Interactive Synthesis of Code-Level Security Rules

A Thesis Presented
by

Leo St. Amour

to

The Department of Computer Science

in partial fulfillment of the requirements
for the degree of

Master of Science

in

Computer Science

Northeastern University
Boston, Massachusetts

April 2017

This material is based upon work supported by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Assistant Secretary of Defense for Research and Engineering.

To my family, who has always given me their love and support.

Contents

List of Figures

List of Tables

Acknowledgments

I would first like to acknowledge Dr. George K. Baah for all of the guidance and feedback that he has given me throughout this process. Without his support, this thesis would not have been possible. I would also like to thank Mark Yeager for voluntarily giving up a weekend to edit and provide feedback on the paper. Next, I would like to thank Dr. Robert K. Cunningham at MIT Lincoln Laboratory for having faith in me and giving me the opportunity to be here today. Finally, I would like to thank Dr. William K. Robertson, my advisor, for being patient and guiding me as I struggled to find direction for this project.

Abstract of the Thesis

Interactive Synthesis of Code-Level Security Rules

by

Leo St. Amour

Master of Science in Computer Science

Northeastern University, April 2017

Dr. William K. Robertson, Adviser

Software engineers inadvertently introduce bugs into software during the development process and these bugs can potentially be exploited once the software is deployed. As the size and complexity of software systems increase, it is important that we are able to verify and validate not only that the software behaves as it is expected to, but also that it does not violate any security policies or properties. One of the approaches to reduce software vulnerabilities is to use a bug detection tool during the development process. Many bug detection techniques are limited by the burdensome and error prone process of manually writing a bug specification. Other techniques are able to learn specifications from examples, but are limited in the types of bugs that they are able to discover. This work presents a novel, general approach for synthesizing security rules for C code. The approach combines human knowledge with an interactive logic programming synthesis system to learn Datalog rules for various security properties. The approach has been successfully used to synthesize rules for three intraprocedural security properties: (1) out of bounds array accesses, (2) return value validation, and (3) double freed pointers. These rules have been evaluated on randomly generated C code and yield a 0% false positive rate and a 0%, 20%, and 0% false negative rate, respectively for each rule.

Chapter 1

Introduction

1.1 Motivation

Developing secure and correct software systems is a difficult endeavor. Software engineers inadvertently introduce security vulnerabilities and undesired behavior, which can potentially be exploited when the software is deployed. Vulnerabilities in the U.S. retailer Target's point-of-sales system resulted in the theft of 40 million account's credit card information [47]. A breach of the Office of Personnel Management computer systems resulted in the theft of 21.5 million people's personal information including social security numbers [19]. The Heartbleed bug [5] was the result of a programming mistake in the OpenSSL library that allowed attackers to leak secret keys [8].

The problem of vulnerabilities has grown more severe with the increase in autonomous systems such as robots and satellite systems, and an explosion in the Internet of Things (IoT). One of the ways to reducing vulnerabilities is to use static analysis tools. Static analysis techniques, if sound, have the advantage of proving the absence of a security vulnerability by checking a program against a specification. Static analysis tools use these specifications or rules that are manually developed a priori and are incorporated into the tool. Manually specifying the rules is a tedious and error prone process. However, since it is important to incorporate human knowledge into the process [25], a fully automatic approach is not desirable.

This work proposes a semi-automatic, interactive approach for synthesizing security rules for C code. The focus of this project is on code-level security properties such as the secure coding standards presented by CERT [3].

1.2 Current Techniques and Limitations

This project addresses the limitaitons of current vulnerability detection techniques by providing a general framework for learning specifications for security properties. There are numerous examples in the literature of tools that identify security vulnerabilities or programming errors. Some tools detect specific types of vulnerabilities such as memory leaks in C or C++ code [26]. Others can be used to detect different types of bugs or vulnerabilities. Hallem et al. [24] presented Metal, a language for specifying program analyses. Xie and Aiken [51] demonstrated how boolean satisfiability solvers can be used to implement analyses such as lock checking. Yamaguchi et al. [52] represents programs as code property graphs and allows for a wide range of vulnerability discovery by manually modeling vulnerabilities as traversals over a graph. The Clang static analyzer [4] is a tool that checks for a variety of bugs from proper API usage to divide by zero detection. These tools all have one thing in common: they are not able to detect bugs if they do not know what they are checking for. Also, many of the existing bug detection tools require an algorithm or a specification to be manually written. This process is both time intensive and error prone.

Recent literature has demonstrated the value of learning security specifications from programs. Kremenek et al. presented a framework for inferring what functions in C programs allocate and release resources [31]. Chang et al. presented an approach for identifying neglected conditions [17]. Chucky [53] and APISan [54] expose missing checks and learn proper Application Programming Interface (API) usage, respectively, by assuming that the majority of uses are correct. While these tools have reduced the need for manual specifications, the approaches are limited.

1.3 Threat model

Within our framework, we assume that we will have full access and control over the source code of an undeployed system. We assume an adversary that (1) cannot modify the source code, (2) has not compromised the toolchain [46], (3) cannot modify the code between compilation and deployment, (4) cannot compromise the hardware that the system will be deployed on, and (5) once deployed, will have control over all inputs to the system.

This threat model allows for us to operate under the assumption that the code we are analyzing is the code that will be deployed. We are able to assume that once the analysis has been completed, that the adversary will not be able to modify the code to introduce bugs that our framework would have otherwise identified.

1.4 Thesis Contributions

This work aims to improve upon the state of vulnerability detection by providing a preliminary framework for interactively learning specifications for a wide range of vulnerabilities. This thesis makes the following contributions:

- **A mechanism for translating C code to Datalog relations**
- **An interactive, general purpose framework for generating security property rules**
- **A proof of concept that security rules can be synthesized from examples**

Chapter 2

Background

This chapter presents the terminology that will be used throughout this paper as well as provide an overview of the various systems that this thesis leverages.

2.1 Program Analysis

Program analysis is the process of analyzing a program's behavior to either discover properties, optimize the program, or verify the program's correctness, and in general the program analysis problem is undecidable [42]. There are two kinds of program analysis techniques: static and dynamic.

2.1.1 Static Analysis

Static analysis techniques analyze a program's behavior over all possible inputs without executing the program. Several tools have been created based upon static analyses. Some of these tools are bug finding tools, that attempt to find bugs by examining the program text. Some of the tools are sound and incomplete. Soundness means that the tool does not miss any bugs, being incomplete means that the the tool could produce false warnings. Other tools may be unsound and incomplete meaning that it can both miss bugs and produce false warnings. The advantage of static analysis tools is that they explore all potential program behaviors. The disadvantage of static analyses is that they make approximations that tend to produce false positives and false negatives. If the tool makes an over-approximation, there is potential for code that is not incorrect to be reported as a bug. Similarly, if the tool makes an under-approximation, it could fail to report an actual bug.

Linters [28] are static analysis tools that flag source code that is suspicious or does not adhere to a style guideline. The Clang static analyzer [4] is a static analysis tool that checks for security and API usage bugs within C, C++, and ObjC code. These checks include divide by zero bugs, null pointer dereferences, dead code checks, and proper API usage (double frees, using `mktemp()` instead of `mkstemp()`, etc). A full list of available checks can be found at [1]. Other static analysis tools exist for detecting various bugs such as memory leaks [26], neglected conditions [17], proper API usage [54], format string vulnerabilities [44], generating test cases [15], or for providing general vulnerability detection frameworks [52, 24].

2.1.2 Dynamic Analysis

Dynamic analysis techniques analyze a program's behavior by executing the program. The advantage of dynamic analysis tools is that they can track runtime information, such as user inputs. For example, if an array of size `5` is being indexed at user defined index and the program is run with user input `6`, a dynamic analysis tool would be able to declare that there is an out of bounds array access. The disadvantage of dynamic analysis tools is that they are unsound, meaning that they are not guaranteed to report all errors.

Fuzzers are dynamic analysis tools for providing valid, invalid, and random inputs to a program and testing its behavior for exceptions or crashes. Examples include American Fuzzy lop [55] and TaintScope [48]. Other dynamic analysis tools exist for detecting various bugs such as memory errors [38], overwrite attacks [39], or for providing frameworks for dynamic instrumentation [35].

2.1.3 Intraprocedural and Interprocedural Analyses

There are two types of scopes for analyzing a program: *intraprocedural* and *interprocedural*. Intraprocedural analyses occur within the context of a single function whereas interprocedural analyses are conducted across all procedures within a program. Interprocedural analyses allow for more precise approximations of program behavior, because they allow information to flow between callers and callees [12].

Flow sensitivity and *context sensitivity* are two of the primary types of program analysis approximations. Flow sensitive analysis preserves the order of statements in the program. A flow insensitive analysis ignores the order of the statements in the program. For example, an analysis technique called points-to analysis determines what variables refer to the same memory location. A flow insensitive points-to analysis could report that pointers `p` and `q` refer to the same location, even if there exists a path down which they do not. Whereas, a flow sensitive analysis could report that pointers `p` and `q` *may* refer to the same location, but it depends on which path is followed. A flow sensitive algorithm allows for a more precise approximation of program behavior.

An analysis that is context sensitive tracks information about the call sites, or where in the code function calls are called. A context insensitive analysis does not account for the context in which functions are called. Context sensitive analysis are more precise because any information learned from analyzing the called procedure can be propagated back to a specific call site rather than all potential call sites.

2.1.4 Control Flow Graphs

A *Control Flow Graph (CFG)* is a directed graphical representation of a program where each node represents a basic block and each edge indicates which blocks can follow which other blocks [12].

Basic blocks are sequences of statements that can only be entered through the first instruction and only be exited through the last instruction[12]. Figure 2.2 is the CFG form of the program in Figure 2.1. The first basic block consists of statements `1`, `2`, and `3`. There are edges from statement `3` to statements `4` and `7` (depending on the user input). These instructions each constitute their own basic block and both of have edges to statement `9`.

```
1 int x,y;
2 scanf("%d", &y);
3 if (3 == y) {
4       x = 2;
5 }
6 else {
7       x = 3;
8 }
9 return x;
```

Figure 2.1: Sample Program

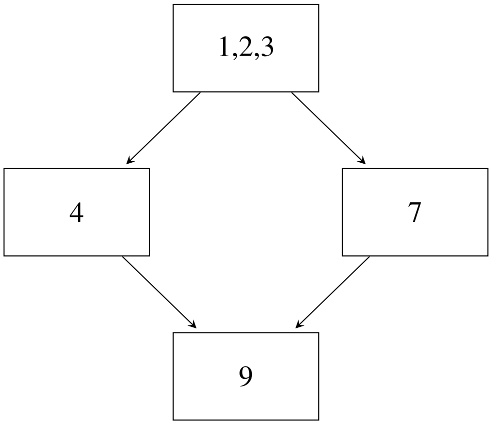


Figure 2.2: CFG of program in Figure 2.1

An interprocedural CFG contains edges between procedures, methods, or functions. If a statement is a call to a function, there is an edge between that statement and the entry of callee. A second edge is created from the exit of the callee back into the caller.

2.1.5 Static Single Assignment (SSA)

Static Single Assignment (SSA) is a program form in which each variable is assigned a value exactly once [43]. Additionally, every use of a variable is dominated by its definition. A node, n_1, in a CFG is dominated by another node, n_2, if every path from the start of the CFG to n_2 goes through n_1 [12], meaning that every path in the CFG from the program entrance to the variable's use passes through the variable's definition. This form is useful for optimizing and analyzing code because it simplifies variables that are re-assigned into separate variables. Take the code snippets below:

Code	SSA
`int foo(int y) {`	`int foo(int` y_1`) {`
`    if (3 == y) {`	`    if (3 ==` y_1`) {`
`        x = 2`	`        `x_1` = 2`
`    } else {`	`    } else {`
`        x = 2`	`        `x_2` = 3`
`    }`	`    }`
`    return x`	`    `x_3` = phi(`x_1`,`x_2`)`
`}`	`    return `x_3
	`}`

The code on the right is the SSA form of the code on the left. In SSA form, new instructions called *phi functions* are introduced. Phi instructions account for variables whose values could depend on which branches in a program are followed. In the code on the right, x_3 is assigned the value of either x_1 or x_2, depending on which execution path is taken. Converting a program SSA is particularly useful when the control flow of a program becomes more complex.

2.2 Logical programming languages

Logical programming languages emerged in the 1970's as a result of research in automatic theorem proving and artificial intelligence [34]. Beginning in 1972, Kowalski and Colmerauer introduced PROgramming in LOGic (Prolog), which was based on the idea that logic can be used as a programming language. Other logical programming languages include SWI-Prolog, DLV, .QL, and Datalog. The power of logical programming languages comes from the insight that logic can have a procedural interpretation. A logical rule $\mathtt{H} \leftarrow \mathtt{C}_1, \ldots, \mathtt{C}_n$ is analogous to defining procedure H. For a body consisting of clauses $\mathtt{C}_1, \ldots, \mathtt{C}_n$ each clause is a call to procedure C_i. This interpretation allowed for logic to be used as a programming language instead of just as specification. Algorithms written in logic programming languages consist of two distinct components: the logic and the control. The logic is "what" problem needs to be solved and the control is "how" the problem is to be solved. Logical programming languages are revolutionary because they allow the programmer to write the logic without being concerned about the control [34]. Section 2.2.1 defines logic programming terminology as well as demonstrates how to write a recursive algorithm in a logic programming language called Datalog.

2.2.1 Datalog

Datalog is a language for writing recursive logic programs. It is a syntactical subset of Prolog, but unlike Prolog, it is not a turing complete language. This section provides a brief overview of Datalog and its associated terminology. The definitions presented here are based on the definitions in [16], which presents a thorough survey on Datalog's syntax, semantics, optimization methods, and extensions.

A logic program consists of a finite set of *facts* and *rules*. A fact is an assertion about the world. For example: "Alex is the parent of Bob". Rules are sentences that allow facts to be deduced from other facts. For example: "If X is the parent of Y and Y is the parent of Z, then X is the grandparent of Z". In this rule, X, Y, and Z are all variables. These variables represent data from facts that can be used to deduce additional facts. Both facts and rules are represented as *horn clauses* which are expressions in the form:

$$\mathtt{H} :- \mathtt{B}_1, \ldots, \mathtt{B}_n$$

where H and $\mathtt{B}_i$ are *literals*. Literals consist of a *predicate symbol* and a set of *terms* which can either be constants or variables. For example, the fact "Alex is the parent of Bob" would be represented by the literal `parent(alex, bob)`. In this example, `parent` is the predicate symbol and `alex` and `bob` are the terms. Similarly, the rule "If X is the parent of Y and Y is the parent of Z, then X is the grandparent of Z" would be represented by the literal:

```
grandparent(X,Z) :- parent(X,Y), parent(Y,Z).
```

A literal, fact, rule, or clause is considered *ground* if it does not contain any variables.

Datalog programs can be considered as queries over a relational database. The set of facts is called the *Extensional Database (EDB)* and the set of rules is called the *Intensional Database (IDB)*. A Datalog program is essentially a mapping from EDB facts to IDB facts. For example, suppose we want to write a rule for deducing ancestor relationships. This rule is broken down into two parts: (1) “If `X` is the parent of `Z`, then `X` is an ancestor of `Z`” and (2) “If `X` is the parent of `Y`, and `Y` is the ancestor of `Z`, then `X` is an ancestor of `Z`”. This problem is represented by the Datalog program in Figure 2.3. Lines 1 through 3 are the set of facts that constitute the EDB. Lines 4 through 5 define the rules used to infer the facts that constitute the IDB. Using a solver (see Section 2.2.2), we can query the EDB to deduce the set of `ancestor` relations.

```
1 parent(alex, bob).
2 parent(bob, charles).
3 parent(charles, david).

4 ancestor(X, Z) :- parent(X, Z).
5 ancestor(X, Z) :-
6     parent(X, Y),
7     ancestor(Y, Z).
```

Figure 2.3: Rules for deriving ancestor relations

Datalog provides a number of benefits: First, Datalog is able to declare recursive relations as demonstrated by the rule for `ancestor` in Figure 2.3 Second, all Datalog queries terminate provided that two safety conditions are satisfied. The first safety condition is that all facts in a Datalog program must be ground. In Figure 2.3, all of the facts (line 1 to line 3) are considered ground because they do not contain variables. The second safety condition is that each variable that occurs in the head of a rule must occur in the body of that rule. Both of the rules for `ancestor(X,Z)` in Figure 2.3 have both variables `X` and `Z` in the right hand side of the horn clause. These safety conditions guarantee that the set of deduced facts is finite [16]. Third, there have been numerous extensions to the Datalog language to improve its expressiveness. Modified versions of Datalog allow for built in predicates such as comparison operators ($=, <, \leq, >, \geq$, etc.) and arithmetic operators ($+$ and $-$). While pure Datalog does not allow negation, there are implementations that allow negative facts. These features make Datalog a powerful and expressive database query language.

2.2.2 Datalog Engines

Datalog engines are used to derive facts based upon a set of rules and input relations. Given a set of relations and a rule, the engine will generate a set of relations that satisfy the rule. The example presented in Figure 2.3 outlines a rule for `ancestor` as well as a set of `parent` relations. Using a Datalog engine such as `LogicBlox`, `pyDatalog`, or `bddbddb` we can derive the following relations:

```
ancestor(alex, bob).
ancestor(alex, charles).
ancestor(alex, david).
ancestor(bob, charles).
ancestor(bob, david).
ancestor(charles, david).
```

A Datalog engine uses the values `alex`, `bob`, `charles`, and `david` to solve for `X` and `Z` in the rule for `ancestor`.

`bddbddb` stands for BDD-Based Deductive DataBase and is a Datalog solver [49]. `bddbddb` is built upon a data structure called a Binary Decision Diagram (BDD). BDDs, first introduced by [14], are directed acyclic graphs that efficiently store large amounts of data that share commonalities. The graph has a single root node and two terminal nodes. One of these terminal nodes represents *true* and the other represents *false*. Each non-terminal node represents an input variable and has exactly two outgoing edges, one that represents true and one that represents false. Traversing the graph for an input will lead to a true or false value for that particular input. `bddbddb` automatically translates Datalog programs and queries into BDD operations. BDDs are an ideal implementation for representing program analyses written in Datalog because BDD operations correspond closely to Datalog's evaluation style and many Datalog programs contain redundant data.

2.3 Program analysis with declarative languages

In recent years, there has been an increase in the applications for recursive declarative programming languages such as Datalog. These applications fall under different domains such as network monitoring, information extraction, data integration, and program analysis [27]. General logic programming languages were identified as an effective means for both writing and conducting static program analyses in the 1990's [20]. Datalog in particular has been used for a variety of program analysis techniques that range from low level analyses such as points-to analysis [50, 32, 13] to higher level analyses such as structural program dependences [22].

Whaley and Lam [50] created the `bddbddb` framework to use Datalog for extracting context-sensitive points-to information from Java programs, a previously unsolved problem. A typical Java program has a context-sensitive call graph with 10^{14} acyclic paths. Whaley and Lam were able to address this problem by exploiting the BDDs ability to handle commonalities amongst contexts.

Bravenboer and Smaragdakis introduced `Doop`, another pointer analysis framework for Java programs [13, 7]. `Doop` differs from `bddbddb` because it uses a strictly declarative specification for programs and analyses, whereas `bddbddb` represents the programs as BDDs. `Doop` is able to provide more precise analyses and address more Java features than previous works. The authors attribute this to their strictly declarative representation as well as various Datalog optimizations.

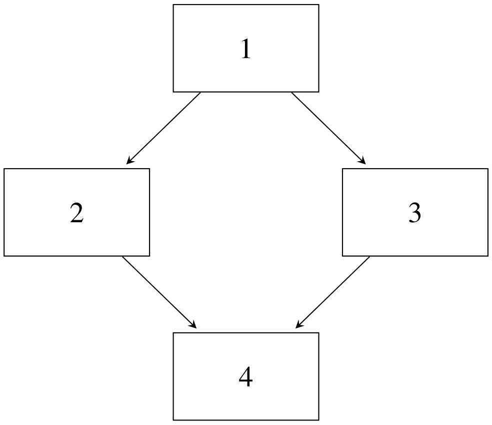
Datalog driven program analysis is not confined to low level analyses such as points-to analysis. Lam et al. [32] used `bddbddb` to write analyses for various web application vulnerabilities. Rules for SQL injections, HTTP splitting, cross-site scripting, and path traversals were written in Datalog. These rules were used to identify 17 previously unknown errors in open source Java web applications. Eichberg et al. [22] used Datalog to define rules to identify logical groupings of program components, or ensembles. Datalog is also used for specifying the architectural, design, and implementation constraints placed on programs.

Finally, `cclyzer` is a tool that uses Datalog for analyzing LLVM bitcode [2]. The relational representation of the LLVM IR used in this project is based on the schema used in `cclyzer` [40].

2.4 Program Synthesis

Program synthesis is the process of automatically synthesizing code from a declarative specification [30]. The general idea is that program synthesis results in a faster and more reliable software development process by allowing a user to focus on the specification rather than the implementation. One form of program synthesis called Inductive Programming (IP) synthesizes programs from examples and background knowledge [23]. Using small sets of examples and human experts, programs can be synthesized to automate repetitive tasks such as processing and transforming data.

Previous research shows the feasibility of using IP to learn, or synthesize, logical horn clauses from a set of example relations. For example, consider the graph reachability problem.



```
edge(1,2).
edge(1,3).
edge(2,4).
edge(3,4).

path(1,2).
path(1,3).
path(1,4).
path(2,4).
path(3,4).
```

Figure 2.4: Sample graph and its corresponding relations

Suppose a graph with a set of nodes and edges represented by `edge(X,Y)` relations which indicate an edge from node `X` to node `Y`. Given this set of relations, can a horn clause, or rule, for a `path` within a graph be synthesized? Figure 2.4 presents the CFG, labeled with basic block IDs, and the set of corresponding Datalog relations for the program in Figure 2.1. The set of relations includes CFG edges and the explicit set of `path` relations, where `path(X,Y)` indicates that there is a path from node `X` to node `Y`. Based on this set of explicit `path` relations, we can learn a general rule. There are numerous systems, which will be discussed in more detail later, that can learn the appropriate horn clause given positive and negative examples of `path` relations. Given the example in Figure 2.4, these systems would output the following rule:

```
path(x,y) :- edge(x,y).
path(x,y) :- edge(x,z), path(z,y).
```

One of the earliest inductive logic programming tools is `FOIL` [41]. Given positive tuples that satisfy the target relation, *P*, `FOIL` calculates a value, called *gain*, for each literal in the logic program. This value influences which literals are chosen for the right hand side of the synthesized horn clause. `Metagol` is a inductive logic programming tool that uses metarules to define what output clauses can look like [37]. `ALPS` is a system that interactively synthesizes logic programming horn clauses [45]. This thesis utilizes `ALPS` for synthesizing security rules.

2.5 Petablox

Petablox is an interactive program analysis framework that accounts for user feedback to guide approximations in analyses [36]. The motivation for Petablox is that analysis writers have the knowledge to design the analyses, but are not able to predict every usage or program specific nuances. Conversely, the user does not have the expertise to write the underlying analysis, but has the knowledge of the program to confirm or deny the analysis reports as real.

An analysis in the Petablox framework is written as a set of Datalog rules. Petablox is able to close the gap between the analysis writer and users by considering two types of rules. Rules can either be *weighted* or *unweighted*. Unweighted rules, or hard rules, are rules that *must* be satisfied. The weighted rules, or soft rules, are rules that do not necessarily need to be satisfied. The goal of Petablox is to find a solution that satisfies all of the hard rules and maximizes the weight of the satisfied soft rules. The hard rules are intended to represent the soundness of the analysis. The soft rules are intended to represent the degrees of approximation. For example soft rules could be written to account for path, flow, or context sensitivity.

When an analysis is run on a program, the user is presented with the report. The user can either “like” or “dislike” each of the results. The system incorporates and generalizes the user feedback as new soft rules. The analysis is run again and the process is repeated until the user is satisfied with the report. Petablox reduces false positives by accounting for user feedback. On average, when users provided feedback for 20% of the reports, 70% of false reports are eliminated while retaining 98% of the true reports [36].

2.5.1 `ALPS`

The `ALPS` system, a component of the Petablox framework, introduces an interactive approach for synthesizing declarative programs. The system asks the user targeted *yes* or *no* questions. Using the relations for the program in Figure 2.1, `ALPS` goes through four iterations of yes/no questions before outputting the correct rule for `path`. The user is asked: `Is path(1,2) true?` (yes), `Is path(1,1) true?` (no), `Is path(1,3) true?` (yes), and `Is path(1,4) true?` (yes). Using an active-learning technique called query by committee and a template guided search, `ALPS` is able to synthesize a Datalog rule given a few relations and asking the user few questions.

`ALPS` is different from previous inductive logic programming works in a number of ways. First, traditional inductive logic programming systems are intended to learn from large input sets of relations. `ALPS` is designed to synthesize a rule based upon a small, representative set of examples. Second, unlike other systems, `ALPS` is designed to synthesize recursive programs. Third, `ALPS` uses a complete, bi-directional search strategy. Full details on the `ALPS` system are presented by Si et al. in [45].

`ALPS` uses a template guided approach to synthesize Datalog rules. An `ALPS` template is a generic datalog rule such as: `A(x,y) :- B(x,y), C(x,y).` In the previous example, ALPS would not have been able to synthesize the rule for `path(x,y)` without having been provided the templates: `A(x,y) :- B(x,y).` and `A(x,z) :- B(x,y), C(y,z).`

Additionally, `ALPS` allows for "rich" templates. A rich template allows for specific relations to be specified within the template. For example, `A(x,z) :- @edge(x,y), C(y,z).` In a rich template, a particular relation is identified by a preceding "@" symbol. `ALPS` uses a template file to aggregate a set of syntactically correct rules that constitute the search space. Using a rich template reduces the search space and increases the efficiency of the synthesis.

`ALPS` has advantages over other logic program synthesis systems, but has a number of limitations. First, the template system is an inherent limitation. While the purpose of the template guided refinement strategy is to restrict the search space while synthesizing rules, it requires the user to have knowledge about the structure of a potential rule. Second, the `ALPS` does not allow for a user to include existing rules as input relations. For example, suppose we want to synthesize a rule for detecting a loop within a graph. The rule for a loop will likely involve the rule for a path. `ALPS` does not have the means for utilizing existing rules, such as the rule for `path` above. In order for a rule for loop to contain `path`, the individual `path` relations must be deduced via pre-processing and supplied to `ALPS`. Third, ALPS does not have a mechanism for built in predicates. Similarly, relations such as `eq(X,Y)`, `lt(X,Y)`, etc. will need to be manually specified.

2.6 The LLVM Compiler Infrastructure

LLVM [9, 33] was originally created as a compiler for lifelong program analysis and transformations, but has since expanded to become an umbrella project for an open-source collection of modular compiler technologies. The goal of the LLVM project was to provide "lifelong code optimizations," which means that optimizations can occur at any stage in the program's life (i.e. link time, install time, run time, or idle time). LLVM accomplishes this goal by providing two components: (1) a language independent intermediate representation and (2) a compiler design that exploits the representation to provide novel capabilities.

The LLVM IR provides a source and target independent representation for programs. It is designed to be low level enough that it is source language independent, yet high level enough to allow for more sophisticated analyses and transformations. The IR uses an infinite set of virtual registers, in SSA form, that can hold values of primitive types (i.e. boolean, various width integers, floating points, and pointers). Memory is manipulated using a load/store architecture and values are transfered between registers and memory via load and store operations.

The LLVM IR has a language-independent type system. The system includes source language independent primitive types: void, bool, int and float as well as derived types: pointers, arrays, structures, and functions. The authors believe that most high-level languages are eventually represented using some combination of the derived types and primitive types (for example a class in C++ is represented by structures, functions, and arrays of function pointers).

A program written in the LLVM IR is made up of a set of functions. Each function is a set of basic blocks and each basic block is a series of instructions. Each basic block ends with a terminator instruction that explicitly specifies its successor basic blocks. Syntactic and semantic documentation for the LLVM IR can be found in the LLVM reference manual [11].

The LLVM compiler architecture allows for transformations at all stages of a program's life by operating on the LLVM IR. While optimizations can be performed by linkers or at run time with just-in-time translators, we will focus on static compilation. A static compiler can perform three key tasks: (1) perform, optional, source language specific optimizations, (2) translate the source program into the LLVM IR, and (3) invoke optional LLVM passes to optimize the code. The optimizations, including custom optimization passes, are built into libraries. These libraries make it easy for front-ends to use them. These optimization passes can iterate through each function in a program, each basic block within a function, and finally each instruction within a basic block.

LLVM has a number of front ends for various languages such as C, C++, ObjC, and Fortran. Clang is a C/C++ front-end for LLVM and has been packaged with LLVM since LLVM 2.6. Clang can load additional optimization passes via a command line argument. It allows for analysts to easily write and insert their passes into the compilation process.

Chapter 3

Synthesizing Security Rules

3.1 Approach

Current approaches for detecting security property violations either require a manual specification [18, 24, 26, 51, 52], or are limited to a specific type of security property [17, 31, 53, 54]. Throughout this work, we shall use the terms property and rules interchangeably. In this chapter, we present our approach to synthesize security rules. The approach eliminates the need to write manual specifications for security properties, and provides a general framework for synthesizing a variety of security rules.

Figure 3.1 shows a dataflow diagram of our approach. The approach consists of three main phases. In the first phase, a program, `P`, is compiled into the LLVM Intermediate Representation (LLVM IR). In the second phase, the LLVM IR is passed to a relational translator, which transforms the IR into a set of Datalog relations. The relations can optionally be processed by `bddbddb` to deduce additional relations from pre-written Datalog rules. In the third phase, the relations and user provided templates are passed to `ALPS`. The user is asked a series of yes/no questions and `ALPS` synthesizes and outputs a Datalog rule. We provide details for phases two and three. Phase one is accomplished using a standard LLVM compiler, and will therefore not be discussed in detail.

3.1.1 Phase Two: Extracting Relations

During the second phase, our technique extracts Datalog relations from a program, `P`. The LLVM IR form of `P`, represented by $\text{IR}_{\mathtt{P}}$ in Figure 3.1, is input into the relational translator. The translator is implemented as a custom LLVM `FunctionPass` [10] to the Clang front-end. The

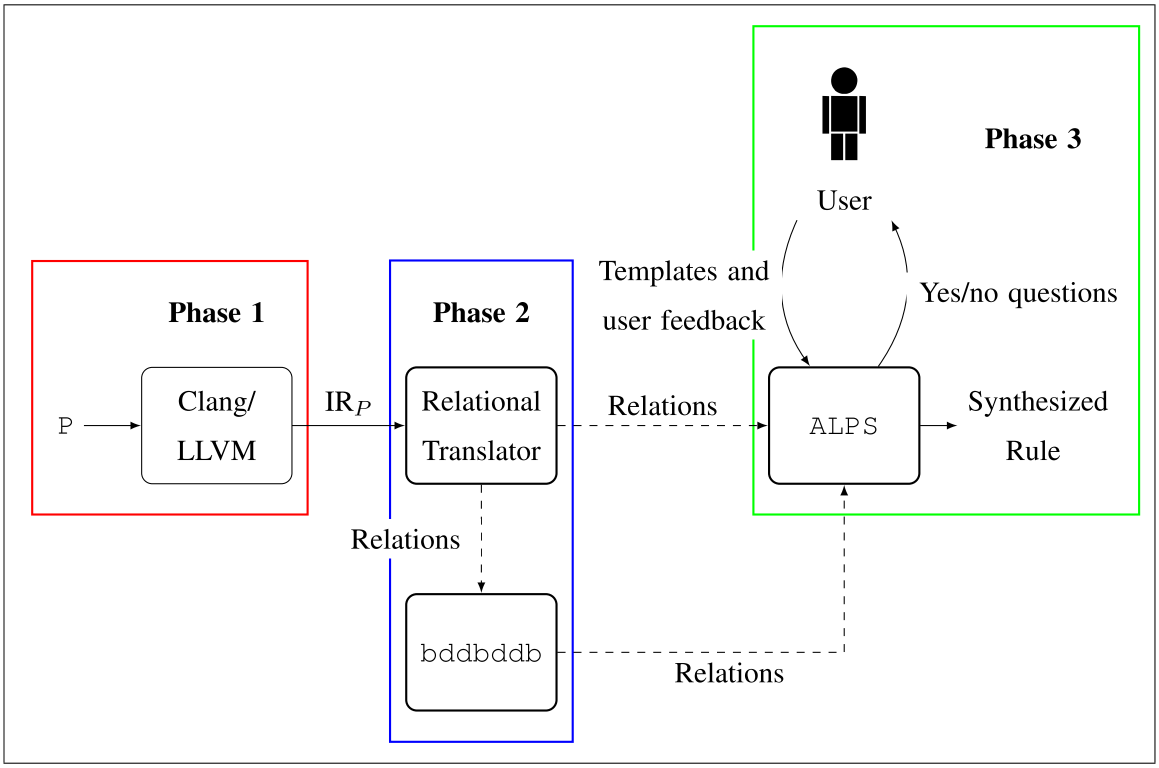


Figure 3.1: System dataflow diagram

pass, iterates through every instruction in IR$_P$and outputs relations detailing information about each particular instruction. In addition, the pass outputs information about IR$_P$ such as control flow, the type system, and global values. These relations will be used to reason about P's behavior. Detailed documentation on the Datalog facts that are extracted from C programs is presented in Appendix A.

```
int main(int argc, char *argv[]) {
    int x = 1;
    int y = x + 2;
    return y;
}
```

Figure 3.2: Example C program, P

The relational translator is designed to be sound so that we can reconstruct the LLVM IR program from the set of relations. Figure 3.2 shows a sample program, P, that assigns the value 1 to variable x, assigns the value x+2 to a variable y, and returns the value of y. Figure 3.3 presents the program P as an LLVM IR program after phase one and the relations extracted for instruction %1 (in red) after phase two. The relations for instruction %1 provide enough information to understand the semantics of the instruction.

In this example, all terms are represented by identifiers, which are strings preceded by a percentage sign (i.e. %i32,%const4, etc). The relations integer_type(%i32) and integer_type_width(%i32, 32) indicate that the identifier %i32 represents an integer type with a bit width of 32. The relations alloca_inst(%1), alloca_inst_align(%1, %const4), alloca_inst_size(%1, %const1), and alloca_inst_type(%1, %i32) indicate that the identifier %1 represents an alloca instruction that allocates memory for %const1 elements of type %i32 with an alignment of %const4. The relations constant(%const1) and constant(%const4) indicate that the identifiers %const1 and %const4 both represent constant values. The relations constant_type(%const1, %i32) and constant_type(%const4, %i32) indicate that constants %const1 and %const4 both have a type of %i32. Finally, the relations constant_value(%const1, 1) and constant_value(%const4, 4) indicate that constants %const1 and %const4 have values of 1 and 4, respectively. Based upon these relations, we can understand the semantics of instruction %1.

LLVM IR	Datalog relations for instruction %1
`%1 = alloca i32, align 4`	`integer_type(%i32).`
`%2 = alloca i32, align 4`	`integer_type_width(%i32, 32).`
`%3 = alloca i8**, align 8`	
`%x = alloca i32, align 4`	`alloca_inst(%1).`
`%y = alloca i32, align 4`	`alloca_inst_align(%1, %const4).`
`store i32 0, i32* %1, align 4`	`alloca_inst_size(%1, %const1).`
`store i32 %argc, i32* %2, align 4`	`alloca_inst_type(%1, %i32).`
`store i8** %argv, i8*** %3, align 8`	
`store i32 1, i32* %x, align 4`	`constant(%const1).`
`%4 = load i32, i32* %x, align 4`	`constant_type(%const1, %i32).`
`%5 = add nsw i32 %4, 2`	`constant_value(%const1, 1).`
`store i32 %5, i32* %y, align 4`	`constant(%const4).`
`%6 = load i32, i32* %y, align 4`	`constant_type(%const4, %i32).`
`ret i32 %6`	`constant_value(%const4, 4).`

Figure 3.3: The LLVM IR form of Figure 3.2 and sample relations

Each identifier in the relational representation of program P must be assigned to a domain. ALPS and bddbddb use domains as a type system. Each term in a literal must fall within a specified domain. In Figure 3.3, the identifiers %1,%const1, and %cosnt4 all fall into the Operand domain and the identifier %i32 falls into the Type (T) domain. This allows for each literal (i.e. `alloca_inst`, `integer_type`, etc) to have a domain assigned to its terms. For example, the literal `alloca_inst_type(inst:O, type:T)` indicates that the first term in the literal must be from the Operand domain and the second term must be from the Type domain. The key domains are Function (F), Operand (O), Type (T), Basic Block (B), and Integer (N). These domains are all intuitively named except for Operand. The Operand domain consists of all LLVM instructions, variables, and constants. An LLVM IR instruction can accept different operands. For example, an `add` instruction accepts two operands, the values to be added together. The operands can either be constant values or registers that contain the value stored in a variable or the result of a previous instruction. To account for this variety, all possible operands fall under a single domain. After Datalog relations are extracted from IR$_P$, all identifiers are processed and assigned to a domain. A full list of domains is presented in Table 3.1.

O	Operand
T	Type
N	Integer
B	Basic Block
F	Function
V	Visibility
L	Linkage
A	Attribute
AO	Atomic Ordering
CC	Calling Convention
CP	Comparison Operator

Table 3.1: Domains

After each literal has been assigned its domains, the user may utilize `bddbddb` to deduce additional relations based on the facts output by the relational translator. The user may have pre-defined rules for relations, such as instruction reachability or integer inequalities, that would not have been otherwise extracted from IR$_P$. This optional step allows users to overcome the limitation discussed in Section 2.5.1 that `ALPS` is unable to account for pre-defined Datalog rules.

3.1.2 Phase Three: Synthesizing Rules

Once the facts have been extracted from the input program and additional facts have been produced using `bddbddb`, we use `ALPS` to synthesize new rules. The inputs to `ALPS` are the set of Datalog facts from phase two, a user provided literal (predicate name and domains for each term) for the desired rule, and a template file. The template files described in Section 2.5.1 are constructed based upon the user's background knowledge about the rule being synthesized. These templates will contain the structure, and occasionally specific clauses, that the user expects the synthesized rule to have. The purpose of these templates is to leverage the user's knowledge to reduce the state space that ALPS searches to find the right rule within a reasonable amount of time. ALPS also asks a series of yes or no questions about what values satisfy the desired rule. After asking a sufficient amount of questions, the user is presented with Datalog rules that encodes the security rule.

For examples, suppose we want to synthesize a rule for identifying LLVM IR memory access or addressing instructions from the LLVM IR in Figure 3.3. According to the LLVM Language Reference [11], the memory access and addressing instructions are: alloca, load, store, fence, cmpxchg, atomicrmw, and getelementptr instructions. Figure 3.3 contains instances of three of these seven instructions.

The user could provide the following literal to ALPS: memory_access_or_addr(O). The user would also provide the following template: A(x) :- B(x). Based upon the input relations from phase two, ALPS will ask a series of questions about values for x that satisfy memory_access_or_addressing(x). If the value for x in question is one of the above instructions, the user answers "yes", otherwise, the user answers "no". After six iterations of questions, ALPS reports that the rule is:

```
memory_access_or_addr(Inst) :- alloca_instruction(Inst).
memory_access_or_addr(Inst) :- load_instruction(Inst).
memory_access_or_addr(Inst) :- store_instruction(Inst).
```

This instruction accurately represents the rule for a memory access or addressing instruction based upon the relations it was trained with. However, the generated rule is not complete. There are additional instructions (fence, cmpxchg, atomicrmw, and getelementptr) that were not represented by the training program. This demonstrates that the "correctness" of a synthesized rule is dependent on the relations it is trained with.

3.2 Applications

To demonstrate our approach to synthesizing security rules, we focused on synthesizing rules for three security properties: (1) out of bounds array accesses, (2) checking the return value of function calls, and (3) double freeing pointers. Our goal is to synthesize these rules and statically check them against a corpus of programs to determine their effectiveness.

The LLVM `mem2reg` optimization is applied to all programs prior to extracting Datalog relations. Applying the `mem2reg` optimization promotes all memory operations to register operations and assures that the program is in SSA form.

3.2.1 Out of bounds array access

Two attributes of a buffer overflow are (1) there is an illegal read or write and (2) that illegal read or write beyond an upper or lower bound [29]. Based on these attributes, we synthesize a rule for an access to an array that exceeds either the upper or lower boundary. Figure 3.4 presents an example program that contains out of bounds array accesses. This program instantiates an integer array, `x` of size `5`. The user provides input, which is assigned to the variable `y`. If `y` is equal to the value `3`, then an integer variable `i` is assigned the value `-1`, otherwise `i` is assigned the value `2`. The program ends by attempting to assign a value to two indices of `x`. These two assignments demonstrate the two ways that an array can be indexed out of bounds. First, `x` is assigned the value `3` at index `i` (line 11). This is potentially an out of bounds array access since the value of `i` is only out of bounds if the user input is `3`. Second, `x` is assigned the value `2` at index `6` (line 12). This is a definite out of bounds access since the index, `6` is larger than the array.

```
 1 int main(int argc, char *argv[]) {
 2     int x[5];
 3     int i,y;

 4     scanf("%d", &y);
 5     if (3 == y) {
 6         i = -1;
 7     }
 8     else {
 9         i = 2;
10     }

11     x[i] = 3;
12     x[6] = 2;

13     return 0;
14 }
```

Figure 3.4: Example: out of bounds array access

3.2.1.1 Synthesized rule

The rule for an out of bounds array access was synthesized from a training program that consisted of various proper and improper array accesses such as the violations in Figure 3.4. These accesses included explicitly indexing the array, similar to line 12 and using an index variable that depends on control flow like line 11. The training program contained example violations within the following scope: (1) all array accesses are on local integer arrays, (2) explicit array accesses that violate either the upper or the lower boundary, (3) variable array accesses that violate either the upper or the lower boundary, (4) out of bounds accesses that occur down various control flow paths, and (5) out of bounds accesses that occur because an index variable whose value depends on control flow paths. The synthesized rule should be able to detect all violations that fall within this scope.

ALPS was run using the following templates:

```
A(X) :- @getelementptr_instruction(X), B(X).
A(V) :- @getelementptr_index(V,W), B(V,X), C(X,Y), D(Y,W).
```

This template was developed based on two key pieces of background knowledge: (1) that an illegal access can exceed the lower boundary (first template) or the upper boundary (second template) and (2) that array accesses are implemented in the LLVM IR via getelementptr instructions.

```
bof(x0:O) :-
    getelementptr_instruction(x0:O),
    negative_index(x0:O).
bof(x0:O) :-
    getelementptr_index(x0:O,x1:N),
    getelementptr_instruction_base(x0:O,x2:O),
    array_size(x2:O,x3:N),
    leq(x3:N,x1:N).
```

Figure 3.5: Synthesized buffer overflow rule

Our technique extracted 462 Datalog facts from the training program. ALPS ran for 13.02 seconds and produced the rule in Figure 3.5. The relations getelementptr_instruction(x0:O) and getelementptr_instruction_base(x0:O, x1:O) are extracted from the input program. They indicate that x0, which falls in the Operand domain, is a getelementptr instruction and that the base address to start indexing from is Operand x1. The leq(x3:N,x1:N) relation is a pre-computed fact that indicates that integer x3 is less than or equal to integer x1. The relations negative_index(x0:O),array_size(x2:O, x3:N) and getelementptr_index(x0:O, x1:N) are user defined rules designed to improve the efficiency of ALPS.

3.2.1.2 User Defined Rules

The relation negative_index(x0:O), presented in Figure 3.6, indicates that the operand, x0, is used as an index for a getelementptr instruction and has a negative value. These rules were synthesized by ALPS. The first rule identifies a situation where the value of the index could be assigned following two branches of an if statement. The second rule identifies a situation where the index is a variable that is assigned a negative value. The third rule identifies a condition where the index is a constant, negative value.

```
 negative_index(x0:O) :-
      getelementptr_instruction_index(x0:O,x1:N,x2:O),
      instruction_uses(x2:O,x3:O),
      negative_constant(x5:O),
      phi_instruction_pair_val(x3:O,x4:N,x5:O).
negative_index(x0:O) :-
    getelementptr_instruction_index(x0:O,x1:N,x2:O),
    instruction_uses(x2:O,x3:O),
    negative_constant(x3:O).
negative_index(x0:O) :-
    getelementptr_instruction_index(x0:O,x1:N,x2:O),
    negative_constant(x2:O).
```

Figure 3.6: Rule for finding a negative index

The relation getelementptr_index(x0:O, x1:N) indicates that a getelementptr instruction x0 from the Operand domain is being referenced at index x1 from the Integer domain. The rule presented in Figure 3.7 was synthesized by ALPS. The first rule identifies getelementptr instructions that have a constant index associated with them. This rule applies to explicit array indices such as line 12 in Figure 3.4. The second rule identifies getelementptr instructions that are indexed at a second instruction that are associated with constant values. This rule applies to variable indices such as line 11 in Figure 3.4. The final rule identifies getelementptr instructions that are indexed at a phi instruction. This rule indicates that an index could be assigned along two branches in a CFG. For example, i is potentially assigned different values in Figure 3.4 depending on the control flow.

```
getelementptr_index(x0:O,x1:N) :-
    getelementptr_instruction_index(x0:O,x3:N,x2:O),
    instruction_uses(x0:O,x2:O),
    constant_value(x2:O,x1:N),
    not_zero(x3:N).
getelementptr_index(x0:O,x1:N) :-
    constant_value(x4:O,x1:N),
    getelementptr_instruction_index(x0:O,x2:N,x3:O),
    instruction_uses(x3:O,x4:O).
getelementptr_index(x0:O,x1:N) :-
    constant_value(x6:O,x1:N),
    getelementptr_instruction_index(x0:O,x2:N,x3:O),
    phi_instruction_pair_val(x4:O,x5:N,x6:O),
    instruction_uses(x3:O,x4:O).
```

Figure 3.7: Rule for finding the integer index of a getelementptr instruction

The relation array_size(x0:O, x1:N), presented in Figure 3.8, indicates that an alloca instruction x0 from the Operand domain is an array with an size of x1 from the Integer domain. The LLVM IR uses a stack based variable system. Any variables in the original source code are allocated space on the stack via an alloca instruction. Intuitively, an array in the LLVM IR is an alloca instruction with a type of array. This rule uses that relationship to find the size associated with the particular array type. This rule is a necessary component of an out of bounds array access because it derives the upper boundary for a particular array.

```
array_size(x0:O,x1:N) :-
    alloca_instruction(x0:O),
    alloca_instruction_type(x0:O,x2:T),
    array_type(x2:T),
    array_type_size(x2:T,x1:N).
```

Figure 3.8: Rule for finding the size of an array

3.2.2 Checking function return values

Verifying the value returned by a function call is a common programming practice. For many function calls, a return value of NULL indicates that an error occurred. If a function returns a pointer and the value of that pointer is not checked, then additional errors, such as NULL pointer dereference may occur. If the result of a function call is stored in a variable and that variable is used later in the program, its value should be verified after the call. Given the importance of checking return values, we applied our technique to synthesize a function return value rule.

Figure 3.9 presents an example program that does not properly validate function return values. This program assigns the return value of a function foo to three variables, x, y, and z. The value of y is compared to 0. The function returns the value z, which is assigned x + y. In this program, there are three calls to the function foo. The function call at line 3 is not a violation because the value of w is not used later in the program. The function call at line 4 is a violation because the value of x is used later in the program, but the value has not been validated. Finally, the function call at line 5 is not a violation because the value of y is used later in the program and it is properly validated.

```
 1 int main(int argc, char *argv[]) {

 2     int w,x,y,z;
 3     w = foo();
 4     x = foo();
 5     y = foo();
 6     if (0 != y) {
 7         return 0;
 8     }

 9     z = x + y;

10     return z;
11 }
```

Figure 3.9: Example: failed return value check

3.2.2.1 Synthesized rule

The rule for verifying that the return value of a function call is checked was synthesized from a training program that consisted of various function calls. The return values of the function calls are either checked or not. The training program contained violations within the following scope: (1) function calls whose return value are used, must be checked down every path of execution and (2) function call does not need to be checked if it is not used. The synthesized rule should be able to detect all violations that fall within this scope.

ALPS was executed using the following templates:

```
A(X) :- B(X).
A(X) :- B(X), C(Y,X), D(Z,Y),
    @icmp_instruction(Y),
    @br_cond_instruction(Z).
A(V) :- B(V), C(W,X), D(X,Y,V), E(Z,W),
    @icmp_instruction(W),
    @br_cond_instruction(Z),
    @phi_instruction(X).
```

This template was developed based on three key pieces of background knowledge: (1) the return value of a function call does not need to be checked if it is not used, (2) if the return value of the call is used, then it must be followed by a comparison whose result is used by a branch instruction, and (3) if the variable that the return value is stored in could have values flowing from two branches of an if statement, then a phi instruction will be involved.

Our technique extracted 465 Datalog facts from the training program. ALPS ran for 10.96 seconds and produced the rule in Figure 3.10. The relations br_cond_instruction(x4:O), icmp_instruction(x1:O),instruction_uses(x1:O, x2:O),phi_instruction(x2:O), and phi_instruction_pair_val(x2:O, x3:N, x0:O) are all extracted from the input program. The semantics of each of these relations is described in Appendix A. The relations call_used(x0:O) and call_not_used(x0:O) are user defined rules designed to improve the efficiency of ALPS.

We will now present the semantics of the second synthesized rule in Figure 3.10. This rule applies to a situation where the value of the function call could flow from two branches of an if statement. The relation call_used(x0:O) indicates that for a function call, x0 to be “good”, it must be used. The relations phi_instruction(x2:O) and phi_instruction_pair_val(x2:O, x3:N, x0:O) indicates that the function call must also be a potential value assigned at a phi instruction, x2. The result of the phi instruction must be used by a comparison instruction, x1, as indicated by the relations icmp_instruction(x1:O) and instruction_uses(x1:O, x2:O). Finally, the relations br_cond_instruction(x4:O) and instruction_uses(x4:O, x1:O) indicate that the result of the comparison, x1, must be used by a conditional branch instruction, x4.

```
good_call(x0:O) :- call_not_used(x0:O).
good_call(x0:O) :-
    br_cond_instruction(x4:O),
    call_used(x0:O),
    icmp_instruction(x1:O),
    instruction_uses(x1:O,x2:O),
    instruction_uses(x4:O,x1:O),
    phi_instruction(x2:O),
    phi_instruction_pair_val(x2:O,x3:N,x0:O).
good_call(x0:O) :-
    br_cond_instruction(x2:O),
    call_instruction(x0:O),
    icmp_instruction(x1:O),
    instruction_uses(x1:O,x0:O),
    instruction_uses(x2:O,x1:O).
```

Figure 3.10: Synthesized function call post-condition rule

3.2.2.2 User Defined Rules

The relation call_used(x0:O), defined in Figure 3.11, indicates that the return value of a call instruction, x0, is used by another instruction at a later point in the program. The relation call_not_used(x0:O) indicates that the return value of a call instruction, x0, is not used by another instruction.

3.2.3 Double free detection

There are a number of API calls that should be called following certain conventions. For example, a call to lock() should be followed by a call to unlock() or a call to close() should not occur without first calling open(). We want to synthesize rules that show a violation of one these conventions. A real world application would be to identify an API violation known as a "double free". A double free is when an allocated memory on the heap has been freed more than once [6]. Double frees lead to memory leaks, which could allow an attacker to write arbitrary values in memory [21]. Figure 3.12 presents an example program that violates this security property (courtesy of the double free CWE [6]). In this program, the pointer allocated at line 10 is freed once

```
call_used(x0:O) :-
    call_instruction(x0:O),
    instruction(x1:O),
    instruction_uses(x1:O,x0:O),
    x0 != x1.

call_not_used(x0:O) :-
    call_instruction(x0:O),
    !call_used(x0:O).
```

Figure 3.11: Rules for identifying used and unused call instructions

on line 12 and again on line 15 without the pointer `buf2R1` being reallocated.

3.2.3.1 Synthesized rule

The rule for a double free was synthesized from a training program that consisted of various double free examples including using an aliased pointer. The training program consisted of violations that occur within the following scope: (1) multiple calls to `free()` within the same function and (2) multiple calls to `free()` on aliased pointers.

`ALPS` was run using the following template:

```
A(V) :- @call_malloc(V), @call_free(W,X),
        B(Y,Z), C(X,V), D(Z,V), E(W,Y).
```

This template was developed based on two key pieces of background knowledge: (1) there must be a call to `malloc()` and (2) there must be a call to `free()`.

Our technique extracted 783 Datalog facts from the training program. `ALPS` ran for 111.25 seconds and produced the rule in Figure 3.13. The relation `call_free(x1:O, x2:O)` indicates that call instruction `x1` is a call to `free()` and that the variable being freed is `x2`. Similarly, the relation `call_malloc(x0:O)` indicates that call instruction `x0` is a call to `malloc()`. Currently, both of these relations need to be manually declared by the user. The relations `alias(x2:O, x0:O)` and `instruction_reaches(x1:O, x3:O)` are user defined rules designed to improve the efficiency of `ALPS`.

```
#include <stdio.h>
#include <unistd.h>
#define BUFSIZE1 512
#define BUFSIZE2 ((BUFSIZE1/2) - 8)

int main(int argc, char **argv) {
    char *buf1R1;
    char *buf2R1;
    char *buf1R2;
    buf1R1 = (char *) malloc(BUFSIZE2);
    buf2R1 = (char *) malloc(BUFSIZE2);
    free(buf1R1);
    free(buf2R1);
    buf1R2 = (char *) malloc(BUFSIZE1);
    strncpy(buf1R2, argv[1], BUFSIZE1-1);
    free(buf2R1);
    free(buf1R2);
}
```

Figure 3.12: Example double free from CWE-415

3.2.3.2 User Defined Rules

The relation `alias(x2:O, x0:O)` indicates that a variable represented by operand `x2` is an alias for the variable represented by operand `x0`. The rule presented in Figure 3.14 was synthesized by `ALPS`. In SSA form, all registers are aliased via either a bitcast or a load instruction. Based on the definition of the LLVM IR pruned SSA form, the synthesized rule for `alias` is sound. This rule will be necessary for dealing with double frees that occur due to freeing a pointer's alias.

The relation `instruction_reaches(x1:O, x2:O)`, define in Figure 3.15, indicates that there is a control flow path from instruction `x1` to instruction `x2`. There are two ways that instruction `x1` can reach `x2`. First, the instructions are in the same basic block and are either sequential or instruction `x1` can reach some other instruction `x3`, which is the instruction before `x2`. Finally, the instructions are in different basic blocks and the basic block of `x1` reaches the basic block of `x2`. This rule is sound because `instruction_next` relations are only extracted from within a basic block.

The relation `basicblock_reaches(x1:O, x2:O)`, defined in Figure 3.16, indicates that basic block `x1` is a predecessor of basic block `x2`. For basic block `x1` to reach basic block `x2`, `x1` must either be an immediate predecessor or `x2`, or reach the immediate predecessor of `x2`. These rules provide the notion of control flow that is necessary for synthesizing a double

```
double_free(x0:O) :-
    call_malloc(x0:O),
    call_free(x1:O,x2:O),
    call_free(x3:O,x4:O),
    instruction_reaches(x1:O,x3:O),
    alias(x2:O,x0:O),
    alias(x4:O,x0:O).
```

Figure 3.13: Synthesized double free rule

```
alias(x0:O,x1:O) :- bitcast_instruction_from(x0:O,x1:O).
alias(x0:O,x1:O) :- load_instruction_address(x0:O,x1:O).
alias(x0:O,x1:O) :- alias(x0:O,x2:O),
    bitcast_instruction_from(x2:O,x1:O).
alias(x0:O,x1:O) :- alias(x0:O,x2:O),
    load_instruction_address(x2:O,x1:O).
```

Figure 3.14: Aliased instruction rule

free rule.

3.3 Evaluation

3.3.1 Methods

To evaluate the synthesized rules, we used python scripts to randomly generate C programs of varied complexity. Each of the programs randomly contain different ways that the synthesized rules can be violated.

To evaluate the rule for out of bounds array accesses, the script generates C programs having a number of characteristics. These characteristics include: (1) an integer array of a random size, (2) a random number of index variables assigned random values ranging from -10 to 1.5 times the array size, (3) varied control flow, (4) random reassignment of index variables, and (5) random array accesses at either an explicit index or one of the index variables. If there is an array access, either explicit or variable, that is negative or is greater than or equal to the size of the array, there is a violation. The generated programs cover the types of violations that fall within the scope that the

```
instruction_reaches(x1:O,x2:O) :- instruction_next(x1:O,x2:O).
instruction_reaches(x1:O,x2:O) :- instruction_reaches(x1:O,x3:O),
   instruction_next(x3:O,x2:O).
instruction_reaches(x1:O,x2:O) :-
    instruction_basicblock(x1:O,b1:B),
    instruction_basicblock(x2:O,b2:B),
    basicblock_reaches(b1:B,b2:B),
    b1 != b2.
```

Figure 3.15: Instruction reachability rule

```
basicblock_reaches(b1:B,b2:B) :- basicblock_pred(b2:B,b1:B).
basicblock_reaches(b1:B,b2:B) :-
    basicblock_pred(b3:B,b1:B),
    basicblock_reaches(b3:B,b2:B).
```

Figure 3.16: Basic block reachability rule

rule was trained.

To evaluate the rule for checking the return value of a function, the script generates C programs having a number of characteristics. These characteristics include: (1) a function, `foo()`, that accepts an argument `x` and returns `x+3`, (2) a set of random variables, initialized to zero, (3) varied control flow, (4) function calls to `foo()` that are either not assigned to a variable, assigned to a variable and checked, assigned to a variable that will not be used, or assigned to a variable and not checked, and (5) returns a summation of all the variables that will need to be used. If the result of a function call is used, but the value has not been checked prior to the use, there is a violation. The generated programs cover the types of violations that fall within the scope that the rule was trained.

To evaluate the rule for identifying a double free, the script generates C programs having a number of characteristics. These characteristics include: (1) assigns the result of a call to `malloc()` to a pointer `x`, (2) a set of random variables that alias to `x`, (3) varied control flow to include if/else statements and goto statements, and (4) random calls to `free()` on `x` or one of its aliases. If there are multiple calls to `free()` on any given path, there is a violation. The generated programs cover the types of violations that fall within the scope that the rule was trained.

The randomly generated test cases are designed to cover all occurrences of the violations within the described scope. However, it is important to note that the resulting programs are simple. Real world code is complex and it is possible that there are scenarios that occur in complex code that the scripts fail to model.

3.3.2 Discussion

3.3.2.1 Out of bounds array access

The synthesized rule in Figure 3.5 is reflective of the conditions for an out of bounds array access. The first rule describes the former condition where if a `getelementptr` instruction is being indexed at a negative value, the boundary has been violated. The second rule describes the latter condition where if a `getelementptr` instruction references an array and the array size is less than or equal to the index, then the boundary has been violated.

Appendix B presents a set of C programs that were developed to test various features of the rule. The programs are designed to test if the rule is able to detect both explicit and variable array accesses that violate either the upper or lower boundary of the array. The programs also test the rule's ability to detect violations that are dependent on control flow. To test the soundness of the rule, it was applied to 1000 randomly generated C programs of varied control flow and array access complexity as described in Section 3.3.1. The test programs varied in length from 12 to 286 lines of code. The test corpus contained a total of 480 violations. The synthesized rule successfully reported all 480 violations and did not report any false positives. It successfully reported all violations and did not report any false positives.

The synthesized rule has some limitations. There are a number of other array access/buffer overflow attributes outlined in [29] that this rule is unable to handle such as arrays that are referenced from containers such as structs, out of bounds accesses in loops, and out of bounds accesses to global arrays. However, it could be possible to synthesize a rule that accounts for these limitations by synthesizing a more complex rules for connecting a `getelementptr` instruction to the allocated array and for connecting an array to its potential index values.

This rule is unable to detect illegal accesses due to more complex indices. For example, the rule is unable to detect violations that occur by adding or subtracting a constant number to/from an index variable. However, this limitation can be addressed by first running compiler optimizations, such as constant propagation, prior to the analysis.

Additionally, this rule is unable to account for array accesses that occur due to runtime values. This limitation could be overcome by first running taint analysis on the code. If the index variable is tainted and the value is unchecked, then there could be a potential out of bounds access.

Finally, the random testing reported a false positive rate of 0%, but this number could potentially be higher. The randomly generated programs could contain invalid array accesses that are unreachable. Based upon the implementation of the random program generator, these accesses are marked as violations when they are generated. For example, the array access in the following code snippet would be reported as a violation, when in fact, it should not be since the statement is logically unreachable. This limitation could be overcome by first optimizing the program to eliminate dead, or unreachable statements in the code.

```
...
if (y > 0) {
    ...
    if (y < 0) {
        x[-1] = 2;
    }
    ...
}
...
```

3.3.2.2 Checking function return values

A function call is considered "good" in the context of this work if its return value is not used, or if it is used that its value is verified via a comparison. The synthesized rule follows this definition. The first rule accounts for function calls whose result is not used. The second and third rules account for function calls whose return values are used. Both rules are satisfied by function calls that are used by compare instructions and followed by a conditional branch, but the difference between the two is that the second rule accounts for a variable having multiple potential values.

Appendix C presents a set of C programs that were developed to test various features of the rule. The programs are designed to test if the rule is able to detect function calls that violate the post condition: return values that are assigned to variables that are used in the future must be validated. The programs are written to contain function calls that are either not used or properly validated, are used but not properly validated, as well as test the rule's ability to detect violations that are dependent on control flow. To evaluate these test cases, a new rule was developed:

```
bad_call(x) :- call_instruction(x), !good_call(x).
```

This rule indicates that a "bad" call is any call instruction that does not satisfy the `good_call` relation. To test the soundness of the rule, it was applied to randomly 1000 generated C programs containing varied control flow and function calls as described in Section 3.3.1. The test programs varied in length from 18 to 378 lines of code. The test corpus contained a total of 894 violations. The synthesized rule successfully reported 720 violations. The rule did not report any false positives, but failed to report 174 violations, yielding a false negative rate of 19.5%.

Inspecting the false negatives identified that the rule is not path sensitive. The example program in Figure 3.17 assigns the value of a call to `foo()` to a variable `x`. After the function call, the user inputs a value that is assigned to the variable `y`. If `y` is equal to zero, the value of `x` is validated and passed to a function, `bar()`. If `y` is not equal to zero, the value of `x` is passed to `bar()` without any validation. The synthesized rule detects `x` being validated down one branch of the if statement and reports that the post condition has been satisfied when there is a path that uses the variable without checking it.

```c
int main(int argc, char *argv[]) {
    int x,y;
    scanf("%d", &y);
    x = foo();
    if (0 == y) {
        if (x == 0) {
            return 1;
        }
        bar(x);
    else {
        bar(x);
    }

    return 0;
}
```

Figure 3.17: Path insensitive function call check

3.3.2.3 Double free detection

The definition of a double free is when `free()` is called on the same memory address twice [6]. The synthesized rule follows this definition. The rule declares that there has been allocated memory, two calls to `free()`, and the operands being passed to free are the same.

Appendix D presents a set of C programs that were contrived to test various features of the rule. The programs are designed to test if the rule is able to detect allocated memory addresses that are freed more than once. The programs also test the rule's ability to detect violations that are dependent on control flow and pointer aliases. To test the soundness of the rule, it was applied to 1000 randomly generated C programs that contained varied control flow and calls to `free()` on a pointer or its aliases as described in Section 3.3.1. The test programs varied in length from 12 to 2044 lines of code. The test corpus contained a total of 356 violations. The synthesized rule successfully reported all 356 violations and did not report any false positives.

The rule has a number of limitations. First, most double frees in the "wild" will occur interprocedurally. The current rule is not able to detect interprocedural double frees. This is because the current implementation of the LLVM pass does not extract relations for interprocedural control flow. Currently, each function produces its own independent control flow graph. I believe that this rule could be expanded to interprocedural analyses by creating edges from a call instruction to the entry of the callee and from the callee's exit to the next instruction in the caller. Additionally, pointer aliasing would have to be improved. Returning to the example in Listing D.8, we would have to develop logic for knowing that `ptr` in function `foo()` is equivalent to `x` for the call `foo(x)`. If the test for Listing D.8 is repeated and the LLVM optimization `inline` is applied to the source, then the current rule is able to detect the double free. This implies that if the control flow graph and aliasing were implemented, the rule would be able to report interprocedural violations such as Listing D.8.

Second, this rule is not able to identify global pointers being double freed. This limitation could be addressed by extending the `alias` rule to account for aliasing between temporary registers and global variables.

Third, while the random testing reported that the rule did not yield any false positives, this rate may be deflated. The randomly generated programs could contain calls to `free()` that are unreachable. As discussed in the previous rules, this limitation can be addrssed by optimizing the program and eliminating unreachable instructions.

Chapter 4

Conclusion

Program analysis tools are not able to detect security violations if they do not know what to look for. Currently, specifications for security properties are manually written, which is error prone and time consuming. In this research, we have presented a novel framework for interactively learning Datalog rules for representing security properties.

Using this framework, we successfully synthesized rules for three, intraprocedural security properties. These rules were tested for their ability to detect violations that fall within the scope for which they were trained. We learned a rule for detecting an out of bounds array access. The rule detects violations, within a function, with no false negatives or false positives. We learned a rule for detecting if the return value of a function call is properly checked. The rule detects violations with a 20% false negative rate because the rule is not flow-sensitive. Finally, we learned a rule for detecting when allocated memory has been freed twice. The rule detects violations on local pointers with no false negatives or false positives. These results indicate that synthesizing logic programming rules for security properties is both feasible and practical.

4.1 Future Work

The novel work presented in this thesis provides many possibilities for future research.

Interprocedural rule synthesis In this work, we focused on an intraprocedural programs to synthesize rules. However, to handle real-world programs requires interprocedural analyses. We intend to expand this work by exploring more complex representations of a program such as interprocedural control flow graphs and program dependence graphs. This will allow us to synthesize more complex, and more realistic, security properties. Furthermore, we intend to expand the scope of the existing security properties, such as detecting illegal array accesses in structs or global arrays, as well as synthesize rules for additional security properties.

Verification and validation of autonomous systems This work has positive implications for the feasibility of synthesizing rules for safety and security properties of autonomous systems or and IoT devices. These systems often interact with the physical world and synthesizing rules to ensure that programs being developed follow specific procedures or do not violate safety constraints will result in more secure and safe systems.

Extending the relational translator The current implementation of the relational translator only accounts for the LLVM instructions and features that apply to C programs. To make this a more practical system, we would need to expand the relational translator to account for other instructions, such as exception handling, that apply to other languages such as C++. This would lead into integrating the relational translator as a front end for a Datalog driven analysis framework such as Petablox. This will allow for the learned rules as well as existing analyses to be run on C/C++ programs.

Bibliography

[1] Available checkers. https://clang-analyzer.llvm.org/available_checks.html.

[2] cclyzer: A tool for analyzing llvm bitcode using datalog. https://github.com/plast-lab/cclyzer.

[3] CERT secure coding standards. https://www.securecoding.cert.org.

[4] Clang static analyzer. https://clang-analyzer.llvm.org.

[5] CVE-2014-0160. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160.

[6] CWE-415: Double free. https://cwe.mitre.org/data/definitions/415.html.

[7] Doop: Framework for java pointer analysis. http://doop.program-analysis.org.

[8] The heartbleed bug. https://www.heartbleed.org.

[9] The LLVM compiler infrastructure project. https://llvm.org.

[10] LLVM documentation. https://llvm.org/docs/doxygen/html.

[11] LLVM language reference manual. http://llvm.org/docs/LangRef.html.

[12] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques*. Addison wesley Boston, 1986.

[13] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. *ACM SIGPLAN Notices*, 44(10):243–262, 2009.

[14] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

[15] C. Cadar, D. Dunbar, D. R. Engler, et al. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[16] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[17] R.-Y. Chang, A. Podgurski, and J. Yang. Finding what's not there: a new approach to revealing neglected conditions in software. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 163–173. ACM, 2007.

[18] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 235–244. ACM, 2002.

[19] J. Davis. Hacking of government computers exposed 21.5 million people. `http://www.nytimes.com/2015/07/10/us/office-of-personnel-management-hackers-got-data-of-millions.html`.

[20] S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems—a case study. In *ACM SIGPLAN Notices*, volume 31, pages 117–126. ACM, 1996.

[21] I. Dobrovitski. Exploit for CVS double free () for linux pserver, 2003.

[22] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini. Defining and continuous checking of structural program dependencies. In *Proceedings of the 30th international conference on Software engineering*, pages 391–400. ACM, 2008.

[23] S. Gulwani, J. Hernandez-Orallo, E. Kitzelmann, S. H. Muggleton, U. Schmid, and B. Zorn. Inductive programming meets the real world. *Communications of the ACM*, 58(11):90–99, 2015.

[24] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *ACM SIGPLAN Notices*, volume 37, pages 69–82. ACM, 2002.

[25] S. Heelan. Vulnerability detection systems: Think cyborg, not robot. *IEEE Security & Privacy*, 9(3):74–77, 2011.

[26] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive c and c++ memory leak detector. In *ACM SIGPLAN Notices*, volume 38, pages 168–181. ACM, 2003.

[27] S. S. Huang, T. J. Green, and B. T. Loo. Datalog and emerging applications: an interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1213–1216. ACM, 2011.

[28] S. C. Johnson. *Lint, a C program checker*. Bell Telephone Laboratories Murray Hill, 1977.

[29] K. J. Kratkiewicz. Evaluating static analysis tools for detecting buffer overflows in c code. Technical report, DTIC Document, 2005.

[30] C. Kreitz. Program synthesis. In *Automated Deduction—A Basis for Applications*, pages 105–134. Springer, 1998.

[31] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 161–176. USENIX Association, 2006.

[32] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12. ACM, 2005.

[33] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[34] J. W. Lloyd. *Foundations of logic programming*. Springer Science & Business Media, 2012.

[35] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[36] R. Mangal, X. Zhang, A. V. Nori, and M. Naik. A user-guided approach to program analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 462–473. ACM, 2015.

[37] S. H. Muggleton, D. Lin, and A. Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. *Machine Learning*, 100(1):49–73, 2015.

[38] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[39] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[40] E. I. Psallida and G. Balatsouras. *Relational representation of the LLVM intermediate language*. PhD thesis, BS Thesis, University of Athens, 2014.

[41] J. R. Quinlan. Learning logical definitions from relations. *Machine learning*, 5(3):239–266, 1990.

[42] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[43] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27. ACM, 1988.

[44] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security Symposium*, pages 201–220, 2001.

[45] X. Si, W. Lee, A. Albarghouthi, P. Koutris, and M. Naik. Interactive synthesis of declarative programs. 2017.

[46] K. Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.

[47] G. Wallace. Target credit card hack: What you need to know. `http://money.cnn.com/2013/12/22/news/companies/target-credit-card-hack/`.

[48] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Security and privacy (SP), 2010 IEEE symposium on*, pages 497–512. IEEE, 2010.

[49] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using datalog with binary decision diagrams for program analysis. In *Asian Symposium on Programming Languages and Systems*, pages 97–118. Springer, 2005.

[50] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *ACM SIGPLAN Notices*, volume 39, pages 131–144. ACM, 2004.

[51] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. *Acm Sigplan Notices*, 40(1):351–363, 2005.

[52] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 590–604. IEEE, 2014.

[53] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 499–510. ACM, 2013.

[54] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik. APISan: Sanitizing API usages through semantic cross-checking. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 363–378. USENIX Association.

[55] M. Zalewski. American fuzzy lop, 2007.

Appendix A

LLVM Relational Representation

This document presents the schema used for representing a program written in the LLVM IR as a set of datalog relations. This schema is based on the `cclyzer` [2] implementation described in [40].

A.1 Control Flow

This section discusses how to represent the control flow of an LLVM program using a relational schema. Control flow is represented as a series of basic blocks. Within each basic block is a series of instructions. The following relations are used to represent the control flow:

- A basic block with an `id` has an entry instruction, `inst`: `basicblock_entry(id, inst).`
- A basic block with an `id` has an exit instruction, `inst`: `basicblock_exit(id, inst).`
- A basic block with an `id` has a predecessor block, `pred`: `basicblock_pred(id, pred).`
- An instruction, `inst`, has a basic block, `bb`: `instruction_basicblock(inst, bb).`
- An instruction `inst` has a sucessor, `next`: `instruction_next(inst, next).`

A.2 Types

This section outlines the relational schema for representing the LLVM IR type system.

- Void Type

  The Void Type has no value. It is represented in the relational schema as the string: `void`.

- Function Type

  The Function Type is analogous to a function signature. The relational schema for a Function Type contains the following information:

  - A type with `id` is a Function Type:

    ```
    fn_type(id).
    ```
  - A Function Type can accept variable arguments:

    ```
    fn_type_varargs(id).
    ```
  - A Function Type has a return type of `ty`:

    ```
    fn_type_return(id, ty).
    ```
  - A Function Type accepts `n` parameters:

    ```
    fn_type_nparams(id, n).
    ```
  - The `i`th parameter of a Function Type has a type of `ty`:

    ```
    fn_type_param(id, i, ty).
    ```

- Integer Type

  The Integer Type is represented in the relational schema as `iN` where `N` is the width of the integer, in bits.

- Floating Point Type

  The Floating Point Type is represented in the relational schema as one of the following strings:

  - `half` - a 16-bit floating point value
  - `float` - a 32-bit floating point value
  - `double` - a 64-bit floating point value
  - `fp128` - 128-bit floating point value (112-bit mantissa)
  - `x86_fp80` - 80-bit floating point value (X87)
  - `ppc_fp128` - 128-bit floating point value (two 64-bits)

- Pointer Type

  The Pointer Type is used to specify a memory location. The relational schema for a Pointer Type contains the following information:

  - A type with `id` is a Pointer Type:

  `pointer_type(id).`

  - A Pointer Type has a component with type `ty`:

    `pointer_type_component(id, ty).`

  - A Pointer Type can be in a specific address space, `addr`:

    `pointer_type_addr_space(id, addr).`

- Vector Type

  The relational schema for a Vector Type contains the following information:

  - A type with `id` is a Vector Type:

    `vector_type(id).`

  - A Vector Type contains `n` components:

    `vector_type_size(id, n).`

  - A Vector Type contains components of type `ty`:

    `vector_type_component(id, ty).`

- Label Type

  The Label Type is represented by the following relation: `label(id).`

- Array Type

  The relational schema for an Array Type contains the following information:

  - A type with `id` is an Array Type:

    `array_type(id).`

  - An Array Type contains `n` components:

    `array_type_size(id, n).`

  - An Array Type contains components of type `ty`:

    `array_type_component(id, ty).`

- Structure Type

  The Structure Type represents a collection of data members together in memory. The relational schema for a Structure Type contains the following information:

  - A type with `id` is a Structure Type:

    `struct_type(id).`

  - A Structure Type may have a name:

    `struct_type(id, name).`

  - A Structure Type may be opaque:

    `opaque_struct_type(id).`

  - A Structure Type has `n` fields:

    `struct_type_nfields(id, n).`

  - The `i`th field of a Structure Type has type `ty`:

    `struct_type_field(id, i, ty).`

A.3 Global Variables

This section outlines the relational schema for representing and LLVM IR Global Variable.

The relational schema for a Global Variable contains the following information:

- A variable `id` is a Global Variable:

  `global_variable(id).`

- The variable has a type, `ty`:

  `global_variable_type(id, ty).`

- The variable has a name:

  `global_variable_type(id, name).`

- The variable has an alignment of `n`:

  `global_variable_alignment(id, n).`

- The linkage type of the variable,`link`:

  `global_variable_linkage_type(id, link).`

- The visibility of the variable, `vis`:

  `global_variable_visibility(id, vis).`

- The variable may be initialized to some `val`:

  `global_variable_initializer(id, val).`

- The section, `sect`, that the variable is stored in:

  `global_variable_section(id, sect).`

- The thread-local mode of the variable, `mode`:

  `global_threadlocal_mode(id, mode).`

- A Global Variable may be declared constant:

  `global_variable_constant(id).`

A.4 Aliases

This section outlines the relational schema for an LLVM IR global alias.

The relational schema for an Alias contains the following information:

- A global value `id` is an Alias:

  `alias(id).`
- An Alias has a type, `ty`:

  `alias_type(id, ty).`
- An Alias has a name:

  `alias_name(id, name).`
- The linkage type of the Alias, `link`:

  `alias_linkage_type(id, link).`
- The visibility of the Alias, `vis`:

  `alias_visibility(id, vis).`
- An Alias, `id` has an aliasee, `aliasee`:

  `aliase_aliasee(id, aliasee).`

A.5 Functions

This section outlines the relational schema for representing an LLVM IR Function.

The relational schema for a Function contains the following information:

- The Function can be define as `id`:

  `function(id).`
- The Function has a type (signature), `ty`:

  `function_type(id, ty).`
- The name of the Function: `function_name(id, name).`
- The Function linkage type, `link`:

  `function_linkage_type(id, link).`

- The Function visibility, `vis`:

  ```
  function_visibility(id, vis).
  ```
- A calling convention, `conv`:

  ```
  function_calling_convention(id, conv).
  ```
- If the Function has an unnamed address:

  ```
  function_unnamed_addr(id).
  ```
- The Function has an alignment of `n`:

  ```
  function_alignment(id, n).
  ```
- If applicable, the garbage collector:

  ```
  function_gc(id, gc).
  ```
- The Function may have a personality function, `fn`:

  ```
  function_pers_fn(id, fn).
  ```
- For each function attribute,`attr`:

  ```
  function_attribute(id, attr).
  ```
- For each return attribute, `attr`:

  ```
  function_return_attribute(id, attr).
  ```
- The Function may be in custom section, `sect`:

  ```
  function_section(id, sect).
  ```
- The `i`th parameter of the Function has a `param_id`:

  ```
  function_param(id, i, param_id).
  ```
- For each parameter attribute of the `i`th parameter, `attr`:

  ```
  function_param_attr(id, i, attr).
  ```
- The Function has `n` parameters:

  ```
  function_nparams(id, n).
  ```

A.6 Instructions

This section outlines the relational schema for representing the various LLVM IR Instructions. Instructions take in operands that are either variables, constants, or other instructions. To represent this relationship, the relation `instruction_uses(id, oper)` is produced for each operand, `oper` for instruction `id`. If the operand is a constant with a value of `val`, the relations `constant(id)` and `constant_value(id, val)` are produced. If the value of the constant is negative, the relations `negative_constant(id)` and `negative_constant_value(id, -val)` are produced.

A.6.1 Terminator Instructions

A basic block must end with a Terminator Instruction. This sub-section outlines the schema for representing terminator instructions.

- `ret`

  A `ret` instruction with an `id` will produce the following relations:

  - `id` is a ret instruction:

    `ret_instruction(id).`
  - If the return type is `void`:

    `ret_instruction_void(id).`
  - Otherwise, the instruction returns `val`:

    `ret_instruction(id, val).`

- `br`

  A `br` instruction with an `id` will produce the following relations:

  - `id` is a `br` instruction:

    `br_instruction(id).`

  If the instruction is a conditional branch:

  - `id` is a conditional branch:

    `br_cond_instruction(id).`
  - The instruction has a condition, `cond`:

    `br_cond_instruction_condition(id, cond).`
  - If the condition is true, go to `true_label`:

    `br_cond_instruction_iftrue(id, true_label).`
  - If the condition is false, go to `false_label`:

    `br_cond_instruction_iffalse(id, false_label).`

  If the instruction is an unconditional branch:

  - id is an unconditional branch:

    `br_uncond_instruction(id).`

  - The branch goes to label:

    `br_uncond_dest(id, label).`

- switch

  A switch instruction with an id will produce the following relations:

  - id is a switch instruction:

    `switch_instruction(id).`

  - The instruction has a condition, cond:

    `switch_instruction(id, cond).`

  - The instruction has a default label:

    `switch_instruction_default(id, label).`

  - The instruction has n cases:

    `switch_instruction_ncases(id, n).`

  - The ith case has a value, val:

    `switch_instruction_case_value(id, i, val).`

  - The ith case has a destination label:

    `switch_instruction_case_label(id, i, label).`

- indirectbr

  An indirectbr instruction with an id will produce the following relations:

  - id is an indirectbr instruction:

    `indirectbr_instruction(id).`

  - The instruction jumps to address:

    `indirectbr_instruction_address(id, addr).`

  - The instruction has n possible labels:

    `indirectbr_instruction_nlabels(id, n).`

  - The ith destination goes to label:

    `indirectbr_instruction_label(id, i, label).`

- unreachable

An `unreachable` instruction with an `id` will produce the following relations:

- `id` is a unreachable instruction:

  `unreachable_instruction(id).`

A.6.2 Binary Operation Instructions

This sub-section outlines the schema for representing binary operation instructions. This sub-section will account for both Binary Operations and Bitwise Binary Operations.

All binary operation instructions with an `id` and an operation, `op` will produce the following relations:

- `id` is an `op` instruction:

  `op_instruction(id).`
- The instruction has a first operand, `oper`:

  `op_instruction_first_operand(id, oper).`
- The instruction has a second operand, `oper`:

  `op_instruction_second_operand(id, oper).`

For example, if the operation is `add`, the following relations would be produced:

- `add_instruction(id).`
- `add_instruction_first_opearnd(id, oper).`
- `add_instruction_second_opearnd(id, oper).`

The valid operations are:

```
 add   fadd   sub   fsub
 mul   fmul  udiv   sdiv
fdiv   urem  srem   frem
 shl   lshr  ashr    and
 or     xor
```

A.6.3 Vector Operation Instructions

This sub-section outlines the schema for representing vector operation instructions.

- `extractelement`

An `extractelement` instruction with an `id` will produce the following relations:

  - `id` is an extractelement instruction:

    `extractelement_instruction(id).`

  - The instruction is extracting an element from `vector`:

    `extractelement_instruction_base(id, vector).`

  - The instruction extracts from index, `i`:

    `extractelement_instruction_index(id, i).`

- `insertelement`

  An `insertelement` instruction with an `id` will produce the following relations:

  - `id` is an insertelement instruction:

    `insertelement_instruction(id).`

  - The instruction is inserting an element into `vector`:

    `insertelement_instruction_base(id, vector).`

  - The instruction is inserting the value `val`:

    `insertelement_instruction_value(id, val).`

  - The instruction inserts at index, `i`:

    `insertelement_instruction_index(id, i).`

- `shufflevector`

  A `shufflevector` instruction with an `id` will produce the following relations:

  - `id` is a shufflevector instruction:

    `shufflevector_instruction(id).`

  - The instruction shuffles `vec1`:

    `shufflevector_instruction_first_vector(id, vec1).`

  - with `vec2`:

    `shufflevector_instruction_second_vector(id, vec2).`

  - The instruction accepts a vector mask, `mask`:

    `shufflevector_instruction_mask(id, mask).`

A.6.4 Aggregate Operation Instructions

This sub-section outlines the schema for representing aggregate operation instructions.

- extractvalue

  An extractvalue instruction with an id will produce the following relations:

  - id is an extractvalue instruction:

    ```
    extractvalue_instruction(id).
    ```
  - The instruction is extracting a value from agg:

    ```
    extractvalue_instruction_base(id, agg).
    ```
  - The instruction accepts n indices:

    ```
    extractvalue_instruction_nindices(id, n).
    ```
  - The ith index has a value of val:

    ```
    extractvalue_instruction_index(id, i, val).
    ```

- insertvalue

  An insertvalue instruction with an id will produce the following relations:

  - id is an insertvalue instruction:

    ```
    insertvalue_instruction(id).
    ```
  - The instruction is inserting a value into agg:

    ```
    insertvalue_instruction_base(id, agg).
    ```
  - The instruction is inserting val:

    ```
    insertvalue_instruction_value(id, val).
    ```
  - The instruction accepts n indices:

    ```
    insertvalue_instruction_nindices(id, n).
    ```
  - The ith index has a value of val:

    ```
    insertvalue_instruction_index(id, i, val).
    ```

A.6.5 Memory Operation Instructions

This sub-section outlines the schema for representing memory access and addressing operations.

- alloca

  An alloca instruction with an id will produce the following relations:

  - `id` is an alloca instruction:

    `alloca_instruction(id).`

  - The instruction has an alignment of `n`:

    `alloca_instruction_alignment(id, n).`

  - The instruction allocates a space of size `n`:

    `alloca_instruction_size(id, n).`

  - The instruction allocates space for type `ty`:

    `alloca_instruction_type(id, ty).`

- `load`

  A `load` instruction with an `id` will produce the following relations:

  - `id` is a load instruction:

    `load_instruction(id).`

  - The instruction has an alignment of `n`:

    `load_instruction_alignment(id, n).`

  - The instruction has an ordering of `ord`:

    `load_instruction_ordering(id, ord).`

  - If the instruction is volatile:

    `load_instruction_volatile(id).`

  - The instruction loads from `addr`:

    `load_instruction_address(id, addr).`

- `store`

  A `store` instruction with an `id` will produce the following relations:

  - `id` is a store instruction:

    `store_instruction(id).`

  - The instruction has an alignment of `n`:

    `store_instruction_alignment(id, n).`

  - The instruction has an ordering of `ord`:

    `store_instruction_ordering(id, ord).`

  - If the instruction is volatile:

    `store_instruction_volatile(id).`

  - The instruction stores the value `val`:

    `store_instruction_value(id, val).`

  - The instruction stores to `addr`:

    `store_instruction_address(id, addr).`

- `fence`

  A `fence` instruction with an `id` will produce the following relations:

  - `id` is a fence instruction:

    `fence_instruction(id).`

  - The instruction has an ordering of `ord`:

    `fence_instruction_ordering(id, ord).`

- `cmpxchg`

  A `cmpxchg` instruction with an `id` will produce the following relations:

  - `id` is a cmpxchg instruction:

    `cmpxchg_instruction(id).`

  - The instruction has a sucess ordering of `succ_ord`:

    `cmpxchg_instruction_success_ordering(id, succ_ord).`

  - The instruction has a failure ordering of `fail_ord`:

    `cmpxchg_instruction_failure_ordering(id, fail_ord).`

  - If the instruction is volatile:

    `cmpxchg_instruction_volatile(id).`

  - The instruction modifies memory at `addr`:

    `cmpxchg_instruction_address(id, addr).`

  - The instruction compares the value at `addr` to `cmp`:

    `cmpxchg_instruction_cmp(id, cmp).`

  - If the comparison is equal, instruction writes `new` to

    `addr`: `cmpxchg_instruction_new(id, new).`

- `atomicrmw`

  An `atomicrmw` instruction with an `id` will produce the following relations:

- `id` is an atomicrmw instruction:

  ```
  atomicrmw_instruction(id).
  ```

- The instruction has an ordering of `ord`:

  ```
  atomicrmw_instruction_ordering(id, ord).
  ```

- If the instruction is volatile:

  ```
  atomicrmw_instruction_volatile(id).
  ```

- The isntruction applies the operation `op`:

  ```
  atomicrmw_instruction_opeartion(id, op).
  ```

- The operand to the operation is `val`:

  ```
  atomicrmw_instruction_value(id, val).
  ```

- The instruction has an `addr` to modify:

  ```
  atomicrmw_instruction_address(id, addr).
  ```

- `getelementptr`

  A `getelementptr` instruction with an `id` will produce the following relations:

  - `id` is a getelementptr instruction:

    ```
    getelementptr_instruction(id).
    ```

  - If the instruction is declared in bounds:

    ```
    getelementptr_instruction_inbounds(id).
    ```

  - The instruction refers to an element with base `addr`:

    ```
    getelementptr_instruction_base(id, addr).
    ```

  - The instruction has `n` indices:

    ```
    getelementptr_instruction_nindices(id, n).
    ```

  - The `i`th instruction has a value of `val`:

    ```
    getelementptr_instruction_index(id, i, val).
    ```

A.6.6 Conversion Operation Instructions

This sub-section outlines the schema for representing conversion operations.

All conversion operation instructions with an `id` and an operation, `op` will produce the following relations:

- `id` is an `op` instruction:

```
op_instruction(id).
```

- The instruction converts value `val`:

  ```
  op_instruction_from(id, val).
  ```

- The instruction has a from type of `ty`:

  ```
  op_instruction_from_type(id, ty).
  ```

- The instruction has a to type of `ty`:

  ```
  op_instruction_to_type(id, ty).
  ```

For example, if the operation is `trunc`, the following relations would be produced:

- `trunc_instruction(id).`
- `trunc_instruction_from(id, val).`
- `trunc_instruction_from_type(id, i32).`
- `trunc_instruction_to_type(id, i16).`

The valid operations are:

```
   trunc       zext      sext    fptrunc
   fpext      fptoui    fptosi   uitofp
   sitofp    ptrtoint  inttoptr  bitcast
 addrspacae
```

A.6.7 Other Operation Instructions

This sub-section outlines the relational schema for instructions that do not fall into other categories.

- `icmp`

  An `icmp` instruction with an `id` will produce the following relations:

  - `id` is an icmp instruction:

    ```
    icmp_instruction(id).
    ```

  - The instruction has the condition `cond`:

    ```
    icmp_instruction_condition(id, cond).
    ```

  - The instruction has a first operand, `oper`:

    ```
    icmp_instruction_first_operand(id, oper).
    ```

  - The instruction has a second operand, `oper`:

    ```
    icmp_instruction_second_operand(id, oper).
    ```

- `fcmp`

  An `fcmp` instruction with an `id` will produce the following relations:

  - `id` is an fcmp instruction:

    ```
    fcmp_instruction(id).
    ```

  - The instruction has the condition `cond`:

    ```
    fcmp_instruction_condition(id, cond).
    ```

  - The instruction has a first operand, `oper`:

    ```
    fcmp_instruction_first_operand(id, oper).
    ```

  - The instruction has a second operand, `oper`:

    ```
    fcmp_instruction_second_operand(id, oper).
    ```

- `phi`

  A `phi` instruction with an `id` will produce the following relations:

  - `id` is a phi instruction:

    ```
    phi_instruction(id).
    ```

  - The instruction has a type, `ty`:

    ```
    phi_instruction_type(id, ty).
    ```

  - The instruction has `n` pairs:

    ```
    phi_instruction_npairs(id, n).
    ```

  - The `i`th pair has a value of `val`:

    ```
    phi_instruction_pair_val(id, i, val).
    ```

  - The `i`th pair has a label of `label`:

    ```
    phi_instruction_pair_label(id, i, label).
    ```

- `select`

  A `select` instruction with an `id` will produce the following relations:

  - `id` is a select instruction:

    ```
    select_instruction(id).
    ```

  - The instruction selects based on condition, `cond`:

    `select_instruction_condition(id, cond).`

  - If the condition is true, select the `true_val`:

    `select_instruction_true(id, true_val).`

  - If the condition is false, select the `false_val`:

    `select_instruction_false(id, false_val).`

- `call`

  A `call` instruction with an `id` will produce the following relations:

  - `id` is a call instruction:

    `call_instruction(id).`

  - The function, `func`, being called:

    `call_instruction_function(id, func).`

  - If tail call optimizations can be used:

    `call_instruction_tail(id).`

  - The function return type, `ty`:

    `call_instruction_return_type(id, ty).`

  - For each function attribute, `attr`:

    `call_instruction_fn_attribute(id, attr).`

  - For each return attribute, `attr`:

    `call_instruction_return_attribute(id, attr).`

  - The `i`th argument, `arg`:

    `call_instruction_arg(id, i, arg).`

  - For each parameter attribute for the `i`th argument, `attr`:

    `call_instruction_param_attribute(id, i, attr).`

  - The instruction calling convention, `conv`:

    `call_instruction_calling_convention(id, conv).`

  - The function signature, `sig`:

    `call_instruction_signature(id, sig).`

  - If the instruction is a direct call:

    `direct_call_instruction(id).`

  - If the instruction is an indirect call:

```
indirect_call_instruction(id).
```

- va_arg

  A va_arg instruction with an id will produce the following relations:

  - id is a va_arg instruction:

    ```
    va_arg_instruction(id).
    ```
  - The instruction type, ty:

    ```
    va_arg_instruction_type(id, ty).
    ```
  - The instruction points to an argument list, arg:

    ```
    va_arg_instruction_va_list(id, arg).
    ```

A.7 Other Information

This section outlines the relations that are created to represent the various information that instructions might need. This information includes attributes, calling convention, linkage type, ordering and visibility.

A.7.1 Attributes

Attributes communicate additional information about either a function or a parameter. In the relational schema, an attribute is represented as a string. The string is the name of the attribute. Parameter attributes can be seen in Table A.1 and function attributes can be seen in Table A.2.

zeroext	signext	inreg
byval	inalloca	sret
align	noalias	nocapture
nest	returned	nonnull
dereferenceable	dereferenceable_or_null	

Table A.1: Parameter attributes

alignstack	alwaysinline
builtin	cold
convergent	inaccessiblememonly
inaccessiblemem_or_argmemonly	inlinehint
jumptable	minsize
naked	nobuiltin
noduplicate	noimplicitfloat
noinline	nonlazybind
noredzone	noreturn
norecurse	nounwind
optnone	optsize
readnone	readonly
argmemonly	returns_twice
safestack	sanitize_address
sanitize_memory	sanitize_thread
ssp	sspreq
sspstrong	uwtable

Table A.2: Function attributes

Appendix B

Array Boundary Violation Test Programs

Listing B.1 is a program that contains a correct, explicit array access. Listing B.2 and Listing B.3 are programs that explicitly violate the boundaries of an array. Listings B.4, B.5, and B.6 contain explicit array accesses that depend on control flow. Listing B.7 is a program that contains a correct, variable array access. Listing B.8 and Listing B.9 are programs that contain an index variable that violates the boundaries of an array. Listings B.10, B.11, and B.12 contain variable array accesses that depend on control flow.

The remainder of the programs test features that fall outside of the rule's scope. Listing B.13 and Listing B.14 contain arrays that are accessed at an index variable added to a constant value. Listing B.15 and Listing B.16 contain arrays that are accessed at constant value subtracted from an index variable. Listing B.17 is a program that violates the upper boundary of an array that is a member of a struct. Listing B.17 accesses an array using a loop variable as the index. Finally, Listing B.19 is a program that violates the upper boundary of a global array.

```
int main(int argc, char *argv[]) {
    int x[5];
    x[0] = 3;
    return 0;
}
```

Listing B.1: Proper explicit indexing

```
int main(int argc, char *argv[]) {
    int x[5];
    x[6] = 3;
    return 0;
}
```

Listing B.2: Explicitly exceeding upper boundary

```
int main(int argc, char *argv[]) {
    int x[5];
    x[-1] = 3;
    return 0;
}
```

Listing B.3: Explicitly exceeding lower boundary

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int x[5];
    int y;
    scanf("%d", &y);

    if (12345 == y) {
        x[4] = 3;
    }
    else {
        x[0] = 3;
    }
    return 0;
}
```

Listing B.4: Proper explicit indexing via all execution paths

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int x[5];
    int y;
    scanf("%d", &y);

    if (12345 == y) {
        x[5] = 3;
    }
    else {
        x[0] = 3;
    }
    return 0;
}
```

Listing B.5: Explicitly exceeding upper boundary via specific execution path

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int x[5];
    int y;
    scanf("%d", &y);

    if (12345 == y) {
        x[-2] = 3;
    }
    else {
        x[0] = 3;
    }
    return 0;
}
```

Listing B.6: Explicitly exceeding lower boundary via specific execution path

```c
int main(int argc, char *argv[]) {
    int x[5];
    int i = 1;
    x[i] = 3;
    return 0;
}
```

Listing B.7: Proper index variable

```c
int main(int argc, char *argv[]) {
    int x[5];
    int i = 6;
    x[i] = 3;
    return 0;
}
```

Listing B.8: Index variable exceeds upper boundary

```c
int main(int argc, char *argv[]) {
    int x[5];
    int i = -1;
    x[i] = 3;
    return 0;
}
```

Listing B.9: Index variable exceeds lower boundary

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int x[5];
    int y,i;
    scanf("%d", &y);

    if (12345 == y) {
        i = 4;
    }
    else {
        i = 3;
    }
    x[i] = 3;
    return 0;
}
```

Listing B.10: Proper index variable via all execution paths

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int x[5];
    int y,i;
    scanf("%d", &y);

    if (12345 == y) {
        i = 7;
    }
    else {
        i = 3;
    }
    x[i] = 3;
    return 0;
}
```

Listing B.11: Index variable exceeds upper boundary via specific execution path

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int x[5];
    int y,i;
    scanf("%d", &y);

    if (12345 == y) {
        i = -1;
    }
    else {
        i = 3;
    }
    x[i] = 3;
    return 0;
}
```

Listing B.12: Index variable exceeds lower boundary via specific execution path

```
int main(int argc, char *argv[]) {
    int x[5];
    int i = 1;
    x[i+3] = 3;
    return 0;
}
```

Listing B.13: Addition remains within boundary

```
int main(int argc, char *argv[]) {
    int x[5];
    int i = 1;
    x[i+8] = 3;
    return 0;
}
```

Listing B.14: Addition exceeds upper boundary

```
int main(int argc, char *argv[]) {
    int x[5];
    int i = 4;
    x[i-1] = 3;
    return 0;
}
```

Listing B.15: Subtraction remains within boundary

```
int main(int argc, char *argv[]) {
    int x[5];
    int i = 4;
    x[i-7] = 3;
    return 0;
}
```

Listing B.16: Subtraction exceeds lower boundary

```
struct Foo {
    int x[5];
};

int main(int argc, char *argv[]) {

    struct Foo f;
    f.x[7] = 2;
    return 0;
}
```

Listing B.17: Struct member assignment exceeds upper boundary

```
int main(int argc, char *argv[]) {

    int x[5];

    for (int i = 0; i <= 6; i++) {
        x[i] = 3;
    }
    return 0;
}
```

Listing B.18: Loop variable exceeds upper boundary

```c
int x[5];

void foo() {
    x[7] = 3;
}

int main(int argc, char *argv[]) {
    x[2] = 3;
    return 0;
}
```

Listing B.19: Exceed upper boundary of global array

Appendix C

Function Call Post Condition Test Programs

Listings C.1, C.2, C.3, and C.4 are programs that satisfy the post condition for a function call. The function call is either not assigned to a variable, not used, or properly checked. Listing C.5 and Listing C.6 are programs that do not check the return value at all. Listing C.7 and Listing C.8 are programs that test the rule's ability to detect violations that occur due to control flow.

```
int foo() {
    return 2;
}

int main(int argc, char *argv[]) {
    foo();
    return 0;
}
```

Listing C.1: Function call is not assigned to a variable

```
int foo() {
    return 2;
}

int main(int argc, char *argv[]) {
    int x = foo();
    return 0;
}
```

Listing C.2: Function call result is not used

```c
int foo() {
    return 2;
}

int main(int argc, char *argv[]) {

    int x = foo();
    if (-1 == x) {
        return 1;
    }

    return x;
}
```

Listing C.3: Function call is used and properly checked

```c
#include <stdio.h>

int foo(int x) {
    return x+1;
}

int main(int argc, char *argv[]) {
    int y,x;
    scanf("%d", &y);

    if (3 == y) {
        x = foo(3);
    }
    else {
        x = foo(4);
    }

    if (3 == x) {
        return 1;
    }

    y = y + x;
    return y;
}
```

Listing C.4: The value of a variable depends on control flow, the value is checked either way

```c
int foo() {
    return 2;
}

int main(int argc, char *argv[]) {

    int x = foo();

    return x;
}
```

Listing C.5: Function call is used and not checked

```c
#include <stdio.h>

int foo(int x) {
    return x+1;
}

int main(int argc, char *argv[]) {
    int y,x;
    scanf("%d", &y);

    if (3 == y) {
        x = foo(3);
    }
    else {
        x = foo(4);
    }

    y = y + x;
    return y;
}
```

Listing C.6: The value of a variable depends on control flow, the value is not checked

```c
#include <stdio.h>

int foo(int x) {
    return x+1;
}

int main(int argc, char *argv[]) {
    int y,x;
    scanf("%d", &y);

    if (3 == y) {
        x = foo(3);
        if (3 == x) {
            return 1;
        }
    }
    else {
        x = foo(4);
    }

    y = y + x;
    return y;
}
```

Listing C.7: The value of a variable depends on control flow, not all calls are checked

```
#include <stdio.h>

int foo(int x) {
    return x+1;
}

int main(int argc, char *argv[]) {
    int y,x;
    scanf("%d", &y);

    x = foo(3);
    if (3 == y) {
        if (x == 0) {
            return 0;
        }
    }
    else {
        y = x;
    }

    y = y + x;
    return y;
}
```

Listing C.8: The value is only checked following one execution path

Appendix D

Double Free Test Programs

Listing D.1 and Listing D.2 demonstrate the correct usage of the `malloc()` and `free()` API, accounting for aliased pointers. Listing D.3 and Listing D.4 are programs that contain simple double frees, for a single pointer and aliased pointer, respectively. Listings D.5, D.6, and D.7 contain examples of double frees that are dependent on control flow. These programs include conditionals and `goto` statements. Listing D.8 is a program that double frees a memory address through a function call. Finally, Listing D.9 is a program that double frees a global pointer.

```c
#include <stdlib.h>

int main(int argc, char *argv[]) {

    int *x = malloc(8);
    free(x);

    return 0;
}
```

Listing D.1: Correct usage of `malloc` and `free`

```c
#include <stdlib.h>

int main(int argc, char *argv[]) {

    int *x = malloc(8);
    free(x);

    x = malloc(8);
    free(x);

    return 0;
}
```

Listing D.2: Correct reallocation of a pointer

```
#include <stdlib.h>

int main(int argc, char *argv[]) {

    int *x = malloc(8);
    free(x);
    free(x);

    return 0;
}
```

Listing D.3: Trivial double free

```
#include <stdlib.h>

int main(int argc, char *argv[]) {

    int *x = malloc(8);
    int *y = x;
    free(x);
    free(y);

    return 0;
}
```

Listing D.4: Double freeing via an aliased pointer

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

    int *x = malloc(8);
    int a,b;

    scanf("%d", &a);
    scanf("%d", &b);

    if (12345 == a) {
        if (54321 == b) {
            free(x);
        }
    }
    free(x);

    return 0;
}
```

Listing D.5: Double free with non-trivial control flow

```c
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

    int *x = malloc(8);
    int a,b;

    scanf("%d", &a);
    scanf("%d", &b);

    if (12345 == a) {
        if (54321 == b) {
            int *y = x;
            free(y);
        }
    }
    free(x);

    return 0;
}
```

Listing D.6: Double freeing via an aliased pointer with non-trivial control flow

```c
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char* argv[]) {
    int *x = malloc(8);
    int a;

    scanf("%d", &a);

    if (12345 == a) {
        free(x);
        goto cleanup;
    }

    int b;

    scanf("%d", &b);

    if (54321 == b) {
        goto cleanup;
    }

    free(x);
    return 0;

cleanup:
    free(x);
    return 1;
}
```

Listing D.7: Double freeing occurs following `goto` statement

```
#include <stdlib.h>

void foo(int *ptr) {
    free(ptr);
}

int main(int argc, char *argv[]) {

    int *x = malloc(8);
    foo(x);
    free(x);

    return 0;
}
```

Listing D.8: Interprocedural double free

```
#include <stdlib.h>

int *x;

int main(int argc, char *argv[]) {
    x = malloc(8);
    free(x);
    free(x);
    return 0;
}
```

Listing D.9: Global double free