proposed by Lengunsen et al. [34], we propose the Evolutionary Hardware-oriented Symbolic Execution technique. This technique reduces the runtime overhead of using symbolic execution to verify different versions of hardware designs by first finding the property-violations that are rooted in the change of the design. We propose a three-stage pipeline for the approach. First, the approach performs a change-of-impact analysis to determine which part of the hardware design is changed. Second, the approach performs a mapping from the change of the design to the program paths produced by the symbolic execution stage (Section III-C) in Yippelia. Lastly, we add as a heuristic to Yippelia's search engine III-E to prioritize exploring the paths corresponding to the change of the design. Therefore, instead of spending time finding the old bugs, the Evolutionary Hardware-oriented Symbolic Execution approach prioritizes finding the new property-violations which are caused by the change of the design. This approach increases the scalability of symbolic execution in the real-world hardware development cycle, not by finding a single property-violation faster, but by reducing the search space of the property-violations to the *new* property-violations which are rooted in the change of the hardware design.

3) Native Symbolic Execution on Hardware: To mitigate the threat of incompleteness (Section VI-A1) which arises from the compiler optimization in the preprocessing stage (Section III-B), our research group is developing an approach called the Native Symbolic Execution on Hardware. To circumvents the incompleteness problem caused by Verilator, the native symbolic execution approach directly builds a symbolic execution engine for the RTL hardware designs. Although this approach cannot fully mitigate the incompleteness problem as the incompleteness is rooted in the general symbolic execution, this approach eliminates the part of the incompleteness that is caused by Verilator's compiler optimization.

VII. Related Work

A. Overview

The recent emergences of the Spectre [1] and Meltdown attacks [2] have emphasized the importance of security verification at the design phase of hardware. Assertion based verification is one of the most popular techniques to verify the security of hardware designs. To perform the verification, one needs to answer two questions: 1) how to generate assertions encoding security critical properties of hardware designs, and 2) how to verify that assertions are not violated in hardware designs. Recently, several answers to the first question has been proposed in [4], [5], [7], [6], and we will review these answers in Section VII-B.

We in Yippelia propose a partial answer to the second question by triggering an assertion violation in hardware designs. Although we cannot prove the absence of assertion violation, we can in the most time trigger the assertion violation if the assertion is indeed violated by the designs. To automate the process of triggering assertion violations, we propose a strategy to perform symbolic execution techniques on hardware designs. Recent works on applying symbolic execution to hardware designs are reviewed in Section VII-B.

Although software symbolic execution techniques only explore a hardware design for one cycle, a processor can run for an infinite number of clock cycles, resulting a severe path explosion problem [13]. Therefore, heuristic search algorithms toward the assertion violation are needed in the symbolic execution process. Section VII-D reviews several search algorithms and their applications to solving problems in various research areas.

B. Hardware Symbolic Execution

First purposed as a software technique in [35], symbolic execution has been widely applied to hardware recently in [9], [36], [10], [8], [37], [38], [39], etc.

Path explosion [13] is a prevalent problem in the field of applying symbolic execution to hardware. This problem is severe in the context of hardware symbolic execution because of the intrinsic properties of hardware designs. Mukherjee et al. [9] made the first attempt to apply forward symbolic execution to hardware verification, but this method is limited to small-size hardware designs because of the path explosion problem. Spectector [36], a countermeasure for the Spectre attack [1], can detect speculative leaks or prove their absence. However, Spectector [36] is not scalable for verifying large, complex designs because it limits the number and the lengths of symbolic paths it can explore in order to circumvent the path explosion problem.

Zhang et al. tackled the path explosion problem [13] by proposing a recursive, backward search algorithm for hardware-oriented symbolic simulation [10], [8]. In a symbolic tree as shown in Figure 10 representing all symbolic paths of a design, to find a path from the root node to the error node, the algorithm starts with the error node and recursively finds the parent node until it reaches the root node. The algorithm counters the path explosion problem [13] by utilizing an intrinsic property of the tree structure: any node has at most one parent node while it can have a much larger number of child nodes. Zhang et al. implemented this recursive symbolic simulation strategy in Coppelia [8] for automated exploit generation. However, Coppelia [8] runs symbolic simulation on the same processor design once for every clock cycle, thus performing redundant work. Moreover, Coppelia has only been evaluated on processors but not general hardware designs. Meng et al. recently extended the verification scope from the processor core to a whole System-on-Chip (SoC) design. Meng et al proposed RTL-ConTest [33], a framework that first extracts the process flows of the designs as control flow graphs (CFGs) and then performs concolic execution on the CFGs.

Symbolic simulation is often combined with concrete simulation to generate functional tests. Lyu et la. [39] automated the generation of directed tests with a combination of symbolic simulation and concrete simulation. They showed that their approach is more scalable than the state-of-the-art model checking technique [9] because of its lower requirement on memory resource. STAR [37] uses a hybrid approach between symbolic simulation and concrete simulation in RTL level as well to generate functional input vector. However, its performance is limited by the number of unrolling cycles. PACOST [38] also combines symbolic simulation and concrete simulation to generate functional input vectors that specifically covering hard-to-reach states. PACOST [38] handles the problem of unrolling cycles in STAR [37] by using the abstract distance to the target state to approximate the optimal number of unrolling cycles.

C. Security Property Generation

Historically, security properties have been manually generated [40], [41], [42], [43]. These manually-developed properties are then used as an initial set of properties for SCIFinder [4] to semi-automatically find security properties utilizing statistical learning techniques. In addition to propositional properties generated by SCIFinder [4], UNDINE [5] can find properties in the form of linear temporal logic. However, the property generation process is not still semi-automated because both SCIFinder and UNDINE require an initial set of manually-developed properties.

Deutschbein et al. fully automated the property generation process in Astarte [7]. Astarte infers security properties from their relevance with safety-critical control signals, instead of using an initial set of manually-written security properties in previous works [4], [5], so that it can eliminate human efforts in the process. However, the scope of Astarte is limited to closed-source, CISC designs (such as x86 processor designs). To handle this problem, Zhang et al. presented Transys [6], a tool that can automatically translate security properties on one design to equivalent properties on another design. Therefore, properties on CISC designs generated by Astarte [5] can be translated to other processor designs as well. However, the semantic equivalence rate of the translation is low (less than 50%) for assertions related to information flow tracking and processor cores.

The generated security properties are useful for light-weight verification of existing designs, as well as for guiding new secure designs. In this work, we use the security properties to generate exploits for existing designs. Farzana et al. [44] showed an approach to use the security properties to guide the new, secure designs of system-on-chips.

D. Search Algorithms

Dijkstra's [45] and A-star [46] are among the most popular shortest path algorithms. Dijkstra's finds the optimal solution but takes a longer time, while A-star can find a solution in optimal time but the solution is not always optimal [47]. In practice, heuristic search algorithms are used in various areas including motion planning [48], [49], constraint optimization [50], [51], and computational biology [52], [53], [54]. In this work, we utilize various heuristic search algorithms to find a path from the root node to the error node in the symbolic tree.

VIII. Conclusion

This thesis presents Yippelia, a hardware-oriented symbolic execution tool that automatically triggers property-violations in hardware designs. Starting with a RTL hardware design and a property, Yippelia first translates the RTL design to the semantically equivalent software code. Yippelia then runs symbolic execution on the software code for one clock cycle to produce a set of symbolic expressions for each program path. Next, Yippelia optimizes the symbolic expressions and uses them to form a search problem and a Constraint Satisfaction Problem (CSP). Lastly, Yippelia solves the search problem through various heuristic search algorithms and the CSP through the industrial CSP solver.

The secret sauce of Yippelia is twofold. First, Yippelia only runs the expensive symbolic execution on the hardware design only *one* time for *one* clock cycle. So, compared to Coppelia [8] which runs symbolic execution for each clock cycle, Yippelia can save execution time on running symbolic execution especially when finding deep bugs. This feature not only makes Yippelia efficient to verify a single property, but also enables Yippelia to verify a massive amount of properties simultaneously (Section VI-B1). Second, by running the one-clock-cycle symbolic execution, Yippelia obtains all possible atomic paths which consist of one clock cycle. With all the building blocks in hand, Yippelia can easily utilize the techniques from the Artificial Intelligence community, such as the heuristic search algorithms and the Constraint Satisfaction Programming, to form the target multi-cycle path. Having all atomic paths in hands, Yippelia also has the potential to borrow ideas from the software engineering community to perform evolutionary verification which prioritizes finding new property-violations in a new version of the design (Section VI-B2).

We evaluated Yippelia on two hardware designs – the up-down counter (Section II) and the MIPS processor [29]. For the up-down counter design, Yippelia has an average speedup of at least four orders of magnitude on finding deep bugs compared with KLEE [11]. Because we have not finished the implementation of the KPiler (Section III-D2), we did not complete the experiments on the MIPS processor yet.

Acknowledgements

First and foremost, I would like to express my appreciation to my advisor, Professor Cynthia Sturton, for her guidance and constant support during my undergraduate study. Professor Cynthia kindly accepted me as her undergraduate student, guided me step-by-step to perform Computer Science research, and patiently helped me practice my writing and presentation skills. I would have not stepped into the research world and completed this thesis without her mentorship and continuous encouragement.

I would also like to extend my sincere gratitude to other professors at the Department of Computer Science at the University of North Carolina at Chapel Hill, especially Professors Parasara Sridhar Duggirala, Samarjit Chakraborty, and David Plaisted. Professor Sridhar kindly agreed to be the second reader for this thesis, provided very valuable advice before, during, and after the thesis defense, and guided me through my study and my graduate school application and decision. Professor Samarjit provided me with invaluable help and guidance on my study, my graduate school application and decision, and life in general. Professor Plaisted kindly accepted me as his undergraduate TA for three years and provided me with opportunities to guest lecture.

I would also like to thank members at the UNC Hardware Security group for their mentorship, helps, and friendship: Dr. Rui Zhang, Calvin Deutschbein, Alyssa Byrnes, Yufeng Gong, and Kaki Ryan. Lastly, special thanks to my family and friends for their emotional supports and love.

References

[1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 973–990.

[3] D. Price, "Pentium fdiv flaw-lessons learned," *IEEE Micro*, vol. 15, no. 2, pp. 86–88, 1995.

[4] R. Zhang, N. Stanley, C. Griggs, A. Chi, and C. Sturton, "Identifying security critical properties for the dynamic verification of a processor," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 541–554. [Online]. Available: https://doi.org/10.1145/3037697.3037734

[5] C. Deutschbein and C. Sturton, "Mining security critical linear temporal logic specifications for processors," in *2018 19th International Workshop on Microprocessor and SOC Test and Verification (MTV)*, 2018, pp. 18–23.

[6] R. Zhang and C. Sturton, "Transys: Leveraging common security properties across hardware designs," in *Proceedings of the Symposium on Security and Privacy (S&P)*. IEEE, 2020.

[7] C. Deutschbein and C. Sturton, "Evaluating security specification mining for a cisc architecture," in *Proceedings of the International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2020.

[8] R. Zhang, C. Deutschbein, P. Huang, and C. Sturton, "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.

[9] R. Mukherjee, D. Kroening, and T. Melham, "Hardware verification using software analyzers," in *2015 IEEE Computer Society Annual Symposium on VLSI*, 2015, pp. 7–12.

[10] R. Zhang and C. Sturton, "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.

[11] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[12] "Ieee standard for systemverilog–unified hardware design, specification, and verification language," *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315, 2018.

[13] S. Krishnamoorthy, M. S. Hsiao, and L. Lingappan, "Tackling the path explosion problem in symbolic execution-driven test generation for programs," in *2010 19th IEEE Asian Test Symposium*, 2010, pp. 59–64.

[14] C. S. Committee *et al.*, "Iso international standard iso/iec 14882: 2014, programming language c++," Technical report, Geneva, Switzerland: International Organization for . . . , Tech. Rep., 2014.

[15] "Verilator." [Online]. Available: https://www.veripool.org/wiki/verilator

[16] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," 2016.

[17] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," 2016.

[18] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware," 2015.

[19] "Kquery · klee." [Online]. Available: https://klee.github.io/docs/kquery/

[20] S. Russell and P. Norvig, "Artificial intelligence: a modern approach," 2002.

[21] "Kleaver's options · klee." [Online]. Available: https://klee.github.io/docs/kleaver-options/

[22] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[24] G. Van Rossum and F. L. Drake Jr, *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[25] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

[26] D. A. Watt, D. F. Brown, and D. Brown, *Programming language processors in Java: compilers and interpreters*. Pearson Education, 2000.

[27] "python lex-yacc." [Online]. Available: https://www.dabeaz.com/ply/

[28] "About or-tools." [Online]. Available: https://developers.google.com/optimization/introduction/overview

[29] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design MIPS Edition: The*. San Francisco: Morgan Kaufmann Publishers Inc, 2013.

[30] J. Heinrich *et al.*, *MIPS R4000 Microprocessor User's Manual*. MIPS technologies, 1994.

[31] J. Jaffar and J.-L. Lassez, "Constraint logic programming," in *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1987, pp. 111–119.

[32] J. Jaffar and M. J. Maher, "Constraint logic programming: A survey," *The journal of logic programming*, vol. 19, pp. 503–581, 1994.

[33] X. Meng, S. Kundu, A. K. Kanuparthi, and K. Basu, "Rtl-contest: Concolic testing on rtl for detecting security vulnerabilities," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[34] O. Legunsen, "Evolution-aware runtime verification," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2019.

[35] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, p. 385–394, Jul. 1976. [Online]. Available: https://doi.org/10.1145/360248.360252

[36] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled detection of speculative information flows," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1–19.

[37] L. Liu and S. Vasudevan, "Star: Generating input vectors for design validation by static analysis of rtl," in *2009 IEEE International High Level Design Validation and Test Workshop*, 2009, pp. 32–37.

[38] Y. Zhou, T. Wang, H. Li, T. Lv, and X. Li, "Functional test generation for hard-to-reach states using path constraint solving," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 6, pp. 999–1011, 2016.

[39] Y. Lyu and P. Mishra, "Automated test generation for activation of assertions in rtl models," in *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2020, pp. 223–228.

[40] M. Abramovici and P. Bradley, "Integrated circuit security: New threats and solutions," in *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies*, ser. CSIIRW '09. New York, NY, USA: Association for Computing Machinery, 2009. [Online]. Available: https://doi.org/10.1145/1558607.1558671

[41] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin, "Security checkers: Detecting processor malicious inclusions at runtime," in *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*, 2011, pp. 34–39.

[42] ——, "Evaluating security requirements in a general-purpose processor by combining assertion checkers with code coverage," in *2012 IEEE International Symposium on Hardware-Oriented Security and Trust*, 2012, pp. 49–54.

[43] M. Hicks, C. Sturton, S. T. King, and J. M. Smith, "Specs: A lightweight runtime mechanism for protecting software from security-critical processor bugs," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 517–529. [Online]. Available: https://doi.org/10.1145/2694344.2694366

[44] N. Farzana, F. Rahman, M. Tehranipoor, and F. Farahmandi, "Soc security verification using property checking," in *2019 IEEE International Test Conference (ITC)*, 2019, pp. 1–10.

[45] E. DIJKSTRA, "A note on two problems in connexion with graphs." *Numerische Mathematik*, vol. 1, pp. 269–271, 1959. [Online]. Available: http://eudml.org/doc/131436

[46] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. SSC-4(2), pp. 100–107, 1968.

[47] A. Candra, M. A. Budiman, and K. Hartanto, "Dijkstra's and a-star in finding the shortest path: a tutorial," in *2020 International Conference on Data Science, Artificial Intelligence, and Business Analytics (DATABIA)*, 2020, pp. 28–32.

[48] B. Bonet and H. Geffner, "Planning as heuristic search," *Artificial Intelligence. 2001 Jun; 129 (1-2): 5-33.*, 2001.

[49] D. Youakim, P. Cieslak, A. Dornbush, A. Palomer, P. Ridao, and M. Likhachev, "Multirepresentation, multiheuristic a* search-based motion planning for a free-floating underwater vehicle-manipulator system in unknown environment," *Journal of Field Robotics*, 2020.

[50] P. Dasgupta, P. Chakrabarti, A. Dey, S. Ghose, and W. Bibel, "Solving constraint optimization problems from clp-style specifications using heuristic search techniques," *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 2, pp. 353–368, 2002.

[51] M. Braik, A. Sheta, and H. Al-Hiary, "A novel meta-heuristic search algorithm for solving optimization problems: capuchin search algorithm," *Neural Computing and Applications*, pp. 1–33, 2020.

[52] R. M. Karp, "Heuristic algorithms in computational molecular biology," *Journal of Computer and System Sciences*, vol. 77, no. 1, pp. 122–128, 2011.

[53] S. Shatabda, M. H. Newton, and A. Sattar, “Mixed heuristic local search for protein structure prediction,” in *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013, pp. 876–882.

[54] H. MotieGhader, Y. Masoudi-Sobhanzadeh, S. H. Ashtiani, and A. Masoudi-Nejad, “mrna and microrna selection for breast cancer molecular subtype stratification using meta-heuristic based algorithms,” *Genomics*, 2020.