YouVerify: An Intermediate Representation and Framework for Symbolic Execution

Griffin Prechter

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-261
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-261.html

December 17, 2021

Acknowledgement

I am extremely grateful for the unwavering support of both my advisors, Professors
Raluca Ada Popa and Koushik Sen, throughout the 5th Year Masters program. I reached out before my senior year with no experience in research, and both were extremely supportive and motivated me to get involved and to pursue the masters.
Were it not for their support and the opportunity to do the program, I would have
never been able to grow and pursue my interests in the way that I have. I would like to thank Jeongseok Son and Rishabh Poddar for their mentorship while working on ObliCheck, which inspired to me to investigate symbolic execution for this project. Finally, I would like to thank Kevin Laeufer for his support and guidance during my work on YouVerify.

YouVerify: An Intermediate Representation and Framework for Symbolic Execution

by Griffin Prechter

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Raluca Ada Popa
Research Advisor

December 15, 2021

(Date)

* * * * * * *

Professor Koushik Sen
Second Reader

12/15/2021

(Date)

YouVerify: An Intermediate Representation and Framework for Symbolic Execution

by

Griffin Christian Prechter

A thesis submitted in partial satisfaction of the

requirements for the degree of

5th Year Masters of Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Raluca Ada Popa, Co-chair
Professor Koushik Sen, Co-chair

Fall 2021

Abstract

YouVerify: An Intermediate Representation and Framework for Symbolic Execution

by

Griffin Christian Prechter

5th Year Masters of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Raluca Ada Popa, Co-chair

Professor Koushik Sen, Co-chair

Symbolic execution is a popular dynamic program analysis technique in which a program is executed with symbolic, or variable, inputs in place of concrete ones. During the execution of a program, multiple paths may be explored due to the presence of symbolic values, producing several states. These states are guarded by path constraints, conditions that indicate what values the symbolic inputs should take on to produce a given state. Symbolic execution has many applications ranging from testing and bug discovery to program analysis and verification. Recently, leveraging domain specific knowledge, modified symbolic execution algorithms have been utilized to solve computer security problems through program analysis. One example, ObliCheck, implements novel symbolic execution techniques to efficiently verify the obliviousness of algorithms modeled in a subset of JavaScript. In this report, I identify two key challenges in implementing custom symbolic execution algorithms: First, features in modern languages are often difficult to support in symbolic execution frameworks due to the constraints imposed by underlying SMT solvers. Second, it is challenging and time consuming to modify existing symbolic execution frameworks to rapidly prototype custom symbolic execution algorithms. To address these challenges, I present YouVerify, an intermediate representation (IR) for symbolic execution built as an abstraction layer directly above the SMT-Lib family of solvers. YouVerify provides a flexible API that separates the state representation and exploration from the IR language and symbolic interpreter. I provide a verified implementation of the framework with a default symbolic execution algorithm. I additionally provide a prototype implementation of ObliCheck's symbolic execution algorithm modifications to demonstrate YouVerify's effectiveness for prototyping symbolic execution techniques.

For Kim and my parents, Daniela and Chris.

Contents

List of Figures

List of Tables

Acknowledgments

I am extremely grateful for the unwavering support of both my advisors, Professors Raluca Ada Popa and Koushik Sen, throughout the 5th Year Masters program. I reached out before my senior year with no experience in research, and both were extremely supportive and motivated me to get involved and to pursue the masters. Were it not for their support and the opportunity to do the program, I would have never been able to grow and pursue my interests in the way that I have. I would like to thank Jeongseok Son and Rishabh Poddar for their mentorship while working on ObliCheck, which inspired to me to investigate symbolic execution for this project. I would like to thank Vivian Fang, for her help and guidance throughout the program. Finally, I would like to thank Kevin Laeufer for his support and guidance during my work on YouVerify.

Chapter 1

Introduction

Symbolic execution is a popular dynamic program analysis technique where a program is run with symbolic, or variable, inputs in place of concrete ones. During the execution of the program, multiple paths may be followed due to the presence of symbolic values, producing a number of states. These states are guarded by path constraints, conditions that indicate what values the symbolic inputs should take on to produce a certain state. Symbolic execution has a number of applications ranging from testing and verification, to analyzing programs for validating certain properties Recently, leveraging domain specific knowledge, modified symbolic execution algorithms have been utilized to efficiently solve computer security problems through program analysis. In this paper, we will be particularly investigating ObliCheck [21], a modified symbolic execution algorithm to verify the obliviousness of code in JavaScript. In this paper, I identify two key challenges in implementing custom symbolic execution algorithms: first, language features in modern languages are often difficult to support in symbolic execution frameworks due to the constraints imposed by underlying SMT solvers. Second, it is challenging and time consuming to modify existing symbolic execution frameworks to rapidly prototype custom symbolic execution algorithms. To address these challenges, in this paper I present YouVerify, and intermediate representation (IR) for symbolic execution built as an abstraction layer directly above the SMT-Lib family of solvers. I provide a verified implementation capable of symbolically executing complex programs written in the IR. And I additionally provide a prototyped implementation of some techniques presented in ObliCheck symbolic execution algorithm to demonstrate its effectiveness for implementing symbolic execution algorithms.

1.1 Symbolic Execution

Symbolic execution [3, 10, 11, 12] is a dynamic program analysis technique, introduced in the mid '70s, now used in a variety of applications from test generation and verification, to validating computer security properties of programs [1]. Whereas a normal program is run concretely, following a single control path through the code based on input values, symbolic

execution explores a variety of control paths, producing a richer model of the program's overall behavior. This is accomplished by substituting concrete values for symbolic, or variable, ones. With certain values unbound to a singular concrete value, after encountering conditional branching statements that depend on that symbolic value, multiple control paths must be explored to account for the vast domain that the symbolic inputs could occupy. As a running example of symbolic execution, in Figure 1, there is modified sample program from the MultiSE [20] paper (a paper we will dive into later in the report):

```
1:  var x: INT = $symbolic_input('x')
2:  var r: INT = $symbolic_input('r')
3:  var z: INT = $symbolic_input('z')
4:  x = 2 * x
5:  if x > 100:
6:      if z == 1:
7:          r = 3
8:  if r > 1:
9:      z = r - 1
```

Figure 1: A sample program with all symbolic inputs.

In *vanilla* dynamic symbolic execution, the program execution begins with only a single state, where (in our example) all the variables are only their symbolic inputs, and the *path constraint* is **true**. The *path constraint* of a state in symbolic execution acts as a guard for that specific state, specifying a condition that must be met in order for the variables to equal the given values in the state. When the symbolic execution reaches line 4, because `x` is a symbolic value, the new value of x becomes the symbolic *expression* `2 * x`. However, the symbolic execution begins to get particularly interesting when line 5 is reached. As we know, the value of x is purely a symbolic expression, `2 * x`, and thus could take on any integer multiple of 2. When evaluating the condition of the if statement, the decision of whether or not to enter the statement is unclear. The original input value of x could possibly be greater than 50, or it could be less. The essence of symbolic execution is that, because neither path is infeasible, both must be followed. This conditional statement causes the state to branch into 2 new states, with updated path constraints.

Previously, the path constraint was trivially true because there was only one initial state. However, now that the symbolic execution is following two separate paths, variables may take on values that are exclusive to only a single path. The states after the if statement is evaluated by the symbolic execution are shown in Figure 2; the program counter (`pc`) indicates what line that state will execute next. Values of the form `$<name>` indicate the initial symbolic input for that variable name. Note that only the `pc` differs between the two states `S0` refers to the path in which the if statement was entered, and `S1` refers to the path

if it was not. The symbolic execution continues until ultimately we arrive at the final state shown in Figure 3. Each state has a path constraint that now guards unique sets of values for the variables.

```
S0:  ((2 * $x) > 100): {pc: 6, x: (2 * $x), r: $r, z: $z}
S1: !((2 * $x) > 100): {pc: 8, x: (2 * $x), r: $r, z: $z}
```

Figure 2: States of program in Figure 1 after executing line 5.

```
S0:  ((2 * $x) > 100) & ($z==1) & (3 > 1): {pc: 10, x: (2 * $x), r: 3, z: 2}
S1:  ((2 * $x) > 100) & ($z==1) & !(3 > 1): {pc: 10, x: (2 * $x), r: 3, z: $z}
S2:  ((2 * $x) > 100) & !($z==1) & ($r > 1): {pc: 8, x: (2 * $x), r: $r, z: $r - 1}
S3:  ((2 * $x) > 100) & !($z==1) & !(r > 1): {pc: 8, x: (2 * $x), r: $r, z: $z}
S4: !((2 * $x) > 100) & ($r > 1): {pc: 10, x: (2 * $x), r: $r, z: $r - 1}
S5: !((2 * $x) > 100) & !($r > 1): {pc: 10, x: (2 * $x), r: $r, z: $z}
```

Figure 3: States of program in Figure 1 after completion.

SMT Solvers and SMT-Lib [2]

Now that the symbolic execution of the program from Figure 1 is completed, we can see all 6 end states in Figure 3. Observing state S1, we can notice something peculiar about that state in particular: its path constraint is always false. Because its path constraint is the conjunction of all conditional statements along its control flow, and one of those conditions reduces to false, that path is infeasible and will never occur during the execution of the program. This is because, when `$z == 1` and `((2 * $x) > 100)` are true, then the variable `r` is assigned to `3`. Thus, the third conditional on line 8 will *always* be true for that path. Through determining the *satisfiability* of the path constraints we can prune unfeasible paths from the final state, or during symbolic execution.

The task of determining the satisfiability of these path constraints, and for finding symbolic variable assignments for each path is offloaded to **Satisfiability Modulo Theories** (SMT) solvers, which solve the problem of determining whether or not these mathematical formulas are solvable. An extension of boolean SAT problems, these first-order formulas can include real numbers, integers, bit vectors, arrays, strings, and more. Each of these types, or **sorts** belongs to a *theory*, which is a set of specialized methods for solving formulas containing that sort allowing for more efficient solving than traditional theorem prover methods.

A particularly popular families of SMT Solvers fall under the international SMT-Lib standard [2], created with the intention that common standards and a set of benchmarks would facilitate the advancement, evaluation, and comparison of SMT solvers. Thanks to the thorough description of the various theories provided by SMT-Lib, there are a number of SMT-Lib compliant solvers that can be used interchangeably when solving formulae written using the SMT-Lib theories. Two well known solvers in the SMT-Lib family are z3 and cvc4.

We can check the satisfiability of `S4`, to see that it is satisfiable, and using the z3 SMT solver we can get a model that makes the formula satisfiable to produce valid inputs for that state. Plugging the path constraint into z3, we get the model: `{$x: 50, $r: 2, $z: 0}`. Plugging these values into the path constraint we can see that it is true. Next, we can determine the values of the variables at the end of the program by plugging in the symbolic values and simplifying the expressions: `{x: 100, r: 2, z: 1}`.

1.2 Applications of Symbolic Execution

A wealth of work has explored the applications of symbolic execution from test generation, to the verification of program properties, like security requirements. We have discussed a simple example of dynamic symbolic execution, and we've seen a simple example go through an SMT solver to produce a model assigning the initial symbolic inputs. Now, we will look into symbolic execution's applications. Some of the tools mentioned will be explored more in depth in the Related Works Section.

Test Generation and Bug Discovery

Due to symbolic execution's, and similar technique's, ability to explore the space of possible program execution paths, it is a popular technique for test generation and for discovery bugs and issues in programs. Works like EXE [6] and KLEE [5] use symbolic execution to explore a program symbolically in order to discover a program path that leads to an error or undefined behavior in C programs. In their paper, the authors of KLEE analyze popular open source libraries and are able to discover a number of serious vulnerabilities. Through symbolic execution, the authors are able to identify paths that lead to errors and can extract the necessary model to generate an example input to reproduce the error through concrete execution.

In the papers CUTE [18] and DART [9], a technique that merges concrete and symbolic execution known as concolic execution is introduced. With concolic execution, path constraints for conditional statements are gathered during a round of concrete execution. With this, it's possible to discover new execution paths by modifying the gathered path constraint and purposefully switching branching directions to generate inputs that will explore other paths through the program, aiming for maximum code coverage.

Program Analysis and Verification

BitBlaze [22] is a project at Berkeley that works on both the design of a Binary Analysis Platform, and its application to real security problems. The effort works to leverage both static and dynamic program analysis techniques to analyze real code at the bit level. Through this the project strives to unveil malicious security exploits and better prevent future security compromises. Symbolic Execution is one of the methods that is used in BitBlaze's Binary Analysis Platform.

The Binary Analysis Platform (BAP) [4] is an infrastructure for performing program verification and analysis tasks on binary code. BAP exposes all side effects of assembly instructions in an intermediate language, which enables analyses to be written in a syntax-directed manner. BAP is capable of generating verification conditions, a boolean predicate that is true if and only if some property holds over the program's execution for a given input. BAP is also packaged with various other analysis and optimizations involving the intermediate language. In their report, the authors showcase some of the particular applications of the tool, such as generating and checking verification conditions to determine if overflow flags would be set, or if return operations would be overwritten. The authors also perform automatic exploit generation and malware analysis.

In their followup [14] to their initial paper, the authors of Gillian [8] utilize their multi-language platform to verify both the JavaScript and C implementations of the AWS Encryption SDK message header deserialization module, and were able to identify 5 bugs in total. This was achieved by first implementing a language-independent specification of the message header decoder program, followed by a language specific specification for the C and JavaScript implementations. Then, the necessary code annotations were added to the implementations.

There have also been efforts to apply symbolic execution particularly to hardware design and generating or discovering exploits and vulnerabilities in hardware. In *A Recursive Strategy for Symbolic Execution to Find Exploits in Hardware Design* [25] the authors define a strategy for their recursive reasoning with hardware-oriented symbolic execution, and they also showcase heuristics that they used to allow their strategy to succeed. The authors demonstrate the effectiveness of the tool, demonstrating that it's able to identify bugs in a processor design that even industry level tools like Cadence could not identify. In *Kronos* [15], the author produces a MicroTitan SoC that demonstrates a security property known as output determinism, which can show that a system provides noninterference without requiring an SoC's state be fully reset, and utilizes techniques like symbolic execution to formally verify the property.

In another recent work, ObliCheck [21], Son et. al. acknowledge one of symbolic execution's weaknesses in the path explosion problem and devise a modified symbolic execution algorithm leveraging domain specific knowledge. In ObliCheck, symbolic execution (particularly MultiSE [20]) is used to verify the obliviousness property of a number of algorithms seen in computer security applications. Obliviousness is a property that a program does not vary its trace with different private inputs, such that an attacker cannot leverage leaked

information to infer the contents of the private data. Understanding that only certain components of the program are important when reasoning about the obliviousness of algorithms, an aggressive merging and selective un-merging technique are utilized to increase efficiency, maintaining correct results.

1.3 YouVerify: An Intermediate Representation for Symbolic Execution

Largely informed by my experience working on ObliCheck [21], in this project report, I present **YouVerify**, which is a simple imperative intermediate representation for symbolic execution, packaged with a framework for rapidly prototyping and testing symbolic execution algorithms and optimizations. Two crucial lessons learned from working on

YouVerify was designed with a number of core principles in mind. First, YouVerify is intended to be used as an intermediate representation for target languages to compile to. The syntax of the language is three-address code, this representation is commonly used by compilers in order to perform optimizations. Barring some exceptions, in YouVerify most expressions have at most three atomic expressions in use. Second, YouVerify acts as an abstraction layer directly above the popular families of SMT Solvers that fall under the international SMT-Lib standard, rather than targeting a specific higher-level language's features, some of which may not be solvable by SMTLib constraint solvers. Third, YouVerify was designed to be wholly modular and extensible.

Modularity and extensibility are at the core of YouVerify's implementation and motivation. In this paper, I present the initial implementation of the YouVerify IR language, and the YouVerify framework and API. Using the YouVerify framework I implemented vanilla dynamic symbolic execution, in subsequent sections I utilize the API further to implement another symbolic execution algorithm, MultiSE which uses a novel representation of state. Finally, I modify the MultiSE implementation to implement a subset of the enhancements presented in the ObliCheck paper.

The implementation for YouVerify presented in this paper is available on GitHub: https://github.com/gprechter/youverify

Chapter 2

Background and Related Works

Symbolic Execution, and similar techniques, are rich areas of research. There are a variety of tools used for test generation, different symbolic execution algorithms, and some solver aided languages or frameworks aiming to democratize access to implementing custom symbolic execution algorithms. In this section we will cover some these related works and delve deeper into one particular application of symbolic execution, ObliCheck, a project I worked on that strongly motivated YouVerify. To give ObliCheck and it's custom symbolic execution algorithm more context, I will also delve deeper into MultiSE, an efficient symbolic execution algorithm with a unique state representation.

2.1 Symbolic Execution Tools and Solver-Aided Languages

EXE [6] and KLEE [5]

In *EXE* [6], the authors present an effective bug-finding tool that automatically generates test inputs that lead to program crashes through the use of an additionally introduced constraint solver: *STP*. By running programs symbolically, EXE solves the state's path constraint in order to find concrete input values to replicate the crash. The STP solver is a decision procedure for particularly bitvectors and arrays, and makes strides in array optimizations, as reasoning about arrays was a main bottleneck for EXE's efficiency. Of the many optimizations made to the symbolic execution algorithm for EXE, one of the main enhancements is the use of constraint caching coupled with constraint independence. EXE maintains a persistent cache of solver queries, saving time by avoiding repetitive, and potentially very expensive, solver calls. To make the most of EXE's caching scheme, constraints are often divided into smaller, independent subsets when possible allowing for unnecessary constraints to be discarded, and increasing cache hits especially if a symbolic expression appears in multiple distinct larger constraints. Additionally, EXE introduces additional search heuristics for exploring a program's execution path space through

encouraging the exploration of in-frequently visited statements when deciding between different possible states.

In *KLEE* [5], a follow up to EXE, the authors build upon their previous work to introduce a novel symbolic execution tool for test generation and bug detection. KLEE operates on LLVM [13] byte code, interpreting the virtual instruction set. Most operations like addition, simply perform their standard operations on symbolic expressions to create new symbolic expressions. Conditional branches, if both branched states are feasible, clone the state and explore both paths. For handling load and store operations, KLEE maps memory objects to their own distinct STP array. When in the position of dereferencing a pointer that could refer to N objects, KLEE opts to clone the state N times. Like with EXE, KLEE introduces a variety of query optimizations and state scheduling techniques to increase performance. Particularly, KLEE performs expression rewriting, which simplifies constraints, along with constraint set simplification, in which equality constraints cause previous constraints involving the same variables to be rewritten and simplified.

KLEE also builds upon EXE's solver query cache by introducing a counter-example cache, which map constraints to counter examples. For example, a subset of an already unsatisfiable constraint set, cannot be satisfiable; whereas a superset of a satisfiable one is satisfiable. Through this optimization, time spent in the STP solver was reduced from 92% of execution time to only 41%. KLEE also makes enhancements to state scheduling by maintaining a binary tree representing the program path followed by active states. KLEE's algorithm prefers states high in the branch tree, ones with less constraints, which favors breadth rather than depth in path exploration. This also avoids *starvation*, where some state space rapidly creates many new states. KLEE also favors states that are close to uncovered instructions, in an effort to optimize code coverage in particular. KLEE was able to automatically generate tests that on average covered 90% of code for 160 complex examples. KLEE was also able to identify 56 serious bugs, 10 of which were in the `COREUTILS` library.

DART [9], CUTE [18], and Concolic Execution

Concolic Execution is a strategy often used to generate test inputs and verify code. Concolic Execution cleverly blends together concrete execution with symbolic execution to build and understanding

In *DART: Dynamic Automated Random Testing* [9], the authors implement a tool for automatically testing software. The tool first automatically statically analyzes a code to determine its interface, with this model a test driver is automatically generated to randomly test the code module. Dynamic program analysis is then used to understand how the program behaves under random inputs in order to direct subsequent executions along different paths.

To gather knowledge about a program being executed, DART executes a program in a manner called *directed search*. With initially random inputs, during execution an input vector for the next round of execution is gathered, consisting of symbolic constraints from

the branching statement conditions. With these predicates, DART can cleverly choose new paths of the program to execute, with an overall goal of exploring all possible execution paths. In cases where an SMT solver would be unable to solve a given symbolic expression, DART instead falls back on the concrete evaluation of such an expression through maintaining both a concrete memory and a symbolic memory. For interacting with the environment through external functions, these functions will return a random value of their return type and be treated as a 'black-box'.

In *CUTE: A Concolic Unit Testing Engine for C* [18], the authors expand upon the ideas and approach in *DART*, blending together concrete and symbolic execution for test input generation. *CUTE* addresses the problem of generating test inputs using concolic execution where a function contains pointer arguments and takes in a memory graph as its input. First, *CUTE* uses a logical input map, representing the memory graph, to generate a concrete memory graph, and two symbolic states for the pointer values and primitive values. The program is run concretely but symbolic constraints are collected during the execution, by negating a constraint after execution, a new input map can be generated. Unlike *DART*, *CUTE* does not generate random pointer graphs and instead assigns all new pointers to NULL. *CUTE* also assumes that there are no external functions.

Rosette [24]

In the Rosette paper, the authors recognize the efficacy of applying solver-aided tools to domain specific languages in particular due to a variety of factors. Namely, the authors recognize that domain specific languages (DSLs) are especially popular in modern programming, and that programs written with DSLs are typically smaller, and thus easier to be optimized using domain specific knowledge and invariants. Rosette, is a host platform for these solver-aided domain specific languages (SDSLs), as implementing these languages in a host language that is solver-aided can ease the burden of translating constraints. [24, 23] Rosette is implemented using Racket, and the authors demonstrate how it can be used to host SDSLs without the need for constructing a complicated symbolic compiler.

Rosette features a lightweight design, and the decision to implement it in Racket allows it to utilize a lot of that language's meta-programming features. Rather than implementing a compiler to create a solver-aided tool, a SDSL is embedded into Rosette, automatically inheriting its features. One key design decision for Rosette is the idea of only compiling a subset of Racket to constraints. This subset is known as the *symbolic core*. All Rosette programs can make use of the underlying Racket's features, as long as those features not in the symbolic core are evaluated before symbolic compilation. In their paper, the authors demonstrate Rosette's effectiveness by implementing a number of solver-aided systems in their framework in only a few weeks.

Gillian [8]

Gillian [8] is an IR-based platform designed to make the development of symbolic-execution tools more accessible. One of Gillian's highlights is the intermediate language that the authors implemented known as GIL, which operates over a parametric memory model of a given target language. In order to use Gillian to operate symbolic execution over a given target language, a developer would need to provide a concrete and symbolic memory model along with a compiler from the target language to GIL preserving its memory semantics.

Gillian's symbolic execution engine was implemented by the authors in OCaml. Gillian also distinctly separates the variable store and the memory model of symbolic execution, automatically handling the variable store and only leaving the concrete and symbolic memory model for developers to handle. To demonstrate the effectiveness of the platform, the authors employ Gillian to create symbolic testing tools for JavaScript and C: Gillian-JS and Gillian-C respectively. In their follow-up paper Gillian Part 2 [14], both of these instantiations of Gillian are exercised to find vulnerabilities in AWS code. In order to implement a new compiler and memory model in Gillian, the authors suggest that a developer must have an in-depth understanding of the language standard, a working knowledge of OCaml, and a basic understanding of the Gillian interface.

2.2 MultiSE [20]

As mentioned previously, one of the most pernicious challenges with using symbolic execution is the path explosion problem. With a symbolic execution engine that performs no path merging, each state is split in two with each conditional expression that relies on symbolic variables. Because of this, the complexity of symbolic execution exponentially increases with the number of conditional expressions, like if statements. In MultiSE [20], the authors propose a novel symbolic execution technique for incrementally merging state without the need for any auxiliary variables.

State merging is a technique that attempts to mitigate path explosion by merging certain paths at join points. This is done through the introduction of new symbolic values, referred to as *auxiliary variables*. As an example, if there are two states where the a variable takes on two distinct values, the values can be replaced with an auxiliary variable. Then, in the path constraint, the old values of the variable can be represented. MultiSE identifies the most significant of the problems facing the use of auxiliary variables as being their lack to represent values outside of the domain of a constraint solver, as clearly these variables cannot be added to the path constraint.

Addressing the weaknesses of merging through auxiliary variables, MultiSE proposes a novel state representation through *value summaries*, which are guarded symbolic expressions. Each value summary contains a variable, and then a mapping of path constraint, value pairs. As an example, `x -> {(p, v1), (!p, v2)}` is a simple value summary. It represents two possible states for the variable `x`. If the *predicate* p is true, then `x` would take on

the value *v1*. Otherwise, `x` would be equal to *v2*. Bypassing auxiliary variables MultiSE is capable of continuing symbolic execution even in the face of values without auxiliary variables. Simplification, and concrete evaluation is also much simpler without auxiliary variables, saving expensive time in a solver. With an auxiliary variable, even a variable with only concrete values cannot be evaluated concretely. Additionally, MultiSE uses a new algorithm for updating value summaries to perform merging incrementally.

When performing symbolic execution in MultiSE, first, the value summary for the program counter is inspected. A predicate-value pair is chosen and then the statement indicated by the value for the program counter is executed symbolically. First, let's consider the case that the statement is a conditional branch, `if r > 1:`. MultiSE would then need to look at all the predicate-value pairs for the variable `r`. We will assume that the value summary for `r` is `{(!p5, $r), (p5, 3)}`, plugging these into the conditional expression we get: `{(!p5, $r > 1), (p5, true)}`. Now, the path constraints for the program counter can be updated. Assuming the current path constraint is `p3`, MultiSE updates the path constraint using the predicate-value pairs from the conditional expression:`p6 = p3 | ((!p5 & $r > 1) | (p5 & true))`. The pc value summary is then updated with the new pairs where the pc guarded with `p6` is the taken state, and the pc guarded with `!p6` is the not-taken state.

Other than conditional branching statements, the manner in which an assignment statement changes state differs as well. Again, a program counter predicate-value pair is selected: `(p6, 10)`, now referring to an assignment statement: `z = r - 1`. We will again assume that the value summary for `r` is `{(!p5, $r), (p5, 3)}`, plugging these into the expression we get: `{(!p5, $r - 1), (p5, 2)}`. For this assignment expression though, since the program counter is guarded by `p6`, the new values of `z` must also be guarded by `p6`. The old values of `z` are thus guarded by `!p6`, and the updated value summary becomes: `{(!p6, $z), (p6 & !p5, $r - 1), (p6 & p5, 2)}`. Another enhancement of MultiSE is its use of binary decision diagrams to represent path constraints, which provides an efficient way of avoiding exploring infeasible paths. A guard's BDD representation is checked first to determine if the boolean formula is false, before its satisfiability is checked by a solver.

To demonstrate how MultiSE differs from typical dynamic symbolic execution, we again refer to the example in Figure 1 in the report's introduction. The resulting state, in value summary representation, in shown in Figure 1.

2.3 ObliCheck [21, 17]

Most distributed computer systems leverage encryption as a tool for transmitting sensitive data over the network. In order to prevent attackers from inspecting the contents of messages, strong encryption algorithms are used. While the content of this data may be occluded from an attacker, there are many attributes about the data transfer that they could observe in the clear. In fact, an attacker can actually infer information about private data by analyzing network or disk traffic. Such attacks are known as access pattern leakage attacks. A family of

```
{
    pc -> {(true, 10)}
    x  -> {(true, 2$x)}
    z  -> {(p1 | p3, $r - 1), (p2 | p4, $z), (p5, 2)}
    r  -> {(!p5, $r), (p5, 2)}
}
p1 = 2$x <= 100 & $r > 1
p2 = 2$x <= 100 & $r <= 1
p3 = 2$x > 100 & $z != 1 & $r > 1
p4 = 2$x > 100 & $z != 1 & $r <= 1
p3 = 2$x > 100 & $z == 1
```

Figure 1: Value summaries for variables of program in Figure 1 after completion.

algorithms that are data-oblivious algorithms protect against these kinds of attacks. These data-oblivious algorithms aim to ensure that publicly available traces are independent of private inputs in order to close these side channels and prevent against leakage. Oblivious algorithms are already in use today. Before ObliCheck [21], traditional pen-and-paper proofs were used to verify the obliviousness of these algorithms. It's important to note that oblivious algorithms can be difficult to design and understand, as algorithm developers must balance efficiency and security. In *ObliCheck* [21], Son et. al. propose a tool for automatically verifying the obliviousness of distributed data processing algorithms. The tool provides an API enabling designers to implement their algorithm, and check its obliviousness. In addition to determining the obliviousness of an algorithm, ObliCheck can provide a proof of the algorithm's obliviousness, or a counter-example. The tool is implemented using Jalangi [19], a MultiSE symbolic execution framework. The tool is able to achieve up to 260x speed increases over tradition symbolic execution methods, and does not sacrifice accuracy.

Since later in this paper, we will be implementing a subset of the ObliCheck symbolic execution algorithm enhancements, I will demonstrate how those features work in more detail below through following an example from my class report for ObliCheck[17]. For example, below is a trivial oblivious program for demonstration purposes, shown with the final program state after being run in unmodified MultiSE:

```
var privIn = readPrivateInput();
var buf = [];
if (privIn)
    buf.push(0);
else
    buf.push(1);
encAndSend(buf);
```

On the final line, an API call moves the private data from unobservable to observable space, the symbolic execution state of the program is as follows:

```
{
    pc = [{true, 7}],
    privIn = [{true, sym0}],
    buf = [{sym0, [0]}, {!sym0, [1]}],
    buf_len = [{true, 1}]
}
```

For this oblivious execution, regardless of the private input value, the length of the buffer is always 1. Because of this, this algorithm is oblivious. Because of the path explosion problem, as algorithms become more complex and the length of buffers increases the runtime of MultiSE suffers.

Optimistic State Merging

In order to mitigate the path explosion problem and decrease the burden on one's machine when running symbolic execution, domain specific insights about the nature of the input values and inner-workings of the programs was leveraged in order to increase efficiency. One of the key contributions of ObliCheck is the recognition that the contents of the output data will be **encrypted** and therefore is often not important to be retained, this can allow symbolic execution to scale in a feasible way.

If a given variable will not contribute to the outcome of the obliviousness property, then this information can be generalized to increase efficiency. In ObliCheck, such variables are aggressively merged such that the number of states that are present is significantly less. In the previous example that we saw, the new symbolic execution state would be as follows:

```
{
    pc = [{true, 7}],
    privIn = [{true, sym0}],
    buf = [{true, [sym1]}],
    buf_len = [{true, 1}]
}
```

The value of the buffer is now a newly introduced symbolic variable, since we don't need to keep track of the original values. The number of states in the program has now halved; as programs increase in complexity, the effect of reduced paths is more pronounced.

However, by aggressively merging, it's possible to lose path specific information. It's possible that variable can be merged that isn't directly involved in the trace for an algorithms verification condition but that is still involved in the control flow of a program such that it makes an algorithm not oblivious. This domain specific merging scheme on its own could

incur a false positive, where the program is incorrectly flagged as being not oblivious. Take the example program:

```
var privIn = readPrivateInput();
var buf = [];
if (privIn)
    buf.push(0);
else
    buf.push(1);

if (buf[0] == 0)
    buf.push(1);
if (buf[0] == 1)
    buf.push(0);

encAndSend(buf);
```

In this program, the second set of if statements depends on the value of the first element pushed to the buffer. Note, however, that the *length* of the buffer is always the same. If the value first added to the buffer was optimistically merged it would be impossible for the symbolic execution framework to know that buf[0] was indeed either 0 or 1; it could believe that the value falls outside of that narrow domain. The final state of the program is now as follows:

```
{
    pc = [{true, 7}],
    privIn = [{true, sym0}],
    buf = [{true, [s1, s2, s3]}],
      buf_len = [ {s1 != 0 && s1 == 1 || s1 == 0 && s1 != 1, 2},
    {s1 != 0 && s1 != 1, 1} ]
}
```

Iterative Un-merging

Domain specific merging could drastically reduce the memory and runtime used by symbolic execution. The use of domain specific merging did however lead to false negatives, as important information was lost that occluded the program's true obliviousness. As seen in the example from the previous sub-section, had the variable 's1' not been introduced, and that value not had been merged, the program would have correctly been flagged as oblivious. Un-merging and re-execution must be performed to determine the true status of the program.

The first step in un-merging specific problematic auxiliary variables is by analyzing the verification condition's path constraints. Since, during symbolic execution, there is no way

of knowing all of the ways that a variable will be used, it is unclear on wether or not a variable should be merged. Fortunately, the variables that may have led to a false positive case are present in the path condition of the verification condition. In the case of the prior example, 's1' is present in the path condition for the buffer's length. Symbolic variables that were introduced by OSM present in the path condition denote that the result could be a false negative. After identifying the target variables, they are flagged to not be merged, and the program is re-executed. Once the program is flagged as oblivious, or there are no more variables in the verification condition to un-merge, a conclusion can be made about the obliviousness of the program.

```
{
    pc = [{true, 7}],
    privIn = [{true, sym0}],
    buf = [{sym0, [0, 1]}, {!sym0, [1, 0]}],
    buf_len = [{true, 2}]
}
```

In ObliCheck [21], optimistic state merging significantly mitigated the path explosion problem, and iterative un-merging recovered any lost accuracy. With iterative un-merging, since all variables are merged on the first execution, it is extremely fast to execute, even symbolically.

Chapter 3

YouVerify

In ObliCheck [21], domain specific knowledge regarding the types of algorithms being analyzed was leveraged in order to mitigate the path explosion problem facing symbolic execution. For the ObliCheck project specifically, the symbolic execution framework Jalangi [19] was augmented with ObliCheck's custom algorithm involving aggressive optimistic state merging and iterative state un-merging. Largely informed by my experience working on ObliCheck, in this project report, I present **YouVerify**, which is a simple imperative intermediate representation for symbolic execution, intended to be framework for rapidly prototyping and testing symbolic execution algorithms and optimizations.

3.1 Design

Design Principles

YouVerify was designed around a number of core principles. First, YouVerify is intended to be used as an intermediate representation for target languages to compile to. The syntax of the language is three-address code, this representation is commonly used by compilers in order to perform optimizations. Barring some exceptions, in YouVerify most expressions have at most three atomic expressions in use. Second, YouVerify acts as an abstraction layer directly above the popular families of SMT Solvers that fall under the international SMT-Lib [2] standard, rather than targeting a specific higher-level language's features, some of which may not be solvable by SMTLib constraint solvers. Third, YouVerify was designed to be wholly modular and extensible.

A limiting factor in the application of some symbolic execution algorithms to a broader set of problems and programs is the language in which those programs are implemented in. Many symbolic execution tools and frameworks operate only on a single target language, whether that be C, JavaScript, or another language. For these tools and frameworks, the benefits of their specific implementation optimizations and techniques can only be reaped by analyzed programs in the target language. If one wanted to leverage MultiSE's value

summary representation to symbolically execute C code for instance, they would not be able to use the existing implementation, Jalangi. Instead, the algorithm must be implemented fresh, or by modifying another existing symbolic execution tool. The benefit of utilizing an intermediate representation like YouVerify is that the problem of target language can be separated from the underlying symbolic execution algorithm.

The YouVerify language also is intended to be an abstraction layer directly above the SMT-Lib [2] family of solvers, in order to avoid generating un-solvable constraints. Every operator in YouVerify is directly reflected by the SMTLib standard, and thus supported by all SMTLib compliant solvers. By adhering to this principle, all of the implemented theories in YouVerify cover the complete set of operators available, and a user can decide what solver to use as long as it supports the SMTLib standard. There is not concern that an algorithm implemented in YouVerify will utilize some language feature that is not supported by the underlying solver because every statement or expression written in the language is supported by SMTLib. The ability to easily change out solvers also lends itself to the frameworks interoperability and modularity, eliminating the reliance on a single solver.

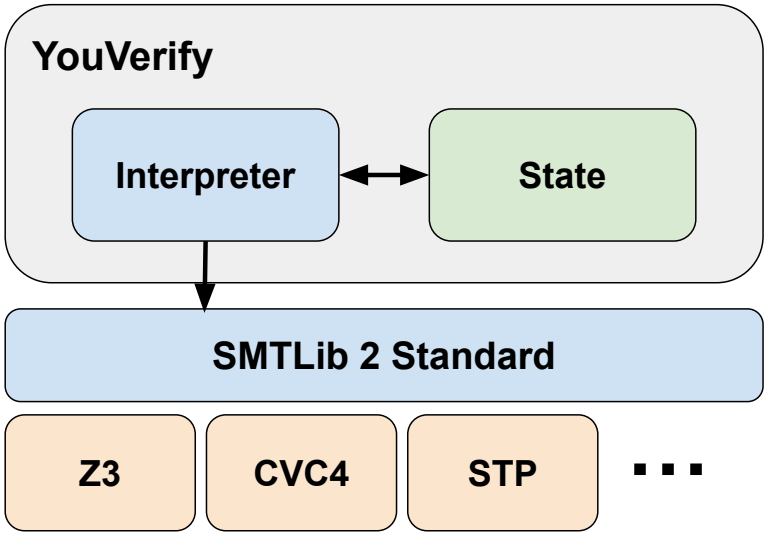
Modularity and extensibility are at the core of YouVerify's implementation and motivation. As mentioned earlier, it's easy to swap out solvers using YouVerify. In addition to this, it is also straightforward to add additional sorts and operators later on to the framework if perhaps an unimplemented theory is desired or a new theory becomes available in the SMTLib standard. For simple operators like unary and binary ones, there is easy support for adding them using prefix and infix notation without the need for modifying the parser or the framework itself. For more complex operators, it is just as straight forward to add builtin functions.

The symbolic executor, which traverses through the state and executes statements, is largely static, and is not intended to be modified or extended. However, the state itself is fully modular and must only adhere to the provided API such that it is able to plug into the symbolic executor. Because there is much variation in the way that the state in particular is represented by different symbolic execution tools and algorithms, and how the path space of a program is explored, these algorithm dependent decisions are left up to the state entirely which the executor accesses through the API.

System Architecture

All YouVerify programs are interpreted as a series of statements that perform operations on the program *state*. The state is at the heart of YouVerfiy's architecture. As we've seen in our discussion of background and related works, the manner in which state is represented varies widely between different symbolic execution techniques and algorithms. In standard dynamic symbolic execution, there are multiple states, all with their distinct symbolic variable stores and state is copied when a branching statement is reached. On the other hand, with MultiSE, there is a collection of value summaries for each variable that encodes the variety of program states. In some implementations state can be merged, and in others it is not. Because of this variety in the representation and management of state, YouVerify treats the state as

being separate from the symbolic interpreter and interacts with it through an API that will be described in more detail below.



This separation of responsibility allows the main execution loop for YouVerify to be very simple, as it can be seen in Figure 1. In YouVerify, the symbolic execution of a program will continue as long as the 'next state', retrieved from the state using the `update_state` method returns a `True` value in Python. As shown in the exploration of various symbolic execution techniques, how the state space is explored is a matter of interest when developing symbolic execution algorithms. For instance, the approaches see in EXE [6] and KLEE [5] use different heuristics in order to explore the program path space in order to achieve the desired result.

```
while state.update_state():
    stmt = state.current_statement
    stmt.exec(state)
```

Figure 1: Main execution loop for YouVerify interpreter.

3.2 Language Features

Each YouVerify program (file extension `.yvr`) is composed as follows, divided into two sections:

```
<GLOBAL DECLARATIONS>
...
<STATEMENTS>
...
```

At the top of the program file, there are the global definitions consisting of **global variable declarations** along with **function declarations**, and **record type declarations**. Functions can only be declared globally. Below all of the declarations are the statements, or the body of the program.

Variable Declarations

```
x: INT # Integer type declaration
b: BOOL # Boolean type declaration
bv: BV[32] # A bit vector of length 32 bits
arr: ARRAY{INT} # Array type declaration of array with integer elements
rec: point     # record type declaration
```

A variable declaration is written using the following syntax:

```
<variable name> : <built-in sort | custom record type>
```

Theories and Sorts

By default, the **Core**, **Integer**, **BitVector**, and **Array** theories and their included sorts are included and implemented in YouVerify.

Core Theory and Boolean Sort

The first theory that is implemented is the core theory, which consists of the Boolean sort and various unary, binary, and the only ternary operator. Below is an example of declaring two boolean variables and assigning them to the true and false constant.

```
t, f: BOOL
t = true
f = false
```

Integer Theory

The integer theory includes common arithmetic operators and the Integer Sort. Below is an example of declaring an integer variable and assigning a constant to it.

```
x: INT
x = 2021
```

BitVector Theory

The BitVector theory includes the BitVector sort, along with many functions over bit vectors. In YouVerify, all operators over bit vectors are implemented as built-in functions rather than infix notation operators. When declaring a BitVector variable, and creating a BitVector value, the *length* of the bit vector has to be specified as shown below:

```
x: BV[32]
y: BV[8]
x = BV{32, 2021}
y = BV{8, 31}
```

Array Theory

The Array theory is unique because it has an arity of two and is specified along with another sort for its key type and its value type. In YouVerify, all arrays only have the Integer sort for its key type. The value type must be specified when declaring a variables and allocating a new array.

```
# Declare and initialize an array of length 10 to bitvectors of width 8
# and value 0.
# Declare a pointer to an array of integers.
# Assign it to an array of undefined length.

x: ARRAY{BV[8]}
y: INT*
x = ARRAY[10]{BV{8, 0}}
x = ARRAY[]{0}
```

Statements

Every statement can optionally be labeled: `LABEL <LABEL_NAME>: <STATEMENT>`. Every labeled statement is made available as a branch destination to be used by conditional and unconditional branching statements. Within a given scope (i.e. global, function local), there **must not** be duplicate label names. That is, each statement that is labeled within a scope must have a unique name.

Assignment Statements

The most basic statement is the assignment statement which is written as follows:

```
<ASSIGNMENT TARGET> = <EXPRESSION>
```

The right hand side of an assign statement can be any valid expression, so long as it evaluates to the same type as the assignment target.

The ***assignment target*** can be one of the following three:

1) An identifier which corresponds to a variable. `x = 10`.

2) An array index expression, which corresponds to storing into a certain index in an array: `arr[10] = 10`.

3) A record index expression, which corresponds to assigning to an element in a record: `rec.x = 10`.

4) A pointer dereference expression, which corresponds to storing it into the index of the array the pointer is pointing to: `*arr = 10`.

Branching Statements

There are two flavors of branching statements **conditional** and **unconditional**.

Conditional Branching Statements

A conditional branch statement depends on a condition and will either increment the program counter by one to the next statement if the condition is false and will set the program counter to the line number of the labeled expression if the condition is true. The conditional expression can be any valid expression that evaluates to the `BOOL` sort.

```
if <CONDITIONAL EXPRESSION> goto <LABEL NAME>
```

Unconditional Branching Statement

The unconditional branching statement is a short hand way of writting `if true goto <LABEL NAME>`:

```
goto <LABEL NAME>
```

Function Call Statements

A function call is expressed not as an expression but rather as it's own individual statement. There are two flavors of the funciton call statement, the first is if the function call return value is supposed to be assigned to a variable in the caller frame, and the second is if the function call return value is **not** supposed to assign to a variable in the caller frame.

Function Call then Assign Statement

```
call <ASSIGNMENT TARGET> = <FUNCTION NAME> ( <ARGUMENTS> )
```

Function Call with No Assign Statement

```
call <FUNCTION NAME> ( <ARGUMENTS> )
```

Any function can be called using this notation.

Return Statements [*Funcitons Only*]

Depending on the desired behavior, a return statement can either have a value to return or not.

```
return
--------------
return x
```

A return statement without a return value can be used to return from a function without a declared return type. If a function has a return type, the return statement with a value of the corresponding type must be used.

Assume Statements

An assume statement is utilized by appending a given condition to the path constraint, acting as a conditional branching statement branching to the end of the program. The statement following the `assume` keyword must evaluate to the `BOOL` sort.

```
assume <CONDITIONAL EXPRESSION>
```

Expressions

Atomic Expressions

There are a few atomic expressions that consist of the following categories:

Concrete Values

```
10  # Value of type INT sort
true # Value of type BOOL sort
false  # Value of type BOOL sort
BV{4, 32} # Value of type BV sort, with value and length
ARRAY[]{INT}  # An ARRAY sort of integers with indefinite length
ARRAY[10]{INT} # An ARRAY sort of integers with a fixed length of 10
```

Variable Identifier

```
x
b
point
factorial_value
```

Symbolic Values

```
$sym{BOOL} # A symbolic value of sort BOOL with a random unique name
$sym{b, BOOL} # A symbolic value of sort BOOL with the name 'b'
```

For symbolic values without a specified name, every time an atomic expression with no name is evaluated a NEW symbol with a unique name is created.

Array Index Expressions

```
arr[<INDEX>] # Where INDEX must be an integer sort, and must be within bounds
```

Record Element Index Expression

```
<variable name> . <element name>
```

Pointer Dereference Expression

```
<variable name> *
```

Unary Expressions

```
<UNARY OPERATOR> <ATOMIC EXPRESSION>
```

Binary Expressions

```
<ATOMIC EXPRESSION> <BINARY OPERATOR> <ATOMIC EXPRESSION>
```

Ternary Expressions

Only `ite`.

```
<ATOMIC EXPRESSION> ? <ATOMIC EXPRESSION> : <ATOMIC EXPRESSION>
```

Example Program

In this section we will cover one of the test cases in YouVerify that covers many of the aforementioned language features, including all of the sorts (Boolean, Integer, Array, BitVector). The example program can be seen in Figures 2 and 3. In the example, a player is playing a game that consists of a one dimensional level. The level consists of 8-bit BitVector values, which are the points that the player gets for landing on that tile of the level. If the tile is equal to 0, then the level is over. The player can choose to either press or not press the button to skip a tile whenever they choose. The function `below_max_length` determines wether or not an array contains a character 0 in it using pointer arithmetic to navigate the array. The function `sum` calculates the sum of an array. Both of these functions are used to add assertions to a *fully symbolic* level. Next, a *fully symbolic* array of button presses is generated and then the `play_game` function is called to calculate the score. To see this example in more detail please view it on YouTube here.

```
define below_max_length(arg: BV[8]*, max_length: INT) -> BOOL:
    null_terminated, cond: BOOL
    i: INT
    val: BV[8]
    null_terminated = false
    i = 0
    LABEL LOOP: if i >= max_length goto END
    val = *arg
    i = i + 1
    arg = arg + 1
    cond = val == BV{0, 8}
    null_terminated = cond | null_terminated
    goto LOOP
    LABEL END: return null_terminated

define sum(arr: ARRAY{BV[8]}, len: INT) -> INT:
    s, t, i: INT
    elem: BV[8]
    i = 0
    s = 0
    LABEL LOOP: if i >= len goto END
    elem = arr[i]
    call t = bv2nat(elem)
    s = s + t
    i = i + 1
    goto LOOP
    LABEL END: i = 0
    return s
```

Figure 2: Comprehensive Example from YouVerify test case suite.

```
define play_game(buttons: ARRAY{BOOL}, map: ARRAY{BV[8]}) -> BV[16]:
    i: INT
    map_elem: BV[8]
    score, temp: BV[16]
    cond: BOOL
    i = 0

    score = BV{0, 16}

    LABEL LOOP: map_elem = map[i]
    if map_elem == BV{0, 8} goto END_LOOP

    cond = buttons[i]
    if cond goto END_COND
    call temp = concat(BV{0, 8}, map_elem)
    call score = bvadd(score, temp)
    LABEL END_COND: i = i + 1
    goto LOOP
    LABEL END_LOOP: return score

buttons: ARRAY{BOOL}
level: ARRAY{BV[8]}
max_level_length: INT
is_below_max_length: BOOL
level_sum, final_score: INT

level = $sym{level, ARRAY{BV[8]}}
max_level_length = 5
call is_below_max_length = below_max_length(level, max_level_length)
call level_sum = sum(level, max_level_length)
assume is_below_max_length
assume level_sum == 100
buttons = $sym{buttons, ARRAY{BOOL}}
call final_score = play_game(buttons, level)
```

Figure 3: Comprehensive Example from YouVerify test case suite (continued).

3.3 Syntax

In this section I will cover the syntax of YouVerify. As mentioned later in the report, the grammar was implemented in the parser generator ANTLR, and is available in the GitHub project. Figure 4 describes the set of sorts that are currently available in YouVerify. As mentioned previously, as of now only the core, integer, bit-vector, and array theories are supported in YouVerify.

sort	$\rightarrow$	simple_sort | `ARRAY{`simple_sort`}` | simple_sort`*`
simple_sort	$\rightarrow$	`BOOL` | `INT` | `BV[`int`]` | r
where		
R		is a set of records
r		is an element of R
int		is a positive integer

Figure 4: YouVerify's syntax for the set of sorts, S.

The context free grammar for YouVerify is on the subsequent page in Figure 5.

program	→	global_decls stmts
global_decls	→	global_decls global_decl | global_decl
global_decl	→	record | func | var_decl
var_decl	→	x : s | x : r
args	→	args , var_decl | var_decl
record	→	r (args)
func	→	`define` f (args) : func_body | `define` f (args) `->` var_decl : func_body
func_body	→	func_body stmt | func_body `return` | func_body `return` expr | stmt | `return` | `return` atomic_expr
stmts	→	stmts `LABEL` ℓ`:` stmt | stmts stmt | `LABEL` ℓ`:` stmt | stmt
stmt	→	y `=` expr | y`[`atomic_expr`]` `=` expr | $z.e$ `=` expr | `*`y `=` expr | `if` expr `goto` ℓ | `goto` ℓ | `call` x `=` f (params) | `call` f (params) | `assume` expr | `assert` expr
params	→	params , atomic_expr | atomic_expr
expr	→	atomic_expr | $\propto$ atomic_expr | atomic_expr $\bowtie$ atomic_expr | atomic_expr `?` atomic_expr `:` atomic_expr | x`[`atomic_expr`]` | $z.e$ | `*`x
atomic_expr	→	x | c | `$sym{` id , s `}` | `$sym{` s `}`

where

V	is a set of variables
F	is a set of functions
R	is a set of records
C	is the set of constants
L	is the set of statement labels
S	is the set of sorts
x, y, z	are elements of V
e	is an element in a record in R
f	is an element of F
r	is an element of R
ℓ	is an element of L
c	is an element of C
s	is an element of S

Figure 5: YouVerify's syntax.

3.4 Implementation Details

The YouVerify symbolic interpreter is implemented in Python, a widely used programming language that has been consistently growing in popularity in the recent past. Python was chosen for my implementation because of the familiarity that many developers have with the language [7]. As YouVerify is intended to be easily accessible and modifiable, it was important to choose a language that was easily accessible. Additionally, there are a wealth of libraries to choose from during the implementation of the interpreter, and the many libraries available could make future implementations and enhancements to the framework easier.

The implementation of YouVerify is available publicly on GitHub here: https://github.com/gprechter/youverify.

Lexing and Parsing with ANTLR

For generating the parser for the YouVerify language, I used the ANTLR [16] parser generator. This powerful tool was used to read and process `.yvr` files according to the specified ANTLR grammar. The ANTLR parser generator is widely used by a variety of projects, and has a wealth of options and developer support.

For YouVerify, a program is converted to a parse tree which is explored by a *visitor*, which walks the program parsed by the generated parser in order to convert it to an abstract syntax tree that will be interpreted later by the symbolic interpreter. The heart of the parsing process is this walking of the parse tree using ANTLR's automatically generated tree walker. During the tree walking, the program's functions, variables, records, labels, and statements are all identified. Another important benefit of the ANTLR ecosystem is that an ANTLR grammar can be used to generate a parser for any of the supported target languages. While I chose to implement YouVerify in Python, it's possible to utilize the implemented ANTLR grammar if one were to develop a C++ or Java implementation.

Representing Symbolic Values and Expressions with pySMT

pySMT is a Python library that allows for the use of symbolic expressions and values that comply with the SMT-Lib format. All of the major sorts, including those that were implemented in YouVerify, have abstract representations in pySMT. When solving a symbolic expression requiring the use of a solver, pySMT offloads that processing to a chosen solver.

One of the main benefits of using pySMT for the implementation of YouVerify is the fact that pySMT is solver-agnostic, and does not rely on a given solver. The framework supports a number of SMT-Lib solvers by interfacing with them through a standard API. This falls closely inline with the design principles of YouVerify, allowing for modularity and interoperability. If a new solver is to be integrated with YouVerify, as long as a developer interfaced it properly with the API laid out by pySMT, and it adhered to the SMT-Lib standard, then it could be used instead of another solver.

During symbolic execution, YouVerify uses pySMT to compose symbolic expressions and make solver queries to check satisfiability. pySMT provides methods for converting its native SMT formula abstractions into SMT-Lib formats. One benefit of pySMT is the way that it represents symbolic expressions not as only trees, but as directed acyclic graphs (DAGs), recurring sub-expressions are reused and not copied, conserving the memory space needed to represent symbolic expressions. pySMT provides methods of emitting symbolic expressions to SMT-Lib both as a *DAG-ified* version and as a standard tree based expression. As for YouVerify, by default, SMT-Lib based expressions are printed without DAG-ification because they are easier to read. However, optionally, the expressions can be printed in the optimized format. It's important to note however that, while the expressions are printed in an un-optimized, easier to read format, behind the scenes, the symbolic expressions are maintained in optimized form and when framework solver queries are made, pySMT, and YouVerify emit the SMT-Lib expressions in the optimized format.

Framework Implementation

The YouVerify framework is implemented primarily as an abstract syntax tree that represents all of the various statements and expressions in the language, along with an execution loop that invokes the actions that each statement takes. The main execution loop interacts with the implemented symbolic execution algorithm through the state which implements the `State` abstract class. Specifically, as long as the state provides a true value from having its `update_state` method called, YouVerify will assume that there are additional statements that need to be executed. The statement to be executed is retrieved from the state using the `current_statement` method, during the execution of the statement, variables can be retrieved and stored from and to the state. Finally, YouVerify will perform an action to modify the program counter for the current state or cause the state to split due to a conditional branch. These methods that modify the program counter of branch state are the `advance_pc`, `jump`, and `conditional_branch` methods. The interaction between the main execution loop, the AST for the YouVerify language and the abstract state is the core of YouVerify's implementation and extensibility.

3.5 Symbolic Execution Algorithm

In this section we will cover the operational semantics of a selection of statements for the default symbolic execution algorithm, which is often referred to as vanilla dynamic symbolic execution. I will describe the rules as modifying the symbolic execution state Σ. Σ consists of any non-negative number of sub-states that consist of the following (ϕ, `pc`, σ), where ϕ is the path constraint for that sub-state, `pc` is the program counter, and σ is a mapping between identifiers and their values for that sub-state.

The symbolic execution algorithm traverses the program by selecting a sub-state from the symbolic execution state and executing the statement that that sub-state is currently on,

which subsequently modifies the state. The initial value of the state, Σ', is $\{(true, 0, \{\})\}$. The operational semantics for the different statements is shown in Figure 6. The current statement is retrieved from the program Pgm using the program counter: $Pgm(\text{pc})$.

CONDITIONAL BRANCH

$$\frac{(\phi, pc, \sigma) \in \Sigma \quad Pgm(pc) = (\text{ if } e \text{ goto } \ell) \quad \sigma(e) = b_e}{\Sigma \longrightarrow (\Sigma \setminus (\phi, pc, \sigma)) \cup (\phi \wedge b_e, \ell, \sigma) \cup (\phi \wedge \neg b_e, pc + 1, \sigma)}$$

UNCONDITIONAL BRANCH

$$\frac{(\phi, pc, \sigma) \in \Sigma \quad Pgm(pc) = (\text{ goto } \ell)}{\Sigma \longrightarrow (\Sigma \setminus (\phi, pc, \sigma)) \cup (\phi, \ell, \sigma)}$$

ASSUMPTION

$$\frac{(\phi, pc, \sigma) \in \Sigma \quad Pgm(pc) = (\text{ assume } e) \quad \sigma(e) = b_e}{\Sigma \longrightarrow (\Sigma \setminus (\phi, pc, \sigma)) \cup (\phi \wedge b_e, pc + 1, \sigma)}$$

VARIABLE ASSIGNMENT

$$\frac{(\phi, pc, \sigma) \in \Sigma \quad Pgm(pc) = (x \texttt{ = } e) \quad \sigma(e) = v_e}{\Sigma \longrightarrow (\Sigma \setminus (\phi, pc, \sigma)) \cup (\phi, pc + 1, \sigma[x \ \mapsto \ v_e])}$$

ARRAY INDEX ASSIGNMENT

$$\frac{(\phi, pc, \sigma) \in \Sigma \quad Pgm(pc) = (x[y] \texttt{ = } e) \quad \sigma(e) = v_e \quad \sigma(y) = i}{\Sigma \longrightarrow (\Sigma \setminus (\phi, pc, \sigma)) \cup (\phi, pc + 1, \sigma[x \ \mapsto \ \text{store}(x, i, v_e)])}$$

Figure 6: Operational Semantics for the Default Symbolic Execution Implementation in YouVerify.

The first rules we will cover is the VARIABLE ASSIGNMENT and ARRAY INDEX ASSIGNMENT rules. In both of these, the statement that the program counter is on is an assignment to either an atomic variable or an array index respectively. In both situations, the program counter is incremented once to the next statement, and the variable store σ is updated to either be the value of the expression e, v_e or an SMT-Lib array where the value of the expression is stored into the array at the index i. With these two statements, the path constraint is unchanged.

Next, the CONDITIONAL BRANCH adds two additional sub-states to the state Σ, where is one case the path constraint is changed to be the logical and of the previous path constraint and the boolean value of the expression b_e, and the new program counter for the sub state is the destination of the label provided in the statement. The other sub-state is the logical and of the previous path constraint and the negation of the boolean value, and the program counter is simply incremented. The variable store of either new state is unchanged. The UNCONDITIONAL BRANCH only results in one sub-state being added to Σ, where the new program counter is the value of the label.

Finally, the ASSUMPTION statement is a mechanism of modifying the path constraint of the current sub-state by conducting a logical and of the previous path constraint and the value of the boolean expression, b_e.

The symbolic execution algorithm will continue to traverse the program in this way until there are no sub-states in the state for which the program counter is less than the number of statements.

3.6 API

There are a number of components to the YouVerify API for implementing and modifying symbolic execution algorithms. First we will look at the abstract `State` class which enables a developer to implement custom symbolic execution algorithms. A developer can inherit and modify the existing symbolic execution state, `DefaultState`, or they can implement one from scratch as long as they implement all the abstract methods. The `State` class and its abstract methods and their descriptions are shown on the following pages.

In addition to using the abstract `State` class to modify the way that state is represented and the program execution path space is explored, a developer might want to modify how certain operators function in YouVerify. As we will see in Chapter 5, implementing ObliCheck in YouVerify required modifying functions to handle value summaries.

Modifying the behavior of individual operators is **not** supported out of the box, but modifying the behavior of unary or binary functions as groups is supported. Of course, because the project is open source, a developer could individually modify the functions is desired. These wrappers are presented below as well, `unary_operator_wrapper` and `binary_operator_wrapper`.

The API documentation was generated using Sphinx.

API Documentation

class State.**State**

Abstract class template for a representation of State in *YouVerify*.

abstract advance_pc(*i: int*)

This abstract method is invoked when the framework wants to advance the program counter.

Parameters **i** – The number of statements to advance through.

Returns None

abstract assume(*cond: object*)

This abstract method is invoked when the framework wants to add an assumption to the current state.

Parameters **cond** – The condition to be affixed to the path constraint.

Returns None

abstract conditional_branch(*cond: object*, *destination: int*)

This abstract method is invoked when the framework reaches a conditional branching statement. Here, a developer would likely want to implement state splitting.

Parameters

- **cond** – The condition guarding the branch.
- **destination** – The destination program counter for the branch.

Returns None

abstract property current_statement

This abstract property is invoked to retrieve the next statement to be executed by the symbolic interpreter.

Returns The next `Statement` object to be *executed*.

Return type Statement

abstract jump(*destination: int*)

This abstract method is invoked when the framework reaches an unconditional branching statement.

Parameters **destination** – The destination program counter for the branch.

Returns None

abstract load_variable(*var: str*)

This abstract method is invoked when an identifier is evaluated and a variable is to be fetched from the state's variable store.

Parameters **var** – The identifier of the variable that will be retrieved.

Returns The value of the variable that was retrieved.

abstract store_variable(*var: str*, *val: object*)

This abstract method is invoked when the framework wants to assign a value to a variable.

Parameters

- **var** – The identifier of the variable being assigned to.
- **val** – The value to be assigned to the variable.

Returns None

abstract update_state()

This abstract method is invoked by the framework when before the next statement is to be executed. This gives the `State` object the opportunity to modify the backend and setup for the next statement execution. A developer would modify this to change how the paths of a program are explored and how state is handled.

Returns `True` if there is more state to be executed, `False` otherwise.

Return type bool

`Wrappers.`**unary_operator_wrapper**(*f: function*)

The wrapper takes in the original unary operator as its parameter and optionally returns a new function with the new behavior.

Parameters `f` – The unary function to be wrapped.

Returns A new unary function.

`Wrappers.`**binary_operator_wrapper**(*f: function*)

The wrapper takes in the original binary operator as its parameter and optionally returns a new function with the new behavior.

Parameters `f` – The binary function to be wrapped.

Returns A new binary function.

Chapter 4

Testing and Verification

In order to verify the effectiveness of the framework and the correctness of the default dynamic symbolic execution implementation, many unit tests were written for YouVerify. All of these tests are available at the project GitHub here: https://github.com/gprechter/youverify/tree/master/tests.

4.1 Overview of Tests

Concrete execution is made available in YouVerify through the simplification and concrete evaluation of *symbolic* expressions. For each of the included sorts, Booleans, Integers, Bit Vectors, and Arrays there are unit tests for each of the operators. Additionally there are tests for additional YouVerify language features, such as functions, records, assume/asserts, and pointers/pointer arithmetic for arrays.

There are also symbolic tests which test the functionality of the symbolic execution in the face of branching state with the presence of symbolic inputs and symbolic input dependent branching statements. Some of these symbolic tests were taken from popular symbolic execution papers like KLEE, SAGE, and MultiSE. Finally, there are negative tests to ensure that the framework handles issues involving invalid typing, division by zero, and out-of-bounds array accesses.

4.2 Testing Setup and Process

Tests (files with the `.yvr` extension) are organized into folders by category and most tests are paired with a companion `.expected` file. The YouVerify framework is packaged with a testing framework in the `TestUtil.py` file. JSON objects are utilized to compare the results of executing programs using YouVerify and the expected results. The `.expected` file contains a string representing a JSON object that becomes de-serialized by the testing utility and compared to the concrete output of the final program state produced by YouVerify.

As an example, I have included an example test for the Arrays sort in Figure 1 and the `.expected` file as well in Figure 2 that tests a complex program populating and reversing an array.

To run a suite of tests, run the `TestUtil.py` file as follows, specifying the directory containing the desired YouVerify and `.expected` files as the command-line argument:

```
python TestUtil.py \tests\<sub-directory>
```

Running the test utility will report the success of each of the tests and provide a summary at the end of executing all of the tests.

```
arr: ARRAY{INT}

define populate(arr: ARRAY{INT}, len: INT):
    i: INT
    i = 0
    LABEL LOOP: if i >= len goto END
    arr[i] = i
    i = i + 1
    goto LOOP
    LABEL END: return

define reverse(arr: ARRAY{INT}, len: INT) -> ARRAY{INT}:
    i, j: INT
    new_arr: ARRAY{INT}
    new_arr = ARRAY[10]{0}
    i = 0
    LABEL LOOP: if i >= len goto END
    j = len - i
    j = j - 1
    new_arr[j] = arr[i]
    i = i + 1
    goto LOOP
    LABEL END: return new_arr

arr = ARRAY[10]{0}
call populate(arr, 10)
call arr = reverse(arr, 10)
```

Figure 1: Example test case for YouVerify arrays.

```
{ "arr": [9,8,7,6,5,4,3,2,1,0] }
```

Figure 2: Expected final program state for test case in Figure 1.

4.3 Evaluation on KLEE Test Cases

In order to further verify the default symbolic execution algorithm, a number of tests were implemented from the KLEE [5] symbolic execution tool. These tests can be found on GitHub here. The KLEE project executes each of the tests and then uses variables to test and verify branch coverage, and also inspects the number of final states at the end of the symbolic execution. This is done in the KLEE project to verify the correct functionality of the symbolic execution. The tests ported from KLEE to YouVerify can be found here; these tests cover many features of the YouVerify language including: **Arrays**, **Bit Vectors**, **Integers**, **Branching Statements**, and **Assumptions**. The results from these tests are shown in Table 1.

Table 1: Results from running KLEE test cases.

Test File	Final Valid Paths	Correct Coverage & Paths
test_and.yvr	2	YES
test_cache.yvr	6	YES
test_const_arr-idx.yvr	1	YES
test_expr_complex.yvr	2	YES
test_expr_mul.yvr	2	YES
test_expr_simple.yvr	2	YES
test_feasible.yvr	2	YES
test_hybrid.yvr	1	YES
test_mix.yvr	12	YES
test_mixed_hole.yvr	2	YES
test_multiindex.yvr	2	YES
test_new.yvr	2	YES
test_noncontiguous_idx.yvr	2	YES
test_position.yvr	2	YES
test_sub_index.yvr	2	YES
test_update_list_order.yvr	2	YES
test_var_idx.yvr	2	YES

4.4 Framework Coverage Testing

For determining the code coverage of the test suite, I used the `Coverage.py` tool, which is available here.

For calculating the coverage of the entire test suite over the source code and symbolic execution implementation use the follow command line argument:

```
coverage run TestUtil.py \tests\
```

This command runs the entire test suite while gathering coverage information. The results of determining the coverage of the test suite for my default dynamic symbolic execution implementation can be found on the GitHub page here.

Chapter 5

Prototyping ObliCheck with YouVerify

To demonstrate YouVerify's effectiveness for prototyping symbolic execution algorithms, I implemented an algorithm taking advantage of domain specific knowledge similar to the one presented in ObliCheck [21] using the framework and API. In order to do this, MultiSE's [20] state representation and incremental merging were implemented in YouVerify for a subset of YouVerify's statements and expressions; then additional changes were made to the symbolic execution algorithm to implement techniques like Optimistic State Merging and Iterative State un-merging from the ObliCheck paper.

5.1 Implementation Overview

The most significant change that needed to be made for implementing ObliCheck was implementing MultiSE's symbolic execution semantics and then modifying them in order to implement the optimistic state merging and iterative un-merging techniques. For implementing MultiSE, I utilized the PyEDA library for electronic design automation, available here. This library provides an implementation for Binary Decision Diagrams which are used in the implementation of MultiSE for representing the path constraints guarding values in the value summaries.

As shown earlier in this report, MultiSE presents a novel state representation through its value summaries, which allow for efficient, incremental state merging without the use of auxiliary variables. Compared with the aforementioned default implementation of dynamic symbolic execution, the state representation for MultiSE is drastically different. In YouVerify these changes are easily made through implementing a new State class that adheres to the necessary API that YouVerify provides. Particularly, the methods for `store_variable()`, `update_state()` and `condition_branch()` are modified. Additionally, for the desired operators, a *wrapper* must be added to handle the fact that operators now do not deal with single values but rather value summaries (for instance in the case of binary operators).

As for prototyping ObliCheck's unique state merging semantics, a number of modifications were made. The first step of the ObliCheck algorithm I prototyped was the Optimistic State Merging technique, which automatically merges variables aggressively by leveraging the domain specific knowledge that the actual values of private data is hidden from an attacker. To accomplish this, two new statements were added `begin_merge` and `end_merge`. With these statements, it's possible to specify which statements can merge variables and which cannot. In the future, this would ideally be done automatically through control flow analysis. In the `store_variable` API call, the method is modified to optimistically merge values that are *not* the length of a buffer, as specified in the ObliCheck paper. Individual merge points during execution needed to be tracked. When merging occurred, it's possible to record what statement caused that merging.

In the `next_state` method, responsible for the state exploration of the symbolic execution algorithm, it might be the case that due to Optimistic State Merging, the algorithm resulted in a false positive due to missing information. This is shown in the ObliCheck [21] paper through the `Tag&Apply` example. During the subsequent iterative un-merging step, it's also not necessary to change the main execution loop of YouVerify, as the API provides all the necessary control over the execution to perform the required re-executions and un-merging. The path constraints for the verification condition are inspected and the variables are extracted; if the result was not-oblivious and a variable introduced through optimistic state merging was present in the path constraint, it's possible that the result is a false negative. In this example, `update_state` flags the statements that introduced an OSM variable and will re-execute not merging any flagged variables.

The implementation of Optimistic State Merging and Iterative State Un-merging using YouVerify can be found as a branch to the GitHub project here: https://github.com/gprechter/youverify/tree/api_oblicheck. Implementing the two techniques on top of MultiSE required **54 lines** of code.

5.2 Evaluation

To evaluate my implementation of Optimistic State Merging (OSM) and Iterative State Un-merging (ISU), a selection of slightly modified benchmarks from the ObliCheck paper were ported to YouVerify and the following runtime statistics were gathered. Each of these examples covers one of the common scenarios shown in the ObliCheck paper. One of the tests is oblivious and will be identified correctly as oblivious with OSM and ISU. One of the tests is **not** oblivious, and will be identified correctly with OSM and ISU. Finally, `Tag&Apply.yvr` is an oblivious algorithm but important information is occluded during the OSM step, so ISU needs to be introduced to un-merge, re-execute and achieve the correct result. The length of the private data was 8 for each of the benchmarks.

Table 1: ObliCheck Implementation Runtime Speedup

Source File	**MultiSE**	**ObliCheck (OSM)**	**ObliCheck (OSM + IUM)**
`Tag.yvr` (Oblivious)	70s	5.2s	5.4s
`Tag.yvr` (Not Oblivious)	49s	8.7s	38s
`Tag&Apply.yvr` (Oblivious)	4409.1s	34.8s	2004.4s

In Table 2 MultiSE results in the correct result, either oblivious (◯) or not oblivious (×). In the `Tag&Apply.yvr` example, with OSM there is an erroneous false negative, but this is corrected with OSM and IUM.

Table 2: ObliCheck Implementation Accuracy

Source File	**MultiSE**	**ObliCheck (OSM)**	**ObliCheck (OSM + IUM)**
`Tag.yvr` (Oblivious)	◯	◯	◯
`Tag.yvr` (Not Oblivious)	×	×	×
`Tag&Apply.yvr` (Oblivious)	◯	×	◯

Chapter 6

Conclusion

In this project report, I presented YouVerify, a new IR and framework for developing custom symbolic execution tools and algorithms. Having learned about symbolic execution and the process required for modifying symbolic execution algorithms through my work on ObliCheck, I identified a number of features to strive for while developing the IR and framework. Namely, YouVerify is purposefully a simple intermediate representation rather than just a framework on an existing language to leave the door open to developers implementing compilers from a target language to the IR. YouVerify is also designed to be directly supported by the underlying family of SMT-Lib solvers such that no language features cannot be represented. Perhaps most importantly, YouVerify was designed to be modifiable and extensible and is packaged with an API that is intended to give a developer a lot of control on the symbolic execution algorithm without having to worry about the interpreter.

Another major component of the project was verifying the implementation of YouVerify to ensure that it operated properly. I hope that the provided test suite is also able to help developers guide and verify their own implementations of symbolic execution algorithms in the future. Finally, some of the symbolic execution enhancements from ObliCheck (optimistic state merging & iterative state un-merging) were implemented on-top of a MultiSE implementation to demonstrate the effectiveness of the framework and its API.

Takeaways

I learned a lot working on YouVerify during my 5th Year Masters. One of the highlights of working on the project was the emphasis placed on testing and verifying the functionality of the framework. Test-driven development served as a north star to guide the implementation, and I believe that focusing on testing has made me a better software engineer.

I also got a lot of exposure to symbolic execution and similar techniques. Having worked on ObliCheck, that project inspired me to work on this project with the aim of enhancing the experience of working with the technique, but actually having worked on the framework and making decisions about the API was a very rewarding process and taught me a lot about the different optimizations that symbolic execution tools use.

Bibliography

[1] Roberto Baldoni et al. "A Survey of Symbolic Execution Techniques". In: 51.3 (May 2018). ISSN: 0360-0300. DOI: 10.1145/3182657. URL: https://doi.org/10.1145/3182657.

[2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org. 2016.

[3] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. "SELECT—a Formal System for Testing and Debugging Programs by Symbolic Execution". In: *SIGPLAN Not.* 10.6 (Apr. 1975), pp. 234–245. ISSN: 0362-1340. DOI: 10.1145/390016.808445. URL: https://doi.org/10.1145/390016.808445.

[4] David Brumley et al. "BAP: A Binary Analysis Platform". In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 463–469. ISBN: 978-3-642-22110-1.

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI. San Diego, California: USENIX, 2008, pp. 209–224. URL: http://dl.acm.org/citation.cfm?id=1855741.1855756.

[6] Cristian Cadar et al. "EXE: Automatically Generating Inputs of Death". In: *ACM Trans. Inf. Syst. Secur.* 12.2 (Dec. 2008). ISSN: 1094-9224. DOI: 10.1145/1455518.1455522. URL: https://doi.org/10.1145/1455518.1455522.

[7] Klint Finley. *Python Is More Popular Than Ever*. 2020. URL: https://www.wired.com/story/python-language-more-popular-than-ever (visited on 11/30/2020).

[8] José Fragoso Santos et al. "Gillian, Part i: A Multi-Language Platform for Symbolic Execution". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 927–942. ISBN: 9781450376136. DOI: 10.1145/3385412.3386014. URL: https://doi.org/10.1145/3385412.3386014.

[9] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed Automated Random Testing". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. Chicago, IL, USA: Association for Computing Machinery, 2005, pp. 213–223. ISBN: 1595930566. DOI: 10.1145/1065010.1065036. URL: https://doi.org/10.1145/1065010.1065036.

[10] W.E. Howden. "Symbolic Testing and the DISSECT Symbolic Evaluation System". In: *IEEE Transactions on Software Engineering* SE-3.4 (1977), pp. 266–278. DOI: 10.1109/TSE.1977.231144.

[11] James C. King. "A New Approach to Program Testing". In: *SIGPLAN Not.* 10.6 (Apr. 1975), pp. 228–233. ISSN: 0362-1340. DOI: 10.1145/390016.808444. URL: https://doi.org/10.1145/390016.808444.

[12] James C. King. "Symbolic Execution and Program Testing". In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252. URL: https://doi.org/10.1145/360248.360252.

[13] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis Transformation". In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004, p. 75. ISBN: 0769521029.

[14] Petar Maksimović et al. "Gillian, Part II: Real-World Verification for JavaScript and C". In: *Computer Aided Verification*. Ed. by Alexandra Silva and K. Rustan M. Leino. Cham: Springer International Publishing, 2021, pp. 827–850. ISBN: 978-3-030-81688-9.

[15] Noah Moroze. "Kronos: Verifying leak-free reset for a system-on-chip with multiple clock domains". PhD thesis. Massachusetts Institute of Technology, 2021.

[16] Terence Parr and Kathleen Fisher. "LL(*): The Foundation of the ANTLR Parser Generator". In: *SIGPLAN Not.* 46.6 (June 2011), pp. 425–436. ISSN: 0362-1340. DOI: 10.1145/1993316.1993548. URL: https://doi.org/10.1145/1993316.1993548.

[17] Griffin et al. Prechter. "CS 294-163 Final Paper — ObliCheck: An Automatic Verification Tool for Oblivious Algorithms". In: (2019).

[18] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: A Concolic Unit Testing Engine for C". In: *SIGSOFT Softw. Eng. Notes* 30.5 (Sept. 2005), pp. 263–272. ISSN: 0163-5948. URL: https://doi.org/10.1145/1095430.1081750.

[19] Koushik Sen et al. "Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2013. Saint Petersburg, Russia: Association for Computing Machinery, 2013, pp. 488–498. ISBN: 9781450322379. DOI: 10.1145/2491411.2491447. URL: https://doi.org/10.1145/2491411.2491447.

[20] Koushik Sen et al. "MultiSE: Multi-Path Symbolic Execution Using Value Summaries". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, pp. 842–853. ISBN: 9781450336758. DOI: 10.1145/2786805.2786830. URL: https://doi.org/10.1145/2786805.2786830.

[21] Jeongseok Son et al. "ObliCheck: Efficient Verification of Oblivious Algorithms with Unobservable State". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2219–2236. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/son.

[22] Dawn Song et al. "BitBlaze: A New Approach to Computer Security via Binary Analysis". In: *Proceedings of the 4th International Conference on Information Systems Security*. ICISS '08. Hyderabad, India: Springer-Verlag, 2008, pp. 1–25. ISBN: 9783540898610. DOI: 10.1007/978-3-540-89862-7_1. URL: https://doi.org/10.1007/978-3-540-89862-7_1.

[23] Emina Torlak and Rastislav Bodik. "A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages". In: *SIGPLAN Not.* 49.6 (June 2014), pp. 530–541. ISSN: 0362-1340. DOI: 10.1145/2666356.2594340. URL: https://doi.org/10.1145/2666356.2594340.

[24] Emina Torlak and Rastislav Bodik. "Growing Solver-Aided Languages with Rosette". In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming Software*. Onward! 2013. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pp. 135–152. ISBN: 9781450324724. DOI: 10.1145/2509578.2509586. URL: https://doi.org/10.1145/2509578.2509586.

[25] Rui Zhang and Cynthia Sturton. "A Recursive Strategy for Symbolic Execution to Find Exploits in Hardware Designs". In: *Proceedings of the 2018 ACM SIGPLAN International Workshop on Formal Methods and Security*. FMS 2018. Philadelphia, PA, USA: Association for Computing Machinery, 2018, pp. 1–9. ISBN: 9781450358330. DOI: 10.1145/3219763.3219764. URL: https://doi.org/10.1145/3219763.3219764.