A MODEL CHECKER FOR JAVA BYTECODE, WITH NOVEL APPLICATIONS

A Dissertation
Presented to
The Academic Faculty

By

Burak Sahin

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Computer Science

Georgia Institute of Technology

August 2017

A MODEL CHECKER FOR JAVA BYTECODE, WITH NOVEL APPLICATIONS

Approved by:

Dr. William Harris, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Alessandro Orso
School of Computer Science
Georgia Institute of Technology

Dr. Hadi Esmaeilzadeh
School of Computer Science
Georgia Institute of Technology

Date Approved: July 27, 2017

Yesterday I was clever, so I wanted to change the world.

Today I am wise, so I am changing myself.

Hz. Mevlana Celaleddin Rumi

To my belowed parents,

Fatma & Hulusi

ACKNOWLEDGEMENTS

I have been very fortunate to work with Professor William R. Harris as his research assistant. It has been a great pleasure and honor to work with him and being in his team. Words falling behind to describe his support, guidance and humane attitude towards his students. I would like to thank him for his support and advise. I admire his personality and advising approach, and I wish to be a mentor like him.

I wouldn't be here if my mother's and my father's indescribable support and their prays for me. I have always felt their support in the predicament times even they are thousands of miles away. I am very lucky to be their offspring and raised by them. I would like to express my endless gratitude to them and my brothers for their support and love.

Finally, I also would like to thank Professor Alex Orso and Professor Hadi Esmaeilzadeh for serving in my Master of Science Dissertation Committee.

TABLE OF CONTENTS

Acknowledgments v

List of Tables viii

List of Figures ix

Chapter 1: Introduction and Background 1

1.1 Program Analysis 1

1.1.1 Dynamic program analysis 1

1.1.2 Static program analysis 2

Chapter 2: Overview 5

2.1 Overview 5

2.2 Foundations 9

2.2.1 Interpolants 9

2.2.2 Path-invariant tree based lazily symbolic model checking 10

2.2.3 Program lifting into the logical space 10

Chapter 3: SAVERIA 11

3.1 Target language 11

3.1.1 Program structure 11

3.1.2 Program semantics 13
3.2 Formal logic 14
3.2.1 Symbolic Representation of Program Semantics 16
3.2.2 Interpolation 17
3.3 Program Modelling 20
3.3.1 Program summaries 21
3.3.2 Program Verification Algorithm 24

Chapter 4: Results 28
4.1 Test Environment 28
4.2 Evaluation 28

Chapter 5: Conclusion 30

Chapter 6: Future Work 31

References 33

LIST OF TABLES

4.1 The benchmark performance for CAMPY 29

LIST OF FIGURES

1 SAVERIA 3

2 A simple `Lock` example 6

3 Unwinding *CFG* into *Interpolant Tree* 8

SUMMARY

We believe that we can extend program analysis tool to improve software security via verifying the given safety properties. The work done for this thesis is at the intersection of programming languages and security which is called software security.

In this work, we have designed and developed an automated static program analysis tool which can check whether the given program satisfies the required safety properties for the Java bytecode. Using the combination of model checker and symbolic execution with lazy abstraction, we have been successful to validate whether a program satisfies the given properties or not.

Our focus is to automatically prove or disprove whether a given JVM bytecode program satisfies the required properties. We have developed a property verification tool for Java applications which is a base framework for our future studies.

CHAPTER 1
INTRODUCTION AND BACKGROUND

1.1 Program Analysis

Software behaviour can be automatically analyzed and verified with respect to the given properties like safety, correctness etc. via program analysis [1]. Through program analysis, it is possible to automate software testing. Furthermore, program analysis have many applications in security such as bug finding, vulnerability detection, malware analysis and so forth.

In this section, the different approaches and techniques are described to accomplish the aforementioned purposes. Several techniques have been proposed and developed for property verification. The rest of this chapter will introduce the various techniques used to perform program analyses with dynamic analysis (program execution in environment) or static analysis (source code or bytecode).

1.1.1 Dynamic program analysis

Dynamic program analysis is a popular technique to analyze properties of programs performed at run-time. Dynamic program analysis offers many advantages like automated and continuous testing and reporting which reduces the cost of the test and maintenance. Dynamic program analysis provides many advantages including, but not limited to the list below:

- It can be performed on the applications which you cannot access either the source code or the bytecode. For example, a remote web services or web applications can be tested via dynamic program analysis.
- It can detect the vulnerabilities caused by environment or configuration other than the

application itself.

- It does not depend on the application environment such as programming language unlike static program analysis. For example, static analysis tool mostly targets a specific programming language such as Java or C/C++ source code and sometimes bytecode. However, applications written in different languages can be tested with same dynamic program analysis tool. For example, a web service provides same functionality written in different languages can be analyzed with one dynamic analysis tool.

Since analysis requires program execution on a real or virtual processor, effectiveness and coverage of dynamic program analysis depends on the test cases and inputs provided. The disadvantages of the dynamic program analysis are including, but not limited to the list below.

- Dynamic program analysis cannot guarantee the full coverage of the software (limited to inputs or test-cases provided); therefore it cannot address everything lacks in the aspect of security.

- The common perception about dynamic analysis is it is usually applied at later phases of development life cycle than static code analysis. Thus, cost and time for fixing the vulnerability is more than in earlier phases.

1.1.2 Static program analysis

Unlike dynamic approach, static program analysis does not require executing the program. Static program analysis can perform analysis using something close to either source code or object code. Since our target programming language is Java, we have developed our tool to analyze the JVM bytecodes. The primary advantages of static program analysis is depicted below:

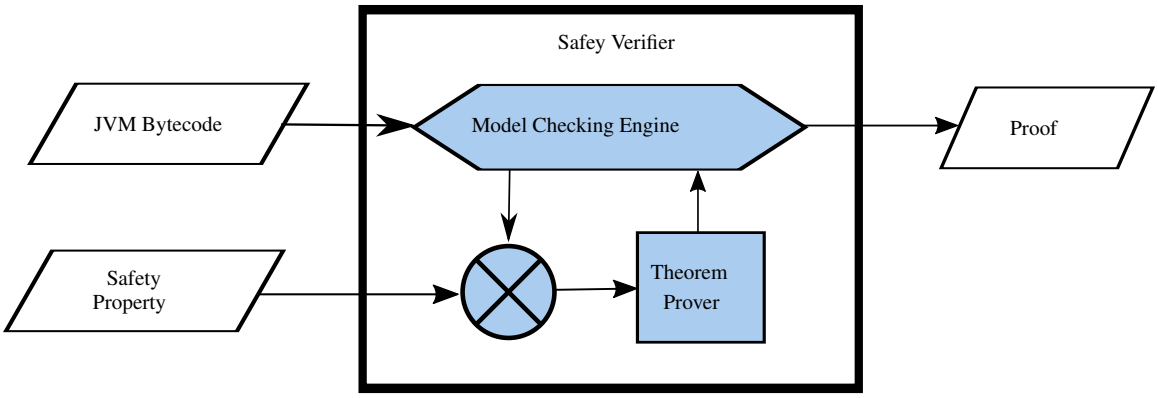


Figure 1: SAVERIA

- Static program analysis can determine facts that hold over program states in all possible runs.
- Static program analysis can detect the unreachable code, unused variables, uncalled functions.

We have chosen to implement our tool using static analysis techniques because our aim is to verify the correctness of a given program for the corresponding formal specification. Fig. 1 shows that our static analysis tool – SAVERIA– takes JVM bytecode and user specified safety property as inputs, and proves/disproves whether the bytecode satisfies the safety property with the help of theorem prover. Using combination of static analysis techniques listed below, we have been able to practically prove whether the program satisfies the formal specification. If the formal specification is not satisfiable by the program behaviour, we can disprove the correctness of the software by generating an input on which he program performs an error.

Model checking is a technique to verify the correctness of systems with respect to the given model formula in temporal logic. Normally, model checkers are used for the finite-state systems because they exhaustively explore and check all the possible paths. This situation can cause bloating and prevents them from using in infinite-state systems.

Symbolic execution is an approach to determine how the inputs affect the program execution behaviour. Symbolic execution treats inputs as symbolic values instead of assigning

actual values.

Lazy abstraction is a paradigm which enumerates the paths on demand. Instead of exhaustively search then refining abstract model, lazy abstraction proposes to refine the single abstract model if required [2].

Lazy symbolic model checking is a technique to apply model checking on infinite-state systems [3]. Symbolic model checker works on symbolic variables instead of explicit values [4].

CHAPTER 2

OVERVIEW

In this chapter, we describe a brief overview of our technical approach to design our static model checker for Java bytecode to verify given safety properties with safety *invariants*. By design principles, this model checker should be easily extensible to complexity verification and DoS attack vulnerability detection for library written in Java programming language. SAVERIA supports many programming languages features such as unbounded inputs, unbounded multi-dimensional arrays, library objects and inter-procedural calls (including recursions).

2.1 Overview

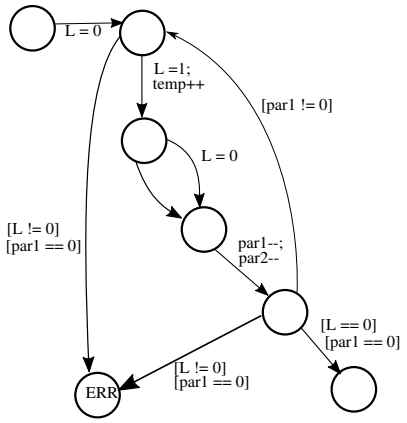
SAVERIA is a lazy interpolant-based infinite-state symbolic model checker for Java applications using counter-example model checker approach [2, 3]. To briefly present the algorithm, we walk through on a simple Java code fragment in Fig. 2a which has been inspired by the previous works on *lazy abstraction* paradigm [2, 3]. We can assume that upon entering the loop in the code fragment, the `L` is always free because if the `L` is acquired by some other process, *acquireLock()* function will wait till the `L` is freed. We want to demonstrate how SAVERIA can prove that the code fragment will always release the *Lock* – `L` – on leaving the loop in the code fragment in Fig. 2a when *par1* > 0.

$$\{\texttt{par1} > 0\}\ \textsc{SafeLock}\ \{\texttt{L} = 0\} \tag{1}$$

Our safety property for SAVERIA to prove the correctness of the program is $\texttt{L} = 0$ where the precondition is *par1* > 0 in Eqn. 1. Fig. 2b is the corresponding control-flow graph (denoted `CFG`) of the source code in Fig. 2a. The vertices in the `CFG` represents the control

```
1  // SafeLock function takes
2  // two integers par1 and par2
3
4  while (par1 != 0) {
5      acquireLock();
6      temp++;
7
8      if(par1 > 0)
9          releaseLock();
10
11     par1--;
12     par2--;
13 }
```

(a) `SafeLock` example



(b) `CFG` of `SafeLock`

Figure 2: A simple `Lock` example

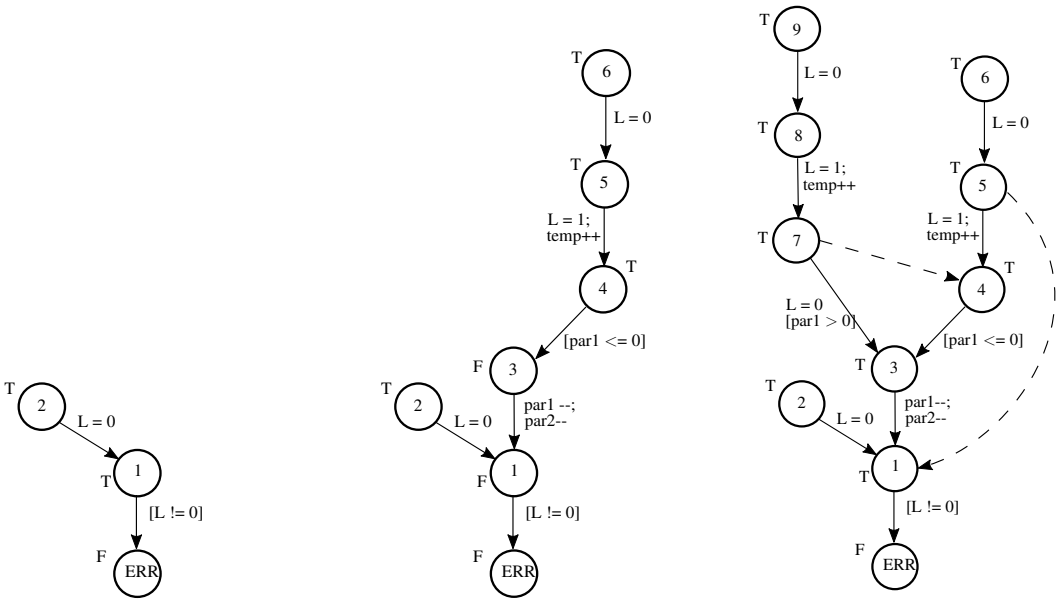
locations in the source code. SAVERIA takes the `CFG` and unwinds it into the path-invariant trees as shown in Fig. 3. An path-invariant tree is formed by the paths of a program labeled with the path invariants to decide the path reachability [5]. Our path-invariant tree is rooted at error location, and each path in the tree is a path from the initial location to the error location. Each vertex labeled as FALSE by default except for the initial location which is labeled as TRUE. Upon reaching the entry point for the loop, SAVERIA weakens the path *invariants* for each control location and merges the labeled path into the path-invariant tree. Having a FALSE *invariant* for the *error location* means that the erroneous path is unreachable; hence, SAVERIA proves that the program satisfies the given safety property.

First, SAVERIA uses Soot [6] for control-flow construction and to lift the bytecode to a *Static Single Assignment* (SSA) intermediate representation. In SSA form, each variable assigned exactly once which makes easier to establish the logical equivalences during static program analysis. Then, SAVERIA takes the `CFG` and unwinds it into the path-invariant tree which contains the expanded single path traces.

`ERR` vertex represents the program state which is inconsistent with safety property. SAVERIA starts unwinding process with expanding the `ERR` vertex using breadth first search (BFS). SAVERIA maintains an expansion queue to store expanded vertices. Each created vertex is pushed into the expansion queue, and SAVERIA pops the first vertex in the expansion

queue to expand the path leading to it. Therefore, the first path explored is the shortest path to the ERR vertex – the path which skips the loop shown in Fig. 3a. To show that this particular path is not feasible, we have to prove that the ERR is unreachable via labeling as FALSE in this single execution path. To generate path *invariants* for a given path, we have leveraged Interpolant Theorem Prover [7]. An *interpolant* for a given path is a path *invariant* for each control location starting with TRUE for initial location and FALSE for the ERR vertex. Therefore, having a FALSE *invariant* for any control location in any single execution path means that the control location and the rest of the path is unreachable which implies the path is infeasible (see Defn. 1). Note the inconsistency between the assumption *par1* > 0 and the control location (line 4). To skip the body of for loop, the variable $par1$ should be equal to 0. However, our assumption is $par1$ is greater than 0. After generating *interpolants*, SAVERIA weakens the *invariants* along the path and labels the ERR vertex as FALSE in Fig. 3a.

SAVERIA resumes expansion at the *vertex 1* (line 4). It is currently the only available condition location which has unexplored branch leading another execution path. The shortest path now executes the loop once, but skips releasing the lock (line 9) via jumping to the head of the if statement (line 8). Again, all the vertices are labeled with FALSE *invariant* except for initial location for this single execution trace. To skip the body of if statement (line 9), the if condition should not hold. On this particular path, the if condition (line 8) does not hold which is quite opposite of our assumption. Since there is inconsistency between the assumption and the rest of the path after the control location at *vertex 4* (line 8), the rest of the path is weakened with FALSE invariant. Upon refutation of the path (proving the ERR location is unreachable), this path is merged in to the path-invariant tree as shown in Fig. 3b. The *vertices 5* and the *1* corresponds to the same program control location (line 4). There are currently two vertices which have unexplored branches – *3* (end of if statement at line 8) and *5* (line 4). Since the *vertex 3* has explored before the *vertex 5*, the *vertex 3* takes place before in the expansion queue than the *vertex 5*. Therefore, SAVERIA will continue to unwind



(a) first erroneous trace (b) second erroneous trace (c) tree termination

Figure 3: Unwinding *CFG* into *Interpolant Tree*

CFG with expanding the other branch leading to the *vertex 3*. This time, the expanded path will take the path releasing the lock (line 9). This makes $\mathtt{L} = 0$ in the final program state. However, the final state for erroneous path is $\mathtt{L} \neq 0$ in CFG (Fig. 2b). Once again, error path is infeasible. The path and the invariants along the path are showed in Fig. 3c

At the final stage, the *vertices 1* and *5* correspond to the same control location (line 4). In Fig. 3, the dashed line means that the path invariant at *vertex 1* implies the path invariant at *vertex 5*. Using this implication relation, we can say that *vertex 1* covers the *vertex 5*. As long as this implication relation holds, SAVERIA will not expand the other branches leading into the *vertex 5* . The *vertices 4* and *7* also represent the same control location (line 8). Fig. 3c also shows that the path invariant at *vertex 4* implies the path invariant at *vertex 7*. If this relation will not be broken in future, the *vertex 7* will not be expanded into the other branches. While SAVERIA explores the vertices in the CFG, it puts the explored ones in to the expansion queue. SAVERIA pops the next vertex, and continues to the expansion if and only if it has an unexplored branch and it has not covered by any other vertex. Finally, expansion queue is empty; in other words, there isn't left any vertex which is not covered

and doesn't have unexplored branch. As a result, SAVERIA finds no path which leads to the error location. Fig. 3c contains the path-invariant tree which is the proof for the correctness of the program.

2.2 Foundations

SAVERIA depends on Craig interpolants to reason about the correctness of the program.

2.2.1 Interpolants

SAVERIA refines the path *invariants* using *interpolants* generated by an interpolating theorem prover upon refutation of the program paths [3, 8]. To understand our approach, first we will present how an interpolant works in *First Order Logic* and what it means in terms of verification.

Definition 1. *For the given two well formed formulae φ_A and φ_B in some logic, if $\varphi_A \bigwedge \varphi_B$ is inconsistent, there is a reverse* interpolant *φ_I in the context with the following properties*

- *φ_I includes only the common non-logical symbols of both φ_A and φ_B*
- $\varphi_A \models \varphi_I$
- $\varphi_B \models \neg\varphi_I$

Example 1. *For all x, y and z, let $\varphi_A = x \leq y \wedge y < z$ and $\varphi_B = x - z - 5 > 0$. Then we have the following properties $\varphi_A \models x \leq z$ and $\varphi_B \models \neg(x \leq z)$. One* interpolant *for these two formulae is $\varphi_I = x \leq z$ where $\varphi_B \wedge \varphi_I$ is inconsistent.*

The use of *interpolants* in program analysis relies on the inconsistency detection. Having a FALSE *invariant* for a single execution path in the CFG means that there are at least two formulae which are inconsistent with each other as shown in Ex. 1. In verification, this means the current execution trace is infeasible; therefore, the path does not satisfies the formal specification in the abstract model. SAVERIA proves or disproves the correctness of a system using *interpolants* described in the Defn. 1.

2.2.2 Path-invariant tree based lazily symbolic model checking

Our approach is to prove infeasibility of a path via refuting the path using interpolation based theorem prover [8] which generates an *interpolant* for each occurrence of a control location and for the error location. Refutation of a path happens upon generating FALSE *invariant* for any of the control location or for the error location. If there exists any FALSE *invariant* for any location along the path, the rest of the locations on the path also labeled as FALSE. Upon refutation, SAVERIA merges this trace into the traces tree. While merging each refuted path, the disjunction of the *interpolants* for the same control location is assigned as the *invariant* for the location. If there exists such a path which error location cannot be labeled with FALSE *invariant*, the path is feasible. Upon reaching a feasible path, SAVERIA decides the program satisfies the given properties if there exists such a single trace which error location cannot labeled with FALSE *invariant*.

The interpolants generated by the theorem prover for the corresponding control locations are used for the cover relation. The main benefit of the lazy abstraction paradigm applying on symbolic model checking is to enable model checking on infinite-state systems. Instead of unwinding each vertex, SAVERIA unwinds only the uncovered vertices in the path-invariant tree using lazy abstraction paradigm. In addition, lazy abstraction prevents SAVERIA from bloating and consuming more resource – memory and computation power.

2.2.3 Program lifting into the logical space

SAVERIA models program semantics as symbolic formulas – representing the instructions in the path as series of logical formulae. Therefore, SAVERIA can use theorem provers [8] to reason about the behavior of the path and prove the correctness of the program with respect to the given safety properties.

CHAPTER 3

SAVERIA

In this chapter, we introduce the technical details of SAVERIA. On the next sections, we present the target language in § 3.1 for SAVERIA and the formal logic basis § 3.2 for SAVERIA. Since CAMPY [9] runs SAVERIA, the target language and formal logic basis for SAVERIA is same in our previous published work CAMPY [9]. CAMPY uses the output of SAVERIA as its input. Hence, the sections § 3.1 and § 3.1.2, which are also defined for SAVERIA, are taken from our previously published work CAMPY [9].

3.1 Target language

In this section, we present the definition of the structure (§ 3.1.1) and semantics (§ 3.1.2) of the programs that SAVERIA takes as input.

3.1.1 Program structure

A program $\mathcal{P}$ is a set of instructions Δ. Each instruction $\delta \in \Delta$ is a either

- *Test statement* which tests the program state and creates branches in the program flow
- *Update statement* which updates the program state
- *Invoke statement* which calls a procedure
- *Return statement* which returns from a call.

Let $\texttt{procnms}$ be a space of *procedure names*; let Locs_B, Locs_C, and Locs_R be disjoint sets of branch, call, and return control locations, and let the set of all control locations be denoted $\texttt{Locs} = \mathsf{Locs}_B \cup \mathsf{Locs}_C \cup \mathsf{Locs}_R$, with a distinguished *initial location* l_i and *final location* l_f. Let $\mathsf{proc} : \texttt{Locs} \to \texttt{procnms}$ map each control location to the procedure that contains

it, with $\mathsf{proc}(l_i) = \mathsf{proc}(l_f)$. Let $\mathsf{entry} : \texttt{procnms} \to \texttt{Locs}$ map each procedure to its entry control location and $\mathsf{exit} : \texttt{procnms} \to \texttt{Locs}$ map each procedure to its exit control location. The space of all variables is denoted Vars. The space of all instructions is denoted Δ. For each control location $\texttt{L} \in \texttt{Locs}$ and sequence of locations $s \in \texttt{Locs}^*$, s is an *immediate suffix* of $\texttt{L} :: s$.

A program statement either tests and updates state, calls a procedure, or returns from a call. A pre-location, instruction, and branch-target-location is a *branch statement*; i.e., the space of branch statements is denoted $\mathsf{Brs} = \mathsf{Locs}_B \times \Delta \times \texttt{Locs}$. For each branch statement $b \in \mathsf{Brs}$, the pre-location, instruction, and post-location of b are denoted $\mathsf{PreLoc}[b]$, $\texttt{Instr}[b]$, and $\mathsf{BrTgt}[b]$, respectively.

A pre-location, call-target procedure name, and return-target control location are a *call statement*; i.e, the space of call statements is denoted $\mathsf{Calls} = \mathsf{Locs}_C \times \texttt{procnms} \times \texttt{Locs}$. For each call statement $c \in \mathsf{Calls}$, the pre-location, call target, and return target of c are denoted $\mathsf{PreLoc}[c]$, $\mathsf{CallTgt}[c]$, and $\mathsf{RetTgt}[c]$, respectively. The call entry point of c is denoted $\mathsf{entry}(c) = \mathsf{entry}(\mathsf{CallTgt}[c])$.

A return location represents a *return statement*; i.e., the space of return statements is denoted $\mathsf{Rets} = \mathsf{Locs}_R$.

The space of all statements is denoted $\mathsf{Stmts} = \mathsf{Brs} \cup \mathsf{Calls} \cup \mathsf{Rets}$. Each control location is the target of either potentially-many branch statement or exactly one call statement. For each call statement $c \in \mathsf{Calls}$ and return statement $r \in \mathsf{Rets}$ such that $\mathsf{CallTgt}[c] = \mathsf{proc}(\mathsf{PreLoc}[r])$, r *returns to* c. A program $\mathcal{P}$ is a set of statements in which for each branch location $\texttt{L} \in \mathsf{Locs}_B$ and location $\texttt{L}' \in \texttt{Locs}$, there is at most one branch statement, denoted $\mathsf{BrAt}[P](\texttt{L}, \texttt{L}')$ with $\mathsf{PreLoc}[b] = \texttt{L}$ and $\mathsf{BrTgt}[b] = \texttt{L}'$. The language of programs is denoted $\mathcal{L}$.

3.1.2 Program semantics

Visible execution state of a function call is denoted a store. A run of a program $\mathcal{P}$ is a sequence of stores that are valid along an interprocedural path of $\mathcal{P}$. A nesting relation over indices models the matched calls and returns of along a control path [10]. For each $n \in \mathbb{N}$, let the space of positive integers less than n be denoted $\mathbb{Z}_n$.

Definition 2. *For each $n \in \mathbb{N}$ and $\rightsquigarrow \subseteq \mathbb{Z}_n \times \mathbb{Z}_n$ such that for all indices $i_0, i_0', i_1, i_1' \in \mathbb{Z}_n$ with $i_0 \rightsquigarrow i_0'$ and $i_1 \rightsquigarrow i_1'$, either $i_0 < i_1 < i_1' < i_0'$, $i_1 < i_1' < i_0 < i_0'$, or $i_1 < i_0 < i_0' < i_1'$, $\rightsquigarrow$ is a* nesting *relation over n.*

For each $n \in \mathbb{N}$, the nesting relations over $\mathbb{Z}_n$ are denoted $\mathsf{Nestings}[n]$. For each $i, j < n$ and nesting relation $\rightsquigarrow \in \mathsf{Nestings}[n]$, we denote $(i, j) \in \rightsquigarrow$ alternatively as $i \rightsquigarrow j$.

A control path is a sequence of control locations visited by a sequence of branch statements, calls, and matching returns.

Definition 3. *Let program $\mathcal{P} \in \mathcal{L}$ and control locations $\mathtt{L} = [\mathtt{L}_0, \ldots, \mathtt{L}_{n-1}] \in \mathtt{Locs}^*$ be such that the following conditions hold.*

***(1)** For each $0 \leq i < n$ such that $\mathtt{L}_i \in \mathsf{Locs}_B$, there is a branch statement $b \in \mathcal{P}$ such that $\mathsf{PreLoc}[b] = \mathtt{L}_i$ and $\mathsf{BrTgt}[b] = \mathtt{L}_{i+1}$.*

***(2)** There is a nesting relation $\rightsquigarrow \in \mathsf{Nestings}[n]$ such that the domain and range of $\rightsquigarrow$ are exactly the indices of the call and successors of return locations in L. For all $0 \leq i < j < n$ such that $i \rightsquigarrow j + 1$, there is some call statement $c \in \mathcal{P}$ such that $\mathtt{L}_{i+1} = \mathsf{entry}(c)$ and $\mathtt{L}_{j+1} = \mathsf{RetTgt}[c]$, and some return statement $r \in \mathcal{P}$ such that $\mathtt{L}_j = \mathsf{PreLoc}[r]$.*

Then $[\mathtt{L}_0, \ldots, \mathtt{L}_{n-1}]$ is a path *of $\mathcal{P}$.*

For each program $\mathcal{P} \in \mathcal{L}$, the space of paths of $\mathcal{P}$ is denoted $\textsc{Paths}[\mathcal{P}]$, and the set of all paths is denoted Paths. For each path $\pi \in \mathsf{Paths}$, we denote the locations and nesting relation of π as $\mathtt{Locs}[\pi]$ and $\rightsquigarrow_\pi$, respectively.

Let the space of program values be the space of integers; i.e., the space of values is $\mathsf{Values} = \mathbb{Z}$. SAVERIA can verify programs that operate on objects and arrays in addition

to integers. An evaluation of all variables in Vars is a store; i.e., the space of stores is $\mathsf{Stores} = \mathsf{Vars} \to \mathsf{Values}$.

For each instruction $\delta \in \Delta$, there is a transition relation $\rho_\delta \subseteq \mathsf{Stores} \times \mathsf{Stores}$. For each branch statement $b \in \mathsf{Brs}$, the transition relation of the instruction in b is denoted $\rho_b = \rho_{\texttt{Instr}[b]}$. The transition relation of an instruction need not be total: thus, branch statements can implement control branches using instructions that act as `assume` instructions. The transition relation that relates each store at a callsite to the resulting entry store in a callee is denotes $\rho_C \subseteq \mathsf{Stores} \times \mathsf{Stores}$. The transition relation that relates each calling store, exit store of a callee, and resulting return store in the caller is denoted $\rho_R \subseteq \mathsf{Stores} \times \mathsf{Stores} \times \mathsf{Stores}$.

For each space X, sequence $s \in X^*$, and all $0 \leq i < |X|$, let the ith element in X be denoted $s[i] \in X$. Let the first and last elements of s in particular be denoted $\mathsf{Head}[s] = s[0]$ and $\mathsf{last}[s] = s[|s| - 1]$.

A run of a program $\mathcal{P}$ is a sequence of stores Σ and a path p of equal length, such that adjacent stores in Σ satisfy transition relations of statements of P at their corresponding locations in p.

Definition 4. *Let $\mathcal{P} \in \mathcal{L}$ be a program, let $\Sigma = \sigma_0, \ldots, \sigma_{n-1} \in \mathsf{Stores}$ be a sequence of stores, and let $\pi \in \textsc{Paths}[P]$ be such that $|\texttt{Locs}[\pi]| = n$, such that the following conditions hold:*

***(1)** For each $i < n - 1$, $(\sigma_i, \sigma_{i+1}) \in \rho_{\mathsf{BrAt}[P](\mathbb{L}_i, \mathbb{L}_{i+1})}$.*

***(2)** For each $i < j < n - 1$ such that $i \rightsquigarrow j + 1$, $(\sigma_i, \sigma_{i+1}) \in \rho_C$, $(\sigma_i, \sigma_j, \sigma_{j+1}) \in \rho_R$.*

Then Σ is a run *of q in P.*

For each path $\pi \in \mathsf{Paths}$, the space of runs of π is denoted $\mathsf{Runs}[\pi] \subseteq \mathsf{Stores}^*$.

3.2 Formal logic

Our approach uses formal logic to model the semantics of programs and synthesize summaries to prove or disprove that a given program satisfies a safety formula. A *theory* is

a vocabulary of function symbols and a standard model. For each theory $\mathcal{T}$ and space of logical variables X, let the spaces of $\mathcal{T}$ terms and formulas over X be denoted $\mathsf{Terms}[\mathcal{T}](X)$ and $\mathsf{Forms}[\mathcal{T}](X)$, respectively. For each formula $\varphi \in \mathsf{Forms}[\mathcal{T}](X)$, the set of variable symbols that occur in φ (i.e., the *vocabulary* of φ) is denoted $\mathsf{Voc}(\varphi)$. Each term constructed by applying only function symbols in $\mathcal{T}$ is a *ground term* of $\mathcal{T}$; i.e., the space of ground terms of $\mathcal{T}$ is $\mathsf{GTerms}[\mathcal{T}] = \mathsf{Terms}[\mathcal{T}](\emptyset)$.

For all vectors of variables $X = [x_0, \ldots, x_n]$ and $Y = [y_0, \ldots, y_n]$, the formula constraining the equality of each element in X with its corresponding element in Y, i.e., the formula $\bigwedge_{0 \leq i \leq n} \mathtt{x}_i = \mathtt{y}_i$, is denoted $X = Y$. For each vector of terms $T_Y = [t_0, \ldots, t_n]$, the repeated replacement of variables $\varphi[\ldots[t_0/x_0]\ldots t_{n-1}/x_{n-1}]$ is denoted $\varphi[X/T_Y]$. For each formula φ defined over free variables X, the substitution of Y in φ is denoted $\varphi[Y] \equiv \varphi[Y/X]$.

A *domain* is a finite set of values. For each theory $\mathcal{T}$, a *model* of $\mathcal{T}$ is a domain D and a map from each k-ary function symbol in $\mathcal{T}$ to a k-ary function over D. The standard model of a theory $\mathcal{T}$ is a distinguished model of $\mathcal{T}$. The domain of the standard model of $\mathcal{T}$ is denoted $\mathsf{Dom}[\mathcal{T}]$.

For theories $\mathcal{T}_0$ and $\mathcal{T}_1$, $\mathcal{T}_1$ is an *extension* of $\mathcal{T}_0$ if the vocabulary of $\mathcal{T}_0$ is contained by the vocabulary of $\mathcal{T}_1$ and the standard model of $\mathcal{T}_0$ is the restriction of the standard model of $\mathcal{T}_1$ to the vocabulary of $\mathcal{T}_0$. For all theories $\mathcal{T}_0$ and $\mathcal{T}_1$ whose standard models are equal on all symbols in the common vocabulary of $\mathcal{T}_0$ and $\mathcal{T}_1$, the combination [11] of theories $\mathcal{T}_0$ and $\mathcal{T}_1$ is denoted $\mathcal{T}_0 \cup \mathcal{T}_1$. We only consider theories $\mathcal{T}$ with standard model m that maps to domain D such that for each element $d \in D$, there is a ground term $\mathsf{term}[d] \in \mathsf{GTerms}[T]$ such that $m(\mathsf{term}[d]) = d$ (e.g., theories of arithmetic), along with their combinations with the theory of uninterpreted functions (EUFLIA).

For each theory $\mathcal{T}$, formula $\varphi \in \mathsf{Forms}[\mathcal{T}](X)$, and assignment m of X to the domain of $\mathcal{T}$, m *satisfies* φ if φ evaluates to TRUE under m combined with the standard model of $\mathcal{T}$ (denoted $m \vdash_{\mathcal{T}} \varphi$). For all $\mathcal{T}$ formulas $\varphi_0, \ldots, \varphi_n, \varphi \in \mathsf{Forms}[\mathcal{T}]$, we denote that

$\varphi_0, \ldots, \varphi_n$ *entail* φ_n as $\varphi_0, \ldots, \varphi_n \models_{\mathcal{T}} \varphi$. A $\mathcal{T}$-formula φ is a *theorem* of $\mathcal{T}$ if $\models_{\mathcal{T}} \varphi$.

Although determining the satisfiability of formulas in theories required to model the semantics of practical languages, such as LIA, is NP-complete in general, solvers have been proposed that often efficiently determine the satisfiability of formulas that arise from practical verification problems [7]. Our approach assumes access to a decision procedure for EUFLIA, named EUFLIASAT.

3.2.1 Symbolic Representation of Program Semantics

The semantics of $\mathcal{L}$ can be represented symbolically using LIA formulas. In particular, each program store $\sigma \in \mathsf{Stores}$ corresponds to a LIA model over the vocabulary Vars, denoted m^σ. For each space of indices I and index $i \in I$, the space of variables Vars_i denotes a distinct copy of the variables in Vars, as does Vars', which will typically be used to represent the post-state of a sequence of transitions. For theory $\mathcal{T}$, the space of program *summaries* is $\mathsf{Summaries} = \mathsf{Forms}[\mathcal{T}](\mathsf{Vars}, \mathsf{Vars}')$.

A safety constraint S is an comparison relation over the final stores of Vars. For each instruction $\delta \in \Delta$, there is a formula $\psi[\mathtt{i}] \in \mathsf{Forms}[\text{LIA}](\mathsf{Vars}, \mathsf{Vars}')$ such that for all stores $\sigma, \sigma' \in \mathsf{Stores}$, $(\sigma, \sigma') \in \rho_{\mathtt{i}}$ if and only if $m^\sigma, m^{\sigma'} \vdash \psi[\mathtt{i}]$. There is a formula $\psi_C \in \mathsf{Forms}[\text{LIA}](\mathsf{Vars}, \mathsf{Vars}')$ such that for all stores $\sigma, \sigma \in \mathsf{Stores}$, $(\sigma, \sigma') \in \rho_C$ if and only if $m^\sigma, m^{\sigma'} \vdash \psi_C$. There is a formula $\psi_R \in \mathsf{Forms}[\text{LIA}](\mathsf{Vars}_0, \mathsf{Vars}_1, \mathsf{Vars}_2)$ such that for all stores $\sigma_0, \sigma_1, \sigma_2 \in \mathsf{Stores}$, $(\sigma_0, \sigma_1, \sigma_2) \in \rho_R$ if and only if $m^{\sigma_0}, m^{\sigma_1}, m^{\sigma_2} \vdash \psi_R$.

Each program $\mathcal{P}$ and logical formula S over the initial state of $\mathcal{P}$ and final state of all Vars define safety specification problem. The problem is to decide if over each run r of $\mathcal{P}$, the initial state of $\mathcal{P}$ and the final states of Vars in r satisfy S. For theory $\mathcal{T}$, the space of safety constraints is denoted $\mathsf{Constraints}[\mathcal{T}] = \mathsf{Forms}[\mathcal{T}](\mathsf{Vars})$.

Definition 5. *For each extension $\mathcal{T}$ of* LIA*, program $\mathcal{P} \in \mathcal{L}$, safety constraint $S \in \mathsf{Constraints}[\mathcal{T}]$, and path $\pi \in \text{PATHS}[\mathcal{P}]$, if for each run $r \in \mathsf{Runs}[q]$, it holds that $m^{\mathsf{Head}[r]}, \mathsf{Stores} \mapsto m^{\mathsf{last}[r]}(\mathsf{Vars}) \vdash S$, then q* satisfies *S. For each path $\pi \in \text{PATHS}[\mathcal{P}]$*

it holds that π satisfies S, then $\mathcal{P}$ satisfies S, denoted $P \vdash S$. The safety-satisfaction problem $(\mathcal{P}, S)$ is to determine if $\mathcal{P} \vdash S$.

While we present our verifier SAVERIA for a simple language whose semantics can be modeled using only LIA, practical languages typically provide features that can only be directly modeled using LIA in combination with the theories of uninterpreted functions and the theory of arrays. The complete implementation of SAVERIA supports such language features (see § 4).

3.2.2 Interpolation

Tree-interpolation problems formulate the problem of finding valid invariants for all runs of a particular program path that contains calls and returns [5]. The branching structure in the tree-interpolation problem models the dependency of the result of a function call on the effect of the path through the callee combined with the arguments provided to the callee by the caller.

Definition 6. *For theory $\mathcal{T}$, a* $\mathcal{T}$-tree-interpolation problem *is a triple (N, E, C) in which:*

- *N is a set of* nodes.
- *$E \subseteq N \times N$ is a set of* edges *such that the graph $T = (N, E)$ is a tree with root $r \in N$.*
- *$C : N \to \mathsf{Forms}[\mathcal{T}](X)$ assigns each node to an $\mathcal{T}$* constraint.

For each tree-interpolation problem $T = (N, E, C)$, an interpolant *of T is an assignment $I : N \to \mathsf{Forms}[\mathcal{T}](X)$ from each node to a $\mathcal{T}$ formula such that:*

- *The interpolant at the root $r \in N$ of T entails* FALSE. *I.e., $I(r) \models_{\mathcal{T}}$* FALSE.
- *For each node $n \in N$, the interpolants at the children of n and the constraint at n entail the interpolant at n. I.e., $\{I(m)\}_{(m,n)\in E}, C(n) \models_{\mathcal{T}} I(n)$.*

- *For each node n, the* vocabulary *at n is the common vocabulary of all descendants for n and all non-descendants of n. The vocabulary of the interpolant at n is contained in the vocabulary of n. I.e.,*

$$\mathsf{Voc}(n) = \bigcup_{(m,n)\in E^*} \mathsf{Voc}(C(m)) \cap \bigcup_{(m',n)\notin E^*} \mathsf{Voc}(C(m'))$$

$$\mathsf{Voc}(I(n)) \subseteq \mathsf{Voc}(n)$$

For theory $\mathcal{T}$ and variables X, the space of all tree-interpolation problems whose constraints are $\mathcal{T}$ formulas over X is denoted $\textsc{Itp}[\mathcal{T}, X]$. For each tree-interpolation problem $T \in \textsc{Itp}[\mathcal{T}, X]$, the nodes, edges, root, and constraints of P are denoted $\mathsf{N}[T]$, $E[T]$, $r[T]$, and $\mathsf{Ctrs}[T]$, respectively. The conjunction of all constraints in T is denoted $\mathsf{Ctr}[T] = \bigwedge_{n\in\mathsf{N}[T]} C(n)$. A model of $\mathsf{Ctr}[T]$ is referred to alternatively as a model of T.

For each tree-interpolation problem T and node $n \in \mathsf{N}[T]$, the tree-interpolation problem formed by the restriction of T to the subtree with root n is denoted $T|_n$. The procedure $\textsc{TMrg}$ takes two tree-interpolation problems (N, E, C) and (N', E', C') with (N', E') a subtree of (N, E) and constructs a tree-interpolation problem in which the constraint for each node is the conjunction over constraints for all nodes in N'. I.e., $\textsc{TMrg}((N, E, C), (N', E', C')) = (N, E, C'')$ with $C''(n) = C(n)$ for each $n \in N \setminus N'$ and $C''(n) = C(n) \wedge C'(n)$ for each $n \in N'$.

Definition 7. *For a tree-interpolation problem a cover relation $\bowtie \subseteq N \times N$ is a tuple in which:*

- *M is a mapping function which maps the each $n \in N$ to its corresponding control location $\mathtt{L}$: $M(n) = \mathtt{L}$.*
- *The path ancestor relation for $m, n \in \pi$ is denoted $m \sqsubset n$ where m is an ancestor of n in a path π if the depth of n is less than the depth of m.*

- *The ancestor relation for the nodes pointing same control location $M(m) = M(n)$ in the tree is denoted $m \sqsubseteq n$ where the depth of n is less than the depth of m.*

- *A node n is covers another node m (denoted $(n, m) \in \bowtie$) if and only if $m \sqsubseteq n$ and $\varphi(m) \models \varphi(n)$.*

When a node is covered, then all of its ancestors in the paths leading to the covered node are also covered. A node can only be covered by only one node. A covered node cannot cover others. When a new cover relation added such as $(n, m) \in \bowtie$, SAVERIA checks if there are other records in such as $(m, \cdot) \in \bowtie$. If such relation exists, then SAVERIA removes the cover on the covered nodes and their ancestors. Since the *invariants* are estimated, they are approximate; hence we cannot establish transitivity on implications.

For theory $\mathcal{T}$ and $\mathcal{T}$-interpolation problems U_0 and U_1 containing the same nodes and edges, U_0 is as weak as U_1 if for each node n, the constraint in U_0 for n is as weak as the constraint for n in U_1 and the vocabulary of the constraint in U_1 is contained by the vocabulary of the constraint in U_0.

Definition 8. *For theory $\mathcal{T}$, variables X, and $U_0, U_1 \in \textsc{Itp}[\mathcal{T}, X]$, if for $N = \mathsf{N}[T] = \mathsf{N}[U]$, $E[T] = E[U]$, and for each $n \in N$, **(1)** $\mathsf{Ctr}[U_0](n) \models \mathsf{Ctr}[U_1](n)$ and **(2)** $\mathsf{Voc}(\mathsf{Ctr}[U_0](n)) \subseteq \mathsf{Voc}(\mathsf{Ctr}[U_1](n))$, then U_1 is* as weak as U_0.

Because weaker interpolation problems have weaker constraints per node, they admit *fewer* interpolants.

Lemma 1. *For theory $\mathcal{T}$, variables X, all $U_0, U_1 \in \textsc{Itp}[T, X]$ with common nodes N such that U_1 as weak as U_0, and all $I : N \to \mathsf{Forms}[\mathcal{T}](X)$ such that I is an interpolant of U_1, I is an interpolant of U_0.*

For theory $\mathcal{T}$, an interpolating theorem prover takes a $\mathcal{T}$-interpolation problem T and returns either a $\mathcal{T}$-model or a map from the nodes of T to $\mathcal{T}$-formulas. An interpolating theorem prover is sound if it only returns a valid model or interpolant of its input.

Definition 9. *For theory $\mathcal{T}$, variables X, let effective procedure $t : \textsc{Itp}[\mathcal{T}, X] \to (X \to \mathsf{Dom}[\mathcal{T}]) \cup (\mathsf{N}[T] \to \mathsf{Forms}[\mathcal{T}](X))$ be such that for each tree-interpolation problem $U \in \textsc{Itp}[\mathcal{T}, X]$, **(1)** if $t(U) : X \to \mathsf{Dom}[\mathcal{T}]$, then $t(U)$ is a model of U; **(2)** if $t(U) : \mathsf{N}[T] \to \mathsf{Forms}[\mathcal{T}](X)$, then $t(U)$ are interpolants of U. Then t is a* sound *interpolating theorem prover for $\mathcal{T}$.*

Previous work has presented an algorithm EUFLIAITP that solves a given EUFLIA tree-interpolation problem T by invoking an interpolating theorem prover for EUFLIA a number of times bounded by $|\mathsf{N}[T]|$ [5].

In § 3.3, we describe an approach for proving that a program whose semantics are expressed in a theory $\mathcal{T}_0$ satisfies a constraint expressed in an extension $\mathcal{T}$. To simplify the presentation of our approach, we fix $\mathcal{T}_0$ to be LIA, and fix $\mathcal{T}$ to be an arbitrary extension of LIA. However, our approach can be applied using any theory for $\mathcal{T}_0$ that satisfies the above conditions: in particular, our actual implementation of SAVERIA uses the combination of the theories of linear arithmetic, uninterpreted functions with equality, and arrays as its base theory.

3.3 Program Modelling

In this section, we describe a safety property verifier SAVERIA which CAMPY, a complexity verification tool, [9] is built on. CAMPY [9] takes the output of the SAVERIA which is a tree interpolation problem and performs complexity analysis on the paths indicated unsafe by SAVERIA. For complexity analysis, CAMPY adds axioms to the erroneous path, and then CAMPY either approves the result of SAVERIA or decides the path is safe. In § 3.3.1, we define the space of the summaries that SAVERIA deduce to prove given a program satisfies a given constraint. In § 3.3.2, we introduce the property verification algorithm for SAVERIA. Once again, the section § 3.3.1 are taken from our previously published work CAMPY [9] because SAVERIA works on same language and summaries. CAMPY takes the output of SAVERIA as input.

3.3.1 Program summaries

SAVERIA, given a program P and safety constraint S, attempts to infer summaries of the behavior of $\mathcal{P}$ that imply that all paths of $\mathcal{P}$ satisfy S. A program summary is a map from each control location `L` to a symbolic summary of the effects of all runs from the entry point of `L`'s procedure to `L`. The space of program summaries is denoted $\mathsf{ProgSums} = \texttt{Locs} \to \mathsf{Summaries}$. Program summaries are inductive for $\mathcal{P}$ and S if they imply that all runs of $\mathcal{P}$ satisfy S.

Definition 10. *For program $P \in \mathcal{L}$ and safety constraint $S \in \mathsf{Constraints}[\mathcal{T}]$, let $\Phi \in \mathsf{ProgSums}$, be such that:*

***(1)** for each procedure $f \in \texttt{procnms}$,*

$$\mathsf{Vars} = \mathsf{Vars}' \models \Phi(\mathsf{entry}(f))$$

***(2)** $\Phi(l_f) \models \mathsf{Constraints}[\mathcal{T}]$;*

***(3)** For each branch statement $b \in P$,*

$$\Phi(\mathsf{PreLoc}[b])[\mathsf{Vars}_0, \mathsf{Vars}_1], \psi[b][\mathsf{Vars}_1, \mathsf{Vars}_2] \models \Phi(\mathsf{BrTgt}[b])[\mathsf{Vars}_0, \mathsf{Vars}_2]$$

***(4)** For each call statement $c \in P$ and each return statement $r \in P$ that returns to c,*

$$\Phi(\mathsf{PreLoc}[c])[\mathsf{Vars}_0, \mathsf{Vars}_1], \psi_C[\mathsf{Vars}_1, \mathsf{Vars}_2], \Phi(r)[\mathsf{Vars}_2, \mathsf{Vars}_3], \psi_R[\mathsf{Vars}_1, \mathsf{Vars}_3, \mathsf{Vars}_4] \models \Phi(\mathsf{RetTgt}[c])[\mathsf{Vars}_0, \mathsf{Vars}_4]$$

Then Φ are inductive *summaries for $\mathcal{P}$ and S.*

The space of inductive summaries for program $\mathcal{P} \in \mathcal{L}$ and safety constraint $S \in$

$\mathsf{Constraints}[\mathcal{T}]$ is denoted $\mathsf{Ind}[\mathcal{P}, S]$. Inductive summaries are evidence of safety constraint satisfaction.

Lemma 2. *For each program $\mathcal{P} \in \mathcal{L}$ and constraint $S \in \mathsf{Constraints}[\mathcal{T}]$, if there are inductive summaries $\Phi \in \mathsf{Ind}[\mathcal{P}, S]$, then $\mathcal{P} \vdash S$.*

For path π, a visible suffix of π is a sequence of locations in π connected over only branch edges, nesting edges, and return edges.

Definition 11. *For each path $\pi \in \mathsf{Paths}$, let $L \in \mathtt{Locs}^*$ be such that $\mathsf{last}[L] = l_f$ and there is some function $m : \mathbb{Z}_{|L|} \to \mathbb{Z}_{|\pi|}$ such that for each $0 \leq i < |L|$, $L[i] = \pi[m(i)]$, if $L[i] \in \mathsf{Locs}_B$ or $L[i] \in \mathsf{Locs}_R$, then $L[i+1] = \pi[m(i)+1]$, and if $L[i] \in \mathsf{Locs}_C$, then for $j < |\pi|$ such that $i \rightsquigarrow_\pi j$, $L[i+1] = \pi[j]$. Then L is a* visible suffix *of π.*

For each path $\pi \in \mathsf{Paths}$, let the space of visible suffixes of π be denoted $\mathsf{Suffixes}[\pi] \subseteq \mathtt{Locs}^*$. For program $\mathcal{P} \in \mathcal{L}$, the visible suffixes of all paths of $\mathcal{P}$ are denoted $\mathsf{Suffixes}[\mathcal{P}] = \bigcup_{\pi \in \textsc{Paths}[\mathcal{P}]} \mathsf{Suffixes}[\pi]$.

Path summaries are sets of visible suffixes of a program's paths, with each visible suffix s mapped to a summary of the effect of all runs of s.

Definition 12. *For program $\mathcal{P} \in \mathcal{L}$, let $Q \subseteq \mathsf{Suffixes}[\mathcal{P}]$ be visible suffixes of paths of $\mathcal{P}$, and let $\Phi : Q \to \mathsf{Summaries}$. Then (Q, Φ) are* visible suffix summaries *of $\mathcal{P}$.*

For program $\mathcal{P} \in \mathcal{L}$, the space of all visible suffix summaries of $\mathcal{P}$ is denoted $\mathsf{VisSums}[\mathcal{P}]$. For all visible suffix summaries $\Phi \in \mathsf{VisSums}[\mathcal{P}]$, the visible suffixes and summary map of Φ are denoted $\mathsf{Suffixes}[\Phi]$ and $\mathsf{Sums}[\Phi]$.

If visible suffix summaries soundly model the semantics of the paths of which the summaries are subsequences, then the summaries are valid.

Definition 13. *For program $\mathcal{P} \in \mathcal{L}$, let visible-suffix summaries $\Phi \in \mathsf{VisSums}[P]$ be such that for each visible suffix $s \in \mathsf{Suffixes}[\Phi]$, **(1)** for each procedure $f \in \mathtt{procnms}$, if $\mathsf{Head}[s] = \mathsf{entry}(f)$, then*

$$\mathsf{Vars} = \mathsf{Vars}' \models \mathsf{Sums}[\Phi](s)$$

(2) *for each branch statement* $b \in \mathcal{P}$ *such that* $\mathsf{BrTgt}[b] = \mathsf{Head}[s]$ *and* $s_0 = \mathsf{PreLoc}[b] :: s \in \mathcal{P}$*,*

$$\mathsf{Sums}[\Phi](s_0)[\mathsf{Vars}_0, \mathsf{Vars}_1], \psi[b][\mathsf{Vars}_1, \mathsf{Vars}_2] \models$$
$$\mathsf{Sums}[\Phi](s)[\mathsf{Vars}_2/\mathsf{Vars}_0]$$

(3) *and each call statement* $c \in \mathcal{P}$ *such that* $s_0 = \mathsf{PreLoc}[c] :: s \in \mathsf{Suffixes}[\Phi]$ *and return statement* $r \in \mathcal{P}$ *such that* $\mathsf{proc}(c) = \mathsf{proc}(r)$ *and* $s_1 = r :: s \in \mathsf{Suffixes}[\Phi]$*,*

$$\mathsf{Sums}[\Phi](s_0)[\mathsf{Vars}_0, \mathsf{Vars}_1], \psi_C[\mathsf{Vars}_1, \mathsf{Vars}_2]$$
$$\mathsf{Sums}[\Phi](s_1)[\mathsf{Vars}_2, \mathsf{Vars}_3], \psi_R[\mathsf{Vars}_1, \mathsf{Vars}_3, \mathsf{Vars}_4] \models$$
$$\mathsf{Sums}[\Phi](s)[\mathsf{Vars}_0, \mathsf{Vars}_4]$$

Path summaries define inductive summaries of $\mathcal{P}$ when they define summaries of all paths of $\mathcal{P}$.

Definition 14. *For each program* $\mathcal{P} \in \mathcal{L}$ *and constraint* $S \in \mathsf{Constraints}[\mathcal{T}]$*, let* $\Phi \in \mathsf{VisSums}[\mathcal{P}]$ *be valid suffix summaries. Let* $\mathcal{P}' \in \mathsf{ProgSums}$ *be such that for each location* $\mathtt{L} \in \mathtt{Locs}$*,*

$$\Phi'(\mathtt{L}) = \bigvee \{\mathsf{Sums}[Sum](t) \mid t \in \mathsf{Suffixes}[S], \mathsf{Head}[t] = \mathtt{L}\}$$

If Φ' *are inductive summaries for* $\mathcal{P}$ *and* S *(Defn. 10), then* Φ *are* inductive visible-suffix summaries *for* $\mathcal{P}$ *and* S*.*

SAVERIA attempts to prove that a given program $\mathcal{P}$ satisfies a given safety property S by inferring inductive visible-suffix summaries for $\mathcal{P}$ and S.

Input : A program $P \in \mathcal{L}$ and safety specification $S \in \mathsf{Constraints}[\mathcal{T}]$.
Output : A decision as to whether $\mathcal{P}$ satisfies S.

```
1  Procedure SAVERIA(P, S)
2      Procedure SAUX(Φ)
3          switch UNWIND[P](Φ) do
4              case TRUE: do return TRUE ;
5              case π: do
6                  switch REFINE[P, S](π, Φ) do
7                      case FALSE: do return FALSE ;
8                      case Φ: do
9                          COVER[P](Φ) ;
10                         return SAUX(Φ) ;
11                     end
12                 end
13             end
14         end
15     return SAUX((∅, ∅)) ;
```

Algorithm 1: SAVERIA: a safety verifier. SAVERIA uses procedures UNWIND$[\mathcal{P}]$, REFINE$[[,]P, S]$, and COVER$[\mathcal{P}]$ which are described in § 3.3.2.

3.3.2 Program Verification Algorithm

Alg. 1 shows pseudo-code for the main algorithm of SAVERIA which takes a program $\mathcal{P}$ and a safety constraint S (line 1) to show whether $\mathcal{P}$ satisfies S. SAVERIA is based on counter-example guided refinement loop approach as in [2, 3]. SAVERIA uses SAUX (line 2 — line 14)function which takes the visible summaries Φ. SAUX calls three core algorithms which are UNWIND (line 3), REFINE (line 6) and COVER (line 9) functions. First, SAVERIA checks that if Φ summarizes the program P. If so, UNWIND returns true, and SAVERIA halts and returns TRUE (line 4) to indicate $\mathcal{P}$ satisfies S. Otherwise, UNWIND returns a path $\pi \in \mathcal{P}$ such that Φ does not summarize. Then, SAVERIA calls REFINE (line 6) procedure to to generate *interpolants* and refine path *invariants* along the path π. If, REFINE functions returns either FALSE or visible suffix summaries Φ. If it returns FALSE, then SAVERIA halts and returns FALSE (line 7) to indicate that $\mathcal{P}$ does not satisfy S. Otherwise, SAVERIA takes Φ and calls the COVER (line 9) function to check the cover relation over updated Φ by REFINE.

Input : Visible suffix summaries Φ.
Output : TRUE to indicate Φ summarizes the program $\mathcal{P}$, or a path $\pi \in \mathcal{P}$ not summarized by Φ.

```
1  Procedure UNWIND[P](Φ)
2      Procedure UNWINDAUX(expq, ⋈)
3          if expq = ∅ then return TRUE;
4          n := expq.pop() ;
5          if (·, n) ∉ ⋈ ∨ (·, (n ⊏ ·)) ∉ ⋈ then
6              if (Tgt[M(m)] × ρ_i × PreLoc[M(n)]) = ∅ then
7                  φ(n) := TRUE ;
8                  return π_n ;
9              else
10                 forall (Tgt[M(m)] × ρ_i × PreLoc[M(n)]) do
11                     π_m := m ⟶_{ρ_i} π_n ;
12                     φ(m) := FALSE ;
13                     expq.push(m) ;
14                 end
15             end
16         end
17     return UNWINDAUX(expq, ⋈) ;
```

Algorithm 2: UNWIND[$\mathcal{P}$] procedure takes the visible suffix summaries Φ, and returns either **(1)** TRUE to indicate the program is safe or **(2)** a path $\pi \in \mathcal{P}$ which is not summarized by Φ.

Alg. 2 presents pseudo-code for the algorithm of UNWIND[$\mathcal{P}$]. UNWIND[$\mathcal{P}$] defines and uses the procedure UNWINDAUX (line 2 — line 16) the takes expansion queue $expq$, and the cover relation set $\bowtie$ of the program $\mathcal{P}$. $expq$ contains nodes $n \in N$ to expand $\texttt{Locs} \in \mathcal{P}$ into $\pi \in$ PATHS[$\mathcal{P}$]. First, UNWINDAUX checks if there is any node in $expq$ to expand (line 3). If $expq$ is empty, then UNWINDAUX returns true (line 3) to indicate that all paths $\pi \in$ PATHS[$\mathcal{P}$] is summarized by the visible suffixes Φ. Otherwise, UNWINDAUX takes the first none n in the expansion queue (line 4, and checks whether n is covered (line 5). If n is directly covered by another node, or any of its descendants are covered, then n is not expanded. If n is not covered, UNWINDAUX checks if $M(n)$ is the initial location (line 6). If n is the initial location, then the label for n is assigned as TRUE (line 7), and UNWINDAUX returns a unique path (line 8). If n is not covered, a node m is created for each location which are previous locations of $M(n)$ in the CFG (line 10). An edge is created

Input : A path $\pi \in \mathcal{P}$, and visible suffix summaries Φ.
Output : FALSE to indicate π does not satisfy S, or visible suffix summaries Φ of $\mathcal{P}$.

```
1  Procedure REFINE[P, S](π, Φ)
2      Procedure REFINEAUX(⋈)
3          if I(π ∧ ¬S) = FALSE then
4              forall n ∈ π do
5                  φ(n)' := φ(n) ∨ I(n) ;
6                  if φ(n)' ⊭ φ(n) then
7                      ⋈ := ⋈ \(·, n) ;
8                      expq.push(n) ;
9                  end
10                 φ(n) := φ(n)' ;
11             end
12             return Φ ;
13         else return FALSE ;
14     return REFINEAUX(⋈) ;
```

Algorithm 3: REFINE$[\mathcal{P}, S,]$ procedure takes a path π and the visible suffix summaries Φ of $\mathcal{P}$, and returns either **(1)** FALSE if there is no inductive summaries of the path π or **(2)** visible suffix summaries Φ of $\mathcal{P}$ merged with the inductive summaries of π.

which contains the transitions ρ_i from m to n (line 11). The label of m is assigned to FALSE (line 12), and m is pushed into the expansion queue (line 13).

Alg. 3 presents pseudo-code for the algorithm of REFINE$[\mathcal{P}, S]$. REFINE$[\mathcal{P}, S]$ defines and uses the procedure REFINEAUX (line 2 — line 13) which takes the cover relation. First, REFINEAUX checks whether the path π is safe or not (line 3). REFINEAUX takes a path and the negation of the safety property. Interpolants for the path π is generated by a theorem prover [12]. The result of the generated *interpolants* is either **(1)** FALSE to prove that the path $\pi \wedge \neg S$ is infeasible or **(2)** TRUE to prove $\pi \wedge \neg S$ is feasible. If the path is safe, the final program state does not end with the error state $\neg S$. Let the initial location and the final location in a path are donated l_i and l_f respectively. In this case, all nodes in the path π are traversed from $M(m) = l_i$ through $M(n) = l_f$ (line 4 — line 11). The disjunction of the current invariant of the node n and the generated interpolant for n (line 5) is compared with the current invariant (line 6). If the disjunction is weaker than the current invariant, n's covered by information is removed from cover relation. The node n is pushed again

Input : A program $P \in \mathcal{L}$ and safety specification $S \in \mathsf{Constraints}[\mathcal{T}]$.
Output : A decision as to whether $\mathcal{P}$ satisfies S.

```
1  Procedure COVER[P](Φ)
2      Procedure COVERAUX(N, ⋈)
3          forall n ∈ N do
4              forall m ⊑ n do
5                  if (·, m) ∉ ⋈ ∨ (·, (m ⊏ ·)) ∉ ⋈ then
6                      if φ(m) ⊨ φ(n) then
7                          ⋈ := ⋈ ∪ (n, m) ;
8                          ⋈ := ⋈ \ (m, ·) ;
9                      end
10                 end
11             end
12         end
13     return COVERAUX(N, ⋈)
```

Algorithm 4: COVER$[\mathcal{P}]$ takes the visible suffix summaries Φ and maintains the cover relation of the nodes $n \in N$.

to expand, if there is any direct ancestor left to expand (line 8). Note that, pushing into expansion queue is performed for all nodes $m \sqsubset n$ where any m has at least one ancestor left unexpanded.

Alg. 4 contains pseudo-code for the algorithm of COVER$[\mathcal{P}]$. COVER$[\mathcal{P}]$ defines and uses the procedure COVERAUX (line 2 — line 12) which takes the nodes of path-invariant tree. First, for all nodes $n \in N$, COVERAUX traverses all nodes $m \sqsubseteq n$ (line 4 — line 11). For a node m such that m or any of its descendants is not covered (line 5), COVERAUX performs an implication check whether n is weaker than m (line 6). If m is stronger than n, then n covers m and this relation is added to the cover relation (line 7. Since a covered node cannot cover others, the relation for m covers others are removed from the cover relation. For the nodes covered by m, since they are not covered anymore, they are pushed into the expansion queue if they or their ancestors in the same path have any unexpanded ancestor left.

CHAPTER 4

RESULTS

In this chapter, we will talk about the empirical results for CAMPY. In § 4.1, we will present our test environment. In § 4.2, we will discuss the results presented in Table 4.1 which is taken from CAMPY [9].

4.1 Test Environment

All the experiments are performed on the machine with the following properties

- 16 cores
- 1.4 GHz
- 132 GB RAM

4.2 Evaluation

We have collected variety of benchmarks from various online coding exercise and challenge platforms [13, 14, 15]. Since our final product is CAMPY, only CAMPY is experimented on the benchmarks and its performance measurements are collected. In Table 4.1, performance measurements both time and memory are shown. The correctness of results and performance measurements for CAMPY promises its applicability. Considering CAMPY performs additional computation over the output of SAVERIA, SAVERIA requires even less time and memory. This makes SAVERIA worth to provide as a tool which other tools can be plugged in SAVERIA as CAMPY.

Table 4.1: The results of evaluating CAMPY. Each benchmark program is associated with two rows: the first row contains data for verifying that the program satisfies the tightest bound found; the second row contains data for verifying that the program does not satisfy the looses bound found. The column titled "Name" contains the benchmark's name; the column titled "LoC" contains the number of lines of source code; the column titled "Loops" contains the number of loops in the benchmark; the column titled "Nesting." contains the maximum nesting depth of loops in the benchmark. The column titled "Bound" contains the bound provided to CAMPY. Under heading "Performance", the column titled "Time" contains the time used by CAMPY; the column titled "Memory" contains the peak amount of memory used by CAMPY. "-" indicates that CAMPY timed out on the benchmark and did not return a definite result.

Program Structure				Bound	Performance	
Name	LoC	Loops	Nesting		Time (s)	Mem (MB)
Array2	39	4	2	n^2	5.3	12.4
				n	5.7	13.3
BirthdayCandles	43	4	2	$t \cdot n$	3.1	15.7
				t	2.9	14.4
Bit	61	2	2	n^2	2.8	13.6
				n	3.1	13.6
CodeChefJava	45	4	2	$3 \cdot t \cdot s$	3.2	46.0
				t	2.9	36.0
DRGNBOOL	51	3	2	$n \cdot (a + b)$	5.1	25.4
				$(a + b)$	3.3	14.5
FibonacciIterative	18	1	1	n	2.7	13.8
				10	3.3	14.1
Ideone	42	2	2	$n \cdot a$	3.5	15.1
				n	2.6	13.4
JewelAndStone	37	3	3	$t \cdot x \cdot y$	4.6	15.2
				$x \cdot y$	3.0	13.8
Jewel2	40	3	2	$350 \cdot t$	3.5	16.8
				10	3.1	13.3
LIS	47	3	2	$n \cdot (\log n)$	3.4	14.6
				n	2.8	13.1
Loops_1	30	2	2	$t \cdot n$	3.4	36.0
				n	2.8	34.0
Pie	34	3	2	$t \cdot 2n$	2.9	14.1
				$t \cdot n$	3.1	13.8
Scroll	58	3	2	$t \cdot (\log a + \log b)$	3.1	13.6
				t	2.9	16.1
SmartSieve	31	2	2	$n^2 + \log n$	4.4	20.2
				10	3.7	18.1
Sweet	125	2	2	$t \cdot n$	3.5	14.2
				n	6.4	20.5
Test0	22	2	1	n	2.7	13.1
				10	2.9	13.1
Test1	45	3	2	$2 \cdot t \cdot n$	3.7	15.2
				n	3.2	12.8
Test2	36	3	2	$n \cdot t$	3.8	13.9
				n	3.2	13.1
Test3	40	4	2	$3 \cdot t \cdot n$	4.3	10.6
				n	3.1	10.8
Test4	37	3	2	$c \cdot n$	3.1	14.7
				c	3.8	16.1
Test5	77	6	3	$t \cdot (n + n^2)$	3.4	54.8
				n^2	2.8	138.0
Test6	36	4	3	$n^2 \log n$	4.1	52.9
				n^2	5.3	47.3

CHAPTER 5
CONCLUSION

We have presented and automated property verifier, SAVERIA, which can prove whether a program satisfies the desired property. Since CAMPY runs on SAVERIA, the correctness of CAMPY relies on SAVERIA. If SAVERIA does not work properly, then it is not expected for CAMPY to work as expected. The soundness and correctness of CAMPY implies that SAVERIA is sound as a verification tool. The success of CAMPY thrives us to extend SAVERIA to verify different properties of Java programs.

CHAPTER 6
FUTURE WORK

There are plenty of extensions can be developed in the future to improve the security and privacy for software written in Java programming language. The common feature of all these extensions are to find bugs, detect vulnerabilities (by implementation or design) and disclose malicious behaviours.

- ***DoS Attack Vulnerability Detection:*** Using current complexity verifier and safety checker, upon modelling the current synchronization in Android [16] is possible to detect the entry points to critical paths which could lead to freezing and eventually DoS attacks in Android.

- ***Partial Information Leakage Detection:*** Given set of sensitive information and leak functions, it is possible to extend the current safety verifier to detect whole even partial information leakage. The leak detection would be able to detect for only the given sensitive information source and leak vectors.

- ***Cross Channel Attacks Based on Timing:*** It is possible to gain information about program behaviour or whether the inputs are sensitive or not based on the runtime of the program on different inputs.

REFERENCES

[1] Wikipedia, *Program analysis — wikipedia, the free encyclopedia*, [Online; accessed 4-April-2017], 2017.

[2] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," *ACM SIGPLAN Notices*, vol. 37, pp. 58–70, 2002.

[3] K. L. McMillan, "Lazy abstraction with interpolants," *International Conference on Computer Aided Verification*, pp. 123–136, 2006.

[4] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.-J. Hwang, "Symbolic model checking: 1020 states and beyond," *Information and computation*, vol. 98, no. 2, pp. 142–170, 1992.

[5] M. Heizmann, J. Hoenicke, and A. Podelski, "Nested interpolants," *ACM Sigplan Notices*, vol. 45, pp. 471–482, 2010.

[6] *A framework for analyzing and transforming Java and Android applications*, `https://sable.github.io/soot/`, Accessed: 2016, 2016.

[7] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS*, 2008.

[8] K. L. McMillan, "An interpolating theorem prover," *Theoretical Computer Science*, vol. 345, no. 1, pp. 101–121, 2005.

[9] A. Srikanth, B. Sahin, and W. R. Harris, "Complexity verification using guided theorem enumeration," *ACM SIGPLAN Symposium on Principles of Programming Languages*, pp. 639–652, 2017.

[10] R. Alur and P. Madhusudan, "Adding nesting structure to words," *J. ACM*, vol. 56, no. 3, 2009.

[11] G. Nelson and D. C. Oppen, "Simplification by cooperating decision procedures," *TOPLAS*, vol. 1, no. 2, 1979.

[12] *Z3prover/z3 - github*, `https://github.com/Z3Prover/z3`, Accessed: 2015 Nov 7, 2015.

[13] *Programming competition, programming contest, online computer programming*, `https://www.codechef.com/`, Accessed: 2016 June 14, 2016.

[14] *LeetCode online judge*, https://leetcode.com/, Accessed: 2016, 2016.

[15] *Programming competitions and contests, programming community*, https://www.codeforces.com/, Accessed: 2016 June 14, 2016.

[16] H. Huang, S. Zhu, K. Chen, and P. Liu, "From system services freezing to system server shutdown in android: All you need is a loop in an app," *ACM SIGSAC Conference on Computer and Communications Security*, pp. 1236–1247, 2015.

[17] S. Gulwani, K. K. Mehra, and T. Chilimbi, "Speed: Precise and efficient static estimation of program computational complexity," *ACM Sigplan Notices*, vol. 44, pp. 127–139, 2009.

[18] *CVE - CVE-2011-3191*, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-3192, Accessed: 2015 July, 2015.