HY2: A HYBRID VULNERABILITY ANALYSIS METHOD

by

Emma Sewell

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Carleton University
Ottawa, Ontario
August 2023

Abstract

Software vulnerabilities remain an ever-present problem. Factors such as software complexity, size, and diversity of vulnerabilities drive the need for automated vulnerability analysis solutions. Past vulnerability analysis methods struggle with nondeterminism and uncertainty introduced by the environment and external dependencies. In this thesis, we present our vulnerability analysis method Hy2 to address this problem. Hy2 is a double hybrid of runtime verification and model checking and dynamic and static analysis. It approaches the problem of building an abstraction of program behavior with decompilation and uses full-system emulation to handle undecidability and address environmental side effects. We discuss the limitations of past vulnerability analysis methods that motivated the creation of Hy2 and detail its design and implementation. We present an evaluation of our method on several real-world programs to demonstrate its practicality and effectiveness. We uncovered 18 reported and several unreported vulnerabilities in the programs evaluated, demonstrated our method's ability to handle concurrency, and described limitations and potential improvements to Hy2.

Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

Software vulnerabilities remain an ever-present and costly problem. In 2022 alone, over 25 thousand vulnerabilities were published, which was a 20% increase over the previous year [24]. Vulnerabilities are exploitable flaws in the design or implementation of a program or the environment it interacts with. Flaws compromise the safety and security of software, opening the door for exploitation and introducing other risks such as data loss. The problem of detecting flaws in software is complicated by many factors. Large codebases and the variety of available software make manual analysis infeasible. Fast-paced development cycles further complicate this task, as any changes to software present the possibility of new errors being introduced [79]. Finally, external dependencies make it insufficient to consider the program in isolation, as interactions between the program and external functions may introduce faults, therefore the behavior of these dependencies must also be considered.

The severity and prevalence of this problem has motivated a wide range of potential solutions, including automated vulnerability analysis methods. Such methods include those that use dynamic analysis with techniques such as fuzzing and runtime verification and static analysis with techniques such as symbolic execution and model checking. Each of these methods comes with a set of unique limitations. A common limitation of methods that use a model to describe program behavior is state space explosion, occurring when the number of program states grows beyond what can be feasibility explored [69] [33]. Secondly, unknowns introduced from various sources such as environmental side effects [33] [85] [35], aliasing [65], and nondeterminism [55] cause undecidability, meaning that a solution may not be able to come to a conclusion about the properties of a particular program. Finally, there is the problem of balancing scale and granularity, with methods needing to balance exhaustively analyzing a program and scaling to larger programs [33] [62] [38]. In order to scale, methods typically sacrifice either granularity or exhaustiveness. In sacrificing granularity they represent program behaviors in a more abstract manner, for instance with patterns of function and library calls, and by sacrificing exhaustiveness they may exclude parts of the code from the analysis.

Past static analysis methods are typically very rigid in that they cannot handle program states they cannot reason about. For example, if a program makes calls to a closed-source library, a static analysis tool may not be able to analyze the program because it cannot reason about the behavior of the calls to this library and the paths that might be taken based on values returned from these calls [41]. Some methods may handle this problem by requiring that a model of the environment be created prior to performing the analysis. The scope of this model ranges from describing the entire system to operating system components such as the file system, to simply external libraries. This task requires additional effort from the user, and incompleteness or errors in the model may introduce erroneous results [35] [55].

This work argues that an approach that incorporates dynamic analysis could potentially address the rigidity of static analysis methods through a novel hybrid of model checking and runtime verification techniques. Static methods start with source code, converting it to an intermediate representation to build an abstraction. Our method uses an orchestrated system to analyze program execution with instruction-level granularity, converting this to an IR to build a comparable abstraction. We believe this approach allows us to overcome many of the limitations of past works, primarily we focus on the problem of undecidability.

To support these claims, we present Hy2. In this work we frame the problem of vulnerability analysis, explore our design and implementation of Hy2, assess the effectiveness of our method and demonstrate that it is potentially practical in real-world applications.

1.1 Contributions

In summary, this work makes the following contributions:

- We introduce a new method of vulnerability analysis that combines runtime verification and model checking as well as dynamic and static analysis methods. We use full-system emulation to observe environmental side effects. We use decompilation methods to create a higher-level control-flow-based abstraction of program behavior comparable to that used by static-analysis methods. We introduce a secondary static analysis step to reach branches that would not be observed during the normal execution of the program. Finally, we do not consider path constraints in our model; instead, we introduce a separate feasibility analysis step that determines whether reported errors may be possible in the target program, and under what conditions they may occur. We argue that our method is able to fill the gap previously described and expanded on in Chapter 3.

- We implement a property language that allows for arbitrary properties to be described in simple language and subsequently checked against the models of arbitrary programs. We create a method for allowing application-specific semantics to be introduced into generic properties.

- We present the implementation of our method, Hy2, which is written in over 14 thousand lines of python code and provides web and command-line interfaces for a user to interact with our framework.

1.2 Outline

The rest of this thesis is organized as follows. In Chapter 2 we present a detailed background and describe related work in the area of vulnerability analysis. In Chapter 3 we explore the problem of vulnerability analysis in depth and present our motivation for this work. Chapter 4 details the design of our method, explaining our design goals and how they were realized with the creation of our method. In Chapter 5 we describe the implementation of our method in detail. In Chapter 6 we present an evaluation of our method through the presentation of several case studies. In Chapter 7 we discuss and analyze our method and its effectiveness, describe potential areas of future work, and conclude the thesis.

Chapter 2

Background and Related Work

This chapter presents the background required to understand this work and describes related work in the area of vulnerability analysis. Section 2.1 describes various program analysis methods used in the area of vulnerability analysis. Section 2.2 summarizes the field of formal verification. Sections 2.3 and 2.4 describe the logic systems on which vulnerability analysis methods are based on, these being propositional and temporal logic. Section 2.5 presents background on model checking, including the various techniques used relevant to this work. Section 2.6 provides insight into the field of runtime verification, and past efforts to combine model checking with runtime verification. Finally, Section 2.7 summarizes past efforts using fuzzing, Section 2.8 summarizes past efforts using symbolic execution, and Section 2.9 summarizes past efforts using binary analysis platforms.

2.1 Program Analysis

Before we can discuss the various methods of vulnerability analysis, we must first understand the fundamental analysis methods they are based on. In this context, program analysis refers to the techniques used to automate the analysis of computer programs.

2.1.1 Control Flow

Control flow describes the path of the instruction pointer through a program [83]. For the purpose of program analysis, control flow can be used to derive legal orderings of events in the program. Basic blocks are a set of instructions that the instruction pointer moves through sequentially. Basic blocks are a commonly used and effective foundation for program analysis. A control flow switch, such as a conditional statement, a function call, or a return instruction causes the program to jump to a different address and describes the transitions between basic blocks. We can describe this with a control flow graph, a language-independent graph that describes the data and control flow of a program. Basic blocks make up the vertices in this graph, and jump statements or control flow switches make up the edges. An execution path or code path through the graph describes the path in the program from the entry point to an exit node. Path coverage describes the process of deriving a set of code paths that cover every possible path.

2.1.2 Data Flow Analysis

Data-flow analysis is the analysis of the flow of data within a program [61]. Specifically, it is the analysis of how data is passed between variables. It is considered complementary to control-flow analysis, typically operating on a control-flow graph to determine how data propagates through a program.

We describe the data propagation for specific variables in terms of liveness and availability. We describe variable lifetimes in terms of definition and use, where *definition* refers to the program point where the value is introduced in the program, and *use* refers to program points where the variable was accessed [61]. We say a variable is *live* at a particular program point if there is a path from the program point to a use of the variable, without a redefinition of the variable [61]. We say an expression e is available at a particular program point p if, for every path from the entry node to p, e is evaluated, and there are no redefinitions of the expression's variable dependencies after the last evaluation of e. If we consider the case of conditional branches on the path before the program point p, we want to ensure that the expression means the same thing across each branch before analyzing it at p. Lastly, we define the reachability of a variable v at a particular program point p as follows: if there is a path from the definition of v to p on which v is not redefined then we say that the variable v reaches node p. With these definitions, we can precisely describe the movement and exchange of data in relation to the control flow of a program. To reduce the computational cost of data flow analysis an abstraction is often used. For instance, data flow could be abstracted by merging information across all instances of the same program point, for each point in the program, without considering control flow. Interprocedural context-sensitive data-flow analysis is expensive, so an approximation is often used [78]. It is expensive because it considers sequences of calls through the program that can influence the context of an operation on a variable. This analysis is worst-case exponential to the number of acyclic paths through the code.

Alias analysis, or aliasing is performed using the data-flow logical framework. It is the analysis of whether two variables ever point to the same location in memory. This can sometimes be undecidable without executing the program. Aliasing can be classified by flow sensitivity and context sensitivity. Context-sensitive analysis increases the precision of the aliasing, using contextual information to differentiate invocations of blocks of code. This may include using the calling function as context. Flow-sensitive analysis considers the order of statements within a procedure when computing aliases. This analysis requires a control-flow graph or transition system. By contrast, to be flow insensitive means to not consider the order of statements, ignoring conditional statements and loops.

2.1.3 Intermediate Representations (IR)

An Intermediate Representation (IR) is a representation of a program that is between the source language and machine code [82]. Its simple structure is designed for optimization and code generation in the context of compilers. There are several types of

IRs, these being as follows:

- *Graph IRs* which are directed acyclic graphs (DAGs), based on simplified Abstract Syntax Trees (ASTs). ASTs model relationships between tokens (i.e. keywords) in statements in the source, representing the structure of the program. Semantics graphs are one such example, these being ASTs transformed into a graph with annotations and type checking performed.

- *Linear IRs* which consist of an ordered sequence of instructions. Each instruction in the sequence takes the form of a tuple which contains the operator and arguments. This representation uses an unbounded number of registers, allowing for the observation of the lifetime of a value. This lifetime begins with the first point the register is written and ends at the last point the register is used.

- *Stack IRs* which are similar to linear IRs except they don't use registers and instead use a stack with push and pop instructions to handle intermediate values.

ASTs are also considered an IR; however, it is a very high-level representation of the source and is impractical for efficient compilation. However, ASTs are used for static analysis tools such as Coverity [51].

Intermediate languages (IL) are a subset of IRs that are often used for static analysis. Intermediate languages are output languages of compilers that generate another language instead of machine instructions. These languages break down higher-level C syntax into simpler constructs. Replacing the source with an IL serves to remove ambiguity and create a representation of programs that is independent of coding conventions, making it better suited for static analysis. These languages are more high level than other IRs, encoding type information and declarations [70].

2.1.4 Decompilation

Decompilation is the process of translating an executable binary back into source code [39]. This is the opposite process of *compilation* which is performed by compilers to translate source code into machine code. Decompilers are not to be confused with disassemblers, which serve a similar function except disassemblers transform executable code to a lower-level language, typically assembly, instead of a higher-level source language. There are many different methods of decompilation, and at a high level they may follow the following process:

1. Disassembly: Convert the machine code into an intermediate representation (IR). This is typically assembly in this context, but sometimes a higher-level architecture-neutral IR [84].

2. CFG Generation: Generate a control flow graph for each subroutine or function. The instructions within the basic blocks in this graph take the form of a set of IR expressions.

3. Variable Operation Analysis: Merge multiple instructions to create more complex expressions. For example, if you have a sequence of arithmetic operations where each instruction operates on the output of the previous instruction, it would combine these into a single larger expression.

4. Data Flow Analysis: Trace the definitions and use of registers and local memory accesses and replace these with a named variable.

5. CFG Reduction: Perform a control-flow analysis, removing dead instructions and structuring IR into higher level-constructs, such as conditional statements and loops.

6. Code Generation: We convert the analyzed CFGs to an approximation of the original source code.

Whereas past works typically use some form of *compilation* as described in Section 2.5, our method is built upon the process of decompilation. Chapter 4 describes how we incorporate these processes into our method in detail.

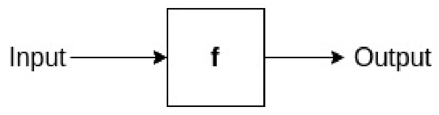
2.1.5 Emulation and Simulation

Although the terms emulation and simulation are often used interchangeably, they have different definitions in the field of computing. An emulator attempts to replicate the internals of a device whereas a simulator attempts to replicate the behavior of the device [81]. For the case of systems, emulators simulate the hardware of the device, allowing for software to run on the emulator the same as it would run on the original device. A prominent example is QEMU, an open-source emulator capable of emulating user space and entire systems through dynamic instruction translation. CPU simulators take in programs as a sequence of instructions and follow the program's execution flow, and for each instruction perform the equivalent operation. CPU simulators are useful for program analysis as instructions can be executed independently of the rest of the program.

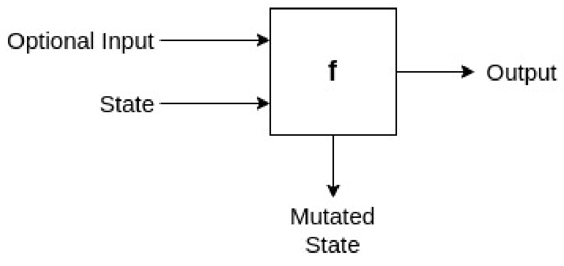
2.1.6 State Transformers

Whilst typically functions map inputs to outputs (Figure 2.1a), stateful functions or state transformers map states to states (Figure 2.1b). A state transformer performs an operation that mutates an input state to produce an output state and optional value which is represented as: $ST : S \to S'$.

State transformers are used to define formal definitions of instructions and semantics, which together serve as a processor which simulates the behavior of a particular



(a) Stateless Function



(b) Stateful Function

Figure 2.1: Illustrating the difference between a stateless function and a stateful function

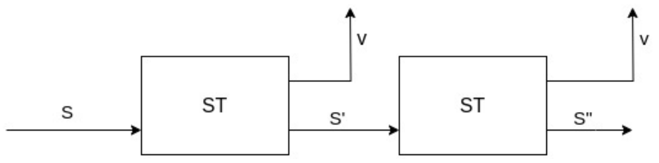


Figure 2.2: State transformers take in the program state and mutate it to produce the new program state and an optional output.

machine. This method of program simulation is commonly used in verification efforts and symbolic execution.

One such example of a method that uses this approach is REDFIN [67]. REDFIN is an assembly language. The execution of instructions in this language involves simulating the effect of each instruction on the program state, this being the processor and memory. Mokhov et al. chose to represent state transformers explicitly, instead of implicitly, so they can represent and manipulate the program state with symbolic values, allowing for formal verification to be performed. Through the implementation of instructions and semantics of REDFIN as state transformers, they are defining its requirements, describing exactly how it will behave.

2.2 Formal Methods

In this section we describe formal verification at a high level. Formal verification is the process of proving or disproving the correctness of a program for a set of formal specifications using mathematical proofs [45]. This method is typically only used for safety-critical systems and hardware, as it is time consuming and labor intensive to perform. The use of formal methods is desirable in these instances where faults can have large-scale real-world impacts because they can provide a higher level of assurance over other testing methods.

Functional verification, including traditional test methods, is the practice of testing that a program meets a set of functional requirements by testing it against various inputs and verifying the outputs are as expected. By contrast, formal verification looks at logical requirements and analyzes program logic to attempt to identify inputs that violate these requirements. Formal methods have the properties of soundness and completeness. A proof system is sound if nothing that is provable is in fact false (i.e. no false negatives), and complete if anything that is true is provable (i.e. no false positives). As performing formal verification may be impractical or infeasible, model checking is often posed as an alternative. Model checking is able to prove the program has certain properties, but cannot prove its correctness. We discuss model checking in Section 2.5.

2.3 Propositional Logic

A proposition is a statement for which one can assign a truth value. Propositional logic [53] describes propositions and the relationships between propositions, referred to as sentences. Atomic propositions are statements whose truth value is not dependent on that of another proposition. Sentences are constructed from atomic propositions and connectives. Connectives include operators such as or ($\vee$), and ($\wedge$), and not ($\neg$). An interpretation is the assignment of a truth value to every proposition within a sentence. Approaches to model checking and symbolic execution are typically based on propositional logic.

2.4 Temporal Logic and Safety Properties

Temporal logic is a logic system for describing properties that are qualified in terms of time, typically discrete time [54]. Linear Temporal Logic (LTL), a type of temporal logic, is logic reasoning about complete paths through a system, describing events along these paths. It allows for us to come to conclusions about a trace. LTL is able to describe safety and liveness properties.

Temporal Safety Properties are properties that describe behaviors that should never happen [29]. Temporal safety properties take the form $\Box\neg(E)$, or henceforth and forever not E, where E describes a set of sequences of events that should never occur in any execution of the program. Alternatively, safety properties can also be described as $G\neg E$, or globally not E. Safety properties deal with the reachability of some undesirable condition.

We can encode temporal property Φ as a finite state machine, or more precisely as 5-tuple $\Phi = (Q, Q_s, \delta, Q_f, L)$, where:

- Q is a set of events
- Q_s is a set of initial states
- Q_f is a set of final states
- δ is a transition relation $\delta \subseteq Q \times L \times Q$
- L is logic whose sentences represent a set of program states, and $\phi \in L$ denotes the set $[\![\phi]\!] \subseteq \Sigma$, meaning part of the property Φ is a subset of the alphabet of the model.

The final states in the finite state machine encoding of the property Φ are typically referred to as `ERROR` states, meaning that if the property is in this state we are in some undesirable or dangerous condition. The relation $(q, \phi, q') \in \lambda$ represents the execution of a statement from the program state λ where Φ holds and causes the temporal safety automaton to transition from state q to q'. If the state of Φ ever reaches an `ERROR` state it means that the property does not hold in the program, so we have determined the undesirable behavior to be possible in the program. Temporal safety properties can be refuted by a finite behavior, meaning if property Φ does not hold in P then we can produce a finite length trace demonstrating its noncompliance, called a counterexample.

Temporal logic differs from propositional logic in that temporal logic describes the truth value at every variable at each point in time. As opposed to assigning a single truth value to every variable, we can describe the truth value of a variable at every point in time. Whilst propositional logic can be expressed using temporal logic, temporal concepts can not be expressed in propositional logic.

2.5 Model Checking

This section describes the field of model checking in depth, discussing different techniques used for model checking and providing examples of past works that use these techniques. Model checking is the practice of constructing a model of a given system or component and checking temporal logic properties against it to verify they hold [62]. This model is derived from the target hardware or software, taking the form of a transition system. This model is typically described as the 5-tuple: (AP, S, S_0, R, L) where:

- AP is a finite set of atomic propositions
- S is a finite set of states
- S_0 is a set of initial states
- R is a transition relation, such that $R \subseteq S \times S$

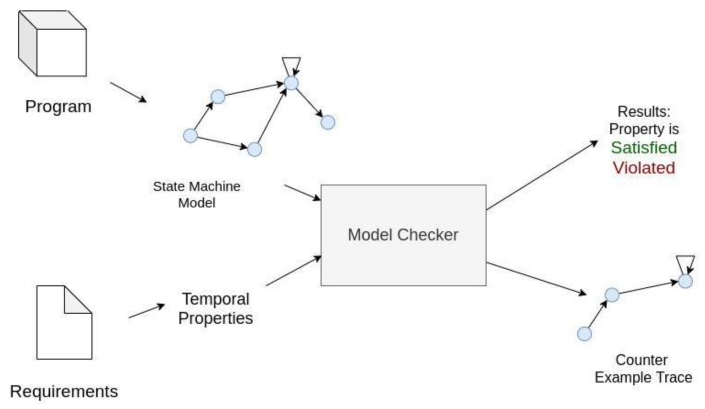


Figure 2.3: The architecture of a model checker at a very high level. It takes in a program model and a temporal property and returns a verdict as to whether the property is satisfied in the model.

- L, $L : S \rightarrow 2^{AP}$, is a labeling function, describing for each state $s \in S$ the set of propositional variables that hold in it. Propositional variables describe fixed sets of atomic propositions, which are arbitrary boolean properties that describe the system being modeled.

The problem of model checking is that of computing whether for model M and property ϕ, M models ϕ which is written as $M \models \phi$. The program M models ϕ if it can generate a trace π that matches a word, with a word being a finite sequence of symbols from the alphabet Σ in the language of the automaton encoding of ϕ.

We present the following trivial example of the model checking process for a satisfiability problem. Given the program shown in Figure 2.4a, we wish to check that the property $A \equiv z \leq 3$ holds in the program. We convert it to SSA form as shown in Figure 2.4b.

From this form we may derive a tertiary representation capturing control flow as shown in Figure 2.4c. Our property A can be rewritten as $A \equiv z_1 \leq 3$ and we can derive the following proposition P:

$$P \equiv x_1 == y_0 \wedge x_2 == 2 \wedge x_3 == x_1 + 1 \wedge ((x_1 > 3 \wedge z_1 == x_2) \vee (x_1 \leq 3 \wedge z_1 == x_3)) \quad (2.1)$$

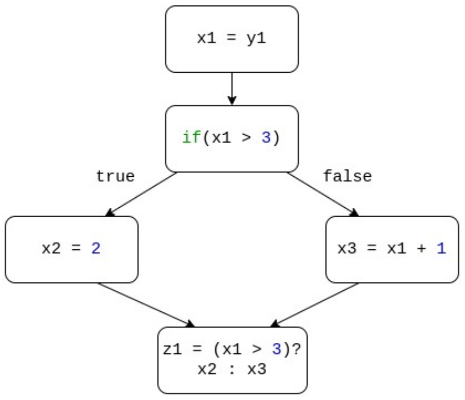
For each of the two branches, we can derive an SSA formula as follows. We

```
x = y
if(x > 3)
    x = 2
else
    x = x + 1
z = x
```

(a)

```
x1 = y1
if(x1 > 3)
    x2 = 2
else
    x3 = x1 + 1
z1 = (x1 > 3) ? x2 :
 ↪  x3
```

(b)



(c)

Figure 2.4: Example showing the conversion of the original program (a) to SSA form (b) and then finally to model describing control flow (c).

represent the two branches of program execution with the two propositions β_1, and β_2, from which we can derive the two cases: σ_1, and σ_2.

$$\alpha_1 = x_1 == y_0 \land x_2 == 2 \land x_3 == x_1 + 1 \tag{2.2}$$

$$\beta_1 = x_1 > 3 \land z_1 == x_2 \tag{2.3}$$

$$\beta_2 = x_1 \leq 3 \land z_1 == x_3 \tag{2.4}$$

$$\sigma_1 = \alpha_1 \land \beta_1 \tag{2.5}$$

$$\sigma_2 = \alpha_1 \land \beta_2 \tag{2.6}$$

For these two cases we have the following:

Case 1: $\sigma_1 = \alpha_1 \land \beta_1$, $z_1 == x_2, x_2 == 2$, $z_1 \leq 3$

Case 2: $\sigma_2 = \alpha_1 \land \beta_2$, $z_1 == x_3, x_3 == x_1 + 1$, $x_1 \leq 3 \therefore x_3 \leq 3 + 1$, so $z_1 \leq 4$, so if $z_1 == 4 \longrightarrow z_1 > 3$, then it is possible for z_1 to be greater than 3, so $P \land \neg A$ is satisfiable because there is a solution that has $z_1 \geq 3$.

Thus, we have proved that the property A does not hold in P with the following execution scenario $(x_1 : 3, y_1 : 3, x_3 : 4, z_1 : 4)$.

Methods for deriving a model for the purpose of model checking are varied. Some methods require that the user themselves create the model of the target program or system, often from specifications. Other methods are automatic, performing static analysis to automatically derive a model from the program source. Such methods typically use some form of IR to derive a simplified representation of the program.

Model checking often struggles with the problem of state space explosion. State space explosion refers to the problem that as the number of state variables in the system increases, the size of the system state space increases exponentially, thus making it impossible to represent and check these models in practice with finite computational resources. There are typically two approaches to handling this problem: abstraction and composition, which we describe in detail in the following subsection.

2.5.1 Explicit-State versus Abstraction-based Model Checking

Explicit-state, also referred to as concrete state, or execution-based model checking involves the exhaustive search of the concrete state space of a program [62]. Explicit-state methods typically focus on finding errors rather than proving correctness. Such methods, due to their use of concrete values, do not scale to large state spaces. An example of the size of the state space for such methods, is if we want to verify the correctness of an algorithm that sorts 4 numbers. Assuming these are 8-bit numbers, an explicit-state model checker would generate at least $2^{8\times4} = 2^{32}$ states, this being 1 state per combination of digits. The memory required to model larger state spaces is astronomical so this method is ill-suited for large programs. An example of a model checking tool that uses this method is SPIN. The purpose of SPIN is to prove the correctness of process interactions, and asynchronous process systems [60]. It requires that the model of the program or system under test be derived by the user. The user derives a prototype of the system in the modeling language Promela, and this prototype is verified using SPIN, being refined until correctness can be proven. The model consists of user-defined process templates, which define the behavior of different types of processes. These templates are translated into FSA (Finite State Automata), and the interleavings of these FSA are computed, allowing for it to consider all possible concurrent behaviors.

By contrast, abstraction-based model checking involves generating an abstraction of the state space. The nature of this abstraction varies between individual methods and is specific to the goal of the method. We consider abstraction as a type of approximation. There are generally two classes of approximation: over-approximation and under-approximation. Over-approximation means the abstraction has more behaviors than the real program and as a result, it may produce spurious results. Under-approximation means the abstraction has less behavior than the original system. This is useful for error detection, if there is an error in the under-approximation, then there must be an error in the real system. However, this means that it is possible for some errors to be missed. Generally, the introduction of an abstraction may result in imprecision as a result of the abstraction. Such methods that use abstraction-based model checking typically focus on proving correctness, by proving the absence of errors in the abstract domain. The use of abstraction is relevant to this work and is expanded on further below (Section 2.5.3 and 2.5.2).

2.5.2 Counterexample Guided Abstraction Refinement (CEGAR)

Counterexample Guided Abstraction Refinement (CEGAR) is a method for automatic refinement of the abstraction of a system [44]. It involves constructing a finite state abstraction of the program, analyzing it with regards to a particular property with model checking, and automatically improving this abstraction if the results are erroneous. It repeatedly refines the abstraction until it is able to produce a feasible counterexample, or it reports that the property holds. This method is able to balance precision and scale, improving the precision of the abstraction with regards to the particular property being verified, whilst keeping the overall state space relatively small. Examples of model checking tools that use this method include BLAST, SLAM, MAGIC, and MOPED.

BLAST [37] is a model checking tool for C programs. It works by converting C programs into the intermediate language CIL, which breaks down C constructs into simpler abstractions. It builds a Control Flow Automata (CFA) from the converted program. A CFA is a directed graph in which the edges correspond to program operations, and nodes correspond to program counter values, referred to as control points in the program. This graph is similar to a control flow graph, except in a control flow graph nodes correspond to operations and edges correspond to control points. From this CFA it builds an Abstract Reachability Tree (ART), iteratively refining the reachable region over each branch, by adding proceeding predicates. ARTs are trees that represent a portion of the reachable state space of a program. The nodes in these trees are labeled as: $n : (q, s, \delta)$, where q is a CFA location, s is the current call stack, and δ is a reachable region which consists of a boolean formula describing constraints on data on the path. Each edge is labeled with an instruction. A path in the tree corresponds to a program execution, and a path formula is a set of constraints that is satisfiable if the path is feasible, meaning the behavior described in the abstraction is actually possible in the program it is describing. The program satisfies a particular property if an error configuration is not reachable in the ART. It uses CEGAR to refine its abstraction. Beyer et al. augmented this tool with CCURED, a type-based memory-safety analyzer, to check for memory errors. They applied BLAST to various benchmark programs and test suites, including several Windows device drivers and Linux utilities.

SLAM [34] is a successful tool that uses static analysis to validate the properties of C programs. It works by abstracting C programs as boolean programs. It begins with a coarse program abstraction, using slicing to eliminate parts of the program that are relevant to the particular property and constructing a boolean encoding of state based on events that cause transitions in the particular safety property. For example, if we consider a property involving file access, we would define file open and file close events that cause the state `file_open` to change from true to false. It uses boolean programs as an abstraction because for such programs reachability and termination are decidable. This method uses the CEGAR algorithm. Upon identifying a potential violation of a particular property, it evaluates the reachability of the undesirable states. It checks if the path is feasible and iteratively produces predicates to refine the path. It then performs model checking on the refined boolean program. It bases predicates on observations rather than invariants. In this work, they additionally create the property language SLIC, a finite state language for stating rules in the form of temporal safety properties, using C syntax. These properties are encoded as security automata and verified through property instrumentation. The iterative nature of SLAM means it can time out and not come to a conclusion. SLAM is the basis for SDV and was used for the verification of Windows drivers.

2.5.3 Model Checking with a Fixed Abstraction

Model checking using type systems refers to a subclass of abstraction-based model checking that uses a finite fixed abstraction [62]. This method of model checking involves computing very coarse invariants over program variables or expressions. The abstraction used is generated by the product of the control flow graph and the type system. For example, if we consider objects such as files and mutexes we can describe their state explicitly as the result of events within the program. We can describe the state of a file as open after `open()` is called or closed after a close call is made. We can describe the state of a mutex as locked after a lock call is made, or unlocked after an unlock call is made. Although this coarseness presents some limitations, it is able to scale very well. However, it is only effective for instances where these coarse invariants are sufficient to describe the particular behavior. Examples of methods that use this method include MOPS and ESP.

ESP is a model checking tool that is designed for scalability [49]. Their approach models only branches on which the program's behaviors that are relevant to the property being verified differ along the arms of the branch. They base their method on property simulation, as it is scalable. For a given C program, they generate a call graph from the source. To identify relationships between variables in this graph, they perform a value-flow analysis, allowing for their approach to be a context-sensitive approximation. They refine the set of relevant interface expressions through a predetermined rule set, matching types, checking value creation patterns, and whether a function emits a particular expression to refine their value flow analysis. Although they claim their analysis is effective and scalable, they were only able to demonstrate its effectiveness for a single property, this being "a file should not be accessed after it is closed", which is very elementary.

Hadjidj et al. [57] present a model-checking tool built upon the MOPED model checker. It supports syntactic pattern matching in the definition of its temporal safety properties. Syntactic pattern matching is the method of using a single keyword to represent a group of functions with the same core functionality. For a target program, it converts the C source to the GIMPLE intermediate language, which is an internal high-level IR used in earlier versions of the gcc compiler. It translates this further into a Remopla model, which is another high-level IL. Finally, it performs a reachability analysis on the product of a given property and model. They evaluated this method using primarily the TOCTOU property on a few major applications including OpenSSH and Apache and were able to discover 3 potential vulnerabilities with a low false positive rate.

Finally, MOPS [42] uses a similar approach to ESP. It converts C programs to a gcc IR, and then further to a CFG. To check a security property against a program, they compute the intersection of the property and the model and check there is no path to an `ERROR` state in this intersection. Notably, this method also uses syntactic pattern matching and is path-sensitive but data-flow insensitive. However, deriving possible assignments for variables in these properties is expensive and error-prone. They performed an extensive evaluation of their tool [32]. The practicality of this method was shown through the verification of a small set of security properties against the entirety of a Linux distribution. This showed that the tool had limited effectiveness at scale, with over 90% of the reported violations being illegitimate, and a large amount of time being needed to be devoted to differentiating between legitimate and illegitimate violations. This demonstrates the limitations of coarse-grained abstractions.

2.5.4 Product Automaton Construction

Product Automaton Construction describes the intersection or union of two or more finite state machines [71]. It describes the intersection N of finite state machines S and P for word $w \in \Sigma^*$ such that N is in state (q, q') after reading w iff S is in state q after reading w and P is in state q' after reading w, or $\delta^*_N((q_0, q'_0), w) = (\delta^*(q_0, w), \delta'^*(q'_0, w))$. If N accepts $L(P) \cap L(S)$ then the set of final states of N is $F_N = F \times F'$.

To illustrate this process we provide the following example (Figure 2.5). We show the intersection of the two finite state machines S, and P (Figures 2.5a, and 2.5b) in Figure 2.5c. Although P has an additional final state $P4$ there is no symbol d, or s accepted by S.

2.5.5 Product Construction for Rule Checking

Product construction, also refered to as product composition, is a method used in model checking to check temporal properties against a state-based program model [52]. This method is well-known and used by tools such as MOPS [42].

Assume we have a pushdown automaton describing the behavior of the program

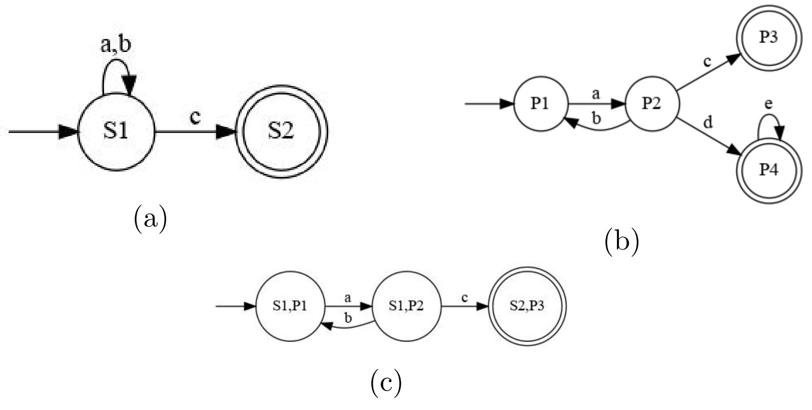


Figure 2.5: The intersection of the FSM in (a), and FSM in (b) forms the FSM shown in (c).

M and a temporal safety property φ. We encode the temporal safety property as a finite state machine S. To compute whether $M \models S$ we perform synchronous product construction with S and M to produce a PDA N describing the intersection of M and S. If there is a path through N to an `ERROR` state then the program violates the property S.

We define the problem of checking S against M as the problem of determining whether there is any path p through S to an `ERROR` state that is also in M, or $L(S) \cap L(M) = \emptyset$. Specifically through product automaton construction, we can create the PDA N describing the intersection of S and M such that $L(N) = L(M) \cap L(S)$. Any traces accepted by M, and S must be a subset of $L(M) \cap L(S)$ and if $L(M) \cap L(S)$ is empty then there are no possible traces accepted by M and S, so P can not violate φ. However, since $L(P) \subseteq L(M)$, then we can not make guarantees that the set $L(M) \cap L(S)$ is empty, as the program may exhibit behaviors that are not modeled by the property. We use an abstraction of this logic in our rule-checking step as described in Section 5.8.

2.6 Runtime Verification

Runtime verification is a method of system analysis that verifies the system as it executes [36]. It involves executing the target program to analyze it, observing the resulting execution trace, building a state-based model of this trace and analyzing said model, by checking safety properties against it. Other types of analysis can be performed alongside runtime verification such as runtime enforcement, which involves taking action in the event of a property violation. It complements model checking which derives a model from source code. Runtime verification has the advantage of being easy to scale and producing no spurious results, but has the limitations of poor coverage and that the code must be executable. By contrast, model checking has the advantages of having good code coverage and can be done earlier in development, because it is static and so does not require working code. However, it has the limitations of undecidability, the potential for false positives and false negatives, and limited scalability. Additionally, model checking must isolate the program from environmental side effects during its analysis, such as randomness and other sources of nondeterminism. This is typically achieved by modeling the relevant parts of the environment. If this environment is not an accurate approximation of the environment the program will be executed in, this can lead to errors in the analysis.

Runtime verification typically performs an analysis of a single execution trace. This method models system or software executions as event systems or transition systems. We define events as observations about the system and a trace as a behavioral abstraction of a single run of the system which consists of a finite sequence of events. The goal of runtime verification is to check traces against security properties. A property describes a potentially infinite set of traces. These properties are most commonly specified with temporal logic, with specifically Linear Temporal Logic (LTL) typically being used.

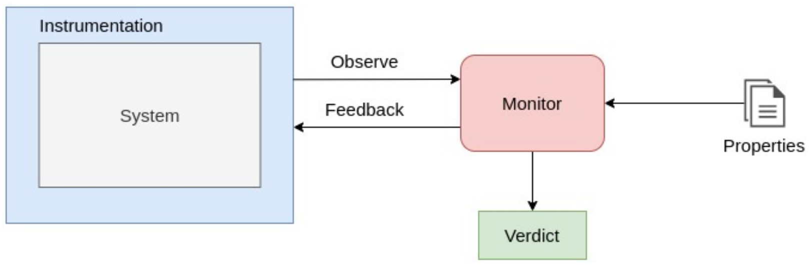


Figure 2.6: The Architecture of Runtime Verification methods at a high level

Runtime Verification methods consist of the following components, illustrated in Figure 2.6:

- System — this refers to the system under test or the system we are analyzing.
- Monitor — this component executes alongside the target system and it analyzes execution traces to determine whether a particular system complies with a particular property.
- Instrumentation — this component is responsible for handling the instrumentation of the running system, including controlling which parts of the system are made visible to the monitor for analysis. It records information from the running system and reports the recorded events to the monitor.

Runtime verification is typically performed online, with verification occurring during the execution of the system, making this a very practical method of analysis.

RV-Match and the work of Bristot et al. are two examples of runtime verification. RV-Match is a runtime verification tool that combines concrete execution with symbolic execution [56]. It verifies the safety of a program by executing it within the model of ISO C-11 and reporting any violations to this model at runtime. It builds an abstract state machine of the program during its execution and analyzes each event, performing consistency checks against the expected state. This tool is limited to certain classes of vulnerabilities and has to see the program executing under the vulnerable conditions to report the fault.

The work “Efficient Verification of the Linux Kernel” [48] by Bristot et al. is a runtime verification method for checking the real-time Linux kernel against a pre-created model based on specifications of its behavior. Their implementation consists of a kernel module that inserts callbacks at the transitions of the model of the specification and records the kernel state at runtime, reporting any transitions that aren’t consistent with the specification. This method allows for verification to be performed with little overhead, however, it is not guaranteed to be exhaustive. The properties that can be verified with this method are limited as their abstraction of system behavior is based on sequences of system calls.

2.6.1 Combining Model Checking with Runtime Verification

The problem of effectively combining model checking techniques with runtime verification techniques is a common research problem [66]. The motivation for combining these two verification methods is that it will allow a developer to choose the extent of verification appropriate for the particular application, with the goal being to allow for the important components of the application to be verified using model checking, and everything else to be verified using runtime verification. This would essentially serve as another method of state-space reduction. Runtime verification can only check a subset of the properties that can be checked with model checking and offers weaker guarantees.

An example of efforts to combine these two techniques is the extension of the explicit-state model checker DIVINE3 with runtime verification [63]. The motivation behind this work was to reduce the effort that goes towards environmental modeling when performing model checking. Model checkers require a complete definition of the system under test, including any environmental effects. This definition is usually created in the same language as the source language. This model must be precise to not introduce any inaccuracy into the verification process. For unknown or undefined functions, model checkers use a fallback implementation which produces undetermined results. Their approach places a virtual OS within the model checker, in which they create a system call interface. Their model checker is capable of operating in two modes: run mode, which explores a single execution to determine event ordering, and verify mode: which uses standard model checking. Their virtual OS has three modes of operation: virtual mode, in which interaction with the outside world is handled by the virtualized OS, pass-through mode, in which system calls are executed on the system, and replay mode, which reads the system call trace recorded in pass-through mode. This design serves two purposes: to identify inconsistency between their virtual OS, and execution within the real OS, and to ensure that all aspects of program execution that are observable to the program are fully representable.

Methods that attempt to combine runtime verification and model checking typically rely on both methods operating in conjunction rather than acting as a single cohesive whole, with some parts of the system being verified with runtime verification and others using model checking. This means that they are incurring the cost of both methods. By seamlessly merging the two methods this cost could be reduced.

2.7 Fuzzing

Fuzzing is a method of fault discovery involving the input of invalid, random, or unexpected data into a system [9]. Fuzzers typically work by mutating input values to reach new branches. The purpose of fuzzing is to stress a system with the goal of observing various exceptions, such as crashes and hangs. Over the past decade, AFL [1] has found many vulnerabilities in a wide variety of different applications. AFL [2] is a brute-force fuzzer built upon a genetic algorithm. AFL detects faults through the spawned process dying due to a signal. It will achieve limited coverage if encryption or compression is used on the input data, or if the relationship between inputs and output is nondeterministic. It has no support for fuzzing network services, background daemons, and applications that require interactions with a user interface. AFL also has limited support for fuzzing command line options. Fuzzing is typically used in combination with other analysis methods, such as symbolic execution. Driller [79] is a fuzzing tool that uses selective concolic execution to generate new input seeds. Concolic execution is the combination of symbolic and concrete execution. The use of fuzzing allows for it to avoid state-space explosion whilst performing symbolic execution, and the use of concolic execution allows it to reach new paths missed by the fuzzer. AFLNet [73] is a grey box fuzzer for protocol implementations. Traditional fuzzing techniques are not effective on protocol implementations, as communication is often stateful and requires a certain sequence of messages to be sent to change states.

2.8 Symbolic Execution

Symbolic execution is a form of static program analysis [33]. Symbolic analysis is an analysis of programs by tracking symbolic values instead of concrete ones. We want to reason about all inputs that take the same path through the program and their potential range of values. We build constraints that characterize conditions for executing a particular path, and the effect of execution of a particular path on the program state. For each path in the program, we define symbolic path conditions. For a given path, the corresponding path condition P is satisfiable iff the path is executable. To evaluate if the path condition is satisfiable and the path can be taken, we use a constraint solver. Symbolic states take the form (E, C) where E maps program variables to symbolic expressions and C is the path condition. For a path P, we define the domain of the path as $D[P]$, this being the set of all inputs that cause the program to take path P, and $C[P]$, the computation for path P or the values that result from the execution of path P. Over the domain of all paths, we can determine if some condition is satisfied and come to a conclusion across all paths for the entirety of the program. An SMT model checker is sometimes used to verify whether a proposition holds along each explored path and if the path is feasible [33].

For the purpose of vulnerability analysis, we can determine if there is a set of concrete input values that cause a particular constraint to be satisfied. In this case, these constraints typically describe the conditions under which memory errors occur. KLEE [40] is a symbolic execution engine, designed to automatically explore program paths. Paths are represented as an execution state, containing the program counter, stack, address space, a list of symbolic objects, and a set of path constraints. The set of path constraints returned can be used to describe the behavior of the program under all possible conditions. Path constraints are used to determine if there are values that cause an error on any of the explored paths and return a concrete input that could be used to trigger it. Many other symbolic execution tools are built upon KLEE. Woodpecker is one such tool. Woodpecker combines static analysis and symbolic execution [47]. Their approach is based on the idea that only a small percentage of code paths are relevant to any given rule. To increase the efficiency of their analysis, they direct symbolic analysis to these paths, pruning redundant paths. This tool allows users to introduce new checkers to describe new rules. One limitation of this approach is that it requires a separate verification run per rule. Sys [38] is a vulnerability discovery tool created with the goal of scaling to large codebases. It uses static analysis to identify potential errors in the source code and then uses symbolic execution to verify that they are legitimate errors. The authors describe the trade-off between exhaustiveness and scale, favoring scale. They reason that it is not necessary to find all errors, only those that are urgent. To achieve this, they focus on finding paths or branches that are likely to have vulnerabilities. Finally, Ferry [85] is a tool for symbolic execution that is designed for program branch exploration. It focuses on exploring state-dependent branches, which are branches whose behavior is dependent on the current program state instead of program inputs. Past symbolic execution methods may be unable to reach or be inefficient in exploring these state-dependent branches. Ferry identifies which program variables describe program state and uses them to describe and reason about internal program states. Through the completion of their evaluation they show that Ferry is able to reach code with complex path constraints and locate vulnerabilities on state-dependent branches.

2.9 Binary Analysis

Binary analysis platforms are systems created for the analysis of software binaries. They are typically extensible and are designed for a wide range of security applications, allowing users to implement new modules for their specific analysis task. Examples of such platforms include BitBlaze, S2E, and BARF.

BitBlaze [77] is a binary analysis platform that combines static and dynamic analysis with symbolic execution. It is designed to be extensible, allowing users to implement their own plugins to augment the functionality of the platform. BitBlaze consists of three components: Vine, TEMU, and Rudder. Vine performs static analysis of binaries, translating machine code into an intermediate language. This component can perform various types of analysis on this IL including control-flow analysis, data-flow analysis, and symbolic execution. TEMU performs full-system dynamic analysis and is built on QEMU. Full-system emulation involves running a full operating system and observing how the target binary is executed within the system. TEMU defines callbacks and using a plugin architecture, allows the analysis to be extended through the creation of plugins analyzing data extracted from these callbacks. It is capable of performing full-system taint analysis to track the flow of data through the system. It extracts operating semantics, these being process and module information and symbol information. Finally, Rudder performs an automatic exploration of the execution space of the program using symbolic execution. Rudder is capable of generating inputs that satisfy a particular program path. Its symbolic execution is guided by concrete execution. Inputs can be marked as symbolic, and Rudder will follow concrete execution flow, symbolically executing operations that operate on these symbolic inputs. This set of analysis tools can be configured to be used in a variety of applications, including vulnerability and malware analysis.

BARF [58] is a binary analysis framework designed for semi-automated analysis of software dependencies. It is designed for extensibility. It is capable of converting binaries to an architectural-neutral IR representation, and performing CFG recovery, this being constructing a CFG from machine code. BARF is integrated with an SMT solver and is capable of performing symbolic execution on paths between basic blocks to reason about possible assignments. Unlike the other platforms described in this section, it only uses static analysis and lacks the ability to perform the same system-level introspection.

S2E [43] is a symbolic execution platform that combines concrete and symbolic execution in a new method referred to as "Selective Symbolic Execution". It is built on the idea that a user might want to explore some components of the software stack in full but not others. This method primarily aims to address the problem of environmental side-effects, allowing the user to specify the target scope within a system's execution space and switching between concrete and symbolic execution depending on whether the execution flow is in scope. When switching from concrete to symbolic execution arguments are made symbolic, and when switching from symbolic to concrete execution arguments are concretized. This concretization may cause branches to be missed; when this occurs, execution backtracks and a different choice of arguments is chosen for the function's execution. S2E defines a set of consistency models for different use cases. Consistency models describe the degree to which to analyze the environment and how it impacts analysis. Extending symbolic execution to environmental dependencies causes a path explosion and doesn't scale generally, while treating the environment as a black-box introduces imprecision in the analysis through missed paths. When analyzing only the relevant parts of the environment, one must consider the trade-off between precision and user effort. Effort is needed to create annotations which are required for values returned by the environment, but the alternative, this being ignoring constraints introduced by the environment and external dependencies introduces imprecision. This method is limited by path explosion, the complexity of consistency models, and the ease of use of the framework. S2E may struggle to reach deep parts of the program due to path explosion, which is caused by complex path constraints or branches dependent on the program state. They state that modifying the environmental consistency model may alleviate path explosion but it requires the definition of annotations for the environmental interface which requires knowledge of the system. The definition of annotations within their framework requires the use of their APIs which they state to be complex. Chipounov showed the applications of the S2E framework in reverse engineering, automated testing, and vulnerability analysis.

Chapter 3

Motivation

In this chapter we provide some insight into the problem of identifying vulnerabilities, contextualizing the problem with a motivating example and explain the potential utility of a new method. Hy2 is a hybrid of runtime verification and model checking created to identify flaws in software. It traces the execution of a target program on an orchestrated system over a set of inputs. Unlike other methods, the execution trace produced by this step has instruction-level granularity, allowing us to perform a series of steps resembling decompilation to derive a control-flow-based abstraction of the observed behavior. We check the derived model against a set of temporal safety properties to identify potential undesirable behavior. Finally, we perform a feasibility analysis to determine whether the behavior described in the reported error trace is possible in the target program.

3.1 Motivating Example

We present a real-world example of a vulnerability that encapsulates the challenges of identifying vulnerabilities automatically. This example shown in Listing 1, is CVE-2023-22809, a privilege escalation vulnerability in sudoedit. For this vulnerability to be exploitable, there must be a sudoers policy allowing for the user to edit a file with elevated privileges. The sudoers policy module selects an editor based on the values of a set of environment variables; in this case, we use `EDITOR` as an example. The sudoers module uses `--` to separate the list of files to be edited from the rest of the command. If for instance the environment variable `EDITOR` is set to `vim -- /path/to/sensitive_file`, it will transform the command line into `vim -- /path/to/sensitive_file -- /path/to/allowed_file` and which will allow for the user to access or modify an arbitrary file, in this case `/path/to/sensitive_file`.

Without insight into the inner workings of the program, it would be very difficult to detect this vulnerability because it is not necessarily an error but more so a mishandling of untrusted input. Furthermore, without the specific environmental conditions required to trigger the affected branch, observing this behavior would not be possible. Past vulnerability analysis methods would likely be ill-equipped to identify this class of vulnerability. Fuzzing methods would be insufficient to uncover this vulnerability as it is akin to a logic error that causes the program to behave in an unexpected way but does not cause any errors or exceptions, and that can only be triggered under very specific conditions, which could only be found by searching what would be a very large input space. Static methods would have to understand environmental side effects and have some understanding of the concepts of environmental variables, user privileges, and file operations to identify this vulnerability. They would also have to have some understanding of control flow to be able to represent that a change in privileges proceeds the opening of the file. Additionally, they would have to have a strong understanding of data flow to be able to identify the source of the file name being opened. Although we can't definitively say that these methods couldn't have found this error, sudo is a heavily audited program [6] and this vulnerability was identified through manual analysis [68]. This suggests the need for new methods that can both describe such undesirable behavior and place emphasis on the severity of such faults.

This is where we see the potential of a method that combines both dynamic and static analysis techniques. The functionality causing the vulnerability is exercised in the integration tests for sudo. Through dynamic analysis, a solution could simply run these tests and observe the vulnerable behavior without any modifications to the test suite or program itself. The input space for any given program is potentially very large; through the use of static analysis techniques, we could precisely describe the behavior of the paths executed for all potential inputs, allowing us to identify the undesirable behavior without specifically executing the program with an input that triggers the vulnerability. A method should be able to describe the undesirable behavior at a high level in such a way that is not specific to a particular application. Broadly speaking, this behavior could be described as “Input from an untrusted source being used in a privileged context”, since environment variables are defined by non-privileged users. In the past, we can find other examples where environment variables have been used in similar manners, such as CVE-2020-15704 [10]. If we were to apply this potential method to other instances of the same type of vulnerability it should be able to detect them as well. We demonstrate that our method meets these requirements and is capable of detecting this error in Chapter 6.

3.2 Exploring the Problem

In this section, we explore the problem of detecting vulnerabilities and why we believe it’s necessary to design a new method to tackle this problem. We identify 3 main areas where we see a gap that needs to be addressed, these being the diversity of types of vulnerabilities, environmental side effects, and scalability.

3.2.1 Diversity of Vulnerabilities

Not only do vulnerabilities remain an ever-present problem, but additionally there are many classes of vulnerabilities that present in different ways and have varying impacts on the vulnerable application. Some vulnerabilities can only be identified with an understanding of program semantics. For example, to identify those involving information leakage we must understand what information within the context of the

```
find_editor(...){
...
ev[2] = "EDITOR";
for(i = 0; i < nitems(ev); i++){
     editor = getenv(ev[i]);
     resolve_editor(editor, ...);
     ...

resolve_editor(...){
     nargv[0] = editor;
     //copy rest of arguments to nargv
     ...
     for (nargc = 1; (cp = wordsplit(NULL, edend, &ep))!= NULL; nargc++) {
     if (nfiles != 0){
          nargv[nargc++] = "--";
          while (nfiles--){
             nargv[nargc++] = *files++;
          }
     ...
     *argv_out = nargv;
     // command_details.argv = argv_out;

sudo_edit(...){
  ...
  setuid(0);
   // Find a temporary directory writable by the user
  set_tmpdir(&user_details.cred);
  ...
  for(ap = command_details->argv; *a != NULL; ap++){
     if (files)
        nfiles++;
     else if (strcmp(*ap, "--") == 0)
        files = ap + 1;
     ...

     // Copy files to temporary user writable directory
```

Listing 1: A snippet of the sudoedit source showing the vulnerable parts of the program

application is sensitive. If there is a string containing a password being written to a log file, we must understand that this string contains a password and that the password is sensitive information. As performing manual analysis is infeasible generally, we still need to tackle these flaws with a generic solution. Thus, we must be able to describe the behavior of these types of vulnerabilities across different applications, instead of a single specific use case. As these may be introduced by a developer oversight and may be the intended behavior rather than the result of an error, solutions that identify faults or logic errors may be ineffective, such as fuzzing, as described in Section 2.7. Additionally, in these cases, why a particular behavior is undesirable must be described by the user, as the vulnerability analysis tool does not have any knowledge of the meaning of variables and fields within a program unless provided as input. This may include specifying which fields contain sensitive information or operations should not be performed without authentication. Thus, we attempt to address this problem by designing a solution that emphasizes flexibility. We achieve this through the use of syntactic pattern matching in our policy language which allows for different symbols to have different definitions depending on the specific application.

Some operating system concepts intrinsically have states which can change over time or in response to other events within the execution of the program. These may include network protocols, files, permissions, and mutexes. It is important to be able to describe this behavior within our framework, to more accurately represent the program's behavior and the cause of undesirable behaviors. Events that are dependent on those that have occurred previously are known to be important in achieving coverage when performing symbolic execution, as described in Section 2.8. When defining security properties, certain behaviors can only be described with temporal logic (Section 2.4), thus its use is necessary. As discussed in Section 2.8, symbolic execution and fuzzing methods are unable to describe these states and struggle with describing programs with an internal state.

Finally, vulnerabilities may be introduced from logic errors, which are errors in program logic that produce unexpected behavior but do not cause the program to crash. These types of errors may result in vulnerabilities such as login bypass. To identify these types of vulnerabilities, we must have some understanding of the expected program behavior. Methods that use techniques such as fuzzing are ill-equipped to identify these classes of vulnerabilities, as they locate faults by identifying crashes and exceptions and do not have insight into the logic of the program, as described in Section 2.7.

These problems motivate us to design an approach that uses temporal logic and thus can describe the evolution and logic of the target program, and to create a robust property language that can describe application-specific semantics in a generic manner.

3.2.2 Environmental Side Effects

Some vulnerabilities are not caused by the target program itself, but by the improper use of or weakness in a dependency. This brings us to the problem of compatibility. Past methods struggle with compatibility, only allowing for the analysis of programs written in certain languages [42] [37], or that use a particular subset of libraries [85], or often require that the program source be provided. This limitation also applies to methods that operate on the program binary but require the use of a specialized compiler for the addition of annotations, for which the source code must also be available [1] [56]. This can become an issue if, for example, the program makes use of a closed-source library. In this case, such methods may be unable to analyze the program, may require the user to manually specify its behavior, or may be unable to analyze the program exhaustively. We are able to eliminate this limitation through the use of dynamic analysis, as we can derive an abstraction of its behavior from its execution, and from its environmental interactions.

Some vulnerabilities involve the misuse of operating system components, such as access control, the file system, and environment variables. To identify these vulnerabilities, an understanding of these concepts is required. This is typically achieved through the creation of a model of the environment. The task of modeling the relevant parts of the environment may require effort from the user, and knowledge of the behavior of the environment as the model produced must be accurate enough to not introduce any inaccuracy into the analysis, as discussed in Section 2.6. Past works have shown that the use of a concrete environment is potentially an effective solution to this problem (Section 2.9). Through the use of a concrete environment, we can exhaustively describe the environmental conditions visible to the program for a particular execution. For the case of external functions being called by the target program, we can automatically represent their side effects visible to the target program, these being input parameters and return values. Secondly, dynamic analysis allows us to eliminate the undecidability that arises when the program encounters an external function that is not included in the environmental model, as we allow for the derivation of a model of the behavior of arbitrary external functions. Past methods that use this approach, particularly those that use symbolic execution still must be able to reason about the relationships between inputs and outputs from calls to external functions. They must be able to reason how environmental side-effects and the logic of these functions affect which branch will be taken in the target program based on the input. This may be computationally expensive, lead to a state-space explosion, or an incomplete program exploration. For some tools an environmental model is still required in addition to the use of a concrete environment. Thus generally such methods are only able to partially overcome this limitation.

A common class of vulnerabilities is those involving improper or unsafe memory access. To detect these vulnerabilities with static analysis an understanding of the structure of virtual memory is necessary, which may be computationally expensive or infeasible to obtain. As discussed in Section 2.1.2, aliasing can lead to undecidability, which introduces the possibility of faults being missed in the analysis. The use of dynamic analysis allows us to overcome this limitation. Dynamic analysis provides us with evidence that the program will behave in a particular way, as opposed to static methods, which must derive a model of control flow from an analysis of the source. This allows us to avoid this undecidability, as we do not have to create a model of the program's memory from the source code alone, but rather are able to derive one from our observations of the execution of the program. As such, our model of the variables within the program describes them through memory locations instead of references to other variables. This allows for a more concrete model of the program's memory, as we model the program in terms of memory accesses instead of variable accesses. This greatly reduces the complexity of our analysis task. These problems motivated us to design a method for which we can observe how the program interacts with a concrete environment.

3.2.3 Scale versus Granularity

Vulnerability analysis methods must be able to balance scale and exhaustiveness. This typically involves a trade-off between the exhaustive checking of small programs and the non-exhaustive checking of large programs. Methods that place emphasis on the discovery of vulnerabilities [38] [47] [43] rather than the exhaustive validation of a program typically skip code and may introduce false negatives, which could potentially be dangerous, as described in Section 2.8. We have the opposite ideology in our method. We favor an approach that is over the code within the user-defined scope, using multiple degrees of granularity to scale. Furthermore, we favor over-approximation when performing data-flow and variable domain analysis to make it less likely that we will introduce false negatives in our analysis. This approach is perhaps less suitable for vulnerability discovery, but rather for a developer attempting to uncover faults in their code. It is conservative and will report on behaviors that are less likely to be exploitable vulnerabilities but are still potential weaknesses in the program.

Past dynamic analysis methods typically favor scale over granularity and are often coarse-grained [48], or require the program be executed with the exact input that triggers the fault to identify it [1] [56]. This makes the use of dynamic analysis methods less desirable. Firstly, a coarse-grained analysis is more likely to produce erroneous results, as it limits the understanding of the target program, specifically the context behind the occurrence of particular operations. Secondly, executing the target program many times may be time-consuming and computationally expensive, and may result in false negatives if the appropriate inputs are not used. To overcome this limitation we need a method with a highly granular abstraction based on control flow, which is able to describe all behavior on a path from a single input example.

3.2.4 Our Approach in Context

In this section we describe our approach to program analysis in comparison to notable past work in the area. We summarize this comparison in Table 3.2. It is worth noting that some of these areas are not explored in the corresponding works, so some of the assigned categorizations are up for interpretation. Within the aforementioned table we describe the features, attributes, and goals of 10 different methods including our own. We select these methods due to their popularity, similiarity, or because they are representative of a larger group. We define features as the capabilities of the tool, attributes as the techniques used for analysis by the tool, and goals as the purpose of the tool. We define each of the properties used to compare the selected methods in detail in Table 3.1.

There are a few notable trends in the table. Firstly, symbolic execution methods have similar properties as they largely have the same limitations. Constraint solvers may not be able to handle all relations or solve all path constraints, which may cause missed branches and other errors, and some do not support programs that use concurrency. The properties “Doesn’t require source code”, “Can handle concurrency”, “Doesn’t require an environmental model”, and “Handles environmental side-effects” are all features of methods that are described as binary analysis frameworks and use a concrete execution environment to observe the program’s interactions with the environment. Additionally, we note that our method is the only one that requires that input examples be provided, except for AFL which requires an initial input example that it mutates to reach new branches. This requirement is one of the main limitations of our method that others do not have. Finally, where our method differs from others is our focus specifically on vulnerability discovery rather than program exploration. Other methods, such as SLAM, MOPS, and KLEE, lack some of the same features as our method because they are not designed for the same type of analysis, making them difficult to compare. The difference between our approach and others is illustrated in our comparison as our use of a concrete memory representation, our property language, and our differing approach to state-space exploration that allows us to handle things like concurrency. We describe and evaluate our approach in depth in the remainder of this thesis.

3.3 Summary

To summarize, we believe that by creating a method that combines runtime verification and model checking, we can address some of the limitations of past methods. To the best of our knowledge, there is no other method that uses decompilation techniques to create a program abstraction then checks properties against it, and as we explain this approach offers unique advantages over past methods.

Property	Description
Can handle concurrency	refers to methods that are able to analyze programs that use multi-threading.
Interprocedural data-flow analysis	is the analysis of the flow of data between functions.
Provides a property language	refers to whether the tool offers a user-facing property language that is designed to allow for the user to describe arbitrary behaviors.
Tolerates undecidability	unknowns introduced into the analysis are handled. This is in contrast to methods that terminate, miss branches, or fail to describe the properties of the program in the face of undecidability[1].
Doesn't require source code	can complete its analysis without access to the program source[2].
Handles environmental side-effects	the analysis accounts for effects of external dependencies and the environment have on the execution of the target program.
Doesn't require an environmental model	refers to methods that don't require the creation of an environmental for the completion of their analysis. This may consist of specifications of components such as memory or the filesystem, or the implementation of handler functions that describe the domains of input and output values of external functions called by the target program.
Multi-language support	refers to methods designed to analyze programs written in different programming languages.
Doesn't require input examples	analysis can be performed without specifying which inputs to execute the program with.
Guaranteed full code coverage	refers to methods that are guaranteed to explore all paths through a program.
Control-flow model	uses a control-flow model to describe program behavior.
Concrete Memory Representation	uses a concrete memory representation in its program model.
Symbolic Execution	refers to methods that are primarily built upon symbolic execution.
Vulnerability-focused	refers to methods designed and implemented specifically for the discovery of vulnerabilities and other errors.

Table 3.1: Definitions for the properties used in our comparison of our method to related work in Table 3.2. [1] This includes methods that can't be run on programs that use particular libraries or functions because they produce nondeterministic outputs. [2] We include methods that analyze LLVM IR in those that require source code as the source code is compiled into this IR so we expect the source must available for the analysis.

	Hy2	S2E	AFL	Sys	Woodpecker	Ferry	MOPS	SLAM	KLEE	BitBlaze
Can handle concurrency	●	●	◐	◐	○	○	○	◐	○	◐
Interprocedural data-flow analysis	●	●	-	●	◐	●	○	◐	●	●
Provides a property language	●	○	○	◐	◐	○	●	◐	○	○
Tolerates undecidability	●	◐	-	○	○	○	◐	○	○	◐
Doesn't require source code	●	●	◐	○	○	●	○	○	○	●
Handles environmental side-effects	●	◐	◐	◐	○	◐	○	○	◐	●
Doesn't require an environmental model^1	●	◐	●	○	○	○	●	○	○	●
Multi-language support	●	●	○	○	◐	●	○	○	◐	●
Doesn't require input examples	○	●	◐	●	●	●	●	●	●	●
Guaranteed full code coverage	○	○	○	◐	○	○	●	●	○	○
Control-flow model	●	●	○	●	●	◐	●	◐	●	●
Concrete Memory Representation	●	◐	-	○	○	○	○	○	○	◐
Symbolic Execution	○	●	○	●	●	●	○	○	●	●
Vulnerability-focused	●	○	◐	◐	●	◐	●	●	◐	○

Table 3.2: A comparison of the properties of relevant related to Hy2. Color Definitions: 🟥 - Feature, 🟦 - Attribute, 🟩 - Goal. Symbol Definitions: ●= provides property; ◐= partially provides property; ○= does not provide property; - = Not applicable. ^{1} - refers to methods that don't require the creation of an environmental for the completion of their analysis, this consisting of specifications of components such as the memory model, or filesystem, or the implementation of handler functions that describe the domains for input and output values for external functions called by the target program.

Chapter 4

Method

In this chapter, we describe and justify the design of our method which we expand on in Chapter 5, which explains our implementation of Hy2.

4.1 Overview

In this section, we present Hy2, a method for uncovering vulnerabilities and other flaws in software binaries with the overall goal of improving their safety and security. Hy2 uses a hybrid of runtime verification and model checking, combined with decompilation techniques to check security properties against arbitrary command-line or systems applications. With our design of this method, we have the following goals:

- It aims to lower the bar for entry of vulnerability analysis, attempting to eliminate some of the configuration tasks and structuring the analysis so it can be performed largely without any knowledge of the inner workings of the program.
- It aims to provide a fine-grained abstraction of program behavior that is suitable for checking arbitrary vulnerabilities whilst still being scalable to larger programs.
- It aims to address the problem of accurately describing OS interactions and environmental side effects.

We show the series of steps performed by our method in Figure 4.1. We build our tool on a full-system emulator, which provides us with a complete view of the

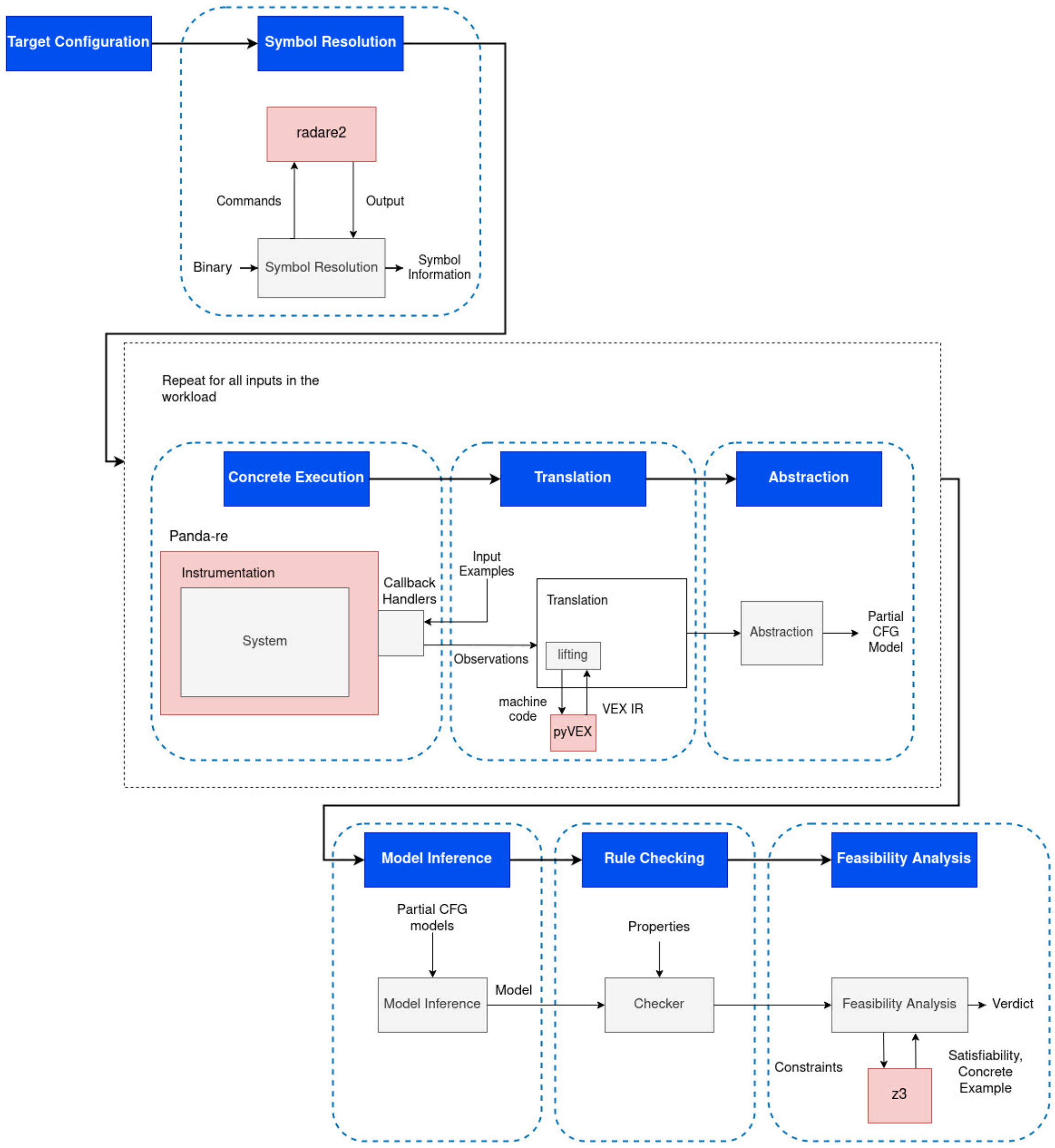


Figure 4.1: The sequence of steps taken by Hy2 in the completion of the analysis of a target program. The architecture of our framework is shown aligned with the corresponding step. External dependencies are highlighted in red and are as follows: for symbol resolution we use radare2 to extract information about the binary, in our process for analyzing single input examples we use panda-re for full-system emulation and instrumentation, and pyVEX for lifting, for our feasibility analysis we use z3 for constraint solving.

environment. Our method traces the execution of the target program on this orchestrated system over a set of inputs. It derives a control-flow-based abstraction from its observations. It builds on this model by performing a data-flow analysis to derive data dependencies between variables and uses a simplified CPU simulator to describe the behaviors of unseen branches. With the program model, it performs reachability checking on temporal safety properties. Finally, it performs a feasibility analysis step, walking backward through the error path to determine if it is possible and defines the conditions under which the error may occur. On top of our framework, we build a rich interface, with a property language allowing for the introduction of user-defined properties and semantics, a wide variety of configuration options to support multiple use cases and a web application to make it easier to interact with the underlying framework.

As shown in Figure 4.1, although Hy2 uses a few pre-existing tools to aid it in its analysis, a majority of it is our own. We chose to create our own binary analysis framework rather than extending an existing one, such as angr, BitBlaze, or S2E, as a key part of our method is that we wish to avoid the limitations of symbolic execution and many of these frameworks are built around it. Additionally, creating our own binary analysis framework gives us the freedom to design and structure our analysis as would be best suited for our specific goals.

We design our method using a combination of model checking and runtime verification techniques. We select this combination because of the valuable properties of dynamic analysis which are described in the previous chapter (Chapter 3). Runtime verification only explores the state space relevant to a particular set of executions, reducing the likelihood of a state-space explosion. We describe this in depth in Section 7.1. Combining this with model-checking techniques allows us to cover more of the state space than just what is observed at execution, allowing us to balance between exhaustiveness and scale. We describe how this is possible in Section 4.5.

Additionally, the use of formal logic that is pre-established builds confidence in our results. Finally, we use the idea of variable domain approximation presented in past works (Section 2.5.3), which reduces the size of the state space and allows us to reason about the behavior of the program even with only a partial model of its state-space. We describe this in detail in Section 7.1.

Unlike past methods, we embrace the concept of decompilation, allowing for the creation of a method that bridges dynamic analysis and model checking, as shown in Figure 4.2. In this figure, we show that rather than translating the source code into an IR, our method builds an IR up from the machine code. Using decompilation techniques allows our method to adapt to many use cases including the analysis of closed-source software. Describing the intersection between static analysis methods and our method is important because we wish to emphasize that these processes mirror each other and highlight the similarity between our abstraction and the one created by static analysis methods. Model-checking methods strictly use static analysis; we are posing our method as a hybrid between runtime verification and model checking and must explain why this is the case. Additionally with dynamic analysis such granularity is not always possible so it is important to highlight. If we were to extract basic blocks from the binary and perform the same analysis as we do on the basic blocks collected at runtime during the execution of the target program, the resulting model would have the same structure, and within each block have the same expressions. However, with dynamic analysis, the model only describes the subset of behaviors observed during execution and is augmented with concrete data.

In the remainder of this chapter, we describe our method in detail, including the architecture of our framework, our model of the execution environment, and the steps in our analysis process.

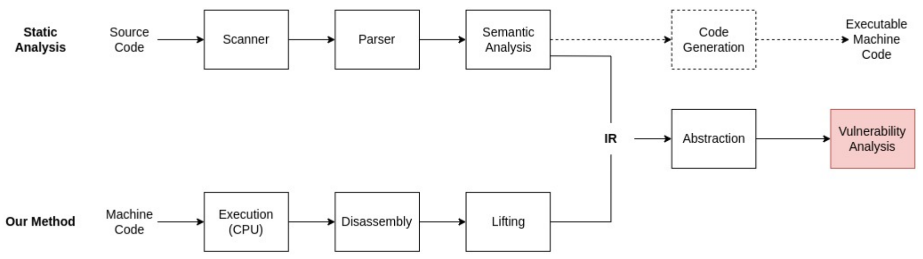


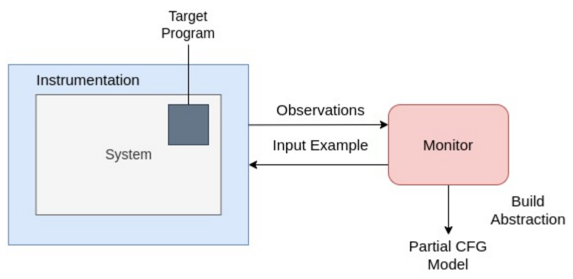
Figure 4.2: Comparing at a high-level the sequence of steps taken by our method to that used by static analysis methods for the purposes of illustrating their intersection.

4.2 Architectural Overview

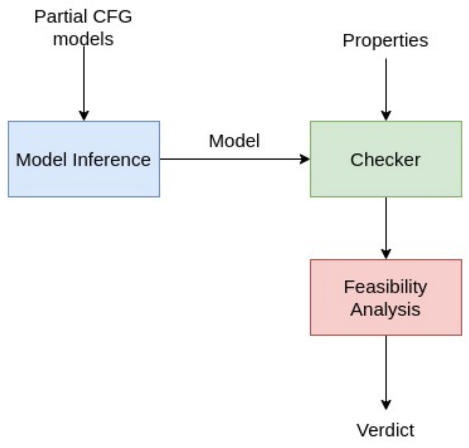
In this section, we describe at a high-level the architecture of our method. Our method is composed of two subsystems, as shown in Figure 4.3. The first subsystem analyzes the execution of a single input example creating an abstraction of its behavior which we consider to be a partial model. This subsystem has a similar architecture to that of runtime verification methods, using an instrumented system for its analysis and performing an analysis of a single execution trace. The second subsystem merges these partial models into a single model that describes all the observed behaviors across the execution of the entire workload and then checks properties against it. When performing an analysis of a program, a user would first exercise the target program for all inputs in the workload creating partial models using the first subsystem, and then create and analyze a model of the program with the second subsystem.

4.3 Environmental Model

In this section, we describe our environmental model, its limitations, and how we structure our analysis around it. One of the motivating factors of our analysis is the flexibility to handle a variety of use cases, as well as strike a balance between scale and granularity. We describe below how our environmental model helps us reach this goal.



(a)



(b)

Figure 4.3: The architecture of our framework consists of the following two subsystems: (a) shows the subsystem for analyzing a single input example and (b) shows the subsystem for analyzing the target across all input examples, from the output from the first subsystem across all input examples.

4.3.1 Definitions

In this subsection we provide definitions for terms we use when describing our method. These terms are as follows:

- Guest System — refers to the operating system being run in the emulator.
- Environment — refers to everything running on the system except the target program.
- Target Program — refers to the program being evaluated. This may not always be a single program, but could be multiple different programs, libraries, or part of the system itelf.
- Input example — refers to an input used to exercise the target program. In this case exercise means to trigger the execution of some part of the target program. Input examples should be designed to exercise the different functionalities of the target program. In this case they should take the form of a command that can be run in the terminal.

- Workload — refers to the set of input examples used to exercise a target program.

- Security Property or Rule — refers to a description of undesirable behavior defined in our property language that can be checked against the model of a program. These properties can be used to describe behaviors that cause vulnerabilities.

4.3.2 Assumptions

To better structure our analysis, we base our model of the environment on the following assumptions:

- **System:** We have a guest system for which we assume we don't know its inner workings but for which we can potentially trace any arbitrary slice of its behavior. Generally we use instructions to describe behaviors and anything that occurs within the system can be described with instructions, so therefore we can describe any subset of the system's execution. Through the use of full-system emulation, we have visibility into the behavior of the entire system, however, we assume that analyzing all of it at an instruction level at any time is infeasible. Even within a single user-space process, the number of instructions increases exponentially when we consider the instructions executed by the functions in the libraries it makes calls to and in the kernel for any system calls made. In reality, although it may be possible to analyze the entire system, it would be extremely inefficient as only a small percentage of behaviors would likely be relevant to the analysis of a target program. Thus, we must be precise when describing the bounds of the program under test. We focus on describing the process or processes that correspond to the execution of the target program. However, since one of the main motivations behind the design of this method was to provide an approach that is generic enough to be applied to almost any target program, we leave it largely up to the user how exactly to define the scope of their analysis. The scope of analysis is limited by computational resources rather than the method itself.

- **Processes:** Each process has an address space containing executable code and data. The program's executable is mapped into memory and within this memory mapping, there are pages. With these pages, we can describe the range of addresses used to map the program's executable code into memory, locating it within the process and isolating its execution. This approach to describing the scope of the analysis can also be applied to shared libraries, as they are executed in the context of the process' address space.

- **Within a Process:** Within a process' address space, we can see its behavior, the behavior of the libraries it makes calls to and also the behavior of the underlying system. For a user-space process, we divide its execution into two parts. Firstly, we have the instructions corresponding to the target binary, which describe exactly which operations it performs during its execution. Secondly, we have calls to libraries and system calls, these describe how it interacts with the system from the perspective of the target program. We make the assumption that we can describe, at a very high level, the operations performed by these calls, rather than what they do specifically, essentially treating them as a black box. For instance, for the case of the `fopen()` call, we assume it is only important that it is opening a file rather than how exactly `fopen()` performs this operation. We assume that the operation is completed as the program specifies, meaning that the library actually performs the operation as described by the input arguments. Since we can see the resulting behavior in the context of the target program, we have evidence that it does. As there may be cases where we wish to examine the behaviors of external functions in our analysis, our framework is flexible enough to adapt. For instance, we can modify our program scope, so our framework only analyzes code executed in the kernel in the context of the target process. This flexibility allows our method to adapt to a variety of potential use cases.

This brings us to the need to handle different types of targets differently within our framework. One such case is for libraries. We see the need to treat libraries as a separate case from executable programs as they have a different structure. For the case of user-space programs, we limit our scope to instructions executed by the associated executable and consider calls to libraries to be out-of-scope, recording the name of functions called and their arguments but not evaluating any of the behaviors of the libraries themselves. Libraries can not be handled the same way as these programs, as they do not necessarily have a single entry and exit point. We treat calls to libraries as separate execution traces, as illustrated in Figure 4.4. These traces begin with a call to the library from a program and end when the execution flow returns to the program. Execution traces with the same entry point are considered to have the same initial state and become part of the same model (Figure 4.5a). Thus, we may have multiple models for a single library (Figure 4.5b). With this method, we can truly partition the behavior of the library from the program making calls to it.

Finally, through our use of full-system emulation, we have the ability to model behaviors that span multiple processes or threads. Our method is able to recognize and model the switch in execution between different threads or processes. Although we are able to model behaviors executing in parallel, we have not taken steps to account for all possible interleavings. Instead, we only account for those that can be derived from our observations. For example, if we observe the sequence of events A, B, A, C, we would not consider sequences containing events $C \rightarrow A$ and $C \rightarrow B$,

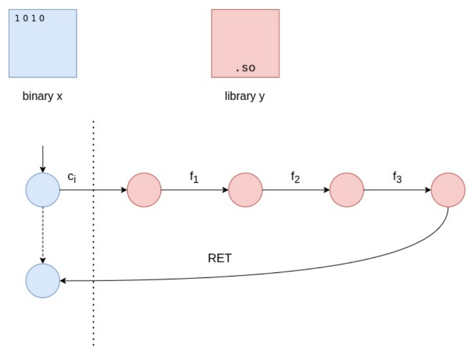


Figure 4.4: Our model of the separation between external dependencies (i.e. libraries) and the target program.

because we have no evidence that A follows C or B follows C allow these sequences may in fact be possible. Although it would be possible to create an exhaustive model describing all possible interleavings, it was decided that this would be left for future work.

4.3.3 Scope

As described previously, the use of full-system emulation gives us a great amount of flexibility with regards to the scope of our analysis. As part of the configuration process, we allow the user to define the scope of the analysis within the entire system. For what we consider a standard use case, this being a program binary, the scope by default is the execution of the target program and the environmental effects that are visible to the target program.

We expand on the idea of flexibility by allowing our analysis to extend to slices of arbitrary libraries. Full-system emulation allows us to derive a model of the behaviors of arbitrary libraries at runtime, rather than explicitly requiring this information to be provided prior to analysis through an environmental model.

Consider Listing 2. In this example, for vulnerability analysis methods that require an environmental model, the user would have to have knowledge of the arbitrary library and be able to precisely describe its functionality. This may be infeasible

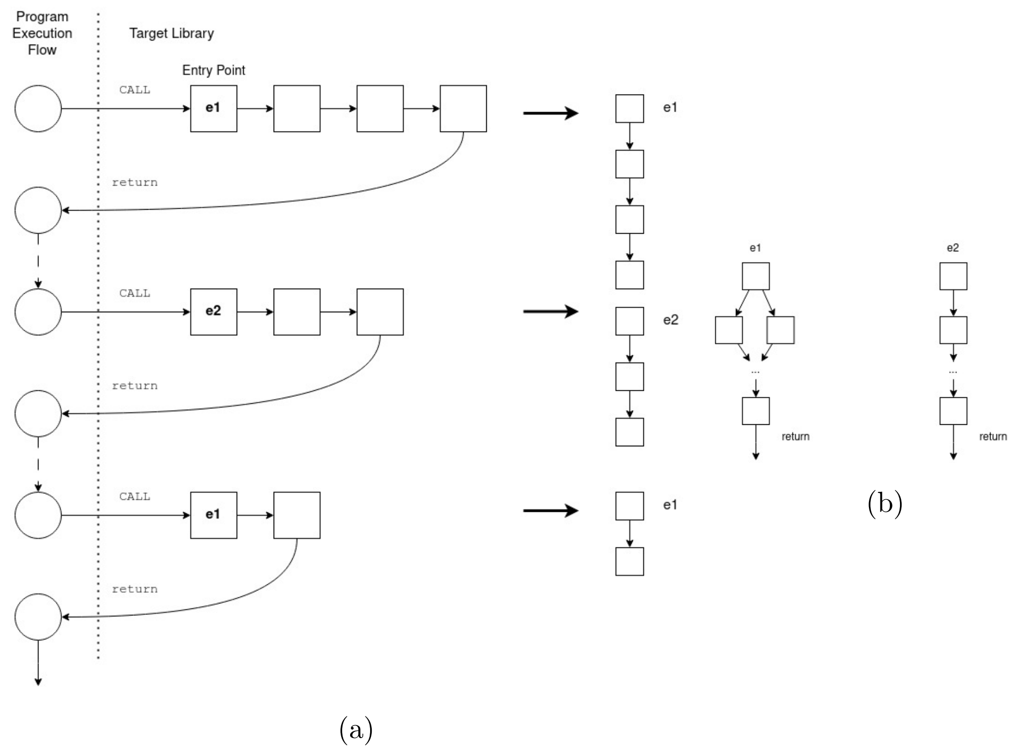


Figure 4.5: Our approach to modeling the behavior of libraries where (a) shows the conversion of behaviors within a library resulting from calls to it made by another program to separate event sequences and (b) shows the conversion of these sequences into separate models.

```c
#include <arbitrarylib.h>

int convert(char *buf, int mode){
    arbconvert_t *a;
    arbformat_t *f;
    ...
    a = arbitraryConverter(mode);
    if(!a){
        return -1;
    }

    f = a.convert(buf);
    //unsafe operation on f
}
```

Listing 2: An example to illustrate the use of an arbitrary external library, and how its unsafe use can effect the target program

in many cases. By allowing our execution flow to jump to and analyze arbitrary functions within this library, we can identify further vulnerabilities resulting from interactions with it without any significant user effort. If for instance there is a case where a fault is caused by the program not validating the input passed into the library function, causing undesirable behavior within the scope of this external function, then this additional functionality would allow for it to be uncovered without requiring that the user have any knowledge of its specific functionality. Additionally, the ability for Hy2 to perform static analysis on unseen branches extends to libraries the program interacts with, allowing us to thoroughly analyze external functions used by the program.

4.3.4 Granularity

Finally, we discuss the granularity of our analysis. By default, there are two levels of granularity used in our analysis: instruction level and call level. Instruction-level granularity is used for the behaviors within the target program, whereas the call-level granularity is used as an abstraction of behaviors that are within the target process but not within the target program. Call-level granularity means we create an abstraction of program behavior using calls to shared libraries and system calls. In some scenarios, this abstraction may be sufficient to describe program behaviors, but past evidence shows that this approach is prone to spurious errors [32]. Thus, we use a two-level approach to granularity, allowing for this method to precisely analyze the functionality within the program and account for its interactions with operating system components. Methods that use static analysis, including those that use symbolic execution, take steps to extensively model the behaviors of the environment including the behaviors of certain standard libraries. We do not do the same, instead, our approach expects that these semantics be incorporated into the definition of the security properties. For instance, in the case of repeated memory allocations and frees, the memory manager will reuse recently freed chunks for efficiency, so it could be the case that a call to $free(x_1)$ followed by $x_2 = malloc(n)$, $x_1 = x_2$, and so any subsequent operations on x_2 would be on the new variable, not the freed x_1. This should be accounted for when describing, for instance, properties involving free. Although this places more responsibility on the user, it eliminates any restrictions as to what we can and can't reason about, as we do not require knowledge of the behavior of each external function. It additionally allows for security properties to introduce application-specific semantics independently of the underlying framework. We also allow for the parts of the program's analysis that have a particular granularity to be configurable. Through our configuration interface, we allow for inclusions and exclusions to be specified. Exclusions are analyzed with call-level granularity and inclusions are analyzed with instruction-level granularity. For calls to unknown libraries, we introduce additional functionality that allows for the user to perform instruction-level analysis of the functionality of individual library functions and describe their functionality, as described in the previous subsection (Subsection 4.3.3) and expanded upon in Section 5.2.

4.3.5 Usage Scenarios

We design our framework to be able to handle a wide variety of use cases. Primarily we consider that the user may not always have access to the same amount of information about the target program. For instance when analyzing a closed-source program the user may not have access to the source and may only be able to provide the program binary. The amount of information provided by the user may have an impact on the depth of analysis Hy2 is able to perform. We envision the following scenarios:

- **User does not provide the program binary** — In the most extreme case we have not been provided with the target binary, and we analyze it solely through its execution on the guest system. This may occur when the target is a component that cannot be separated from the system. When a process is started we get the process mapping, record the addresses of the target pages, trace the execution within these pages and any calls to libraries, or system calls made within the process. In this scenario, we can still check arbitrary properties against our model, except for those that require static data or semantic data. Results will not provide information such as function names, or lines in the program source.

- **User provides the program binary** — We extract function names from the binary and are able to match the blocks executed to a function in the binary. We can match discovered undesirable behavior to locations in the binary. With the binary we are able to perform the optional static analysis step, greatly increasing our program coverage.

- **User provides the program binary with Debugging Information** — In addition to function names, we extract type information from the binary. This type information includes the sizes of static fields (i.e. `char buf[40]`). This serves to eliminate false positives arising from memory access operations on statically allocated memory, as we must infer the size of these fields. We can match discovered undesirable behavior to locations in the source code. This makes it much easier for the user to trace the results back to a particular location in the source code.

- **User provides the program binary with Debugging Information and Header Files** — In addition to what was previously described, we collect information about structs and create a mapping of the fields within structs. This serves to eliminate some of the false positives that may arise. We collect information about constants that describe specific semantics within the program, which can be used to make it easier to describe program-specific properties.

As we have described above, with the addition of each new source of information we have improved our ability to analyze the program and effectively communicate the reasoning behind our results to the user. In the event that the user does not provide the binary Hy2 is still able to analyze the target but it is not able to provide the same depth of analysis as the other scenarios and is more likely to produce spurious results compared to other options. With the exception of the optional static analysis step, we do not lose any features between these scenarios but rather are able to provide more accurate results. As one of the goals of our method is to be generic, this will allow us to accommodate a much wider range of target programs than if we were to require that the user provide source code.

4.3.6 Use Cases

In this subsection, we describe the potential use cases for our framework and describe which usage scenario they may fall under. These use cases are as follows:

- Analysis of a closed source program or library — As our method does not require source code, it has the ability to analyze closed-source software. This use case would likely fall under the "User provides the program binary" scenario described previously. Since our method requires input examples to be provided, some knowledge of how to interact with the target program is required. Additionally, when debug information is not provided the results may be difficult to interpret as the program is described in our IR and machine code, so the user would likely have to have some binary analysis capabilities.

- Integration into a development cycle — Our method could be integrated into a development cycle. In the event that the project performs functional testing, we see the potential for these tests to be repurposed for use as the workload in our framework. A developer would likely be able to provide debug symbols and semantic information so this would fall under the “User provides the program binary with debugging information and header files” usage scenario. Any errors identified would be mapped to the corresponding lines in the source, making them easy to interpret. Additionally, the created configuration and workload for the program could be reused across updates.

- Checking the quality of a patch — We see the potential for our method to be used to check the quality of a patch. Using a subset of test cases, a user could test the affected branches, testing if the vulnerable behavior is still possible. This would likely be similar to the “Integration into a development cycle” use case described above.

- Black-box program exploration — We see the potential for our method to be used for a black-box program exploration, that is analyzing the behavior of a few input examples for a program the user has no knowledge of. For instance, an analyst may wish to test a few inputs of interest, specifically with the goal of identifying vulnerabilities. As the user has a set of inputs they wish to test analysis would require very little user involvement and could be largely automated. This could potentially fall under any of our defined usage scenarios.

- Analysis of interactions between components — As our method has the ability to analyze multiple processes on the guest system, we see the potential for our method to analyze the interaction between processes or different components. This could also potentially fall under any of our defined usage scenarios.

- Analysis of firmware images — We see our framework as, with further development, being useful for analyzing firmware images. In past works, such as that of FirmAE [64] they demonstrated that full-system emulation was an effective method of analyzing firmware binaries automatically. This could be a natural extension to this method. This would fall under the "User does not provide the program binary" scenario as we would be analyzing the system as a whole.

To conclude, the use of full-system emulation allows for the tracing of environmental side effects that are observable to the target program to be configured based on the particular use case.

4.4 Program Abstraction

In this section, we describe our process for building an abstraction of program behavior. This includes configuring and executing the target program and interpreting execution traces to build a control-flow-based abstraction.

4.4.1 Target Configuration

To support our goal of providing a method that can adapt to a variety of use cases, we introduce a wide range of configuration options. These include options for selecting the guest image and target pages and specifying inclusions and exclusions to refine the scope of the analysis. Additionally, the user must provide a set of inputs to exercise the target program and select which properties to check against it. We expand on the different configuration options offered by our framework in depth in Section 5.2.

4.4.2 Symbol Resolution

To address some of the limitations of dynamic analysis, such as the fact that data like type information is not preserved at compile time, we have an optional step to introduce this information and other human-readable symbols into our analysis. We create a table mapping addresses of basic blocks to function names. For each block we observe during the execution of the program, we perform address translation using the base address of the corresponding page in memory and the base address of the binary. Within this mapping, we also store the sizes of each block, which we use to extract new basic blocks from the binary in our static analysis step. We allow for the introduction of arbitrary type information manually and through the inclusion of header files or debugging symbols. We apply this information to variables derived during concrete execution by introducing additional attributes to describe type information. This step serves to improve the readability of our analysis as well as allows us to check type-specific properties or properties with high-level semantics.

4.4.3 Concrete Execution

Our method uses dynamic analysis to analyze the behavior of a target program produced from a given set of inputs. We choose to base our analysis on concrete program executions because it allows us to observe environmental interactions and building our model largely from concrete observations largely eliminates the need for inference about the behavior of the program as we have concrete evidence that it behaves a certain way. We perform this analysis using full-system emulation, as it gives visibility into the state of the entire system and low-level operations, including system calls, library calls, and virtual memory accesses. We perform the following observations about the target:

- At program runtime, we fetch the raw bytes of the machine code corresponding to the basic blocks from the CPU prior to its execution, as shown in Figure 4.6. We store these blocks in a mapping, storing the machine code of a block, indexed by the program counter (pc) and size. We pair this with a sequence describing the order in which these blocks were executed, pairing a block identifier with the thread ID of the thread that was executing it. Retrieving the bytes corresponding to basic blocks executed from memory at runtime, instead of from the binary itself, allows us to precisely observe exactly which parts of the program were executed and nothing more, which is a valuable property. Additionally, we can analyze a target program even if we do not have a copy of its executable or we cannot separate it from the system. The machine code extracted at runtime is the same as what is in the binary, so the functionality described is the same as if we were to perform static analysis on the binary.

- We record the values of the registers prior to the execution of any basic blocks that are within the scope of the instruction-level analysis for the target program.
- We collect information about the pages mapped into process memory, including their names, size, and addresses.
- We record any calls to libraries and system calls made by the target and their arguments, tracking this data by the index of the most recently executed basic block in the sequence with the corresponding thread ID.
- We trace virtual memory reads and writes, recording the program counter (pc) of the operation, the address accessed, the value read or written, and whether the address is on the stack or heap. We store this information with the corresponding basic block instance. Due to the large amount of data this callback produces, we write this information to a file stream instead of storing it in memory. This serves as an abstraction of the program's memory throughout its execution, which is used to populate the values of variables in the model we create in subsequent steps. This abstraction takes the form of a lookup table, which can provide the value at a given address at any given time in the program's execution.

We use the information recorded during our subsequent analysis steps, in which we transform our runtime data into a higher-level abstraction. This abstraction takes

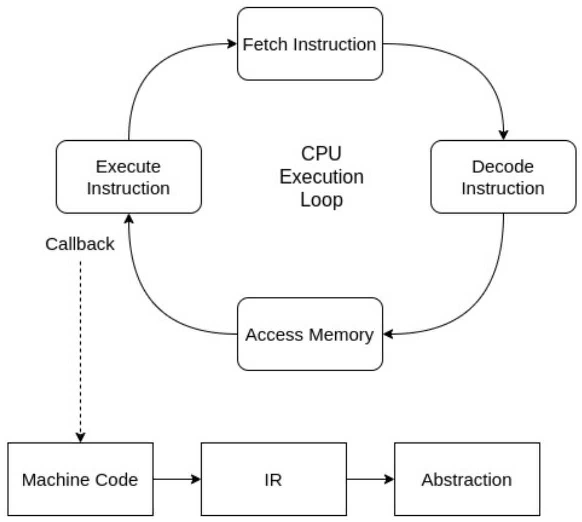


Figure 4.6: The relationship between our analysis process and the CPU execution loop.

the form of a control flow graph which describes the program's behavior during that particular execution. The nodes in this graph correspond to the basic blocks we observed being executed and the edges are derived from the order that these blocks were executed in.

4.4.4 Translation

In this section, we describe our first step in transforming runtime data into a higher-level abstraction, this being the translation of machine code to a higher-level IR. We use an IR because its semantics and composition are consistent across different programs and the structure of its expressions is ideal for our analysis, which we expand on below. The process performed in this step is comparable to the variable operation analysis and data-flow analysis steps in the decompilation process described in Section 2.1.4.

From the program's execution, we have the following data: the set of basic blocks executed as raw bytes, the sequence of basic blocks executed, a mapping of registers, a mapping of virtual memory, a mapping of libraries and system calls made, and a

Name	Description
Get(x)	Get the value stored in register x
LOAD(x)	Load the value store in memory at effective address x
y = Put(x)	Put value x at register y
STORE(y, x)	Store the value x at effective address y
Cmp	Condition, proceeds a conditional jump

Table 4.1: The set of operations that make up the core of our IR.

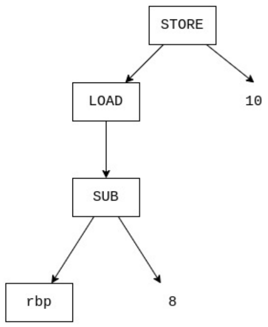
mapping of pages within the process. For each unique basic block whose execution we observed at runtime, we have its raw machine code as a sequence of bytes. We translate this into an intermediate representation (IR), which serves as a low-level abstraction of the program's behavior on which we base our analysis. Each basic block executed when converted to IR results in a series of expressions in the form of `X = EXPR(Y)`, where `X` is a variable, `EXPR` is an operation, and `Y` is a dependency. We describe the core operations of this IR in Table 4.1.

After this translation step, for each basic block, we have a sequence of IR expressions, which we refer to as an IR block. Within each IR block, we have a sequence of operations such that those executed later may depend on the result of those executed earlier. We reduce this set of expressions into a set of expression trees. To perform this reduction we perform a process akin to "Variable Operation Analysis" described in Section 2.1.4, eliminating intermediate expressions by merging them into a single more complex expression. All expressions that have no later expressions depend on them as the root of the trees, and intermediate operations become child nodes. For example, for the expressions defined in Figure 4.7, we define a single variable `t3 = STORE(LOAD(rbp, 8), 10)`, which means store the value 10 at the address stored at `rbp-8`.

These expression trees always have either Put, Store, or a conditional operator at their root, which demonstrates that this process removes expressions that do not impact the program state, as Store indicates a change in the state of a program

```
t1 = SUB(rbp, 8)
t2 = LOAD(t1)
t3 = STORE(t2, 10)
```

(a) IR Expressions

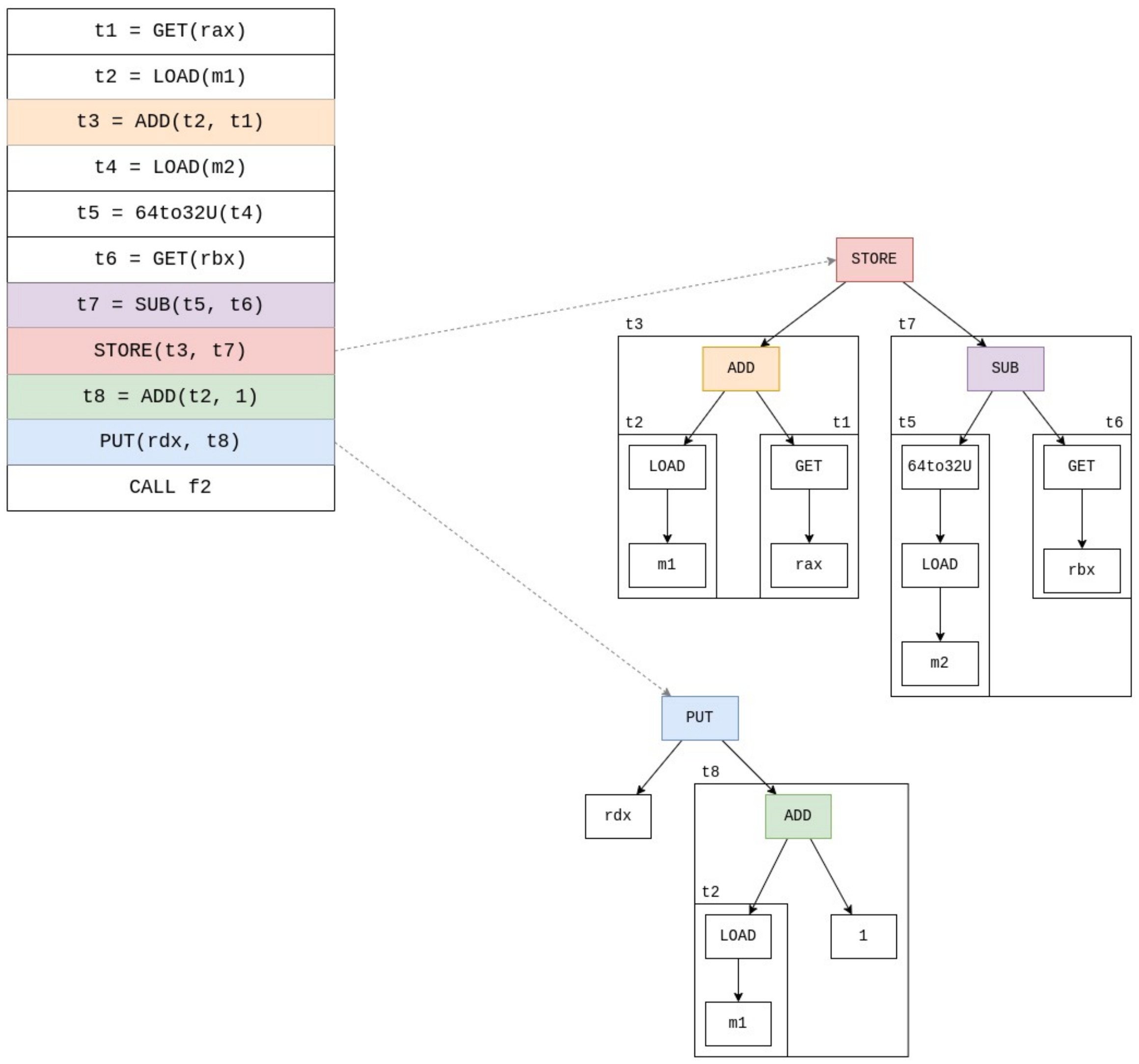


(b) Expression Tree

Figure 4.7: An example showing the merging of a set of IR expressions shown in (a) into an expression tree shown in (b) to eliminate intermediate values.

variable, Put corresponds to setting the values of registers for function calls, and comparison operators are used to control execution flow. We refer to these expression trees as variables or variable expressions, as they are comparable to the variables defined in the program source. In Figure 4.8 we show the simplification of the set of IR expressions over the entire block. The two expressions resulting from this reduction (`mt1 = STORE(Add(t2, GET(rax)), Sub(64to32U(t4), GET(rbx)))`, and `rdx = Put(Add(t2, 1))`) are both significant in describing the program state, as one describes a change in the state of memory, and the other a change in the state of the registers, and thus can be used describe the behavior within the entire block.

Additionally, we treat load instructions or virtual memory reads as significant, creating an additional type to describe their behavior. These operations are treated as significant because they describe a great deal about the nature of a particular variable. For example, for function arguments, if a load operation is not performed before passing a pointer to a function we know we are passing that variable by reference instead of by value. If a condition is in the form: `Cmp(v0, v1)`, where `v0` and `v1` are variable expressions, we can determine that this condition is variant, rather than invariant if it were in the form `Cmp(v0, c)`, where `c` is a constant. The structure of these expressions can also be used to describe whether a variable is on the stack or



```
t1 = GET(rax)
t2 = LOAD(m1)
t3 = ADD(t2, t1)
t4 = LOAD(m2)
t5 = 64to32U(t4)
t6 = GET(rbx)
t7 = SUB(t5, t6)
STORE(t3, t7)
t8 = ADD(t2, 1)
PUT(rdx, t8)
CALL f2
```

Figure 4.8: An example showing the conversion of a set of IR expressions within a basic block to a pair of expression trees

the heap, and if it is at an offset in a block of memory such as a string or an object.

For each block, we have a set of expression trees describing the operations that occur within the block. We design this set of expression trees in such a way that it can be applied as a stateful function, describing the change in program state when the block is executed. This design allows us to pass in the previous program state and derive the state after the execution of the particular block. Operators within these

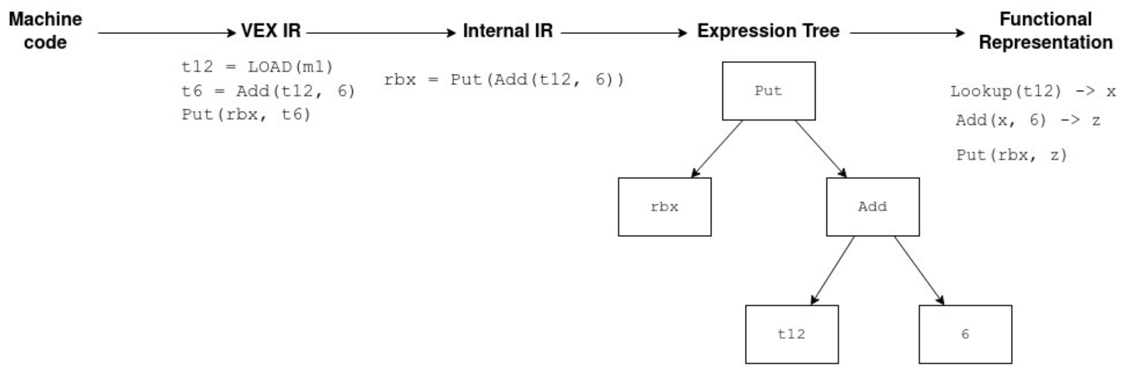


Figure 4.9: An example illustrating our translation process.

expression trees are mapped to a set of handler functions that perform the corresponding low-level operations. We liken this functionality to a CPU simulator, as we are able to simulate the behavior of any given block with the program in a particular program state. This additional functionality serves two purposes. Firstly, since we only record the values of the registers before the execution of a particular block, the functional representation of the block is used to determine their updated values if they are modified within the block. For example, if we have: `rdi = Put(Add(t12, 2))`, where `t12` is a value loaded from memory, we use this functionality to compute the value of `t12 + 2`. Secondly, this functionality is used to partially address the problem of coverage through the addition of a static analysis step, as described in Section 4.4.6. Additionally, this functionality could potentially be extended further to explore the behavior of the program on unseen input values, as this has the ability to simulate the execution of the program on arbitrary input values. With this step we have described the operations that occur within the basic blocks in our target program, now we must describe the relationships between them.

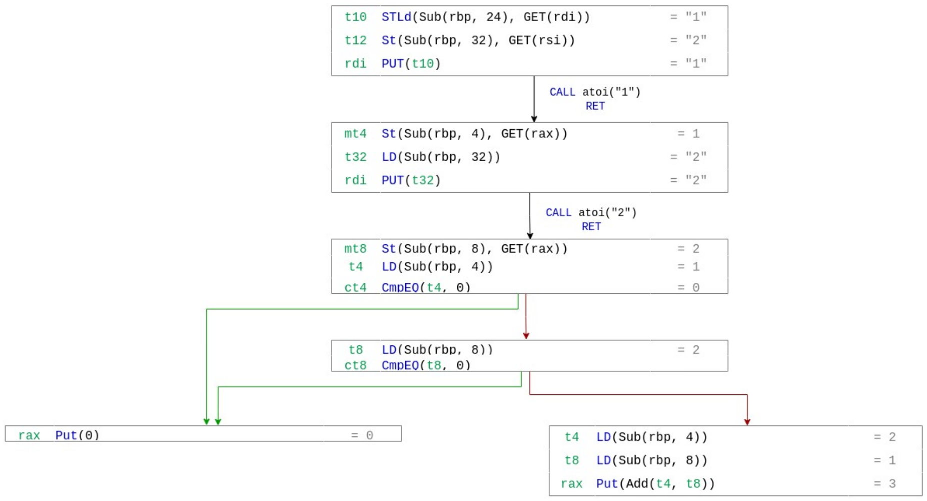
4.4.5 Abstraction

In this subsection, we describe the second part of the process of creating an abstraction of a single execution of the target program. This process involves reducing the sequence of basic blocks triggered at execution into a control-flow graph (CFG).

We use a CFG, rather than representing each program execution as a linear sequence of basic blocks executed, as storing runtime data for each occurrence of a basic block is largely redundant and introduces a substantial overhead for most programs. Further, for programs that perform a file encoding or conversion, a linear representation could be very large as it would be proportional to the size of the program's input, in this case a file. By representing each program execution as a CFG, we have largely eliminated this overhead. A CFG model is able to efficiently describe valid orderings of events within the program and capture constructs such as loops and conditional statements.

To minimize the overhead of the process of building the CFG, we first construct the graph and then introduce the corresponding expressions and data at each of the nodes. Initially, each node is represented by its address, and each edge is represented by a destination address and type of jump statement, either Call, Return, or a Condition. With a few exceptions, we consider nodes with the same pc to be equivalent and merge them. These exceptions are nodes with a different transition type (e.g. a conditional jump statement versus a return statement), or nodes that make a call but not to the same destination, as we consider in these cases the program to be in a different state.

In our final step, we introduce data describing the expressions that are present within each node in the CFG as well as data describing calls to external dependencies, namely shared libraries. Throughout the execution of the target program the control flow jumps to and from external dependencies such as shared libraries. We describe the behavior of these operations with the associated library name, function name, and the arguments the function was called with. This results in a call trace describing all the library calls made, from when the control flow leaves the target program to when it returns. For example, the function `atoi` calls `stroq`, we will see both operations in the associated call trace. We include these calls as part of the structure of the node



```
t10  STLd(Sub(rbp, 24), GET(rdi))        = "1"
t12  St(Sub(rbp, 32), GET(rsi))          = "2"
rdi  PUT(t10)                            = "1"
```

```
CALL atoi("1")
     RET
```

```
mt4  St(Sub(rbp, 4), GET(rax))           = 1
t32  LD(Sub(rbp, 32))                    = "2"
rdi  PUT(t32)                            = "2"
```

```
CALL atoi("2")
     RET
```

```
mt8  St(Sub(rbp, 8), GET(rax))           = 2
 t4  LD(Sub(rbp, 4))                     = 1
ct4  CmpEQ(t4, 0)                        = 0
```

```
 t8  LD(Sub(rbp, 8))                     = 2
ct8  CmpEQ(t8, 0)
```

```
rax  Put(0)                              = 0
```

```
t4   LD(Sub(rbp, 4))                     = 2
t8   LD(Sub(rbp, 8))                     = 1
rax  Put(Add(t4, t8))                    = 3
```

Figure 4.10: An example of the control flow for a function that takes in two strings, converts them to integers and adds them together, returning the result.

in the derived CFG, as shown in Figure 4.11, with there being an edge from the node where the control-flow leaves the target program, to where it returns to the target program. We choose to omit the basic blocks that correspond to the stub functions in the procedure linkage table that make the calls to these external functions from our model because they do not describe any functionality specific to the program.

For each node, we define a set of variables corresponding to the set of expression trees we derive for each block described previously which we store in each node. We perform the following 3 steps for each variable expression in a given basic block:

- Lookup and Compute: If the operation performed consists of a memory access operation, we use our virtual memory mapping to look up the values and addresses accessed at that particular program point during program execution. If the operation performed involves register reads, we compute the value at that particular time step using our functional representation of the variable expression and the current program state. We aggregate the values and addresses observed across all executions of the basic block.

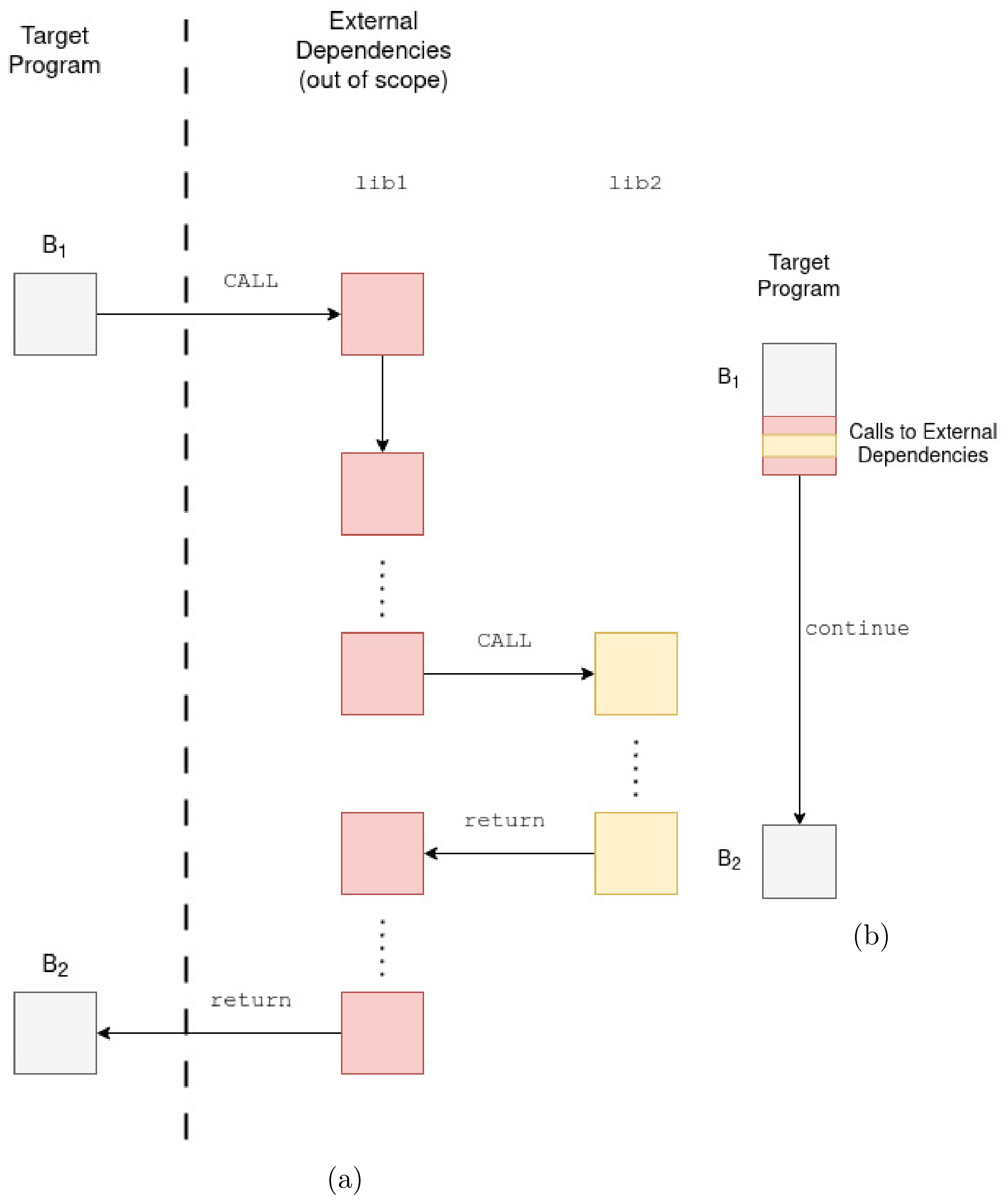


Figure 4.11: Illustrating how we handle calls to external dependencies, such as calls to libraries. Figure (a) shows how calls to external dependencies appear during a program's execution, with control flow leaving the target program when it calls lib1, and leaving lib1 when it calls lib2. Control flow returns to lib1 from lib2 when it reaches the corresponding return instruction, and then from lib1 to the target program. Figure (b) shows how calls to external dependencies are handled in our program abstraction. These calls are represented with call-level granularity, with calls made by lib1 and lib2 being included in the abstraction of the basic block B1, and the edge labeled "continue" to indicate the control flow continues to B2 without a jump statement.

- Derive Dependencies: We compute an approximation of data flow for each variable expression. We consider two types of dependencies, local and global. Local dependencies occur within the same basic block and global dependencies occur anywhere within the program. Our approach to identifying dependencies is designed to be as simple as possible. Initially, we consider dependencies arising from data being written and read from the same address in memory. We later refine these dependencies in subsequent analysis steps. Because we don't represent constraints on data when we aggregate values across executions of a block, this method is an over-approximation and may introduce spurious results later However, it doesn't fail to represent a relationship between variable expressions if one exists, which is consistent with our motivations as discussed in Chapter 3.

- Flatten: We pair the concrete values for each variable expression with a simplified representation of its expression tree. To represent more complex relationships between variables, not lose the structure of the original expression, and improve searchability and portability, we flatten each IR expression tree into a vector. We use this vector to describe how variable instances are related. For example, for the expression `v = STORE(Add(x, 2), y)` we would convert it to the following vector: `["St", "D1", "Add", "D2", "x", "D2", 2, "D1", "y"]`.

With the variable representation selected, it would be very easy to perform additional analysis steps to derive constraints on said variables, however, this process would incur substantial overhead and is not necessary for the completion of our analysis. For each execution of the target program, we produce the described control-flow-based abstraction and save it to a separate file, allowing for any subset of the workload to be modeled and evaluated.

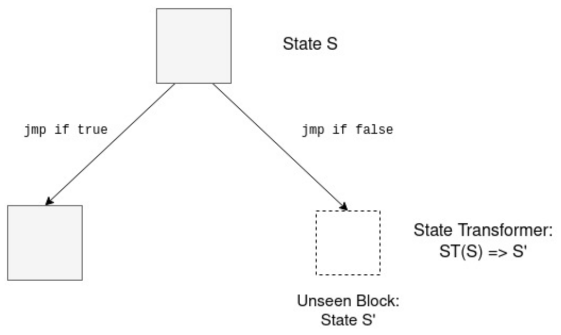


Figure 4.12: Demonstrating the flow of data from a seen to an unseen branch, using a state transformer to mutate the observed program state.

```
1 char *buf;
2 ...
3 buf = malloc(n);
4 if(!buf){
5   log("Error: Memory not allocated");
6 }
7 buf[0] = 'a';
```

Listing 3: Our motivating example for the introduction of hybrid analysis. The point of this example is that we wouldn't be able to determine that there is a potential null pointer dereference error on line 7 without knowing the behavior on line 5, which we are unlikely to observe during the normal execution of the program.

4.4.6 Augmenting our Method with Static Analysis

In this subsection, we describe how we integrated static analysis into our abstraction and translation steps to improve coverage. Unlike other approaches to static analysis, this analysis is based on observations made during our concrete execution step, rather than an analysis of the source code.

As previously described in our translation step, we derive a function describing the transformation performed on the program state within each basic block, which can be applied to the current program state S to derive the next program state S', where the program state describes the state of registers and virtual memory. We realized that this functionality could be used to extend our analysis to unseen branches, deriving the values of variables for unseen blocks.

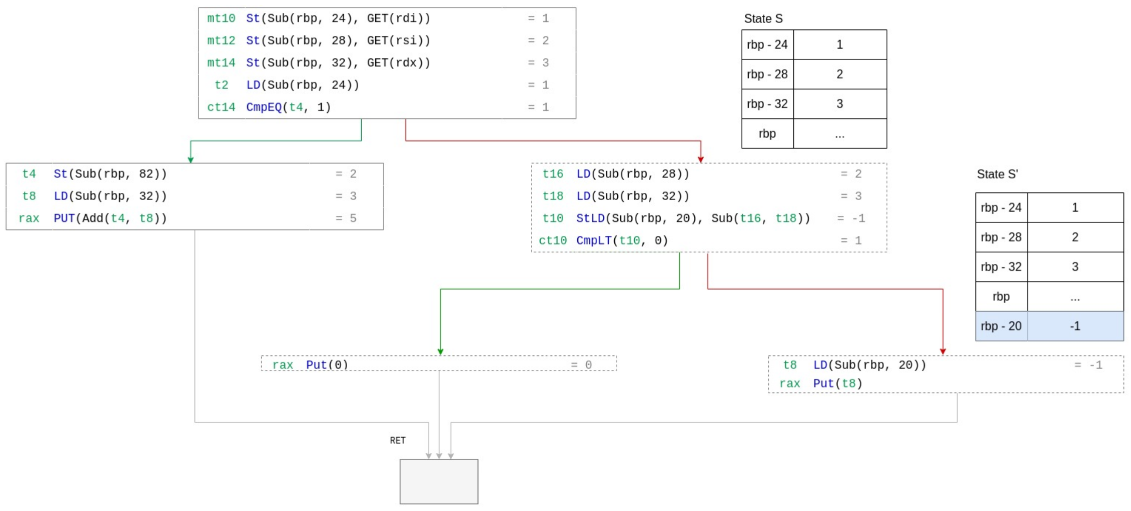
In our translation step, when we reach a conditional jump, we record the address of the basic block of the conditional jump on the branch we did not visit, this being the start of the unseen branch. Then for each of these unseen branches, starting at the block that the conditional jumps to, we iteratively perform the following steps for each unseen block:

- We extract the bytes of machine code of the corresponding basic block from the program binary.
- We translate these bytes into IR, then reduce this IR into our expression tree-based representation.
- We determine the address of the next block, and if it corresponds to a branch we observed during execution, or we are unable to determine the address of the next block, we stop iterating and record the address where the branch converged with the control flow observed at runtime. Otherwise we continue, repeating these steps for each new block.

In the abstraction step, we insert these new branches into our CFG model by connecting these new branches at the conditional jump and the node where the branch converged with our observed model. For each block in these branches, we extract the program state after the last block before the unseen branch and use it as input to our derived state transformer, as shown in Figure 4.12. We solve for each variable and record any changes to the program state. This new program state becomes the input for the next block's state transformer, continuing along the branch until it converges with the observed execution flow.

As we diverge from the observed branch, it becomes more probable that we will be unable to compute the values of variable expressions in these unseen blocks, as we do not derive the values of inputs we have not observed. We still can describe the functionality of these blocks but are unable to determine the values and addresses of the variables within these blocks. For the case of unseen inputs, since we use concrete values to derive data flow, we will be unable to describe the data flow dependencies of these variables in the case we can not derive their addresses and values. This will not introduce any spurious results, however, as in these instances variable expressions only describe operations which we know to be possible to occur and does not attempt to derive their values or estimate data flow which could potentially be incorrect. This is a small downside, given the advantages of this addition, as it greatly increases coverage with very little additional computational cost. In Section 5.6 we evaluate of the effectiveness of this extension, demonstrating that it has very little overhead while resulting in an increase in block coverage.

In Figure 4.13 we give an example of this process on a trivial program that either adds or subtracts a pair of integers depending on the command given and returns the result. If performing subtraction, it checks if the difference is negative, in which case it will instead return 0. In this example, we execute the program with a single input example, hitting the branch that performs addition and using our static analysis method to automatically analyze the branch that performs subtraction. We can see that the new variable `t10` arising from the subtraction is added to our state representation and used in subsequent blocks in the branch. Although it would not be possible to return a negative value on the branch that performs the subtraction, in our abstraction this is the case, as shown. This is an over-approximation and is considered acceptable as we don't care about the actual values of these variables, we are only interested in uncovering data dependencies. Additionally, we can see that the condition checking that difference is greater than 0 always occurs before the subtraction operation, and since our model considers execution order we would be able to identify this constraint applies to this branch. Furthermore, on this new branch,



```
mt10  St(Sub(rbp, 24), GET(rdi))          = 1
mt12  St(Sub(rbp, 28), GET(rsi))          = 2
mt14  St(Sub(rbp, 32), GET(rdx))          = 3
 t2   LD(Sub(rbp, 24))                    = 1
ct14  CmpEQ(t4, 1)                        = 1
```

```
 t4   St(Sub(rbp, 82))                    = 2
 t8   LD(Sub(rbp, 32))                    = 3
rax   PUT(Add(t4, t8))                    = 5
```

```
 t16  LD(Sub(rbp, 28))                    = 2
 t18  LD(Sub(rbp, 32))                    = 3
 t10  StLD(Sub(rbp, 20), Sub(t16, t18))   = -1
ct10  CmpLT(t10, 0)                       = 1
```

```
rax  Put(0)                               = 0
```

```
 t8   LD(Sub(rbp, 20))                    = -1
rax   Put(t8)
```

State S

rbp - 24	1
rbp - 28	2
rbp - 32	3
rbp	...

State S'

rbp - 24	1
rbp - 28	2
rbp - 32	3
rbp	...
rbp - 20	-1

Figure 4.13: An example illustrating our static analysis process, in which we show the mutation of the program state over the unseen branch.

we can see the difference is computed from the second and third input parameters which are the same as what is used to compute the sum on the other branch, so we can label these values as being dependent on these input parameters.

The motivation behind this augmentation is to reach branches that handle exceptions and errors, which we do not expect to see in the normal execution of the program. To demonstrate the motivation behind this additional functionality, we present the example in Listing 3. In this example, it would be unlikely that the call to `malloc` on line 3 would fail, thus we would not be able to observe the behavior of the program when `!buf` is true, and would be unable to determine with certainty that a null-pointer dereference was possible.

4.5 Model Inference

In this section, we describe our process for building a control-flow-based model of the target program by combining the partial CFGs produced for each execution. This is a sharp diversion from runtime verification methods, which typically perform an analysis of a single execution trace. Instead, we are trying to reconstruct the control flow of the entire program from what we have observed executing it. By combining the partial CFG models across separate executions, we are able to build a model that exhaustively reflects the functionality during our analysis across the entire workload. The merging of CFGs creates new execution paths, and we approximate the data flow for these new execution paths by merging data flow information across all nodes that are at the same program point.

The final model has the same structure as the CFGs produced for each program execution.To combine the set of partial CFGs into a single CFG, we iteratively identify and merge equivalent nodes, which allows for the inference of new execution paths. To eliminate impossible execution paths we condition the equivalence of a pair of nodes on their address, this being the value of the program counter at the start of the basic block. For each pair of nodes $z0$ and $z1$, if $z_0 == z_1$, we replace all edges $x \xrightarrow{t} z_0$, with edges $x \xrightarrow{t} z_1$, where x represents an arbitrary node with an in-edge to z_0 and replace all edges $z_0 \xrightarrow{t} x$, with edges $z_1 \xrightarrow{t} x$, where x represents an arbitrary node with an out-edge from z_0.

Within each node, each expression is backed by data aggregated across all executions of the block for the entire workload. This includes merging the values observed at that particular program point across all executions. We perform a second data dependency approximation step when merging nodes, aggregating data dependencies across all instances of the expression. The goal of this step is to identify cases where if for example, we have two variables `x` and `y` and we know that a program point uses `x` and `y`, and a second program point uses `x`, then the second program point could also use `y`. This is an over-approximation, as we do not check whether there is a path constraint that prevents the second program point from using `y`. However, because in our property checking step we only consider valid orders of events which are described by control flow, its impact is minimal. This model is further solidified in the rule-checking phase, when we introduce logic to restrict transitions based on previously visited states. This approach allows us to not only describe the control flow of a single execution trace but also the control flow across multiple executions in a single model.

To illustrate this process we include the example program shown in Listing 4. This example program contains a use-after-free vulnerability, which can be triggered if the user tells the program to delete a student and then print that same student. Assume we have created two test cases to exercise the program, one that calls `deleteStudent()` after creating a student (Figure 4.14a), and the other that calls `printStudent()` after creating a student (Figure 4.14b). We show the final CFG produced by merging these partial models in Figure 4.14c. In this new model we have inferred that it is possible to call `printStudent()` after `deleteStudent()`, and through our data-flow approximation that it is possible for the student being deleted to be the same as the one being printed, thus would be able to identify this vulnerability without observing the required sequence of events. This allows us to model the program with fewer input examples. We can infer program behaviors for new paths we haven't specifically observed, as for different occurrences of a particular node we can see different branches taken over different executions and merging these occurrences creates new paths in the resulting CFG.

4.6 Property Language

In this section, we describe the design of our property language Hy2-lang, first describing its syntax and structure then justifying its design and describing different techniques used to define properties in the language. This language is created to allow for arbitrary properties to be described and tested against arbitrary program models using our framework.

```
student_t *createStudent(int student_id);
int deleteStudent(student_t *s){
   free(s);
}
int printStudent(student_t *s){
   print("Name: %s\n", s->name);
   ...
}
int main(){
    while(1){
       option = INPUT();
       if(option == 1){
          students[student_no] = createStudent(student_no);
          student_no++;
       } else if(option == 2){
          st_no = INPUT();
          deleteStudent(students[st_no]);
       } else if(option == 3){
         student_no = INPUT();
         printStudent(students[st_no]);
       } else {
   ...
```

Listing 4: A snippet of an example program to be used to illustrate the model inference step

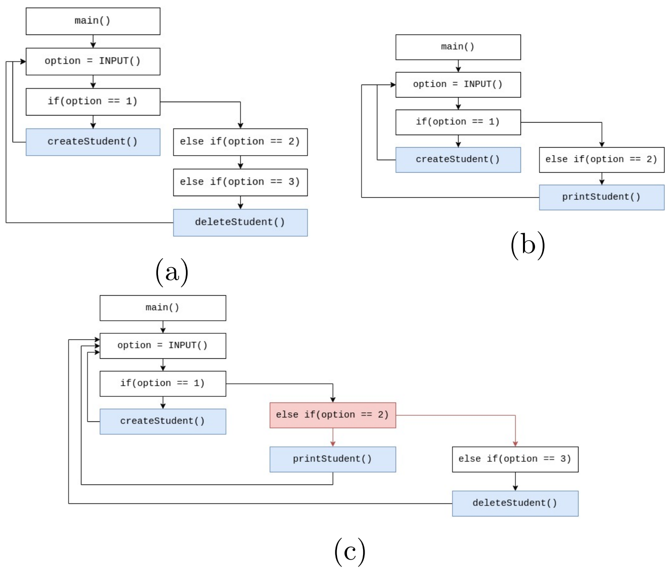


Figure 4.14: Illustrating the merging of separate program executions to infer new execution paths. (a) and (b) show the control flow derived from the execution of two input examples, and (c) shows the merging of these two partial models.

4.6.1 Overview

Common Weakness Enumeration (CWE) [3] and the SEI CERT C coding standard [5] are commonly used standards for describing and categorizing flaws in software and hardware. We aim to design a language that can represent abstractions of these weaknesses in simple language. In our implementation of our property language, we need to handle two classes of undesirable behaviors: those based on ordering constraints, and those based on data constraints. The first class are those that can simply be described by a sequence of events that leads to an undesirable condition. The second class is not only defined by a series of events, but also by constraints on data or the domains of variables. This includes properties such as read out-of-bounds and write out-of-bounds. To describe these properties, we first describe the underlying behavior and then introduce additional propositions describing the domain of values of variables that would cause the undesirable behavior. This second class is important because memory errors are a very common vulnerability. For example, for read out-of-bounds we first describe the behavior of reading from memory and then introduce conditions describing how a memory read can be invalid, in this instance if it is possible for the size of the allocated region of memory to be smaller than the offset at which it is being accessed which would lead to a read out of bounds. We use this approach because the undesirable condition is not directly related to any particular ordering of events, and because our abstraction emphasizes control flow rather than variable domain constraints.

We create a policy language, referred to as Hy2-lang, to implement temporal safety properties or security properties. Past works [42] [57] have used similar representations but are typically more abstract or coarse grained, using sequences of library calls to describe undesirable behaviors. We create Hy2-lang to be compatible with our abstraction of program behavior and to communicate underlying model semantics, expressing properties we derive from our representation of a variable expression. This language is designed to be concise, expressing semantics in simple, and human-comprehensible statements. It allows for the definition of both generic and application-specific properties. Generic properties introduced can be used on arbitrary applications. With continued effort over time, a large library of properties could be created and shared, describing a much wider range of behaviors.

We refer to the implementation of security properties within our language as rules. These rules are an abstraction of the finite-state machine encoding of temporal safety properties. The implementation of these rules involves the decomposition of this finite state machine encoding into a set of equivalent finite event sequences. This abstraction is much simpler and more efficient to check against a program model. The use of such an abstraction will not introduce errors in our analysis, firstly because with the implementation of this language, we take care to introduce semantics to serve as an abstraction for logic that can't be represented in a finite sequence, and secondly because, unlike other approaches, our implementation of the rule-checking step does not stop at the first discovered error [57] and exhaustively explores all paths through the program model.

4.6.2 Syntax

Each rule consists of a set of event sequences, such that each sequence corresponds to a path through the finite state machine encoding of the property. Within these sequences, each event consists of an action and an optional set of attributes and predicates. Actions consist of either a function call or one of a set of keywords that have a specific definition within the context of our language. The technique of using a single keyword to represent a group of operations that serve a similar purpose is referred to as pattern matching. The first type of pattern uses a single keyword to represent a group of functions that all perform the same underlying operation, allowing us to create a single property for all of them instead of a separate one for each one. Such patterns include `ALLOCATE`, `FREE`, `FOPEN`, and other standard operations. For instance, the pattern `FACCESS = {fclose, fwrite, fseek, fgetc, feof, ...}` includes all operations that operate a given file descriptor, and the pattern `FOPEN = {fopen, open, fopen64, open64}`, allows us to represent different instances of the open call with a single symbol. For each of these patterns, we create a mapping of their arguments to a set of keywords across all included function instances. For example, for print formatting functions, we use `"fmt"` to extract all format string arguments across different function instances. We additionally use these keywords to represent other low-level operations, including the following:

- LOAD — corresponds to a memory read operation
- STORE — corresponds to a memory write operation
- VAR — any memory access operation, can be either LOAD or STORE
- RET — corresponds to an operation where the program operates on a return value of a function call
- END — corresponds to the termination of the program
- COND — corresponds to conditional jump operations

Each of the above keywords except END can be paired with a set of attributes that can be extracted from instances of the corresponding events in our program model. For example, for VAR, LOAD, and STORE we have attributes such as base and offset for operations in the form of `LOAD(base+offset)`, and `STORE(base+offset, y)`, which extract the concrete or symbolic values for base and offset of the memory access. Other examples include `address`, `value`, `type`, `label`, and `is_constant`, which is a boolean describing whether the value of a particular variable is hard-coded in the program source. This set of attributes to check for and extract for each event are stored as key-value pairs with the corresponding event within the property definition. When performing the rule-checking step, the keys are used to identify which properties to extract from a given event within the program model, and the values are used to give a unique identifier to the particular instance of the attribute across the entire property definition. The extracted values are considered part of the global state, and we keep track of the set of attributes corresponding to the current path through our model. We additionally allow for propositions involving these attributes to be described, which allows for constraints to be described on the values of these attributes for any attributes in the global state.

We introduce additional logic to serve in the place of transitions that would cause a state machine model of a property to transition to previously visited state, which we refer to as back-edges. By abstracting away these edges in our implementation, it allows for the representation of properties as a finite set of traces, each with a finite length. In their place, we introduce the `NOT(X)` keyword where X is some event and `NOT(X)` can be consumed by any symbol except for X. If the event X occurs at the specified time step for all paths through P then the property cannot be present in our program model. We describe this in more detail in following subsection (Subsection 4.6.3).

Finally, we introduce additional syntax to allow for the definition of data-based constraints. These constraints describe the domain of values that a particular variable expression would have to have for the program to be vulnerable. These take the form of a set of propositions describing the constraints on the set of attributes previously described but are evaluated in secondary feasibility analysis, which is described in detail in Section 4.8.

4.6.3 Property Definitions

In this subsection, we describe how we use our syntax to define security properties and the structure of these rules.

We use the term decomposition to describe the process of transforming a finite state machine encoding of a temporal safety property into a set of elementary sub-automata. In simpler terms, this process is the translation of temporal logic into our more basic property language. Our language cannot represent logical disjunction (i.e. or) and instead we represent this logic with an equivalent set of subexpressions. For example, if we have expression `a(b|c)`, we would dervive two subexpressions `ab`, and `ac`.

As described previously, we replace back-edges with `NOT` events. The `NOT` event is able to serve in place of back-edges because the decomposition step produces a set of automata that will produce traces of a finite length. We check for the `NOT` event in returned traces at the time step specified in the property definition. The exhaustive search used in our implementation checks from each initial state to match every possible successor state in the program model and then eliminates those with the `NOT` event after the completion of our search.

If for instance we have a property that emits the following sequence of events `A, B, NOT(D), E(i)` and paths through the program:

$$\begin{aligned} &P = \{ \\ &\quad P_1 = \{A, B, E(1), E(2)\}, \\ &\quad P_2 = \{A, B, D, E(1)\}, \\ &\quad P_3 = \{A, B, E(1), D, E(2)\}, \\ &\quad P_4 = \{A, B, D, E(1), E(2)\}, \\ &\} \end{aligned} \tag{4.1}$$

We would return the following paths as violating the given property, referred to

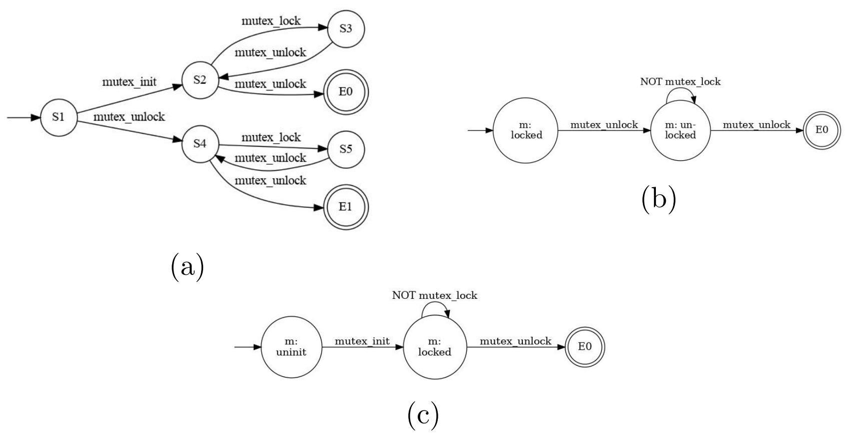
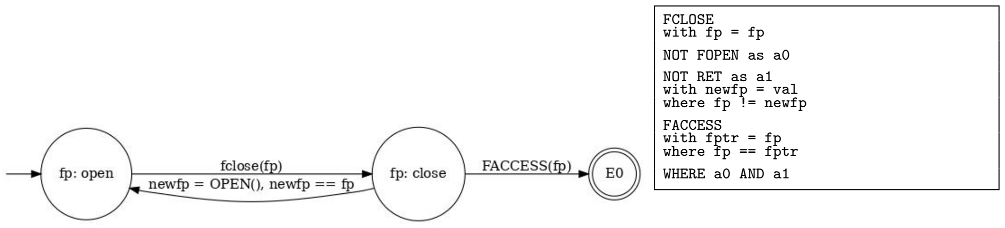


Figure 4.15: Example showing how we decompose properties into elementary FSMs which can be represented with a finite event sequence. (a) - Original FSM (not labeled with states), (b) and (c) - resulting FSMs

as error traces: $A, B, E(1)$, and $A, B, E(2)$ in P_1, $A, B, E(1)$ in P_3. The path $A, B, E(2)$ in P_3 would not be returned as D occurs before $E(2)$. We wouldn't return paths P_2 or P_4 as in both cases they have an event D occuring after B and before $E(i)$. In P_4 event D occurs before both $E(1)$ and $E(2)$ so neither path violates the given property.

The set of all sub-automata will capture the same behavior as the original FSA but are much easier to check against our model of program behavior and describe with simple syntax. In Figure 4.15 we give an example of this process for the property "A mutex should never be unlocked without being locked". We begin with the FSA shown in Figure 4.15a. We decompose this FSA into the two sub-automata shown in Figures 4.15b and 4.15c, one for each branch. We replace the two `mtx_lock` back edges with a `NOT` event, such that if this event occurs on the path between the `mtx_unlock`, or `mtx_init` event and the second unlock event it can't be an instance of unlock without lock. If there is a sequence of events `mtx_init, mtx_lock, mtx_unlock, mtx_unlock`, the first `mtx_unlock` event will be matched by the property instead of the `mtx_init`.

In Figure 4.16 we present an example of the property that "a file that is closed should not be accessed". The reasoning behind this is if a file pointer is closed,



```
FCLOSE
with fp = fp

NOT FOPEN as a0

NOT RET as a1
with newfp = val
where fp != newfp

FACCESS
with fptr = fp
where fp == fptr

WHERE a0 AND a1
```

(a) (b)

Figure 4.16: Example of our property definition for FIO-46: Do not access a closed file.

and there is at least one path through the program where it is possible for it to be accessed without it being reopened, then a violation occurs. Here `FCLOSE`, `FOPEN`, and `FACCESS` correspond to groups of functions that perform the corresponding operations. The `with` syntax describes attributes of each event, and the `where` syntax describes conditions on these attributes. The `RET` event is used to get the value of the file pointer returned from `FOPEN`. Finally, the `WHERE` syntax is used to describe relationships between `NOT` events. This example is merely illustrative; in our actual implementation of the property we use the labels describing data-dependencies to identify uses of the same file pointer instead of their concrete values. Additionally, we don't explicitly represent the state of `fp` in our property; instead, we represent it implicitly through the events `FOPEN` and `FCLOSE` that cause the change of state.

In designing this language we assume that CWE listings, SEI CERT C standards, and past vulnerabilities will be used as reference to describe undesirable behaviors. To derive a property from a CWE entry, a user would have to be able to consider possible patterns of behavior that describe the weakness. To illustrate this process, we consider the example of "CWE-252 Unchecked Return Value".

In this instance, the user would have to consider potential patterns in behavior in which the return value is checked, as shown in Listing 5. The first example shows the sequence of events from passing a variable by reference with the `USE addr` event describing the use of the modified variable, and the second two events show two

```
arg = PUT(addr)
CALL
RETURN
CMP(GET(rax), 0)
USE addr

RETURN
CMP(GET(rax), 0)
STORE(addr, GET(rax))
USE addr

RETURN
STORE(addr, GET(rax))
t0 = LOAD(addr)
CMP(t0, 0)
USE addr
```

Listing 5: Our identified patterns of instructions for checking a return value.

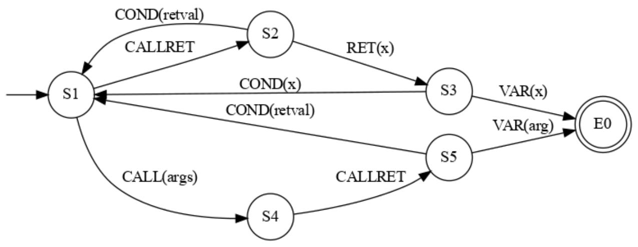


Figure 4.17: Property model for "CWE-252 Unchecked Return Value" with back edges.

different orderings of the case where the program performs some operation on the return value. Then since we want to describe instances where the return value is not checked, we negate the condition events. With the `VAR` event, we are describing instances where the program uses the value returned without first checking its value.

The property shown in Listing 6 is equivalent to the FSM model shown in Figure 4.17, because for every back edge, there is an equivalent NOT event, and for each state with more than one successor state we have defined a separate event sequence that describes a finite path through the original state machine model.

4.6.4 Techniques for Property Definitions

In this subsection, we describe techniques that could be used for defining security properties within our language. The purpose of this language is to allow for the

```
---
CALLRET
with fname = fname

NOT COND as a0
with cond0.dsource = dsource
where cond0.dsource == fname

RET
with returned_from = ret.fname, var.var = var.name, ret.label = label
where returned_from == fname

NOT COND as a1
with ls.var = ls.var, cond.dsource = ls.dsource
where cond.dsource == returned_from AND ls.var == var.var

VAR
with var.label = label
where ret.label like var.label

WHERE a0 OR a1
---
CALL
with fname = fname, call.args = call.args

CALLRET
with ret.fname = fname

NOT COND as a0
with cond0.dsource = ls.dsource
where cond0.dsource == fname

VAR
with var.label = label
where var.label in call.args
---
```

Listing 6: Our property definition for “CWE-252 Unchecked Return Value”.

implementation of generic security properties in simple language. As such, there are a number of different strategies that could be employed to describe vulnerable behaviors. This includes defining properties with various degrees of precision. More specific properties describe a very particular behavior, whereas less specific properties attempt to remain generic and cover an entire range of behaviors with a single event sequence. The former are easier to define for a single case but may be ineffective in describing all possible behaviors that cause an error, and thus may cause Hy2 to miss errors. Such properties produce higher-quality results, as they can more accurately describe the source of the error and produce fewer spurious results. The latter are more difficult to define and result in lower quality results but are less likely to miss potential errors. An example of this is the case of "off-by-one" errors. These errors would be covered by our read out-of-bounds and write out-of-bounds properties, however, these more generic properties do not describe the actual source of the problem. Because of the observations made in our evaluation, it is generally believed that the former has more value than the latter and replacing more generic properties with a set of more specific ones yielded higher quality results. During our evaluation in Section 6, we noted that these strategies had a large impact on the quality of the results.

4.7 Rule Checking

In this section, we describe our method of checking Hy2-lang rules against our control-flow-based model of program behavior. We describe the basis of our approach to checking temporal safety properties against a state-based model and describe our abstraction of this logic designed to improve practicality. We describe our implementation of this step in detail in Section 5.8.

We aim to check a property defined in our language against a control-flow model.

We use the well-established logic of synchronous product composition [71] to approach this problem. Our implementation of this logic is distinct as we omit the step of reducing the initial model M, so it only has symbols in the alphabet of the property P. This makes our verification process more time consuming as we must check whether every event in M triggers a transition in P, and not every event is relevant to the current property being checcked. However, it allows us to build a complete error trace as we traverse the model. This is necessary for the completion of our feasibility analysis step, described in Section 4.8. Whereas past methods either consider variable domains and path constraints in their model or disregard them entirely, we consider them only for specific paths through the program for which we have identified potentially undesirable behavior. The goal of our method is scalability, as such we require a method to be able to scale to larger programs. We describe our approach in detail below.

4.7.1 Property Instrumentation

Property instrumentation in this instance refers to the algorithm used to check security properties against a state-based program model. Assume we have a set of security properties Φ and a control-flow-based model of program behavior P. We wish to check an arbitrary property against our model of the program. We encode this security property $\varphi \in \Phi$ as a finite state machine S. We encode the model of program behavior as a pushdown automaton (PDA) M because the next state in our model is dependent not only on the current state but on past states. For instance, control flow dictates that a function returns to the function that called it. For instance, if we have two functions `foo` and `bar` that both call the function `qux`, our model would be inaccurate if it was possible to call `qux` from `foo`, and `qux` would return to `bar`. To compute whether $M \models S$ we perform synchronous product construction with S and M to produce a PDA N describing the intersection of M and S.

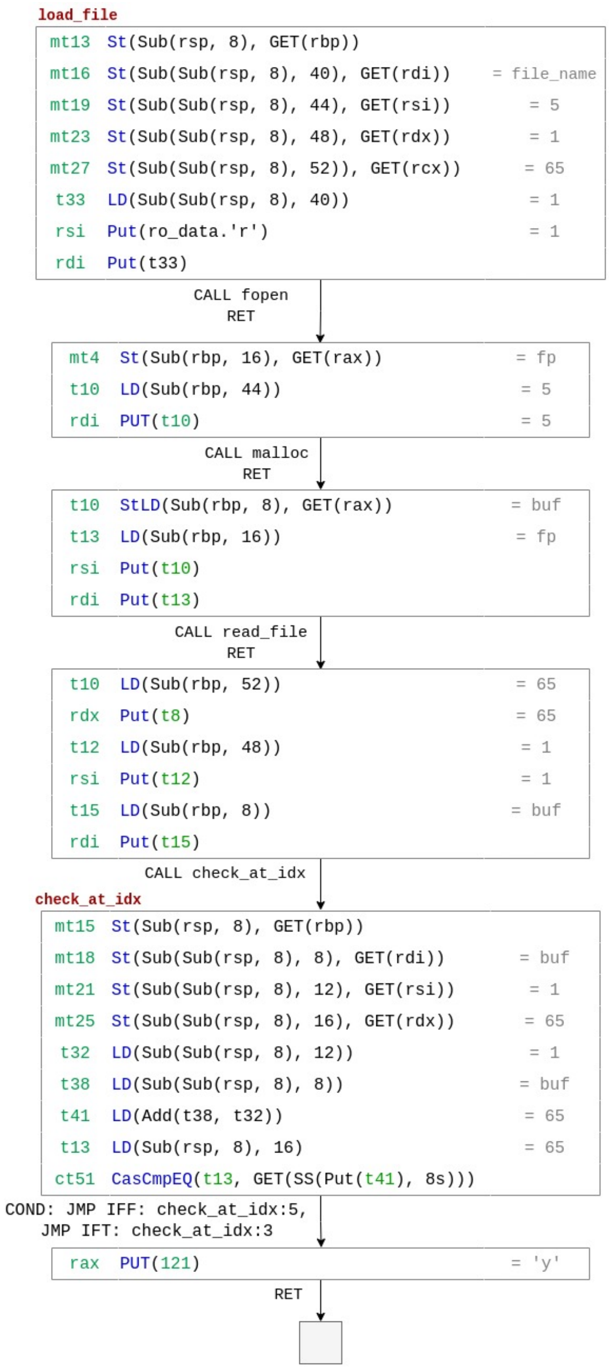
```c
char check_at_idx(char *buf, int idx, int val){
    if(buf[idx] == val){
        return 'y';
    }
    return 'n';
}
int load_file(char *file_name, int n, int idx, int val){
    FILE *fp = fopen(file_name, "r");
    char *buf = malloc(n);
    read_file(fp, buf);
    char rc = check_at_idx(buf, idx, val);
    printf("%c\n", rc);
}
```

Listing 7: The snippet of code used in our product composition example. This program takes in a filename, file size, an index in the file, and a value, reads the contents of the file into a buffer of the given size, and checks whether the character at the given index is equal to the given value.

If there is a path through N to an `ERROR` state then the program violates the property S.

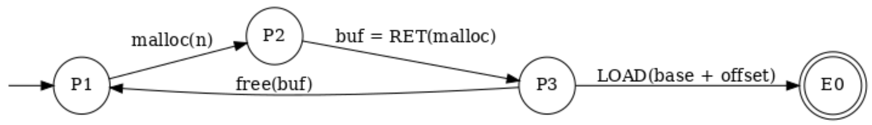
We illustrate this process with an example. In Listing 7 we provide an example of a trivial program and in Figure 4.18 we provide an example of the simplified model of this program. In Figure 4.19a we provide an example of a heap-based read out-of-bounds property φ, encoded as a finite state machine P which we aim to check against the model of our example program. This property has the following alphabet $\Sigma = \{ALLOC, RET, FREE, LOAD\}$. In Figure 4.19b we show how we reduce our model of the program to only contain events in Σ. Finally, in 4.19 we show the intersection of this reduction of M and our property model P, where transitions in P are triggered by their equivalent transitions in M.

To check an arbitrary property φ, encoded as S, against target program M we orchestrate S and M in parallel, exploring them simultaneously, instead of explicitly creating a model N. Through this process we iteratively build N, this being the PDA that describes the intersection of M and S. We begin by identifying events in M that

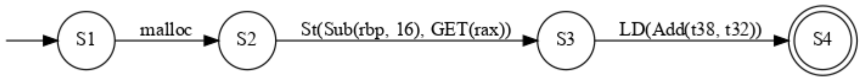


```
load_file
mt13  St(Sub(rsp, 8), GET(rbp))
mt16  St(Sub(Sub(rsp, 8), 40), GET(rdi))     = file_name
mt19  St(Sub(Sub(rsp, 8), 44), GET(rsi))        = 5
mt23  St(Sub(Sub(rsp, 8), 48), GET(rdx))        = 1
mt27  St(Sub(Sub(rsp, 8), 52)), GET(rcx))       = 65
 t33  LD(Sub(Sub(rsp, 8), 40))                  = 1
 rsi  Put(ro_data.'r')                          = 1
 rdi  Put(t33)
                CALL fopen
                   RET
mt4   St(Sub(rbp, 16), GET(rax))               = fp
t10   LD(Sub(rbp, 44))                          = 5
rdi   PUT(t10)                                  = 5
                CALL malloc
                   RET
t10   StLD(Sub(rbp, 8), GET(rax))              = buf
t13   LD(Sub(rbp, 16))                         = fp
rsi   Put(t10)
rdi   Put(t13)
              CALL read_file
                   RET
t10   LD(Sub(rbp, 52))                         = 65
rdx   Put(t8)                                  = 65
t12   LD(Sub(rbp, 48))                          = 1
rsi   Put(t12)                                  = 1
t15   LD(Sub(rbp, 8))                          = buf
rdi   Put(t15)
            CALL check_at_idx
check_at_idx
mt15  St(Sub(rsp, 8), GET(rbp))
mt18  St(Sub(Sub(rsp, 8), 8), GET(rdi))         = buf
mt21  St(Sub(Sub(rsp, 8), 12), GET(rsi))          = 1
mt25  St(Sub(Sub(rsp, 8), 16), GET(rdx))         = 65
 t32  LD(Sub(Sub(rsp, 8), 12))                    = 1
 t38  LD(Sub(Sub(rsp, 8), 8))                   = buf
 t41  LD(Add(t38, t32))                          = 65
 t13  LD(Sub(rsp, 8), 16)                        = 65
ct51  CasCmpEQ(t13, GET(SS(Put(t41), 8s)))
COND: JMP IFF: check_at_idx:5,
   JMP IFT: check_at_idx:3
rax   PUT(121)                                 = 'y'
                   RET
```

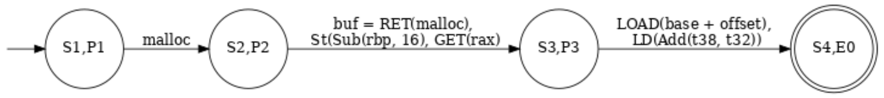
Figure 4.18: An abstraction of the relevant parts of the example program shown in Listing 7 used in our product composition example.



(a)



(b)



(c)

Figure 4.19: Product Composition Example Models: (a) Showing the read out-of-bounds property, (b) showing the model of the subset of events in the program model that are relevant to the particular property being checked, and (c) showing the composition of (a) and (b).

trigger a transition from the initial state in S, this being the state from which the event in M was triggered, paired with the initial state of S becomes the initial state in N. For subsequent states in N we identify transitions $((q_S, q_M), (q_{S'}, q_{M'}), \epsilon)$ in which property S is in state q_S and model M is in state q_M, and accepts event ϵ, causing a transition to state $q_{S'}$ in S and state $q_{M'}$ in M.

We maintain a global state consisting of a call stack and a set of predicates E that describes the state of $N = S \times M$. During our exploration of S and M, to simulate execution flow we perform the following for each step through M: if the current transition in M is a call instruction, then we push the return address to the call stack, and when the current instruction in M is a return instruction, we check that the address of the next state is equal to the address at the top of the call stack and pop from the top of the stack. If M emits a symbol ϵ that satisfies E_i, then we perform a transition in N and compute the set of predicates E_{i+1} that represents the next state in N. We continue until we reach an `ERROR` state in P or when an `ERROR` state is not reachable from the current state in S. Since there is a finite number of states in both S and P then this is guaranteed to come to a conclusion for any given property.

In our abstraction of this logic, we are orchestrating the security property with our model of the program M, such that events in M trigger transitions in the property model. We keep iterating until we reach an `ERROR` state or determine such a state is not reachable in M. This occurs for instance, if we have exhaustively explored M. During our traversal of M, we keep track of the visited configurations of N to avoid cases where this search wouldn't terminate, which may occur in the case of looping behaviors. We also introduce a call stack per thread, so that we can account for different interleavings.

We describe different paths through the model as candidates. We store candidates in a queue, where candidate C holds the global state for a particular traversal and consists of the following:

- the pair (q_M, q_S) where q_S is the current state in S, and q_M is the current state in M
- E_i the set of predicates describing the event in M that will trigger a transition in S
- a call stack describing the configuration of N
- a call trace describing visited configurations

While the queue is not empty, we pop candidate C from the queue and find all possible successor states, adding each one to the queue. If no successor states are found, then an `ERROR` state is not reachable from (q_M, q_S). If S is in an `ERROR` state we add C to the set of results returned. This approach greatly reduces the complexity of this search problem, as we only have to keep track of the current state in S as we traverse the model instead of both S and S. Additionally, unlike past methods this method is exhaustive (it will check all possible paths in the model), as at each step it checks all successor states reachable from the current state for each candidate. The trade-off with this approach is that while it greatly reduces memory usage, regardless of model size, its iterative nature increases the time required for analysis. However, we can compensate for this through the use of multi-threading, as each thread is able to process a different candidate in parallel.

4.8 Feasibility Analysis

This final step takes the results produced in the rule-checking stage and for each result performs a feasibility analysis. For each result returned we have a path through the program that violated the particular property, which we will refer to as the error trace. The abstraction of program behavior M used in our rule-checking step that produces these results is an over-approximation. This is the case because when traversing M in the previous step we do not consider the satisfiability of path constraints. The feasibility step asks whether the behavior described error trace is actually possible in the target program or, more precisely, if it is possible to trigger the reported error with concrete inputs.

We designed this step to be performed after the rule-checking process is completed for several reasons. This step is the only step in our analysis process where we reason about path constraints because constraint solving is expensive. It is only necessary to perform such an analysis on the branches and variables relevant to potential errors, therefore we wait until we have identified relevant paths through the program before performing this analysis. Additionally, we do not expect this step to always be able to reach a conclusion about whether the described erroneous behavior is possible, and by performing this step last, it ensures that any failure of this step does not affect the results returned as we have already identified the potential errors.

To describe the feasibility of a path we label it as either `sat`, `unsat`, or `unknown`. We return all error traces regardless of what this label is, as this analysis step is not exhaustive; it can only reason about logic that is observable to the program. Additionally, whilst our rule-checking step is built on formal logic, this step is not, and thus this conservative approach is necessary. Firstly, we extend each error trace by generating an additional set of paths from the additional program state to the initial state in the error trace. We extract a set of conditions from each error trace, and if there is any pair of conditions for which it is not possible for both of them to have the particular assignment we observed in this trace, we mark it as `unsat`. Secondly, we evaluate the variables in the model involved in the events that cause transitions in the property model. For these variables, we wish to define their domain, or potential range of values for the purposes of performing a feasibility analysis.

Although our properties are control driven, we allow for the definition of an additional set of data-driven predicates, denoted G, as previously mentioned in Section 5.7. We choose to check the satisfiability of these predicates separately than our reachability analysis to reduce the complexity of our rule-checking step. For each relevant variable v, we describe its domain v_D. This domain begins without any constraints, and we iteratively add them to v_D by backtracking through the execution trace, comparing first the variable's identifier to that of those in conditional statements on the path and checking any dependencies the variable has against conditions on the path. For each conditional statement that effects v_D, we build a predicate describing exactly how it constrains v and add it to v_D. We also identify any logic we cannot reason about, such as the variable being checked against the value returned from a function, and return it as part of the result. Finally, we compare the derived variable domain v_D with the associated data-driven predicates $G_v \in G$ and compute whether it is possible to satisfy the resulting proposition. If we are able to satisfy the expression, we return an example of a satisfying assignment. If there are no possible values of v_D that satisfy G_v we label the error trace as `unsat`. We return the derived proposition to explain under what conditions the program does not comply with a

```c
int FLAG = 4;
int MAX_FLAG = 10;

int example(){
    int n = INPUT();
    if (n > MAX_FLAG) {
        ...
    } else {
       addr = malloc(n);
       j = FLAG;
       OUTPUT(addr + j);
    }
    return 0;
}
```

Listing 8: Example snippet used to describe how we analyze variable domains. This program takes in an integer as input which is used as the size of a buffer and then reads from this buffer at a fixed offset.

particular property.

This step is relatively simple as we have already computed the error path during our property-checking step, dependencies between expressions, and how they relate in previous steps. We use these relationships to describe any constraints on the set of relevant variables. Delaying the definition of feasibility analysis may result in an overestimation in the domain of variables but never the underestimation, so this step will never introduce false negatives.

To illustrate the purpose of this step, we present the following example, shown in Listing 8. For a read out-of-bounds vulnerability to occur we must have an event `LOAD(addr + j), j>n`, where `LOAD(addr+j)` is the event of reading from memory at `addr + j`, where addr is the location of the allocated buffer, at the offset `j`. If ever `j > n` then the program is reading outside the boundary of the allocated buffer, meaning we are reading out of bounds. Within this example, we have two variables: n and j. To determine the bounds of n and j we walk backward through our model, building a conditional statement from dependencies of n and j. Here, we have `(n < MAX_FLAG) < FLAG`, which is `(n < 10) < 3`, which simplifies to `n < 3`, so we can conclude that our vulnerable condition `j > n` is satisfied.

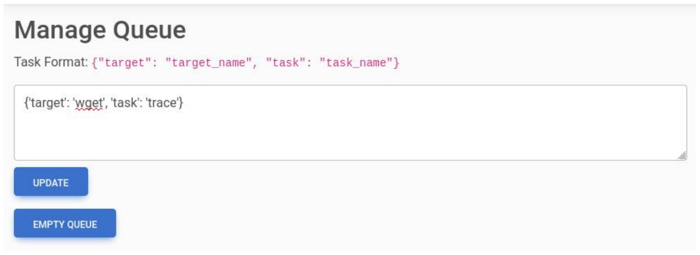
Chapter 5

Implementation

This chapter describes details specific to the implementation of our method, expanding on what was described in Chapter 4. In this chapter, we discuss how the user interacts with our framework and the various tools and techniques used in the implementation of our method. Hy2 is implemented in over 14 thousand lines of python code.

5.1 Interface

In this section, we describe how the user interacts with Hy2. We provide 3 different methods for interacting with our framework. Firstly, we created a web application that provides an interface for modifying the configuration of target programs, and the viewing of results. We implemented this web application in flask, reading saved configuration and result files and formatting them so that they are easier to interpret. We include several examples showcasing the functionality of this application in Figures 5.2 and 5.3, as well as several examples demonstrating our formatting of results in Chapter 6. Secondly, we created a command line interface for accessing the various functionalities within our framework which we show in Figure 5.4. Finally, we implemented a work queue interface (Figure 5.1), which can run analysis tasks as jobs. Each of the major steps in the completion of our analysis process is to be run as a separate task. The work queue allows these jobs to be grouped together and executed sequentially. We discuss this interface further in Section 5.9 and discuss potential improvements in Section 7.4.

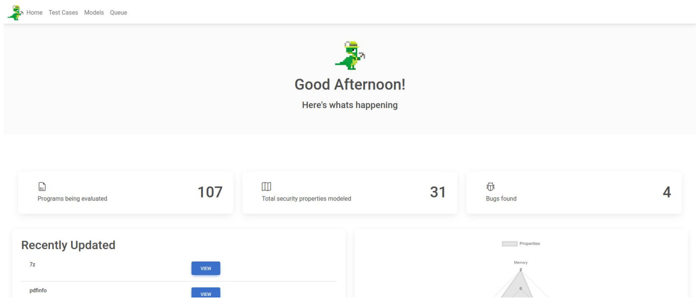


Manage Queue

Task Format: `{"target": "target_name", "task": "task_name"}`

```
{'target': 'wget', 'task': 'trace'}
```

Figure 5.1: Screenshot of our web application showing the configuration interface for the work queue.



Good Afternoon!

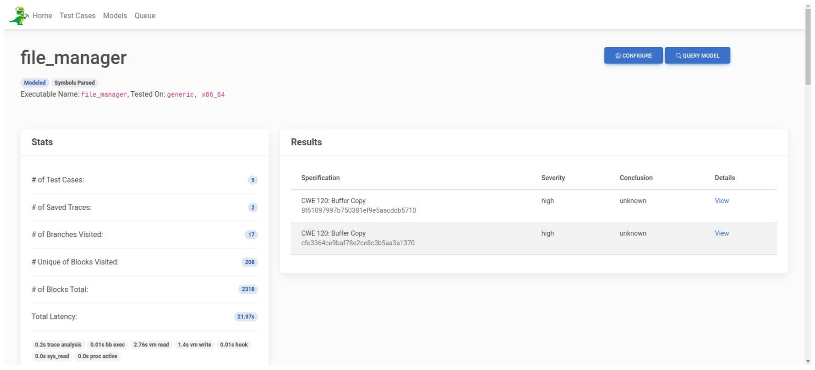
Here's whats happening

Programs being evaluated	107
Total security properties modeled	31
Bugs found	4

Recently Updated

7z	VIEW
pdfinfo	VIEW

Figure 5.2: A screenshot of the home page of our web application.



file_manager

Modeled Symbols Parsed

Executable Name: `file_manager`, Tested On: `generic, x86_64`

Stats

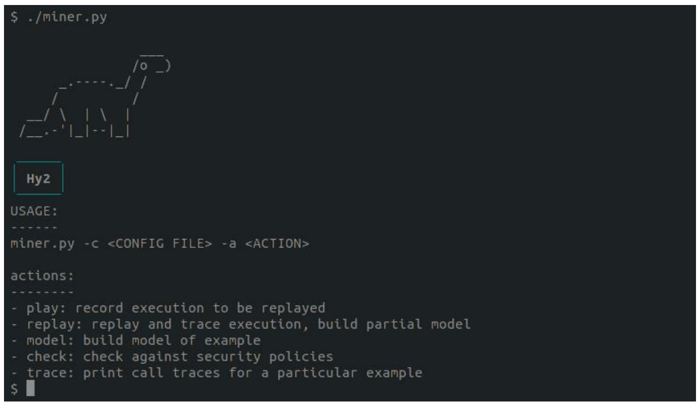
# of Test Cases:	5
# of Saved Traces:	2
# of Branches Visited:	17
# Unique of Blocks Visited:	208
# of Blocks Total:	2318
Total Latency:	21.97s

0.2s trace analysis 0.01s bb exec 2.76s vm read 1.4s vm write 0.01s hook 0.0s sys_read 0.0s proc active

Results

Specification	Severity	Conclusion	Details
CWE 120: Buffer Copy 8f61097997b750381ef9e5aacddb5710	high	unknown	View
CWE 120: Buffer Copy cfe3364ce9baf78e2ce8c3b5aa3a1370	high	unknown	View

Figure 5.3: A screenshot showing a target application profile page in our web application. Some relevant statistics are shown on the left-hand side of the page and results are listed on the right-hand side.



```
$ ./miner.py

                ___
               /o _)
      _.----._/ /
     /         /
  __/ \  | \  |
 /__.-'|_|--|_|

 Hy2

USAGE:
------
miner.py -c <CONFIG FILE> -a <ACTION>

actions:
--------
- play: record execution to be replayed
- replay: replay and trace execution, build partial model
- model: build model of example
- check: check against security policies
- trace: print call traces for a particular example
$ 
```

Figure 5.4: A screenshot showing the usage options for the command line interface created for interacting with Hy2.

5.2 Target Configuration

In this section, we describe how the user can configure this framework to best suit their specific use case. We provide a complete breakdown of all the possible configuration options in Table 5.1. As our analysis is based on concrete execution, we must specify exactly how to execute the target program. This includes specifying the environmental conditions in which the target program will be exercised and specifying a workload to exercise the target program. Configuring the environmental conditions includes specifying which guest image, and which snapshot to execute the target in, specifying how to configure the environment prior to and following execution. To describe the target itself the user must specify the name and location of the target, the name of the target pages within the process and whether the target is a library or a program. They can also refine the target scope by selecting functions by name to exclude from the analysis or by pagename and function name from external dependencies to include in the analysis. Inclusions can take 2 forms depending on whether the symbol resolution step has been performed (Section 5.3). If symbol resolution has been performed, for any function specified all basic blocks within the particular function will be analyzed. If static analysis is enabled, static analysis will also be performed on this function. If symbol resolution has not been performed for the library containing this external function, this method will analyze all functionality, from the time the control flow leaves the target program and hits the target function, until it returns to the target program. This is necessary because we don't know the range of addresses corresponding to the target function at runtime, so we cannot know precisely which blocks correspond to it. They must also specify how much static information to use during the analysis, including configuring symbols information, as described in the following section (Section 5.3).

Additionally, they can also specify semantic labels, mapping labels in properties to specific labels or types in the program source, including function parameters, struct members, and variable names. For example, this can be used to label specific fields as containing sensitive information, which in turn can be used to check for information leaks, as described in Section 6.4.3. For each target program Hy2 requires that a workload be specified, which consists of a set of inputs to exercise the program with. These inputs only need to hit any blocks or branches that the user wishes to test once. Assuming the user wishes to exhaustively test the entire program, this method would need to observe the execution of every block within the program once, namely block coverage. This may not always be feasible or necessary, so this method is able to evaluate any percentage of the target program based on the observed behavior produced by the given workload. We expect that if static analysis is enabled for a given program, then block coverage is feasible.

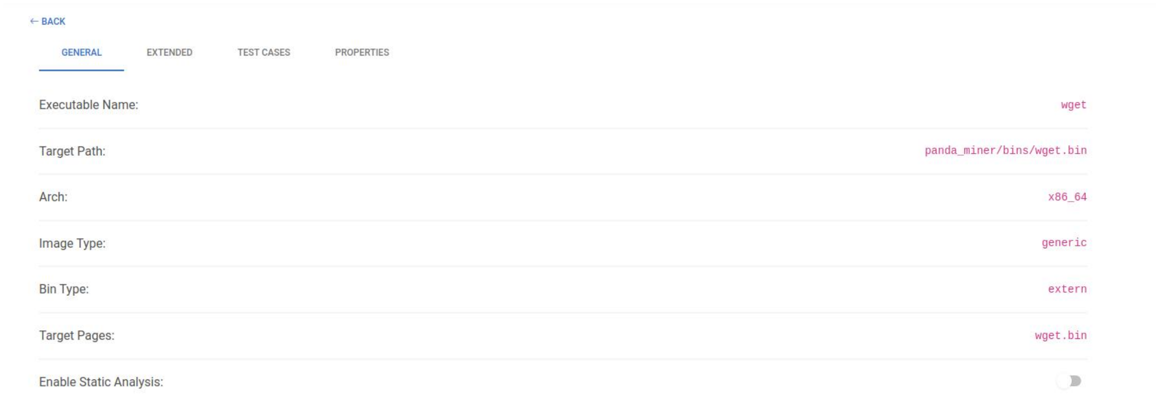
Currently, inputs are expected to take the form of commands, executed using the command line on the guest system. In Chapter 6, we show that integration tests are an effective workload and explore the relationship between the provided workload and results.

Attribute	**Description**
Target Name	The name of the target, used to identify it
Target Path	The path to the target on the guest, or locally
Arch	The architecture of the guest on which to execute the target
Image Type	The guest image to use to execute the target
Bin Type	(external, or built-in) specifies whether the target is on the guest
Target Type	Specifies whether the target program is a library or an executable
Static Analysis Enabled	Specifies whether to enable the optional static analysis step, only available if the target binary is available
Target Pages	The name of the target pages to analyze
Inclusions	Optionally select a list of function and associated pages that are in scope
Exclusions	Optionally select a list of functions that are out-of-scope
Pre-Conditions	Commands to run on the guest prior to executing the target
Post-Conditions	Commands to run on the guest after executing the target
Inclusions - Symbols	Optionally specify a list of function, page pairs to include in the analysis, including the static analysis step if enabled. For this option the target binaries must be available.
Pages for Static Analysis	Pages to include the static analysis step
Labels - Semantic Info	The mapping of symbols in our security properties to specific labels or types in the program source.
Test Cases	The set of inputs used to exercise the target
Property Selection	Specifies the set of properties to check against the target

Table 5.1: Descriptions of the configuration options available in our framework

```
{
    "target": "wget",
    "target_path": "wget.bin",
    "process_name": "wget.bin",
    "recording_name": "sample",
    "arch": "x86_64",
    "image_type": "generic",
    "bin_type": "extern",
    "exclusions": [],
    "type": "prog",
    "target_pages": "wget.bin",
    "inclusions_named": [["sym.iconv", "libc-2.27.so"],
 ↪      ["sym.pcre_exec", "libpcre.so.3"], ["sym.regexec",
 ↪      "libc-2.27.so"], ["sym.iconv_open", "libc-2.27.so"],
 ↪      ["sym.iconv_close", "libc-2.27.so"]],
    "inclusion_paths": {"libc-2.27.so":
 ↪      "/lib/x86_64-linux-gnu/libc-2.27.so", "libpcre.so.3":
 ↪      "/lib/x86_64-linux-gnu/libpcre.so.3"},
    "preconditions": [
        "panda.mount bins",
        "cp /root/bins/libnettle.so.7 /usr/lib/x86_64-linux-gnu/",
        "cd /root/bins/wget",
        "export PERL5LIB=/root/bins/wget"
    ],
    "postconditions": [],
    "command": [
        {
            "cmd": "./Test-iri.px",
            "type": "async"
        }
    ]
}
```

Listing 9: Example of a configuration file for the program wget. In this example wget is being exercised under the conditions described in the intergration test script Test-iri.px which is in a folder that gets mounted on the guest.



Executable Name:	wget
Target Path:	panda_miner/bins/wget.bin
Arch:	x86_64
Image Type:	generic
Bin Type:	extern
Target Pages:	wget.bin
Enable Static Analysis:	

Figure 5.5: A screenshot of the configuration interface for the program wget in our web application.

5.3 Symbol Resolution

One of the limitations of dynamic analysis is that a lot of information about the program is not available at runtime. This includes things like function names that make the analysis human-comprehensible. Information such as struct definitions, type information, and variable names are not typically preserved at compile time. There are classes of vulnerabilities for which dynamic analysis may be less than effective at identifying because it cannot always describe higher-level program semantics. For example, in our analysis, we may see a program reading a string from standard input, but we would not be able to identify that this string is a password, which would be necessary information for identifying information disclosure vulnerabilities. To help with such situations, we added additional steps to introduce source information into our model to improve the readability of our model, improve the accuracy of our results, and increase the number of classes of vulnerabilities we can identify. However, to preserve our ability to perform black-box analysis we make this process optional, as described in the previous section.

For this process we use the radare2 framework [75] to extract information about the binary. We select radare2 because it is light weight, can be interacted with programmatically, and has a wide variety of features in a single wrapper. Firstly, we create a symbol table for the target program, so we can perform symbol resolution on the blocks from the dynamic analysis step. This process involves identifying which parts of a program were executed using human-comprehensible symbols. In this instance, we aim to identify the function names associated with each basic block executed. We create a table mapping addresses to function names, and for each block, we perform address translation using the base address of the binary and the base address of the page in memory. In this mapping, we also store the sizes of each block, and we use the combination of block address and block size to extract bytes corresponding to unseen blocks in the static analysis step.

We extract the location and size of the `rodata` segment to retrieve static constants referenced during execution. At runtime, we can see the address being referenced but not the actual values themselves. This typically includes things such as format strings, which would be important for checking for format strings errors. We perform a check that an address is within the `rodata` segment and if so we extract the value at that address.

We use debug information to assign types to the variables in our model. This is particularly important for statically allocated variables or variables that have user-defined types, as this information is not available otherwise.

Finally, we allow the user to provide a set of C header files from which we extract type information, constants, and information about the fields in structs. For structs, we extract the type and offset of each field within the struct. We extract constants and assign labels to variables that are instances of these constants as they have special meaning within the program, which can be used to describe higher-level semantics. For example, for a program using seccomp we can describe which filters it is using and how they are implemented by mapping the variable's integer values to constants.

5.4 Concrete Execution

In this section, we describe our method of tracing a program's execution. Our method requires a great deal of visibility into the behavior of not only the target program but also the environment it is being executed in. For this reason, we use panda-re [4], a powerful full-system emulator built upon QEMU. Panda-re has many desirable features, including that it is multi-architecture, with support for x86-64, ARM, and MIPS. It is extensible, allowing users to define their own tracing methods and extensions to the platform. Tracing is performed by adding callbacks before significant low-level operations in the emulation workflow such as virtual memory reads and writes and basic block translation.

Panda-re emulates a guest operating system on which we execute target programs. A compatible guest image can be created using the qemu-img tool [74] to convert an existing virtual drive to a qcow2 file. During the evaluation and development of Hy2 we use a Ubuntu 20.04 x86_64 headless server image as well as the default preconfigured images provided by panda-re. We use a snapshot to maintain a consistent system state and restore this snapshot before the execution of each input example. Target programs and their dependencies are stored in a directory that is mounted on the guest at launch, allowing for user-specified programs to be executed on the guest. We interact with the guest by sending commands and receiving output over a serial console. The guest system itself is not configured for the analysis of any particular target, so any dependencies or other setup required must be specified in the preconditions in its configuration file. All the interaction with the guest during the analysis of a target program is handled by Hy2 and performed automatically without user involvement.

The pypandare library [46] is a python library that allows users to extend panda-re's functionality further by allowing for callbacks to be created on-the-fly, and for control of the guest to be performed programmatically. We use pypandare to add breakpoints on the desired operations and implement callback handler functions.

We use the following callbacks to collect data about the instrumented system:

- We use the `before_block_exec` callback, which is triggered before the execution of each basic block and defines a handler function to fetch and dump the bytes of the basic block to be executed first checking that the address of the block is within the target scope. We also use this callback to dump all the values of the registers.

- We use the `on_task_change` callback to check for the start of the target process and collect information about the pages mapped into memory.

- We use the `cb_virt_mem_after_write` and the `cb_virt_mem_after_read` callbacks to trace virtual memory reads and writes. We record the address accessed and the value read or written. We also check whether this value is a pointer.

- We use the `hook_symbol` callback to collect information about the calls made to libraries mapped into memory and their arguments. The calls observed correspond to symbols that are exported.

- We implement handlers for the read and write system calls using the `on_sys_read` and `on_sys_write` callbacks. In the future we could do the same for other system calls if necessary.

We set a limit of 10 minutes for the time which Hy2 waits for the completion of the execution of the target program. If this limit is exceeded we still analyze its execution before that point. This timeout handles instances where a program does not terminate (e.g. infinite loops), allowing our method to always take a finite amount of time for its analysis. Even if this timeout is reached Hy2 will still perform an analysis of the observed execution behavior. We selected this timeout of 10 minutes because we determined through a preliminary analysis that it was sufficient for the completion of program execution as no programs reached this limit. We implemented a data structure ExecCapture that stores all the data collected at runtime that is passed on to subsequent analysis steps.

5.5 Translation and Abstraction

In this section, we describe details specific to the implementation of our abstraction of program behavior. The IR used in our implementation is VEX IR [31]. We translate the machine code of each unique basic block observed at runtime into this IR using the pyVEX library [76]. This IR serves as a low-level abstraction of a program's behavior on which we base our analysis. Its simple structure is designed for optimization and code generation, and is well-suited for our purposes. Additionally, this IR is SSA (static single assignment), meaning each temporary variable is only assigned to once. In this case, since we only process one block at a time, this is only within a block and not for the whole program.

From this IR we perform a further translation step. We split each IR expression into 3 parts: operator, data type, and endianness and encode it as a 3-byte string. Although this second translation step is largely redundant, it allows for a cleaner representation of each operator and allows us to introduce our own additional higher-level operators and abstractions without impacting data flow. We perform the step described in Section 4.4.4 to reduce this set of expressions into a set of expression trees. When performing this step we have a data structure representing each unique block, and within each block, we have a list of expression trees. The expression tree is implemented as a recursive data structure.

For each of these operators within our IR, we implement a handler function that performs the corresponding low-level operation for arbitrary inputs, and for each datatype, we create a conversion function to convert an arbitrary value to that specific type. With this, we are able to create a subsystem for which, for an arbitrary expression, we can simulate its execution. For a particular discrete time step in the execution of the program, we can look up the values operated on in our memory abstraction and compute the value produced by that expression. This functionality has various applications within our framework.

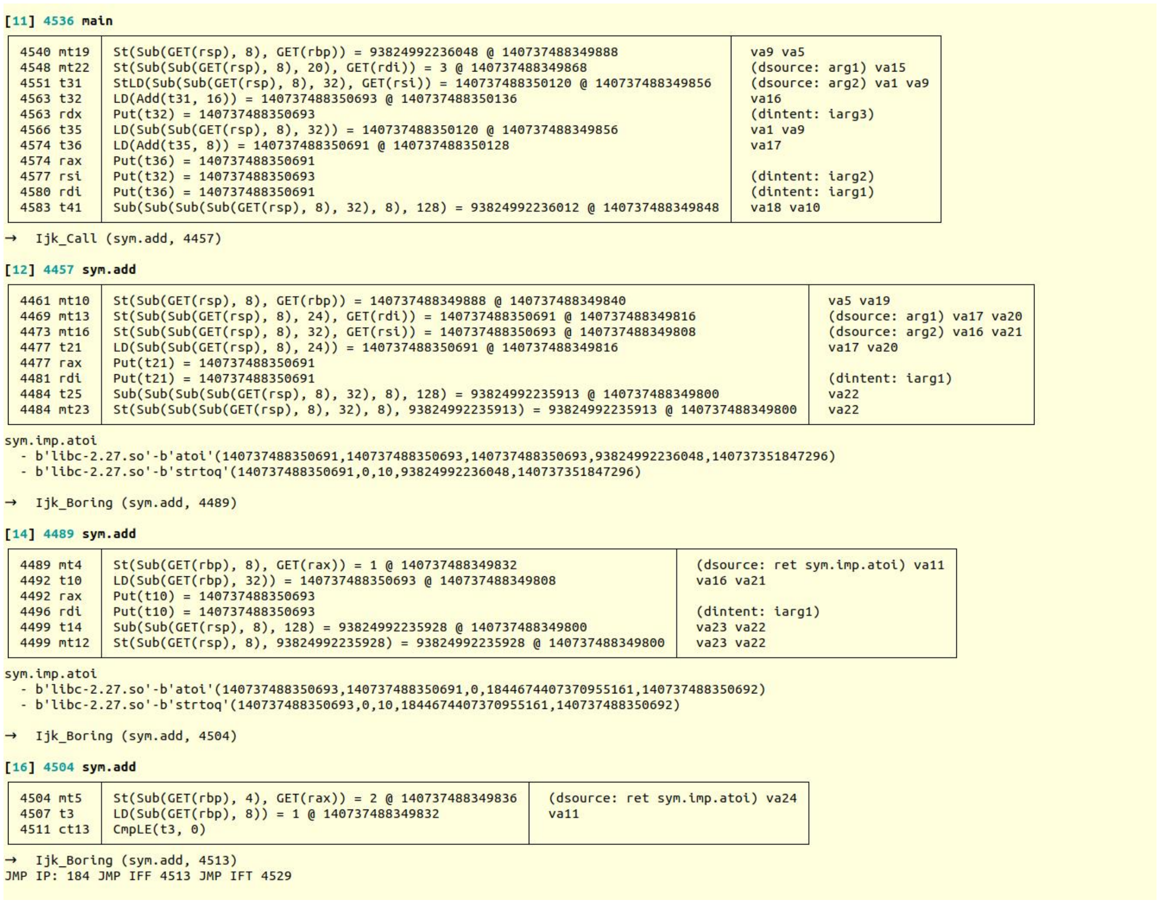
Once we have processed the set of unique basic blocks as described above, we derive a control flow graph from the ordering of the execution of said blocks. From our concrete execution step, we have an event sequence describing the sequence of basic blocks executed. This sequence consists of objects containing the address of the instruction at the start of the basic block and a jump operation, which includes the type of jump statement and the destination of the jump. We aim to transform this event sequence into a graph describing control flow. To achieve this, we iteratively merge nodes with the same address. We refine the edges in this graph to reflect these merges. For a given node x that we merge with y, we remove node x from the graph and redirect all edges with destination x to node y, and all edges with source node x to have source node y. We put the dropped node x into a mapping so that in subsequent steps when we introduce runtime data to y we know to merge this with data from node x. Next, we need to populate the nodes in this graph with the corresponding set of variable expressions found within the basic block.

When we introduce variable expressions to the nodes within our control flow graph, we also introduce the corresponding data observed at runtime across all executions of the basic block. For a given expression, if the operation performed consists of a memory access operation, we use our virtual memory mapping to look up the values and addresses at that particular point during the program's execution. For the case of operations involving registers, we keep track of the program state within the block. If the operation performed by the expression involves modifying a register, we compute the value that register would have at that particular time step using our functional representation of the variable expression and the current program state. If the expression writes to a register and is in the form `Put(x)`, we use our CPU simulator to perform the operation described by `x` to compute its value. For load and store operations we use our virtual memory mapping to look up the values and addresses accessed at the particular program point across all time steps for which the block is executed. We aggregate the values and addresses observed across all executions of the basic block. In this step, we also compute an approximation of data flow for each variable expression. We perform this step as part of the processing of the basic block, rather than as its own step, to reduce overhead. We apply labels to variable expressions to trace the data flow within the program. If a variable accesses a memory location we have seen before we assign it the corresponding label, otherwise we create a new label for the memory location, assigning it to this variable expression, and to subsequent variable expressions that use this location. This is a very primitive abstraction, as we do not consider things like re-assignment and lifetimes. However, the effects of this over-approximation are reduced significantly by the fact that we consider control-flow in our property-checking step, thus considering the reachability between variables using the same memory location. This abstraction is preferable because of its simplicity which allows it to scale effectively. Finally, we convert our expression trees to a vector. We use the structure of this vector to describe the properties of a particular expression in the property-checking step. For example, if we have an expression in the from of `STORE(Add(t12, 4), t32)`, we know we are describing memory access at the offset of 4 within a contiguous block of memory such as an array or a struct. We show an example of the composition of blocks within our model in Figure 5.6. For each variable expression within the block, we have the address of the corresponding instruction, a name, observed addresses and values, an expression describing the operation performed, and labels describing its data-flow dependencies. We use a field `dsource` to label variables that operate on function input arguments or return values, and a field `dintent` to label variables used as arguments in a function call. For the case of function arguments this label describes its position (e.g. `arg1,...,argn`).

Our final abstraction represents blocks, referred to as nodes, as a series of variable expressions and describes transitions between these blocks. The control flow graph we create is stored separately from the underlying node objects. Each node object has an identifier and can be mapped to a corresponding node in the control flow graph. We deserialize and save the CFG produced in this step for each execution of the target program in a separate file.

5.6 Augmenting our Method with Static Analysis

To implement our static analysis step we use our block mapping to select raw bytes from the target binary. We reuse the subsystem and data structures described previously to compute the values of the associated variable expressions across all instances, performing the following additional steps to compute and propagate the values of the variable expressions for the unseen blocks. We record the discrete time step t where we diverge from the observed execution flow. We define a secondary representation of the program state referred to as the SimState, which keeps track of the program state along the unseen branch. We compute the values for variable expressions within the block using our CPU simulator and memory mapping. If a variable reads from memory at address a we first check for the value of a in the SimState and then in our virtual memory mapping at discrete time step t. If a variable writes to memory at address a, we modify the SimState: if a is present we update its value, otherwise we add the address and its value. We use the same method described in 4.4.5 for



```
[11] 4536 main
 | 4540 mt19 | St(Sub(GET(rsp), 8), GET(rbp)) = 93824992236048 @ 140737488349888                     | va9 va5               |
 | 4548 mt22 | St(Sub(Sub(GET(rsp), 8), 20), GET(rdi)) = 3 @ 140737488349868                        | (dsource: arg1) va15  |
 | 4551 t31  | StLD(Sub(Sub(GET(rsp), 8), 32), GET(rsi)) = 140737488350120 @ 140737488349856         | (dsource: arg2) va1 va9 |
 | 4563 t32  | LD(Add(t31, 16)) = 140737488350693 @ 140737488350136                                  | va16                  |
 | 4563 rdx  | Put(t32) = 140737488350693                                                            | (dintent: iarg3)      |
 | 4566 t35  | LD(Sub(Sub(GET(rsp), 8), 32)) = 140737488350120 @ 140737488349856                     | va1 va9               |
 | 4574 t36  | LD(Add(t35, 8)) = 140737488350691 @ 140737488350128                                   | va17                  |
 | 4574 rax  | Put(t36) = 140737488350691                                                            |                       |
 | 4577 rsi  | Put(t32) = 140737488350693                                                            | (dintent: iarg2)      |
 | 4580 rdi  | Put(t36) = 140737488350691                                                            | (dintent: iarg1)      |
 | 4583 t41  | Sub(Sub(Sub(Sub(GET(rsp), 8), 32), 8), 128) = 93824992236012 @ 140737488349848         | va18 va10             |

→  Ijk_Call (sym.add, 4457)

[12] 4457 sym.add
 | 4461 mt10 | St(Sub(GET(rsp), 8), GET(rbp)) = 140737488349888 @ 140737488349840                              | va5 va19                   |
 | 4469 mt13 | St(Sub(Sub(GET(rsp), 8), 24), GET(rdi)) = 140737488350691 @ 140737488349816                     | (dsource: arg1) va17 va20  |
 | 4473 mt16 | St(Sub(Sub(GET(rsp), 8), 32), GET(rsi)) = 140737488350693 @ 140737488349808                     | (dsource: arg2) va16 va21  |
 | 4477 t21  | LD(Sub(Sub(GET(rsp), 8), 24)) = 140737488350691 @ 140737488349816                               | va17 va20                  |
 | 4477 rax  | Put(t21) = 140737488350691                                                                      |                            |
 | 4481 rdi  | Put(t21) = 140737488350691                                                                      | (dintent: iarg1)           |
 | 4484 t25  | Sub(Sub(Sub(Sub(GET(rsp), 8), 32), 8), 128) = 93824992235913 @ 140737488349800                  | va22                       |
 | 4484 mt23 | St(Sub(Sub(Sub(GET(rsp), 8), 32), 8), 93824992235913) = 93824992235913 @ 140737488349800        | va22                       |

sym.imp.atoi
  - b'libc-2.27.so'-b'atoi'(140737488350691,140737488350693,140737488350693,93824992236048,140737351847296)
  - b'libc-2.27.so'-b'strtoq'(140737488350691,0,10,93824992236048,140737351847296)

→  Ijk_Boring (sym.add, 4489)

[14] 4489 sym.add
 | 4489 mt4  | St(Sub(GET(rbp), 8), GET(rax)) = 1 @ 140737488349832                          | (dsource: ret sym.imp.atoi) va11 |
 | 4492 t10  | LD(Sub(GET(rbp), 32)) = 140737488350693 @ 140737488349808                     | va16 va21                        |
 | 4492 rax  | Put(t10) = 140737488350693                                                    |                                  |
 | 4496 rdi  | Put(t10) = 140737488350693                                                    | (dintent: iarg1)                 |
 | 4499 t14  | Sub(Sub(GET(rsp), 8), 128) = 93824992235928 @ 140737488349800                 | va23 va22                        |
 | 4499 mt12 | St(Sub(GET(rsp), 8), 93824992235928) = 93824992235928 @ 140737488349800       | va23 va22                        |

sym.imp.atoi
  - b'libc-2.27.so'-b'atoi'(140737488350693,140737488350691,0,18446744073709551615,140737488350692)
  - b'libc-2.27.so'-b'strtoq'(140737488350693,0,10,18446744073709551615,140737488350692)

→  Ijk_Boring (sym.add, 4504)

[16] 4504 sym.add
 | 4504 mt5  | St(Sub(GET(rbp), 4), GET(rax)) = 2 @ 140737488349836 | (dsource: ret sym.imp.atoi) va24 |
 | 4507 t3   | LD(Sub(GET(rbp), 8)) = 1 @ 140737488349832           | va11                             |
 | 4511 ct13 | CmpLE(t3, 0)                                         |                                  |

→  Ijk_Boring (sym.add, 4513)
JMP IP: 184 JMP IFF 4513 JMP IFT 4529
```

Figure 5.6: An example of the structure of basic blocks within our control-flow-based abstraction of a program. Each block consists of a sequence of labeled expressions and a set of outgoing edges.

Program	Coverage		Analysis Time	
	Dynamic	Hybrid	Dynamic	Hybrid
capsh	28.3%	94.66%	0.47s	3.22s
stat	17.44%	64%	2.4s	28s
ls	20.7%	77%	5.63s	88s
id	16.1%	57.8%	0.71s	8.9s
file	32.13%	51.9%	0.49s	1.16s

Table 5.2: Comparing the coverage and analysis time with dynamic versus hybrid analysis with a single test case on various coreutils programs.

computing data dependencies to compute data dependencies for these new variables.

As this functionality is primarily designed for exploring unseen branches within a function (i.e. intraprocedural analysis) and not the entire program and we don't want to introduce the possibility for our branch exploration to not terminate or introduce significant overhead, we select a maximum depth of 10 for which this search can diverge from observed execution flow. This depth was selected largely arbitrarily and future work should explore different limits.

We tested this addition on several coreutils programs as shown in Table 5.2 and we found that for a single test case, we are to observe a significant increase in coverage. We see different percentages of coverage between these programs as they all do not have the same number of conditional jumps, and because they have different numbers of basic blocks and our max depth parameter limits how far we explore from the observed execution flow. We conclude that although this step does not manage to cover all blocks, it serves its purpose of allowing us to cover branches that would be difficult to observe without unexpected inputs.

5.7 Property Language

In this section, we describe the implementation of our property language Hy2-lang. Security properties for Hy2 are initially defined in a text file using the syntax described in Section 4.6.2. For the interpretation of these property definitions, we wrote a conversion script. This script reads in a property, checks the correctness of the syntax used and converts it to a JSON file. We introduce this conversion step because while the plaintext format is more human comprehensible than the JSON format, it is more complicated to interpret; thus converting it to JSON allows for the parsing and syntax checking to be a one-time cost. The resulting JSON files are then read by the checker and converted to a state-based representation in the property-checking step. Over the course of our evaluation and testing we implemented 23 properties which are listed in A.

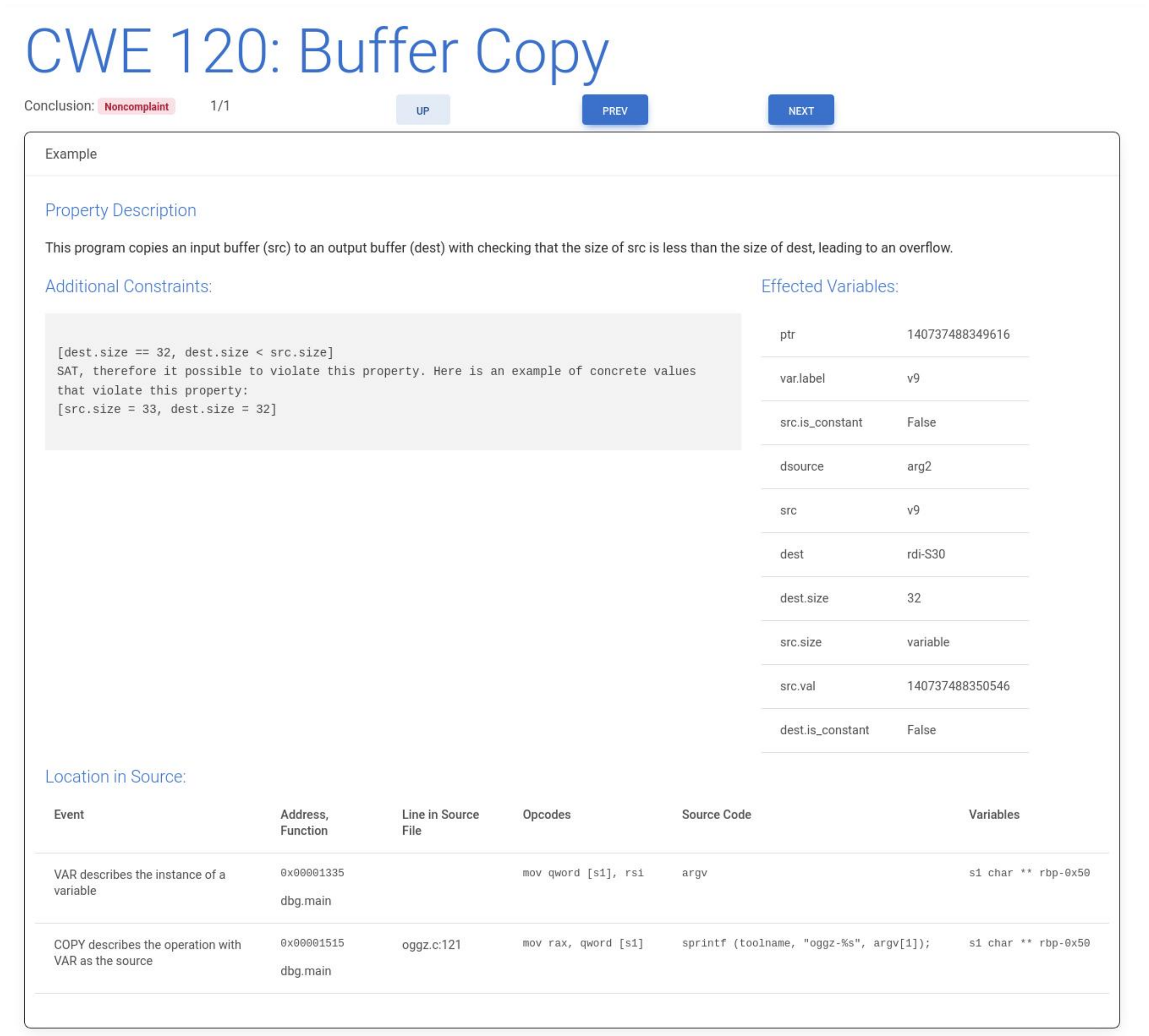
5.8 Rule Checking

In this section, we describe our implementation of the process of checking our security properties against the derived model of the program. We begin by loading the saved model of the program. This model consists of a control-flow graph in which each node can be mapped to an object holding its underlying data. This object consists of a set of variable expressions, a set of calls to external libraries, and a description of its jump statement, which corresponds to its outgoing edges in the control-flow graph. We then load the selected security property to be checked. This property becomes a finite state or a set of finite state machines in which each transition can be mapped to an object, describing the event, the event's attributes, and the set of propositions that must hold to trigger a change of state. To check this property model against our control-flow-based model of the program, we must perform a graph traversal on the program model. We begin by iterating through all node objects to attempt to locate events in the program model that match the initial state in the property model. When we check each node we are checking the node object itself. If the event describes a variable operation, we loop through the set of variable expressions within the model, checking each one. If we are unable to locate events in the program model that match the initial state in the property model, then we can conclude that the property is satisfied in the program and our analysis is completed. If we are able to locate events in the program model that match the initial state in the property model, these become the initial set of candidates with which we populate our candidate queue. Each candidate object stores the current model state, the previous model state, the current property state, an attribute table, a call stack, and a trace of previously visited states. This attribute table describes the values of the set of attributes corresponding to the attributes within the property model, which are used to check against the set of predicates in the property model. We use multi-threading to handle multiple candidates simultaneously, with each thread handling a separate candidate. To check each candidate, we perform a separate depth-first traversal of the program model, beginning from the current model state stored in the candidate object. For each traversal performed for a candidate, we have a secondary queue, which stores branches we have not visited, as we continue along a single path until it is exhausted. We continue our traversal for a particular candidate until this queue is empty. For each event that matches the current property state, we extract its attributes and, if applicable check them against its propositions using the attribute table to hold attributes from previous states. If these propositions are not satisfied, we do not consider this event as a match. Upon the discovery of an event that matches the current property state, a new candidate is added to the candidate queue, with the current global state, and the successor state in the property model. We continue our traversal until all paths are exhausted with the current candidate with the goal of locating all possible paths from the current state to an `ERROR` state.

Upon reaching an `ERROR` state, we convert the error trace, and each node object that causes a change of state in the property to JSON and save it to a file. Finally, we perform a secondary feasibility analysis with this error trace. For this step, we use z3 [50], an SMT solver to build and check the variable domains against the data-driven constraints described in the property model and derive a concrete counterexample if possible. We collect the set of conditions along the error path, along with their satisfiability and present this in our results alongside the `ERROR` trace in order to describe the feasibility of this path. We add this secondary feasibility analysis to the JSON file of the particular result. These results can be viewed via the web application we implement, as described in the following section.

5.9 Results Formulation

In this section, we describe our approach to the presentation of the results of our analysis. One of the challenges of any vulnerability analysis tool is communicating results effectively to the user. They must be able to understand why a particular error was reported and should be given enough information to be able to take steps toward correcting it. This information is usually presented as either concrete inputs that trigger the fault or the locations in the code that cause the fault. We favor the latter over the former, as we believe it is more useful for the developer to see exactly where the problem is in their program rather than having to work backward by tracing the execution for the given inputs. For each potential fault identified, we display the events in the program model that match those in the property, a call trace, the conditions under which the affected branch is reachable, and the constraints on any affected variables if applicable. We also identify the locations within the program binary corresponding to the events in the particular security property violated, reporting on their address. If debugging information is available we are able to provide the exact lines in the source. Finally, we also list the inputs from our original workload that triggered the vulnerable branches. We show an example of the format of our results in Figure 5.7. This application is a prototype and other features such as the prioritization of results based on severity are left to future work.



CWE 120: Buffer Copy

Conclusion: Noncomplaint 1/1

Example

Property Description

This program copies an input buffer (src) to an output buffer (dest) with checking that the size of src is less than the size of dest, leading to an overflow.

Additional Constraints:

```
[dest.size == 32, dest.size < src.size]
SAT, therefore it possible to violate this property. Here is an example of concrete values
that violate this property:
[src.size = 33, dest.size = 32]
```

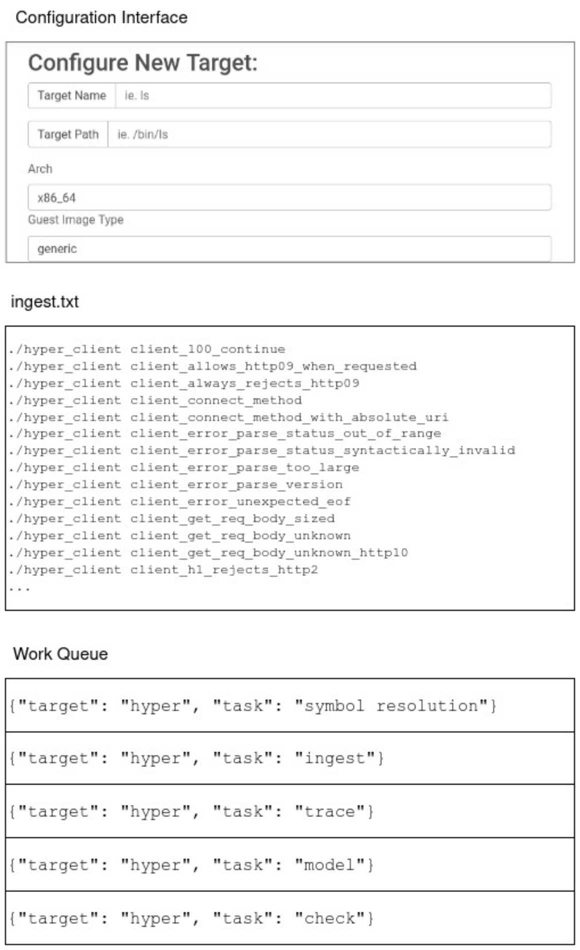
Effected Variables:

ptr	140737488349616
var.label	v9
src.is_constant	False
dsource	arg2
src	v9
dest	rdi-S30
dest.size	32
src.size	variable
src.val	140737488350546
dest.is_constant	False

Location in Source:

Event	Address, Function	Line in Source File	Opcodes	Source Code	Variables
VAR describes the instance of a variable	0x00001335 dbg.main		mov qword [s1], rsi	argv	s1 char ** rbp-0x50
COPY describes the operation with VAR as the source	0x00001515 dbg.main	oggz.c:121	mov rax, qword [s1]	sprintf (toolname, "oggz-%s", argv[1]);	s1 char ** rbp-0x50

Figure 5.7: A screenshot of our web interface showing the format of one of the errors returned.



Configuration Interface

Configure New Target:

Target Name: ie. ls

Target Path: ie. /bin/ls

Arch: x86_64

Guest Image Type: generic

ingest.txt

```
./hyper_client client_100_continue
./hyper_client client_allows_http09_when_requested
./hyper_client client_always_rejects_http09
./hyper_client client_connect_method
./hyper_client client_connect_method_with_absolute_uri
./hyper_client client_error_parse_status_out_of_range
./hyper_client client_error_parse_status_syntactically_invalid
./hyper_client client_error_parse_too_large
./hyper_client client_error_parse_version
./hyper_client client_error_unexpected_eof
./hyper_client client_get_req_body_sized
./hyper_client client_get_req_body_unknown
./hyper_client client_get_req_body_unknown_http10
./hyper_client client_h1_rejects_http2
...
```

Work Queue

```
{"target": "hyper", "task": "symbol resolution"}
{"target": "hyper", "task": "ingest"}
{"target": "hyper", "task": "trace"}
{"target": "hyper", "task": "model"}
{"target": "hyper", "task": "check"}
```

Figure 5.8: The key part of the user's workflow within our framework, which includes configuration, specifying the workload, and setting up the set of jobs that perform the analysis process.

5.10 User Workflow

In this section we describe the steps taken by a user when analyzing a new program in our framework. In Figure 5.8 we show the key parts of the configuration process to illustrate its simplicity.

A user would begin by using the web interface to create a configuration profile for the program they wish to analyze. This process involves filling in fields corresponding to the options described in Section 5.2. Upon the completion of this process, Hy2 will automatically create a template configuration file for the new program with the selected options. Additionally, a symbol resolution task gets added to the work queue which when run will create a mapping of debug symbols and basic blocks within the specified binary, and save it to a JSON file. Next, the user must provide input examples that make up the workload. There are multiple ways this can be done within our framework. The user can specify input examples as commands or as JSON in the file `ingest.txt`. Then they would have to add a job to the work queue with task "ingest". When this job is run Hy2 will read commands specified in `ingest.txt` and create a configuration file for each input example using the template configuration created in the previous step. Input examples can also be added via the web interface however this process is slower as they must be added one at a time.

We anticipate that the task of creating a sufficient set of input examples may be potentially time-consuming. However, a potential way around this may be to reuse an existing set of integration or functional tests. Integration tests determine if software components or features tested separately function correctly when connected. Functional testing determines if a feature or component meets requirements. Both of these types of testing typically exercise large parts of the program, interacting with it as a user would. Additionally, they are typically designed to cover edge cases that we would miss without knowledge of the inner workings of the target program. Unit tests could also be used potentially, but they would be less effective as they only test single components and not the program as a whole. The set of input examples produced by a fuzzer could also potentially be used as input as they are crafted specifically to exercise different paths. In other cases, the user would likely have to read the documentation to understand the different features of the program and create different input examples to exercise each one. The amount of effort required for this task may be comparable to the task of writing functional tests as the role they serve is the same. In the case that the target program can't be interacted with via the command line, and input examples can't be specified as commands, a test harness will need to be created. The test harness, in this case, is a script that is run from the command line to interact with the target program. An example where this might be necessary is when analyzing a network service. The service would be launched and traffic needs to be sent to it.

After specifying the workload the user adds a job to the work queue with the task "trace". When run this job will instruct Hy2 to perform its analysis for all input examples. Once this process has completed the user must run Hy2 with the option "model" to build the program model, and finally with the option "check" to check properties against this model. The results of this analysis can viewed via the web interface.

To summarize, Hy2 takes in a set of configuration options and a set of input examples as input. Optionally, it may take in source code and semantic information. We do not consider the security properties themselves to be input as they can be reused between analyses. In comparison, AFL requires a single input example to mutate and the target program as input. Symbolic execution methods typically require the binary or source code and a set of assertions to check. Such methods may also require the insertion of assertions or annotations in the program source, or the specification of an environmental model, this being a model of the behavior of environmental dependencies, or a set of handler functions that describe the domain of inputs and outputs for external functions called by the program. Although the creation of an environmental model and checkers may be time-consuming, they are largely a one-time cost as they can likely be reused between analyses. The primary disadvantage our method has over other methods with regards to configuration is the need for the user to specify input examples. We discuss this further in our evaluation (Section 6.4) and when discussing the limitations of our method (Section 7.3).

Chapter 6

Evaluation

In this chapter, we present an evaluation of Hy2, through several case studies which serve to highlight its various capabilities.

To evaluate Hy2 we have selected several programs, each which provide a set of regression or integration tests. We aim to demonstrate that this tool can be easily integrated into a development cycle without any modification to the selected programs or to the tool itself. We use the provided set of tests as input to exercise each program and show that we can uncover vulnerabilities affecting it. It is important to note that while the programs selected for evaluation are those that provided a set of integration tests, this method is not limited to exclusively these programs. Alternatively, a program with a set of unit tests or a set of commands could also be used.

We selected programs with known vulnerabilities and primarily evaluate Hy2's ability to identify these vulnerabilities. We use this approach to our evaluation, rather than trying to locate undiscovered vulnerabilities as we believe it is equally important to evaluate which flaws may be missed by this method. During the course of our evaluation, we were able to locate several additional unreported vulnerabilities. We did not report these as later updates eliminated the vulnerable code or the identified faults were not or are no longer exploitable. We evaluate this tool based on coverage, the vulnerabilities identified and missed, the time required for analysis, and the quality of the analysis.

The rest of the chapter proceeds as follows. First we illustrate how we can identify the vulnerability in sudoedit described in our motivating example in Section 3.1. Next, we demonstrate our approach to handling concurrency and identifying vulnerabilities caused by concurrent scenarios. Last, we perform an in-depth analysis of the capabilities of our method, by evaluating it against three different real-world programs.

6.1 Experimental Setup

The entirety of our evaluation was completed on a machine running Ubuntu 20.04 with a single 3.2GHz AMD Ryzen 7 2700 CPU and 32 GiB of RAM. Our guest is running Ubuntu 20.04 and is given 8192 MiB RAM. To exercise the selected programs we used the integration or regression tests included in their respective repositories. These tests are ideal for exercising the target programs as they also set up the environment prior to execution, setting environment variables, changing configurations and creating necessary files. Additionally, the format of these tests allows for us to observe the complete execution of the binary including its command-line interface, which is not possible with unit tests, as they typically only target specific functions.

Each test case is treated as a separate input, which we use to exercise the target binary. For instances where the test suite is composed of a set of files, we copy the test suite into a folder, which then is copied to the guest when it is launched. During the configuration process, we provide a file with a list of commands to our framework, which gets automatically processed into a set of configuration files, this being the workload. During our analysis step these files instruct our framework how to set up the guest and which commands to execute. For example, for the case of wget we specify a list of commands in the form: `./Test-ftp-recursive.px`. In this case the test script itself executes wget whose behavior we analyze. In other cases where the

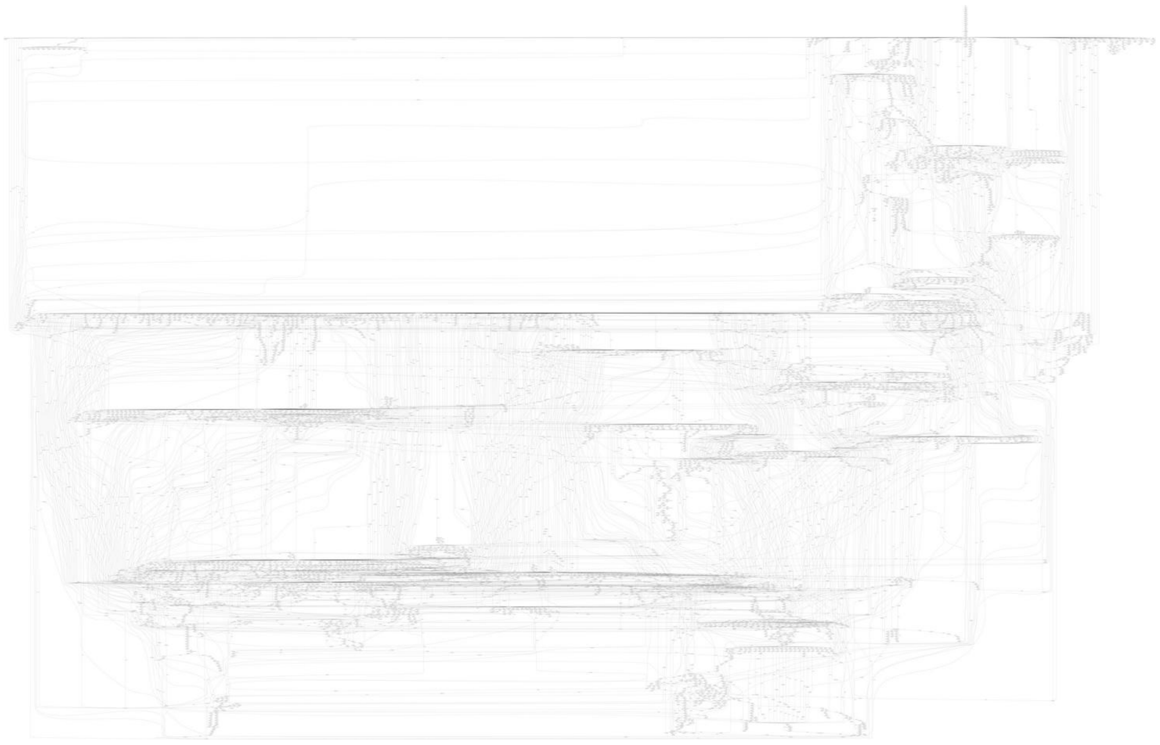
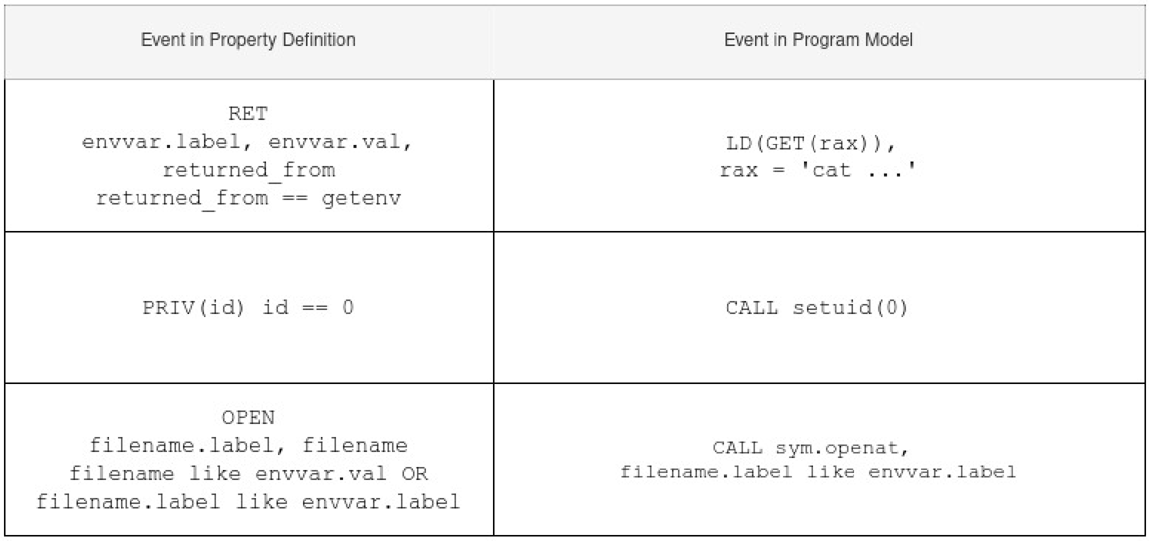


Figure 6.1: The control-flow-based model of the sudoedit program produced by Hy2.

test suite takes the form of a set of commands we pass that list of commands to our framework and each command becomes an input example in the workload.

6.2 Revisiting our Motivating Example

This section demonstrates the effectiveness of our tool against a vulnerability in the sudoedit program as described in our motivating example in Chapter 3. At a high level, this vulnerability involves a privileged operation being performed with data from an untrusted source. We describe the property in our policy language in Listing 10. We represent this behavior as `getenv()` followed by a change of user then followed by a sensitive operation that is influenced by the result of the `getenv()` call. We define sensitive operations in this context as those that access a file or execute a program; however, this definition is not exhaustive. With this property, we are looking for instances where the value that is being returned from `getenv()` is the same as what



Event in Property Definition	Event in Program Model
RET envvar.label, envvar.val, returned_from returned_from == getenv	LD(GET(rax)), rax = 'cat ...'
PRIV(id) id == 0	CALL setuid(0)
OPEN filename.label, filename filename like envvar.val OR filename.label like envvar.label	CALL sym.openat, filename.label like envvar.label

Figure 6.2: The events in the program model that matched on the events in our property "Data from an untrusted source used in a privileged operation".

is being passed into the sensitive operation. If this is the case, then an unprivileged user could be able to influence the behavior of the privileged operation.

Because our method can recognize that the source of `nargv` (Listing 1) is the `getenv()` call and `nargv` is the source of `files` which leads into the `setuid(0)` and `files` being modified with elevated privilege, it reports this as a potential vulnerability. In our model of sudoedit, we are able to see the program calling `getenv()`, then `setuid()`, and finally `openat()`, where the file being opened is like the value returned from the call to `getenv()`, as shown in Figure 6.2.

In the final stage of our analysis, we attempt to evaluate feasibility by examining conditions on the reported error paths. This analysis would report that `strcmp(*ap, "--") == 0`. The patch for this vulnerability checks that the `editor` string does not contain `"--"`, so performing the same analysis reveals a contradiction: `strcmp(*op, "--") == 0` and `!strcmp(nargv[nargc], "--") == 0`, thus the vulnerable path is no longer possible. We would conclude that there can't be an instance where `getenv("EDITOR")` can reach the branch where `files` is set and thus influence the sensitive operation, in this case allowing a user to edit files with elevated privilege.

```
---
RET
with envvar.val = val, envvar.label = label, returned_from =
 ↪  ret.fname
where returned_from == getenv

SETPRIV
with id = id
where id == 0

NOT SETPRIV
with id0 = id
where id0 > 0

OPEN
with filelabel = filename.label, filename = filename
where filelabel like envvar.label
---
RET
with envvar.val = val, envvar.label = label, returned_from =
 ↪  ret.fname
where returned_from == getenv

SETPRIV
with id = id
where id == 0

NOT SETPRIV
with id0 = id
where id0 > 0

EXEC
with filelabel = filename.label, filename = filename, args.label =
 ↪  args.label
where filelabel like envvar.label OR args.label like envvar.label
---
```

Listing 10: Our property "Data from an untrusted source used in a privileged operation" used in our evaluation of sudoedit.

This method of analysis is powerful because the property itself doesn't describe anything specific to the program and can be applied to other programs without modification. Because this property is generic and can be applied to different programs, it may be the case that the behavior described in the property may be the behavior intended by the developer, however, it is still dangerous and we believe it is worthwhile to draw attention to.

To conclude, through the completion of this evaluation we demonstrated our ability to identify the described vulnerability in sudoedit using its preexisting test suite, a generic property, and our hybrid analysis method that is able to describe environmental interactions, thus filling the gap discussed in Chapter 3.

6.3 Concurrency Example

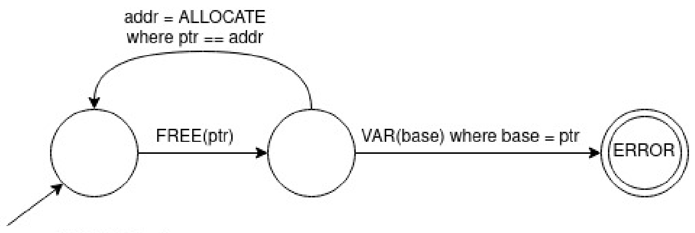
In this section, we present a case study showing how our method handles concurrency. This example is based on CVE-2020-0424 [72], a use-after-free vulnerability in the binder driver. Binder is an Android-specific kernel component that allows for communication between processes [30]. This vulnerability is caused by a race-condition due to improper locking. We show a simplified version in Listing 11 of the vulnerable code for brevity. On line 7 in thread `t2`, the program releases the lock on `work_mutex`. If thread `t1` is able to free `w`, before the check on line 18 in `cleanup_worker_jobs` but after the check on line 14, then a use-after-free will occur when thread `t2` checks `w->type`.

We are able to identify this vulnerability with the property described in Figure 6.3. The `FREE` event corresponds to the set of calls that perform the freeing of memory, with `ptr` matching on the argument corresponding to the address of the memory being freed. The `ALLOCATE` event corresponds to a set of calls that perform the allocation of memory with `addr` being the address of the newly allocated memory. Finally, the `VAR` event corresponds to memory access operations, where the memory access involves arithmetic on a base address. For instance, access to a field in a struct would take the `base + offset`. The proposition `base == ptr` is checking if it is possible for the base address to be equal to the address being freed. Since we are able to identify that it is possible for `FREE` to occur before `USE` we can identify that the property is violated. In Figure 6.4 we show the events in our model that correspond to the events in our property.

As discussed in Section 4.3.2 our method has limitations with regards to how it

```c
pthread_mutex_t work_mutex = PTHREAD_MUTEX_INITIALIZER;

struct WorkItem* dequeue_work_item(struct Worker *worker){
    struct WorkItem *w;
    pthread_mutex_lock(&work_mutex);
    w = dequeue_remove_job(worker->jobs);
    pthread_mutex_unlock(&work_mutex);
    return w;
}

void cleanup_worker_jobs(struct Worker *worker){
    struct WorkItem *w;
    while(1){
        w = dequeue_work_item(worker);
        if(!w){
            return;
        }
        if(w->type > 0){
            ...
        }
    }
}

void cleanup_todo_list(struct Worker *wq){
    struct WorkItem *w;
    int i = 0;
    while(1){
        w = wq->jobs[i];
        if(!w){
            break;
        }
        pthread_mutex_lock(&work_mutex);
        free(w);
        wq->jobs[i] = NULL;
        pthread_mutex_unlock(&work_mutex);
        i++;
    }
}

int main(){
    ...
    pthread_create(&t2, NULL, cleanup_worker_jobs, worker);
    pthread_create(&t1, NULL, cleanup_todo_list, worker);
    ...
}
```

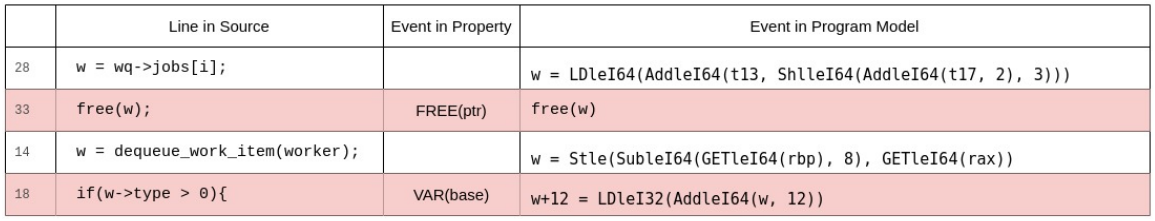
Listing 11: A snippet showing the vulnerable sections of the program used to describe our ability to handle concurrency.



```
FREE(ptr)
NOT addr = ALLOCATE() where addr == ptr
VAR(base) where ptr == base
```

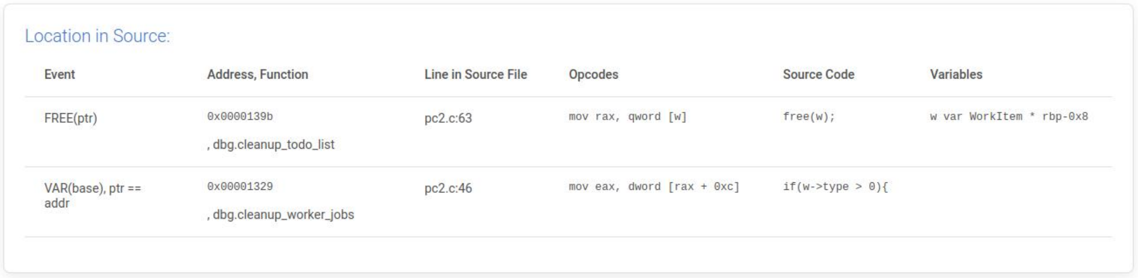
Figure 6.3: The FSM encoding of the use-after-free property used in our evaluation for our concurrency example.

handles concurrency. We do not exhaustively model all possible interleavings, only considering potential sequences of events that can be derived from our observations. In this case, to identify the aforementioned vulnerability we would have to observe at least one iteration of the loop in `cleanup_todo_list` in thread `t1` execute before `cleanup_worker_jobs` in thread `t2`. Although there may be a high probability that we will observe the vulnerable sequence of events in this instance, this may not be the case in other scenarios. Additionally, even if we were to execute this program many times it is not guaranteed that the vulnerable sequence of events will ever occur. As such, this introduces the potential for false negatives when attempting to identify vulnerabilities arising from concurrent scenarios. The task of computing the Cartesian product of the behavior of threads to describe all interleavings may introduce a state space explosion, so we do not attempt to address it here, as our method is designed to be general purpose, although it would be possible to introduce such capabilities. Past methods typically struggle with concurrency [85] [32]. Our method can reason about the behavior of this program even if we don't have a complete model of the state space, allowing us to overcome this limitation. We discuss the implications of this in Chapter 7.



	Line in Source	Event in Property	Event in Program Model
28	`w = wq->jobs[i];`		`w = LDleI64(AddleI64(t13, ShlleI64(AddleI64(t17, 2), 3)))`
33	`free(w);`	FREE(ptr)	`free(w)`
14	`w = dequeue_work_item(worker);`		`w = Stle(SubleI64(GETleI64(rbp), 8), GETleI64(rax))`
18	`if(w->type > 0){`	VAR(base)	`w+12 = LDleI32(AddleI64(w, 12))`

Figure 6.4: The events in our model of the program and the corresponding lines in the source code that match the events in our property.



Location in Source:

Event	Address, Function	Line in Source File	Opcodes	Source Code	Variables
FREE(ptr)	`0x0000139b` , dbg.cleanup_todo_list	pc2.c:63	`mov rax, qword [w]`	`free(w);`	`w var WorkItem * rbp-0x8`
VAR(base), ptr == addr	`0x00001329` , dbg.cleanup_worker_jobs	pc2.c:46	`mov eax, dword [rax + 0xc]`	`if(w->type > 0){`	

Figure 6.5: The presentation of the reported fault in our web application, with the two events that describe the vulnerable behavior being shown with the corresponding address in the binary and line in the source file.

6.4 Case Studies

In this section, we perform an in-depth analysis of the practicality of our method, approaching this evaluation as we expect a user of this tool. We performed our evaluation on the following programs: wget, a command-line tool for downloading files from the internet; htmldoc, a program that converts markdown documents or web pages to various formats; and hyper, an HTTP protocol implementation written in rust. We provide information about these programs in Table 6.1.

For each of these programs we provide the corresponding binary with debug information as input to our framework. We use the regression or integration tests that come with each program as the workload. We analyze each program by executing each input in the workload one at a time, building a single model after the execution of the entire workload. Although some inputs within this workload may exercise the same branches of the program multiple times, we choose to include them regardless

Program	Number of Test Cases	Lines of Code
wget 1.19.5 [11]	87	97K
htmldoc 1.9.11 [26]	21	95K
hyper 0.14.9 [22]	130	16K

Table 6.1: Overview of the scale of the programs evaluated in our case studies. The test cases listed in this table is what we use as the workload for the corresponding program.

so as to not bias our analysis. Wget represents an ideal case, as each test script is a separate entity, exercising a different functionality and will perform the necessary setup and teardown automatically. This includes setting up network services mimicking HTTP and FTP services when testing the associated features. In these instances, these services are configured to exercise particular branches of the program. As a user, configuring our framework for testing would be as simple as providing the list of testing scripts.

The test suites for other programs are less consistent. For programs that perform file conversions, it is common to provide a test suite of files, with the successful conversion of these files serving to indicate the program is functioning correctly. This approach, however, is less than effective, typically missing large portions of the program's functionality. For these cases, instead of providing test scripts we provide these files as input, and for each test we execute the program with the associated file. Since these test suites are not exhaustive, we do not expect them to provide the same quality of results.

In Table 6.2 we summarize the results of our evaluation, including estimated coverage, analysis time, and vulnerabilities identified. We are unable to describe the coverage of the program hyper, as we use radare2 to retrieve information about the basic blocks within the binary, and due to the size and complexity of the compiled crate radare2 hangs and is unable to return this information.

Program	Estimated Coverage	Analysis Time	Reported Vulnerabilities Identified
wget 1.19.5	76.6%	1.8 hours	CVE-2018-20483 [7], CVE-2019-5923 [8]
htmldoc 1.9.11	52%	4.1 hours	CVE-2021-26259 [14], CVE-2021-26252 [19], CVE-2021-23206 [18], CVE-2022-34033 [13], CVE-2021-23180 [16], CVE-2021-23158 [12], CVE-2021-33235 [15], CVE-2021-40985 [23] CVE-2021-23191 [17], CVE-2021-27114 [28], CVE-2022-0534 [27], CVE-2021-20308 [25]
hyper 0.14.9	n/a	7.2 hours	CVE-2021-32714 [21], CVE-2021-32715 [59]

Table 6.2: A summary of the results of our evaluation.

Step	Total Overhead
Total	4.11 hr
Trace Analysis	69.87%
BB Exec Callback	9.58%
VM Read	3.17%
VM Write	9.92%
Library Hook	0.25%
Syscall Hook	< 0.01%
Process Active Hook	< 0.01%
Panda Play	1%

Table 6.3: The time taken to analyze htmldoc broken down by analysis step.

6.4.1 Performance

When performing our evaluation we noted that a large percentage of the analysis time could be traced back to a few sources. We include an exact breakdown of the time taken to complete each step in Tables 6.4 and 6.3. For the case of wget, about 28.32% of the overhead of our analysis comes from the virtual memory read and write callbacks. This is because they are the most commonly hit callback, as virtual memory reads and writes may occur multiple times within any given basic block. As this is a somewhat significant percentage, it suggests that future efforts should go towards eliminating or minimizing our reliance on this callback.

Secondly, we found that not all test cases required the same analysis time. For the case of htmldoc, there is a single test case that accounts for 51.52% of the analysis time

Step	Total Overhead
Total	1.8 hr
Trace Analysis	18.96%
BB Exec Callback	4.74%
VM Read	18.61%
VM Write	9.71%
Library Hook	0.53%
Syscall Hook	< 0.01%
Process Active Hook	0.05%
Panda Play	12.68%

Table 6.4: The time taken to analyze wget broken down by analysis step.

taken. This test case involves converting an HTML document with several different types of elements including a table and images into a PDF file. The performance for behaviors such as processing files is still less than optimal in our implementation. This is partially related to the problem of the virtual read and write callback. However, it is more due to the fact that file processing typically reads in a file in a chunk at time in a loop, and each time this loop is executed the same sequence of blocks is typically executed. This process produces a sequence of basic blocks with length relative to the size of the file. The sequence of basic blocks produced by the htmldoc aforementioned test case was about a million blocks in length, with the number of unique basic blocks only being a small fraction of that at 1041 blocks. It is largely redundant to analyze any of the blocks in the sequence multiple times, as we are only interested in describing data flow rather than the values themselves in our analysis so much of this data is unused. This problem and its implications are, in our view, one the greatest flaws in our implementation. We expand on this further in Chapter 7. It is worth noting that the time required to build the model of program behavior is a one time cost, as this model can be tested against any of our security properties. By contrast, other methods [47][42][57] may build a separate model which requires a separate analysis for each property evaluated.

The time required for the rule-checking step is dependent on the security property.

To be more specific, it is dependent on the number of states in the model of the program that match the initial state in our property and the size of the model of the program. If the number of initial state matches is small, then the rule-checking process takes at most a few minutes, regardless of the size of the model, because the model has a finite number of states which can be explored quickly in parallel. If the number of initial state matches is large, this being numbers approaching a thousand, it becomes very slow, relative to the model size. This is due to the fact that for each candidate we must traverse the model exhaustively. For instance, if the initial state in a property is a `LOAD` event with no attributes or conditions, it will find every single load event in the model, and for each of these it will have to traverse the model looking for successor states. These traversals occur because our rule-checking step attempts to find every single instance of the property in our model of the program. Additionally, in our implementation of the rule-checking step we sacrifice speed for low memory consumption and stability through our depth-first search method, which checks a single path for a single candidate at time, meaning we may traverse the same paths many times across separate candidates.

6.4.2 Vulnerabilities Identified

In our analysis, we were able to identify all reported vulnerabilities for which we had observed the execution of the corresponding branches in the program and we had defined properties to describe the erroneous behavior. We provide a complete summary of the identified errors in Chapter B. For the program wget, we were able to identify CVE-2019-5923, a heap-based buffer overflow vulnerability, and CVE-2018-20483, an information disclosure vulnerability. We discuss the two information disclosure vulnerabilities affecting this version in the following subsection.

For the case of CVE-2019-5923, although we originally identified it with our write out-of-bounds property, we identified the potential to create a more specific property that explained the undesirable behavior more precisely. The write out-of-bounds property is a very weak property as it identifies behaviors that may be undesirable without being able to describe their cause, and as a result it can generate many spurious results.

We show a snippet of the vulnerable function in Listing 12. This vulnerability is specifically due to the fact that the widening of characters during conversion between character encodings can increase their width by up to 3 bytes, so the allocated size of the buffer defined on Line 24 would be insufficient, so it is possible to cause the program to write outside the bounds of this buffer. For example, if we were to call the `do_conversion()` function to convert a string from ISO 8859-15 to UTF-8 encoding, we could trigger this error. We use the example of a sequence of euro signs. The euro sign in ISO 8859-15 is `"\xa4"`, but when it is converted to UTF-8 it becomes `"\xe2\x82\xac"` which is 3 times the length of the original string, instead of 2 as expected.

There were two ways we were able to identify this behavior. Firstly, through our generic write out-of-bounds property and tracing of the behavior of the function `iconv()` in `libc` from when it was called by wget to when it returned we were able to identify the function `internal_utf8_loop()` writing to an index that could be greater than the size of the buffer. Secondly, we create a property that describes the weakness “CWE-176: Improper Handling of Unicode Encoding”. Specifically, we describe improper memory management when performing encoding conversions. When performing conversions between character encodings we must account for character widening. Wide characters (`wchar_t`) are 32 bits and characters are 8 bits, so the size of the buffer to write the converted string to should be at least 4 times that of the source buffer. This new property produces no false positives and was able to identify a second instance of this issue in the program in the function `convert_fname()`.

[1] src/iri.c:125:160 of wget version 1.19.1 available: https://www.gnu.org/software/wget/

```c
static bool do_conversion (const char *tocode, const char *fromcode,
↪  char const *in_org, size_t inlen, char **out)
{
  iconv_t cd;
  /* sXXXav : hummm hard to guess... */
  size_t len, done, outlen;
  int invalid = 0, tooshort = 0;
  char *s, *in, *in_save;

  cd = iconv_open (tocode, fromcode);
  if (cd == (iconv_t)(-1))
    {
      logprintf (LOG_VERBOSE, _("Conversion from %s to %s isn't
↪  supported\n"),
                 quote (fromcode), quote (tocode));
      *out = NULL;
      return false;
    }

  /* iconv() has to work on an unescaped string */
  in_save = in = xstrndup (in_org, inlen);
  url_unescape_except_reserved (in);
  inlen = strlen(in);

  len = outlen = inlen * 2;
  *out = s = xmalloc (outlen + 1);
  done = 0;

  for (;;)
    {
      if (iconv (cd, (ICONV_CONST char **) &in, &inlen, out, &outlen)
↪  != (size_t)(-1) &&
          iconv (cd, NULL, NULL, out, &outlen) != (size_t)(-1))
        {
          *out = s;
          *(s + len - outlen - done) = '\0';
          xfree(in_save);
          iconv_close(cd);
    ...
}
```

Listing 12: A snippet of iri.c from wget showing the vulnerable parts of the do_conversion function. [1]

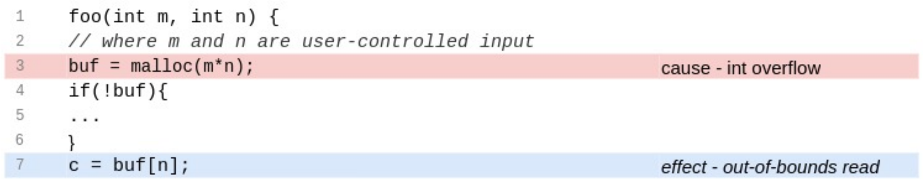
Of the 20 vulnerabilities that affected htmldoc, we were able to identify 14 of them. The three vulnerabilities that we were unable to locate in our analysis were on branches we were not able to cover due to the limited coverage provided by its test-suite. This tool makes a very good candidate for our evaluation because many of these vulnerabilities were caused by improper input validation, which is a very common source of error. Furthermore, we identified a number of additional errors that were not reported but we are able to verify that they may cause faults or dangerous operations. We found several additional read out-of-bounds errors, in `update_image_size()` and `get_cell_size()`, that were caused by the program attempting to read the first byte in an empty string, taking the form of: `if (buf[strlen((char *)buf) - 1] == '%')`. Additionally, we identify several errors in the function `image_load_bmp()` caused from a lack of input validation and error handling. The worst of these errors is that we can set the image dimensions to numbers large enough to cause an integer wraparound when computing the amount of memory needed for the image, `(width * height * depth)` (image.cxx:925), which causes invalid memory accesses and a subsequent crash.

We identify a write out-of-bounds error in `parse_table()` function caused by improper input validation. Although this has the same effect as CVE-2021-23206 [18], the reported error associated with this vulnerability wouldn't have covered this case. The original reported vulnerability sets the table `COLSPAN` attribute to an invalid integer, which causes `atoi()` to return -1, which is not handled by the program and causes an out-of-bounds read. This, however, is not necessary as the same error we discovered, which involves setting `COLSPAN` to any number large enough to cause it to attempt to write outside the bounds of the allocated table struct. For example, a table with `<TD COLSPAN="3404" ROWSPAN="2" ALIGN="CENTER">` would set `COLSPAN` to 3404, which is a valid integer but would still cause a crash. This could be missed by a developer as a proper description of the error was not provided. For instance, if they added a check on the return value of `atoi()` they would have not patched the error completely, leaving the program vulnerable.

In the `write_image()` function, it does not handle errors from the `load_image()` function, causing it to either write uninitialized values to the output, or if `image_load()` returns NULL, attempt to write to invalid addresses causing a crash.

Additionally, there are a potential null pointer dereference errors in `flatten_tree()`, `write_image()`, and a few other functions where memory is allocated and then accessed without checking it is successful. Finally, in many cases the program does dangerous operations that would be considered bad practice, such as the use of the unsafe functions, like `atoi()`, and `atol()`, and the use of `strlen()` on strings that are not null terminated. We report on these cases as well, regardless of whether they are vulnerable to exploitation, as if not addressed, there is the potential they could become vulnerabilities in the future.

With our evaluation for htmldoc and wget we saw a trend emerge. This being that we could describe vulnerabilities in terms of cause and effect. For example, in Figure 6.6 we show a simple read out-of-bounds vulnerability. Although this is a read out-of-bounds vulnerability, it is caused by an integer overflow. Whilst our initial approach of attempting to identify potential instances of read out-of-bounds behavior and backtracking to determine the bounds of the offset being read at was somewhat effective, it is much more effective to describe the source of the vulnerability, this being the integer overflow which leads to the allocation of insufficient memory resulting in the read out of bounds. This approach provides more useful results to a developer, as we are describing the actual bug itself, instead of just the effect that it has on the execution of the program. However, there are some cases where this method of describing vulnerabilities would be ineffective, as it assumes that the effect is always proceeded by some other erroneous behavior, this being the cause, which isn't always the case. This suggests that a combination of the two approaches



```
1   foo(int m, int n) {
2   // where m and n are user-controlled input
3   buf = malloc(m*n);                          cause - int overflow
4   if(!buf){
5   ...
6   }
7   c = buf[n];                                 effect - out-of-bounds read
```

Figure 6.6: An example used to illustrate the difference between cause and effect. In this example, the multiplication of m and n on line 3 could potentially result in a wraparound, causing an insufficient amount of memory to be allocated for buf and resulting in a read out-of-bounds error on line 7.

```c
objs = malloc(sizeof(struct ObjType *) * num_objs);
...
for(int i=0; i < n; i++){
    v = objs[i];
}
```

Listing 13: An example of a pattern of behavior we observed during our evaluation that may result in false positives.

described is necessary, with some properties describing potential causes, and others describing potential effects.

When considering the number of falsely reported errors, we made the following observations. All false positives both matched the behaviors described in our security properties and were valid paths through the program. For some cases, such as with the additional bug in wget these are legitimate weaknesses that hadn't been reported.

A majority of the falsely reported errors are returned by properties for which we use the technique of identifying instances of behavior that could be vulnerable under a certain set of conditions and then for each instance using our feasibility analysis step to determine if these conditions are actually possible. These properties include the read out-of-bounds, write out-of-bounds, and insecure copy. This is the case for these properties because they rely more heavily on the feasibility analysis step, which often fails to describe variable domains accurately. Since our feasibility analysis step is conservative, favoring reporting a potential error if it is unable to identify whether the specified vulnerable conditions are possible; it introduces false positives. We present an example of the types of behaviors that may result in false positives in Listing 13. In this case, we have to determine the variable domains of `num_objs` and `n` as well as the relationship between them, to be able to determine if `i` can ever exceed the space allocated for `objs`. Our feasibility analysis step often fails at this task because our data-flow abstraction only describes that variables are related but not the relationship between them. This causes our feasibility analysis to overlook some constraints on variables, specifically those that are not in the same subroutine and for which the relationship between the variable referenced in the conditional statement and the variable we are trying to derive constraints for is more complex. We expand on the limitations of our feasibility analysis further in Section 7.3.2.

Through the use of concrete execution, our model is built upon sequences of events that we know can which greatly reduces the number of false positives reported, which is a huge advantage. Primarily, what these results suggest is that improvements to our feasibility analysis step must be made if we want increase the overall accuracy of our analysis. Finally future work should go towards performing a quantitative analysis of the false positives returned by our framework, which should include a calculation of the false positive rate.

6.4.3 Case Study: Introducing Semantics

We use the information disclosure vulnerabilities present in wget as a case study, as they are instances where this framework needs to have an understanding of the meaning of particular structures within the program to be able to identify them.

We define information disclosure as fields containing sensitive information being written to an external location. This would include behaviors like writing to log files or the terminal. This definition is perhaps a bit generous, as there are possible cases where this would be legitimate behavior; however it is still a potential insecurity in

```
STORE
with arg.dsource = dsource, arg.dtype = dtype, arg.label = label
where arg.dtype in const.SENSITIVE

LEAK
with buf = buf, buf.label = buf.label
where buf.label like arg.label
```

Listing 14: Our information leak property used in our evaluation of wget.

the program. In Listing 14 we show our definition of this property in Hy2-lang.

In this case, we must be able to define `const.SENSITIVE` in the context of the target program. We consider the url, username, and password from the struct url to match this definition, as credentials meet this definition. In our configuration file for wget, we define these fields as sensitive, introducing an additional attribute that provides a list of fields that are sensitive by type and offset within the struct. During our rule-checking step, we identify variables that match this definition or more precisely are instances of the types in `SENSITIVE`, then we attempt to identify subsequent events where they are leaked.

Due to the nature of our analysis, we cannot reason about the meaning of variables unless this information is provided to us. We see the method of having the user provide definitions for certain attributes within a property such as this one as the best solution, as we expect that relying on things like variable or field names to infer higher-level semantics to be less reliable as naming conventions are not usually consistent between applications. This method also allows for this property to be reused for other programs. Since this framework is designed for the creation of generic properties, it allows for a great deal of flexibility in which properties can be defined.

With the described property (Listing 14) we are able to identify CVE-2018-20483 in wget, which involves writing a downloaded file's origin URL in the metadata of said file on disk, causing the leaking of sensitive information if it is contained in the url. We identify a second instance of this property within the program caused by wget logging the requested url in debug mode. The second information disclosure vulnerability affecting this version, this being CVE-2021-31879 [20], involves sending the Authorization header to a different origin when redirected. We don't have a property that describes this behavior and thus do not locate it. Although it would be possible to create one in this framework, it would likely be specific to this specific scenario, and thus defeat the purpose of this evaluation.

6.4.4 Case Study: Hyper

Finally, we discuss our analysis of the hyper crate, an HTTP library written in rust [22]. This project fits well into our evaluation because it came with a set of integration tests, which we use to exercise the library. To analyze this library we compiled it into the shared object libhyper.so and the integration tests into a separate binary. When we executed this binary, it would load libhyper.so into memory, and we analyzed the behavior within these corresponding pages. The rust compiler greatly complicates our analysis through its optimizations and runtime checks, and because dependencies are statically linked in rust, the library was quite large and contained a lot of code that is considered out of the scope of our analysis. This greatly increased the time required for analysis because these library function were included in the defined the scope so our model describes not only the behavior of hyper but also much of the standard rust libraries.

There are two vulnerabilities in this library that we were able to identify. The first vulnerability, CVE-2021-32714, is an integer overflow vulnerability caused by improper input validation. The second vulnerability, CVE-2021-32715, is an allocation without limit vulnerability also caused by improper input validation.

When describing the first vulnerability, although we can identify a potential overflow in the program, we are unable to identify the effects of this overflow, these being data loss and potentially HTTP request smuggling. We show the vulnerable code in

Listing 15. This vulnerability occurs in the HTTP/1.1 implementation of chunked encoding. If a request or response is recieved with the header `Transfer-Encoding: chunked` set and a chunk size greater than an unsigned 64-bit integer, it will cause an integer overflow in the variable `size`. This fault occurs because the `size` value is not checked when looping through and decoding the hex-encoded chunk size.

The second vulnerability is not strictly a vulnerability in the library itself, rather, it is in how it is used by other libraries that depend on it. The function `to_bytes()` copies a request or response body into a single `Bytes` buffer which it allocates based on packet size. Since this function doesn't perform input validation on the length of the `HttpBody` it can be forced to allocate an arbitrary amount of memory which can cause a crash. For the purpose of our evaluation, we consider this to be a vulnerability in hyper. We were only able to describe this behavior as unconstrained memory allocation, which is too broad to be useful outside of this context. The semantics of rust are vastly different from that of others tested thus far (i.e. C, golang), so we would need to introduce different rules to describe them. For example, we rely on the set of allocate functions in libc to identify memory allocation on the heap, however, in rust when alloc is called, one of these functions is not always called as a result. We would need to be able to identify and understand these instances.

Rust vulnerabilities challenge our handling of the trade-off between precision and scalability. They require bit-level precision to identify, as they can rarely be described by patterns of calls as C vulnerabilities can and typically stem from improper input validation. This requires a completely different mindset than what we had employed previously. As previously discussed, the quality of results we obtain is largely dependent on how we describe a particular behavior. Before performing our evaluation, we did not consider integer overflows and wraparounds on their own to be a vulnerability,

[2]Snippet corresponds to src/proto/h1/decode.rs:205:234 in the hyper crate version 0.14.9 which is available: https://github.com/hyperium/hyper/releases/tag/v0.14.9

```
1      fn read_size<R: MemRead>(
2          cx: &mut task::Context<'_>,
3          rdr: &mut R,
4          size: &mut u64,
5      ) -> Poll<Result<ChunkedState, io::Error>> {
6          trace!("Read chunk hex size");
7          let radix = 16;
8          match byte!(rdr, cx) {
9              b @ b'0'..=b'9' => {
10                 *size *= radix;
11                 *size += (b - b'0') as u64;
12             }
13             b @ b'a'..=b'f' => {
14                 *size *= radix;
15                 *size += (b + 10 - b'a') as u64;
16             }
17             b @ b'A'..=b'F' => {
18                 *size *= radix;
19                 *size += (b + 10 - b'A') as u64;
20             }
21             b'\t' | b' ' => return
   ↪   Poll::Ready(Ok(ChunkedState::SizeLws)),
22             b';' => return Poll::Ready(Ok(ChunkedState::Extension)),
23             b'\r' => return Poll::Ready(Ok(ChunkedState::SizeLf)),
24             _ => {
25                 return Poll::Ready(Err(io::Error::new(
26                     io::ErrorKind::InvalidInput,
27                     "Invalid chunk size line: Invalid Size",
28                 )));
29             }
30         }
31 }
```

Listing 15: A snippet showing the vulnerable function read_size from decode.rs in the hyper crate [2]. The hex-encoded chunk size is read from the request headers as a byte array. To convert the hexadecimal string to an integer, hyper loops through this array, calling read_size on each byte. In read_size on lines 10-11, 14-15, and 17-18 multiplication and addition are performed on size without first checking whether the operation may cause it to overflow.

as there are legitimate use cases for this behavior. Instead, we rely on identifying some sort of critical operation such as memory allocation or memory access that could be affected by it. This approach is not effective for rust programs, as rust considers integer overflows to be a bug and will panic if the program is compiled in debug mode. As a result, we were forced to change our approach, and the behavior we end describing is: *improper validation of arguments used in an operation that might overflow.* For this property Hy2 will identify all of a set of overflow operations, such as Add, Sub, Mul, and Shl and then use our feasibility analysis step to identify constraints on the operands used in these operations. Although this does work, it will return a result for every single instance of the affected operations, which totaled 425 errors. This is not helpful for the user as it would require significant effort to interpret all these results. One potential solution to this problem would be to hide examples where the overflow is not possible in our front-end framework, however, it is not an ideal solution.

6.4.5 A High-Level Comparison to Other Methods

Finally, we compare at a high level the approach we took in our evaluation and our results to that of other methods. Specifically, we focus on methods that use fuzzing and static analysis.

We do not perform a direct comparison to other tools for several reasons. Primarily because some of these tools, as described in Section 3.2.4, are designed for program exploration rather than the discovery of vulnerabilities so a direct comparison may not be possible. For example, for the binary analysis framework S2E to perform this analysis a new module would have to be created and configured which would be a separate work. For other methods, additional checkers would have to be created and the tools themselves configured. Additionally, the way our analysis is structured is fundamentally different. The time taken for the completion of our analysis is the time required to run the program and analyze its execution behaviors over a set of input examples, whereas fuzzing and symbolic execution methods are typically compared by running them for a fixed amount of time. These two approaches are not necessarily directly comparable. Finally, some of the tools would not be able to analyze the programs selected for our case studies as they lack multi-language support. We conclude that such a comparison is beyond the scope of this work and if completed not guaranteed to produce meaningful results. Instead, we describe how various methods might have gone about such an analysis and what results they might have achieved.

For the case of fuzzing tools such as AFL, the user would have to perform some configuration in the form of initial test cases before being able to analyze the target program. Wget provides an interface for fuzzing simplifying this configuration process. For the case of htmldoc, for each of the different conversion options, the user would have to specify the appropriate command line arguments and provide an initial test, in this case, a file to be converted. Additionally, for cases involving documents with images, the user would likely need to perform a separate analysis for each of the image formats (i.e. png, gif, jpeg). As a result, the size of the input space as a whole would be quite large, which would greatly increase the overall time required for analysis. Additionally, we cannot expect fuzzing to be exhaustive and will likely miss some errors. We saw evidence of this in our analysis, with only some of the errors on a particular branch being reported. We saw some evidence of this in our analysis. From the bug reports for htmldoc, we noted that for a majority of the vulnerabilities we evaluated, it was stated that they were found using a fuzzer. We observed that only some of the vulnerabilities on a particular branch were reported. For example, as described in Section 6.4.2 in the function `image_load_bmp()` we were able to identify several other errors that had not been reported, despite the fact a vulnerability was discovered in this function using a fuzzer. This includes additional memory errors which can be triggered with a crafted bmp image with a truncated header caused by the program not checking the size of the image file before reading from it. As these errors do not cause a crash, we expect that they would be overlooked by the fuzzer. The reports themselves also did little to describe why the fault occurred beyond providing an input that caused the crash. In Section 6.4.2, we discussed the additional error uncovered in `parse_table()` and how an improper error description may introduce the potential for incomplete patches. By contrast, Hy2 can describe the locations in the program where the fault occurs and attempts to explain why it occurs with the goal of making it easier for developers to interpret and act on our results. Fuzzing tools would likely be able to uncover most of the reported vulnerabilities we attempt to identify in our case studies as they were almost exclusively memory errors or cause the program to crash. However, we would not expect fuzzers to be able to identify the information leak vulnerabilities in wget as they are not designed to trace the data flow for specific fields.

When considering the capabilities of static analysis methods, we consider methods that use symbolic execution as they are typically designed to identify memory errors and a majority of the errors we encountered were memory errors. We would expect that if configured to do so, these methods would be able to locate a majority of the errors reported, with a few potential exceptions. For the case of the null-pointer dereference vulnerability in the `image_load_jpeg()` function in htmldoc that arises from not checking whether the call to `jpeg_read_header()` in the external library libjpeg.so was successful, static methods would have to model this external library to be able to identify that it is possible for this function to return null. Similarly, for the case of the wget write out-of-bounds vulnerability, the method used would need to have a model of libc, as the actual code that attempts to write out of bounds is in an internal function within this library. Thus to identify this vulnerability the behavior of this function would have to be described. Since libc is a standard library, most tools used in practice would likely have this ability. Finally, for the case of the hyper crate, we expect that if able to analyze rust programs, static methods would be able to uncover the vulnerabilities we considered, with an analysis of the domains of the variables used in the operations that overflow.

With this discussion, we have not considered the limitations of static methods. For instance, both hyper and wget use multi-threading so would not be supported by some symbolic execution and model-checking tools [85][42]. Additionally, with such tools, there is the potential for a state-space explosion to occur, in which case the tool may be ineffective in locating the described vulnerabilities. Our method is at an advantage over static methods as we only describe the state space relevant to a particular set of executions and not exhaustively, thus we are much less likely to encounter this problem. We describe the role of exhaustiveness in our analysis in-depth in Section 7.1.

Finally, in this section we are comparing Hy2 to a generalized description of many different tools; individual tools do not necessarily have all the capabilities discussed. This serves to highlight that an advantage of Hy2 is its versatility, shown through its ability to handle different programs and classes of vulnerabilities.

6.5 Summary

To summarize, with our evaluation consisting of several case studies, we were able to show the effectiveness of Hy2 , as well as identify some limitations of our method which could be addressed in the future.

Chapter 7

Discussion

In this chapter, we re-evaluate the contributions of Hy2, explore the limitations of our method, and describe potential areas of future work.

7.1 The Role of Exhaustiveness

In this section, we discuss the role of exhaustiveness in our analysis. Exhaustiveness in vulnerability analysis methods is important as skipping over parts of a program introduces the possibility of vulnerabilities being missed. Methods that use static analysis, including model checking, are typically exhaustive, meaning they explore the entire state space of a system. Runtime verification methods which use dynamic analysis only explore the state space relevant to a particular execution or set of executions and thus offer weaker guarantees. This extends to our method, which is only able to exhaustively analyze the parts of the system we observe during execution, and we can't guarantee that we have observed all possible program behaviors. Where this method differs from other runtime verification methods is that we combine multiple executions into a single model and are able to describe behaviors we haven't observed at runtime with our static analysis step, hence covering a greater percentage of the state space than simply runtime behaviors.

Despite the fact that we position our method as a hybrid of model checking and runtime verification, we can't strictly call it a model-checking method but rather claim that it uses model-checking techniques. We described our reasoning in Section 4.1. We made the comparison between the abstraction derived by our method and those that use a coarse-grained fixed abstraction as described in Section 2.5.3. We could make the claim that our method could easily be transformed into a purely static analysis method by extending our static analysis step to analyze the entire program instead of single branches, and this would result in the same control-flow-based abstraction as our current method. However, where this method differs from other runtime verification methods with regards to state-space exploration is that we combine multiple program executions into a single control-flow-based model and with our static analysis step, explore branches not seen during program execution, hence covering a greater percentage of the state space than simply runtime behaviors.

We see this as a trade-off between exhaustiveness and practicality. Firstly, dynamic analysis methods are generally considered to produce fewer erroneous results than static methods as all results returned come from concrete observations about the target system [38]. Secondly, the fact that we don't have to consider the entire state space is in many cases beneficial. This method will be able to complete its analysis even if we are unable to exhaustively model the state space of the target program. This is beneficial because it may not always be possible to describe the entire state space. Such cases may include those involving concurrency, randomness, and other sources of nondeterminism, or instances where the state space is so large or complex that modeling it wouldn't be feasible. As described in Section 6.3, we can still describe concurrent behaviors even with an incomplete model of their state space. More broadly, we can describe behaviors even if we can't specifically reason about why they have occurred because we are building a model of the program from data observed from concrete executions of the program.

This trade-off extends to our handling of external functions. The values returned from these functions may influence the path taken by the program. Through our static analysis step, we can consider other program paths that might be taken if a different value is returned by external functions called by the program without specifically having to reason about its behavior. Although this means we can't necessarily understand the relationship between the input arguments passed in and the values returned, it is not necessary for our analysis. Past static-analysis methods abort when they reach an unhandled external function or, for the case of dynamic symbolic execution methods, replace symbolic values with concrete ones causing them to miss some paths [41] [85]. Finally, the use of over-approximation in describing variable domains allows us to consider the behavior of variables within our model over their entire potential range of values from a single execution. This makes our abstraction coarser and the state space we are considering smaller, as variables can be represented with a type definition, with their potential range of values being that of their corresponding type. For example, for a variable that is a 64-bit integer, we would consider it possible for it to have any value between -2^{31} and $2^{31} - 1$.

Lastly, our method is very error tolerant as completeness is not a necessity. Although it may not be ideal, our analysis can continue in the event that we encounter an unknown. For instance, if we are unable to calculate the value at a particular program point we still have a description of its behavior through its variable expression, and in cases where we can't reason about particular behaviors we fall back on concrete values. We expand on these points further in Section 7.3.

7.2 The Value of Full-System Emulation

In this section, we discuss the value of full-system emulation and how it is reflected in our method. As discussed in Section 4.3, the use of full-system emulation grants us the ability to observe the state of the entire system. This allows for the tracing and modeling of arbitrary slices of system execution. Although in our experiments, we only consider environmental effects observable to the program, these being the system and library calls and their arguments, we could extend our analysis beyond this.

The use of full-system emulation is well-suited for the combination of model checking and runtime verification; we can precisely model the functionality we consider to be in-scope, whilst modeling the behavior we consider to be out-of-scope with coarser granularity. The use of the `before_bb_exec` callback allows us to describe exactly which parts of the program were executed at a basic block level without modifications to the code, such as adding assertions or the use of a specialized compiler as done with past dynamic analysis methods [1] [56]. This technique allows us to obtain a fine-grained abstraction of program behavior with similar properties as abstraction-based model-checking methods that use an IR without the source code as described in Section 5.5. Through our use of dynamic analysis, we have a concrete representation of program memory and as such we are able to overcome some of the potential inaccuracies of static analysis, such as aliasing and representing memory. Finally, past model-checking methods may use a fallback implementation which produces undetermined results for unknown functions. We can eliminate the need for this by allowing the user to include arbitrary functions in the analysis scope and thus be able to model unknown functions automatically. We show the potential benefits of this functionality in Section 6.4.2, where we describe how it is used to uncover a memory error in wget for which the actual write out-of-bounds operation occurs in an external dependency.

However, this does come at a cost. Through the use of dynamic analysis and full-system emulation, we are introducing potential environmental side effects into our analysis that may be unknown to the user or beyond their control. This is more so the case for kernel-space programs than user-space programs. The use of dynamic analysis also introduces additional limitations, including computational cost and user effort. We describe these further in Section 7.3.

7.3 Limitations

In this section, we discuss some of the limitations of our method and describe how they could potentially be addressed in the future.

7.3.1 Cost of Analysis

We discuss the performance of this method with regards to the computational resources and time required for the analysis. Since one of the primary goals of this method was to balance precision and scalability, this is an important topic of discussion.

Although our use of full-system emulation undeniably introduces some additional overhead, this is an intrinsic part of our design and the value it adds more than compensates for this additional overhead, as described previously (Section 7.2). In our evaluation, we noted that we could trace some of the inefficiency of our method back to two main sources. First, the virtual memory access callback was responsible for a disproportionately large percentage of the overhead of our method. We discuss this in our evaluation (Section 6.4.1), and in Section 7.4, we discuss how we could approach this issue.

Second, as described previously, analyzing the execution of a program that performs operations like modifying or accessing a file would require the repetition of certain operations for each byte in the file, introducing redundancy. This repetition is reflected in the length of the execution trace produced, and introduces additional overhead when building the CFG model and merging equivalent nodes, as we are handling the blocks that occur within the loop many times unnecessarily. We don't care about each individual byte within the file, instead, we care about the behavior and data flow in the loop. Creating an abstraction to describe these loops would remove this redundancy. Generally, for Hy2 , the time and computational resources required for analysis are proportional to the length of the sequence of basic blocks executed by the target program. Each time a basic block is executed we record concrete data describing its execution at the particular discrete time step and analyze it, thus increasing overhead relative to the number of basic blocks executed. We would expect for larger programs, this sequence would be longer so the overhead would be higher. Although we have evidence that this method is able to scale to some larger programs, including sqlite3 and mandb, it is unable to scale to very large programs such as browsers without breaking them up into smaller components.

It is difficult to compare the time requirements of this method compared to other vulnerability identification methods such as fuzzing and symbolic execution, as they typically are run for an arbitrary amount of time. Past works, such as Ferry and Driller, typically use 6 to 12 hours as a benchmark for the amount of time needed for analysis, which is more than what is needed for Hy2, although these measurements are arguably not comparable. Our analysis will always be completed in a finite amount because we have a finite number of test cases, each which are analyzed in a finite amount of time.

7.3.2 Precision

In this subsection, we discuss the precision of our method and how our design choices within our method may introduce imprecision. One limitation of our analysis is that due to our use of full-system emulation, some program behavior may be influenced by the environment in unexpected ways.For example, a program may execute on different branches depending on the system architecture, and some undesirable behaviors may be specific to a particular architecture. Through our additional static analysis step, we expect to partially rectify this problem however; it is not guaranteed to address this for all cases as our static analysis step works by mutating the current program state and cannot reason about new inputs.

Through our use of dynamic analysis, we are able to eliminate the potential for infeasible paths, as we are building our model upon the sequence of events that we know to have occurred. However, our data-flow analysis is a potential source of imprecision as we propagate labels describing memory locations without checking that the path between the definition at the address and the use of the address is feasible and is not redefined. We discuss this problem further in Section 7.4.

In our rule-checking process, we always favor over-approximation over potentially introducing false negatives in our analysis, and this is known to produce spurious results. This is the case primarily because, upon discovering that a potential counter-example is infeasible we chose to report it anyway with the evidence that is infeasible instead of eliminating it from our results. Additionally, we do not expect absolute accuracy in the completion of our feasibility analysis because it can only reason about logic within the program and it has a limited understanding of the relationships between expressions in the program. Our data-flow analysis describes whether two variable expressions are related, but not how they are related. Finally, our analysis whilst still favoring over-approximation, is imprecise with regards to static data. The primary impact of this limitation is being unable to precisely determine the size of statically allocated memory. To address this limitation we define separate properties for static and dynamic memory errors, introduce methods that allow debugging information to be introduced into our model and thus make the size information available, and underestimate the size of statically allocated variables to reduce the potential for false negatives.

7.3.3 Coverage

In this subsection, we discuss the role of coverage in our analysis. With the design of our method, we can evaluate a program regardless of the coverage achieved by the workload or for any subset of the workload. In simpler terms, we can analyze any subset of the program's functionality, as well as the program as a whole. We see this approach to coverage as a trade-off, as whilst this property may be desirable, it also introduces several limitations. This property could be useful if, for instance, a user wanted to verify that a change to the program source does not introduce any new undesirable behaviors, as it would only require them to test the affected code paths. Additionally, this approach would allow for larger programs to be evaluated as a set of smaller components or, if desirable, for single input examples. However, to make any claims about the properties of the target program you would need to exercise the target program in its entirety, hitting all its basic blocks. To achieve this, our method requires that the user produces a set of inputs that exercises the program's functionality. Although we may expect that a thorough test suite would serve as a sufficient workload, we have evidence that this may not always be available, as shown in our evaluation (Section 6.4). Other methods, such as those that use fuzzing, automatically generate inputs that cover all program branches, and so would not have this additional task. Although, through the addition of a static analysis step, we are able to cover a large number of blocks with fewer input examples and potentially reach branches that would be harder to reach with other methods, this is not guaranteed. For the case of wget, we were able to increase our coverage to 88% over the 76.6% achieved with only dynamic analysis.

Although addressing coverage was not a goal of this method, we recognize its value. Since there are many potential methods that could perform input generation, we leave this problem for future work. For example, this method could work in conjunction with a fuzzing tool. A fuzzer could be used to create a set of inputs that could be passed into our concrete execution step. Additionally, in the report of each discovered error, we provide the set of test cases that triggered the execution of the undesirable behavior. For the case of memory errors, we observed that we could simply pass these test cases into a fuzzer to derive a proof-of-concept crash for the discovered error. We are able to reduce the size of the input space for the fuzzer considerably, allowing it to identify vulnerable inputs very quickly. Although this is not the purpose of this tool, it helps demonstrate that this method has goals that are distinct from that of other methods.

7.3.4 The Role of the User

In this subsection we discuss the role of the user in the analysis process. One of the biggest trade-offs in our analysis is centered around the amount of knowledge and effort required by the user. We wish for our method to provide a great degree of freedom with regards to how the scope of the analysis can be defined and as a result, the configuration process may be more complex. We expect that configuration may require interpretation of the program model and refinement, specifically through the definition of the workload, the selection of external functions to include in the model, modification of the program scope, the introduction of type information, and selection of properties to check. This method may require a greater amount of user involvement as they must be able to describe how to execute their target program and be able to interpret the results. To describe how to execute the target program they must provide a workload, consisting of a set of inputs that exercise the different branches of the program. If the user does not have knowledge of the program this will require additional effort from the user to figure out which inputs trigger the execution of the different parts of the program's functionality, and if the workload is insufficient to exercise, the program we will miss faults.

An effective vulnerability analysis solution must be able to not only discover flaws in the target program, but must also be able to report its findings effectively to the user. The user needs to be able to understand the cause of the reported error, including where in the program a fault may occur and under what conditions. Through the addition of decorators and with debugging and source information, we increase the readability of our results, making it trivial to locate the source of the fault in the source. However, without debugging symbols in the binary, this information would not be available and in these cases, the user would have to have some understanding of what the undesirable behavior is and would have to be able to understand the model enough to trace it back to the affected part of the program source. Since the behavior of the target program is described through a low-level language, in such cases, the results will be much more difficult to decipher. Additionally, we focus our efforts on providing results that will be useful to a developer debugging the source, rather than an individual attempting to find concrete inputs that cause faults. In this case, the user would have to have the knowledge to produce an input from the reported error. With further refinement of the application interface, additional steps can be taken to improve on this, which we describe in Section 7.4.

7.4 Future Work

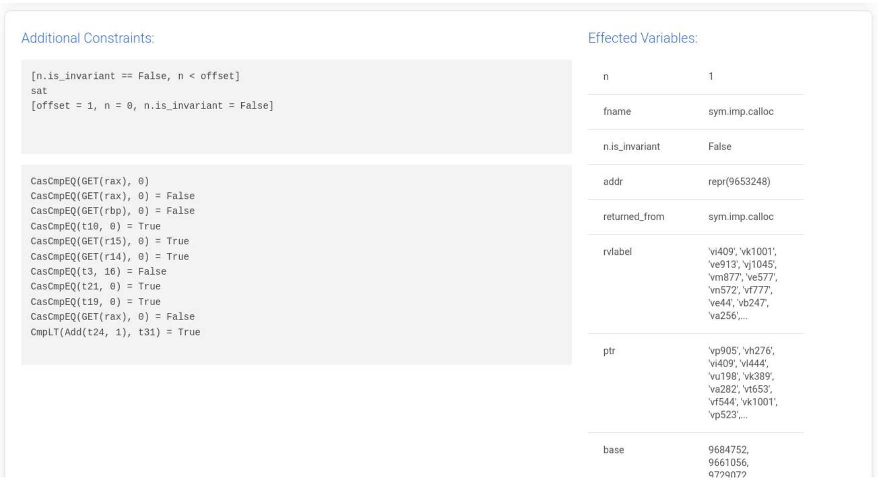
This section describes potential future work with regards to our method and framework. We focus primarily on how we can improve our communication with the user and the performance of our tool.

7.4.1 Improving our User Interface

Our primary method of communicating with the user is through our web interface as described in Section 5.1. This interface is important as our tool requires some degree of user involvement in the configuration process, as they must provide input examples and specify the scope of the analysis. Additionally, the results of our analysis are displayed through this interface, and the user must be able to interpret the results to take steps toward addressing them. Thus, one area of future work lies in improving this interface. Firstly, the interface should provide proper feedback to the user to notify them of errors in the execution of the target program or in their specified configuration of the target. Hy2 has many configuration options, each of which serves a specific purpose, and thus how and when to use them should be made clear to the user to reduce the chance of misconfiguration. Additionally, we expect that the user may desire to create their own security properties, and we see it as worthwhile to potentially create an interface to allow them to do so within our web interface. This would allow for syntax errors to be reported directly. Most importantly, further effort should go towards refining how we present and structure our results. Improvements such as ordering results by severity and hiding results that were decided to be infeasible in the feasibility analysis step would reduce the effort required by the user in interpreting them.Additionally, we noted that our framework tends to return duplicate results, these being the same sequence of events reported multiple times on different paths, and thus taking steps to communicate these occurrences as different paths instead of separate results would be a worthwhile addition. Finally, we see the need to provide a clearer explanation alongside the error trace and feasibility analysis returned. This information is difficult to interpret in isolation without knowledge of the target program and does little to communicate the nature of the potential error. We show an example of this problem in Figure 7.1, in which we can see that although it returns information about the conditions under which the error may occur in the program, it does nothing to explain them. Although the web application was meant to be a proof-of-concept, since we see the user as a central part of the task of vulnerability analysis, improving this area of our framework is critical.

7.4.2 Performance

Through the completion of our evaluation, we identified several opportunities for improvement in the performance of Hy2. Although some of the overhead of our method is unavoidable, such as that which results from the execution of the target, there is



Additional Constraints:

```
[n.is_invariant == False, n < offset]
sat
[offset = 1, n = 0, n.is_invariant = False]
```

```
CasCmpEQ(GET(rax), 0)
CasCmpEQ(GET(rax), 0) = False
CasCmpEQ(GET(rbp), 0) = False
CasCmpEQ(t10, 0) = True
CasCmpEQ(GET(r15), 0) = True
CasCmpEQ(GET(r14), 0) = True
CasCmpEQ(t3, 16) = False
CasCmpEQ(t21, 0) = True
CasCmpEQ(t19, 0) = True
CasCmpEQ(GET(rax), 0) = False
CmpLT(Add(t24, 1), t31) = True
```

Effected Variables:

n	1
fname	sym.imp.calloc
n.is_invariant	False
addr	repr(9653248)
returned_from	sym.imp.calloc
rvlabel	'vi409', 'vk1001', 've913', 'vj1045', 'vm877', 've577', 'vn572', 'vf777', 've44', 'vb247', 'va256',...
ptr	'vp905', 'vh276', 'vi409', 'vl444', 'vu198', 'vk389', 'va282', 'vt653', 'vf544', 'vk1001', 'vp523',...
base	9684752, 9661056, 9729072

Figure 7.1: An example of the presentation of the results of our feasibility analysis. The bottom box of the left-hand side of the page shows the results of the analysis of the satisfiability of path constraints on the error path.

a lot of redundancy that could be eliminated with further refinement. First, the use of the `cb_virt_mem_after_read`, and `cb_virt_mem_after_write` callbacks could be eliminated. In their place, Hy2 could use a similar approach to how we handle registers, collecting values at addresses on the stack at the `before_block_exec` callback and keeping track of updated values when performing computations with expressions within the block.

Second, we identified that looping behaviors, typically introduced by the processing of files, introduced a significant unnecessary overhead in our analysis because our method performs an analysis of the occurrence of each block within the loop. If, for example, the program performs a file conversion a byte at a time, we would see the same set of blocks repeated over every byte in the file. Therefore, we see the potential for the introduction of a loop abstraction to handle these cases. This is a logical step for our method, as this is also a step performed by decompilers as described in Section 2.1.4. The problem of control-flow reduction and loop optimizations is a well-studied problem [80] and existing algorithms could be applied to our control-flow-based abstraction. By deriving an abstraction for looping behaviors, we could represent the

```
m = LOAD(a)
x = STORE(m + 4, c), def(m+4), use(m)
y = STORE(m + 8, d), def(m+8), use(m)
...
z = LOAD(m+8), use(m+8)
```

Listing 16: An example describing a more precise data-flow approximation which could be introduced to improve our program abstraction.

properties of iterators and other looping constructs.

7.4.3 Data-flow Analysis

As discussed in Section 7.3.2, the greatest source of inaccuracy in our analysis was introduced by the over-approximation in our data-flow analysis. Although we were able to identify that variable expressions were related with our data-flow approximation, we were unable to describe precisely how they were related. Specifically, presently our data-flow approximation does not differentiate between data definitions and use, meaning that for some variable x it cannot determine whether x itself is being accessed or if some variable that uses x is being accessed. We provide an example of the improved data-flow analysis in Listing 16. In our current implementation, we only describe x, y, and z as using m but are unable to describe the relationship between them. Consequently, we end up representing a direct relationship between x and z when there isn't one. To improve our data-flow analysis, we want to describe that z is only related to m through y and is not related to x. This change would also increase the effectiveness of our feasibility analysis step, which is also limited by our understanding of the relationships between variables.

7.4.4 Potential Extensions

In the future, it may be worthwhile to extend our framework to handle a wider range of applications. In our evaluation of Hy2, we focused exclusively on user-space command-line applications, but we see the potential for our tool beyond this. Through the creation of a guest image with a desktop environment, we could add support for applications with a graphical interface. This would require further research as QEMU has limited support for interacting with graphical interfaces, so the user would either have to manually interact with the application, or a testing harness would have to be created. Finally, we would wish to improve our handling of applications that use multithreading or multiprocessing as discussed in Section 6.3.

Lastly, a potential future direction is extending the functionality of our CPU simulator framework. Firstly, we could use it to evaluate our program model on arbitrary inputs, allowing for the behavior of the program to be described without exactly executing it, which could have many applications, including deriving variable domains. Secondly, we could integrate it into our feasibility analysis step. If we are able to derive a concrete counterexample for a reported error, we could immediately check them against our model of the program to determine their validity to improve the accuracy of our results.

7.5 Conclusion

In this work, we presented our method of vulnerability discovery Hy2, including the motivation behind it and its implementation. We demonstrated that our method could be applied to various real-world programs in a manner that could be easily integrated into a development cycle, and we were able to identify vulnerabilities of different classes within these programs. We discussed trade-offs we made in the design of our method and how we could address them in the future. We described potential extensions and improvements to our method, primarily focusing on performance and usability. Future improvements to Hy2 could further our progress towards our goal of improving software safety and security.

Appendix A

Evaluation and Implementation Artifacts

This chapter describes the security properties we implemented in the process of our evaluation and provides several implementation artifacts for reference.

A.1 Security Properties

In this section we provide a summary of the properties (shown in Table A.1) implemented over the course of the evaluation of this framework.

A.2 Property Definitions Referenced in our Evaluation

In this section we include the property definitions used in our evaluation. We omit the definition of the "Information Leak" and "Data from an untrusted source used in a privileged operation" as they are described elsewhere.

A.2.1 CWE-125 Out-of-Bounds Read

In Listing 17 we show our implementation of the "Out-of-Bounds Read" property for dynamically allocated memory. This property looks for memory being read at an offset greater than what was allocated for the buffer.

A.2.2 CWE-126 Out-of-Bounds Write

In Listing 18 we show our implementation of the "Out-of-Bounds Write" property for dynamically allocated memory. This property looks for memory being written to at an offset greater than what was allocated for the buffer.

Name	Description
CONC-31C	Destroy locked mutex
CWE-120	Insecure Buffer Copy
CWE-125	Out-of-bounds Read
CWE-252	Unchecked Return Value
CWE-367	TOCTOU Race Condition
CWE-134	Use of External Format String
CWE-415	Double-free
CWE-416	Use-after-free
CWE-476	Null Pointer Dereference
CWE-667	Improper Locking
CWE-176	Improper Handling of Unicode Encoding
CWE-787	Out-of-bounds Write
CWE-349	Data from untrusted source used in privileged operation or Acceptance of Extraneous Untrusted Data With Trusted Data
FIO46-C	Do not access a closed file
POS30-C	Improper Permission Revocation Order
ERR07-C	Prefer functions that support error checking. Unsafe use of atoi
CWE-200	Information Leak
CWE-190	Unsafe Arithmetic in Memory Allocation Operation
CWE-667	Mutex Lock without Unlock
CWE-667	Mutex Unlock without Lock
CWE-190	Integer Overflow or Wraparound
POS-30C	Use of readlink function properly
STR31-C	Use of strlen() with non-null-terminated string

Table A.1: A list of the properties implemented in Hy2-lang over the course of our evaluation and testing.

```
---
ALLOCATE
with n = size, fname = fname, n.is_constant = size.is_constant

RET
with addr = val, addr.label = label, returned_from = ret.fname
where returned_from == fname

NOT FREE
with eol_addr = ptr
where eol_addr == addr

LOAD
with ptr = ls.lhs, base = base, offset = ls.rhs, sop = ls.op,
 ↪  ls.rs.is_constant = ls.rs.is_constant
where (ptr like addr.label OR base == addr) AND (ls.rs.is_constant ==
 ↪  false OR n.is_constant == false)
if n <= offset
---
```

Listing 17: CWE-125 Out-of-Bounds Read

```
---
ALLOCATE
with n = size, fname = fname, n.is_constant = size.is_invariant

RET
with addr = val, returned_from = ret.fname, addr.label = label
where returned_from == fname

NOT FREE
with eol_addr = ptr
where eol_addr == addr

STORE
with ptr = ls.lhs, base = base, offset = ls.rhs, sop = ls.op,
 ↪  ls.rs.is_constant = ls.rs.is_constant
where (ptr like addr.label OR base == addr) AND (ls.rs.is_constant ==
 ↪  false OR n.is_constant == false)
if n <= offset
---
```

Listing 18: CWE-126 Out-of-Bounds Write

```
---
FREE
with ptr = ptr, ptr.label = ptr.label

NOT ALLOCATE as a1
with fname = fname

NOT RET as a2
with faddr = ptr, returned_from = ret.fname
where returned_from == fname AND faddr == ptr

FREE
with ptr1 = ptr, ptr1.label = ptr.label
where ptr.label == ptr1.label

WHERE a1 AND a2
---
```

Listing 19: CWE-415 Double Free

A.2.3 CWE-415 Double Free

In Listing 19 we show our implementation of the “Double Free” property. This property looks for instances where free is called on address after free has been already been called on that same address.

A.2.4 CWE-416 Use-after-Free

In Listing 20 we show our implementation of the “Use-after-Free” property. This property looks for instances where a memory location is accessed after free has been called on it.

A.2.5 CWE-476 Null Pointer Dereference

In Listing 21 we show our implementation of the “Null Pointer Dereference” property. This property looks for instances where a return value is accessed without being first checked that the operation was successful. We use a list of known functions that return null in the event of an error.

```
---
FREE
with ptr = ptr, ptr.label = ptr.label

NOT ALLOCATE as a1
with fname = fname

NOT RET as a2
with returned_from = ret.fname, faddr = addr
where returned_from == fname AND faddr = ptr

VAR
with addr = base, base.label = base.label
where ptr == addr OR base.label like ptr.label

WHERE a1 AND a2
---
```

Listing 20: CWE-416 Use-after-Free

```
---
NULL_RETURN
with fname = fname

NOT COND as a0
with cond0.dsource = dsource, cond0.op = op, rs0.val = rs.val, lsv0 =
 ↪  lsv
where (cond0.dsource == returned_from OR lsv0 == ptr) AND rs0.val ==
 ↪  0 AND cond0.op in ["const.CmpEQ", "const.CmpNE"]

RET
with ptr = val, returned_from = ret.fname, var.var = var.name,
 ↪  ret.label = label
where returned_from = fname

NOT COND as a1
with ls.var = ls.var, lsv = lsv, rs.val = rs.val, cond.dsource =
 ↪  dsource, cond.op = op
where (cond.dsource == returned_from OR lsv == ptr) AND rs.val == 0
 ↪  AND cond.op in ["const.CmpEQ", "const.CmpNE"]

VAR
with ref = val, var.name = name, var.label = label
where ptr == ref OR var.label like ptr.label

VAR
with ldeps = deps
where var.name in ldeps

WHERE a0 OR a1
---
```

Listing 21: CWE-476 Null Pointer Dereference

```
---
STORE
with var = rs.ls, k = rs.rs, sop = rs.op
where sop in ["ADD", "SUB", "MUL", "SHL"]
---
PUT
with var = ls.ls, k = ls.rs, sop = ls.op
where sop in ["ADD", "SUB", "MUL", "SHL"]
---
```

Listing 22: CWE-190 Integer Wrapraround

A.2.6 CWE-190 Integer Wraparound

In Listing 22 we show our implemetation of the "Integer Wraparound" property. We use a list of operations that are known to cause a wraparound to occur.

A.3 Implementation Artifacts

The implementation of Hy2 may be available at `https://github.com/crazyeights225/hy2` or by contacting the author.

Figure A.1: Hy2's mascot

Appendix B

Evaluation Artifacts

B.1 Summary of Vulnerabilities Identified

In the tables below errors with identifier prefixed by `ERR` are new vulnerabilities or flaws that we identified during our evaluation. We describe these errors in detail in Table B.4.

Vulnerability	Class	Missed
CVE-2018-20483	Information disclosure	
CVE-2019-5923	Write out-of-bounds	
CVE-2021-31879	Information disclosure	✗
ERR-01	Write out-of-bounds	
ERR-02	Information disclosure	

Table B.1: Summary of vulnerabilities described in our evaluation of wget

Vulnerability	Class	Missed
CVE-2021-23714	Integer overflow	
CVE-2021-32715	Allocation without limit	

Table B.2: Summary of vulnerabilities described in our evaluation of hyper

Vulnerability	Class	Missed
CVE-2021-26259	Heap buffer overflow	
CVE-2021-26252	Heap buffer overflow	
CVE-2021-23206	Stack buffer overflow	
CVE-2022-34033[1]	Heap overflow	
CVE-2021-33236[1]	Heap buffer overflow	
CVE-2021-23180	Null pointer dereference	
CVE-2021-23158	Double free	
CVE-2021-33235[2]	Heap buffer oveflow	
CVE-2021-40985[2]	Stack buffer under-read	
CVE-2021-43579	Stack buffer overflow	
CVE-2021-23191	Null pointer dereference	
CVE-2022-27114	Integer overflow	
CVE-2022-34035	Heap buffer overflow	✗
CVE-2022-0534	Stack out-of-bounds read	
CVE-2022-0137	Heap buffer overflow	✗
CVE-2021-34121	Out-of-bounds read	✗
CVE-2021-34119	Heap buffer overflow	✗
CVE-2021-26948	Null pointer dereference	✗
CVE-2021-23165	Heap buffer overflow	✗
CVE-2021-20308	Integer overflow	
ERR-03	Integer overflow	
ERR-04	Read out-of-bounds	
ERR-05	Read out-of-bounds	
ERR-06	Write out-of-bounds	
ERR-07	Unchecked Return Value	
ERR-08	Null pointer dereference	
ERR-09	Dangerous operations	

Table B.3: Summary of vulnerabilities described in our evaluation for htmldoc. The pairs [1] [2] are vulnerabilities that effect the same block of code and would not be distinguished from each other in our analysis.

Vulnerability	Summary
ERR-01	Write-out-bounds error from failing to allocate a sufficient amount of memory when performing a character encoding conversion in function `convert_fname()`.
ERR-02	Sensitive information in the url may be written to the console or to a log file when debug mode is enabled in function `http_loop()`.
ERR-03	Failure to check the values of the bmp dimensions in function `image_load_bmp()` may result in overflow when allocating memory, leading to a read out-of-bounds error.
ERR-04	In functions `update_image_size()` and `get_cell_size()`, there is a read out-of-bounds by 1 byte when an empty string is passed in as `buf[strlen(buf)-1]` causes the program to attempt to read the byte at index 0 in an empty string.
ERR-05	In function `image_load_bmp()` it attempts to read values from the BMP header without checking the size of the file, leading to invalid reads if the file is smaller than the size of the standard BMP header.
ERR-06	In function `parse_table()`, when `COLSPAN` is set to a valid integer greater than the size of the allocated table struct, it leads to a write out of bounds.
ERR-07	Failure to check that memory is allocated successfully before accessing it in functions `write_image()`, `flatten_tree()`, among others may lead to null pointer dereference errors.
ERR-08	In function `write_image()` it fails to handle errors from `load_image()`. When `load_image()` does not successfully read the image file, `write_image()` writes uninitialized memory to the output file. When memory for the image is not successfully allocated and null is returned, `write_image()` still attempts to access the image, causing a null-pointer dereference error.
ERR-09	Use of dangerous functions such as `atoi()`, `atol()`, and use of `strlen()` on non-null-terminated strings.

Table B.4: Summary of the new errors we identified in the completion of our evaluation.

Bibliography

[1] american fuzzing lop. https://lcamtuf.coredump.cx/afl/.

[2] american fuzzing lop online readme. https://lcamtuf.coredump.cx/afl/README.txt.

[3] Cwe - common weakness enumeration. https://cwe.mitre.org/.

[4] panda-re/panda: Platform for architecture-neutral dynamic analysis. https://github.com/panda-re/panda.

[5] Sei cert c coding standard. https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard.

[6] Sudo - static analysis. https://www.sudo.ws/security/static_analysis/.

[7] Cve-2018-20483 - cve details, 2018. https://www.cvedetails.com/cve/CVE-2018-20483/.

[8] Cve-2019-5953 - cve details, 2019. https://www.cvedetails.com/cve/CVE-2019-5953/.

[9] Fuzzing, 2020. https://en.wikipedia.org/wiki/Fuzzing.

[10] Nvd - cve-2020-15704, 2020. https://nvd.nist.gov/vuln/detail/CVE-2020-15704.

[11] Wget - gnu project - free software foundation, 2020. https://www.gnu.org/software/wget/.

[12] Addresssanitizer: double-free in function pspdf_export ps-pdf.cxx:945:7 - issue 414 - michaelrsweet/htmldoc - github, 2021. https://github.com/michaelrsweet/htmldoc/issues/414.

[13] Addresssanitizer: heap-based-overflow in write_header() html.cxx:273 - issue 425 - michaelrsweet/htmldoc - github, 2021. https://github.com/michaelrsweet/htmldoc/issues/425.

[14] Addresssanitizer: heap-based-overflow on render_table_row() ps-pdf.cxx:6123:34 - issue 417 - michaelrsweet/htmldoc - github, 2021. https://github.com/michaelrsweet/htmldoc/issues/417.

[15] Addresssanitizer: heap-buffer-overflow on write_node htmldoc/htmldoc/html.cxx:588 - issue 426 - michaelrsweet/htmldoc - github, 2021. https://github.com/michaelrsweet/htmldoc/issues/426.

[16] Addresssanitizer: Segv in file_extension file.c:337:29 - issue 418 - michaelrsweet/htmldoc - github, 2021. https://github.com/michaelrsweet/htmldoc/issues/418.

[17] Addresssanitizer: Segv on unknown address 0x000000000014 - issue 415 - michaelrsweet/htmldoc - github, 2021. https://github.com/michaelrsweet/htmldoc/issues/415.

[18] Addresssanitizer: stack-based-overflow in parse_table() ps-pdf.cxx:6611:25 - issue 416 - michaelrsweet/htmldoc - github, 2021. https://github.com/michaelrsweet/htmldoc/issues/416.

[19] Cve-2021-26252 - cve details, 2021. https://www.cvedetails.com/cve/CVE-2021-26252/.

[20] Cve-2021-31879 - cve details, 2021. https://www.cvedetails.com/cve/CVE-2021-31879/.

[21] Integer overflow in chunked transfer-encoding advisory hyperium/hyper - github, 2021. https://github.com/hyperium/hyper/security/advisories/GHSA-5h46-h7hh-c6x9.

[22] Release v0.14.9 - hyperium/hyper - github, 2021. https://github.com/hyperium/hyper/releases/tag/v0.14.9.

[23] stack-buffer-underflow in htmldoc - issue 444 - michaelrsweet/htmldoc - github, 2021. https://github.com/michaelrsweet/htmldoc/issues/444.

[24] Browse vulnerabilities by date - cve details, 2022. https://www.cvedetails.com/browse-by-date.php.

[25] Bug: buffer-overflow caused by integer-overflow in image_load_gif() - issue 423 - michaelrsweet/htmldoc - github, 2022. https://github.com/michaelrsweet/htmldoc/issues/423.

[26] Github - michaelrsweet/htmldoc: Html conversion software, 2022. https://github.com/michaelrsweet/htmldoc.

[27] Stack out-of-bounds read in gif_get_code() - issue 463 - michaelrsweet/htmldoc - github, 2022. https://github.com/michaelrsweet/htmldoc/issues/463.

[28] Two integer overflow bugs in image.cxx - issue 471 - michaelrsweet/htmldoc - github, 2022. https://github.com/michaelrsweet/htmldoc/issues/471.

[29] V. S. Alagar and K. Periyasamy. *Temporal Logic*, pages 177–229. Springer London, London, 2011.

[30] Android. Using binder ipc — android open source project. https://source.android.com/docs/core/architecture/hidl/binder-ipc.

[31] angr. angr/vex: A patched version of vex to work with pyvex. `https://github.com/angr/vex`.

[32] D. Wagner B. Schwarz, H. Chen. Model checking an entire linux distribution for security violations. Technical report, Berkeley, USA, 2005.

[33] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018.

[34] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, SPIN '01, page 103–122, Berlin, Heidelberg, 2001. Springer-Verlag.

[35] Zuzana Baranová, Jiří Barnat, Katarína Kejstová, Tadeáš Kučera, Henrich Lauko, Jan Mrázek, Petr Ročkai, and Vladimír Štill. Model Checking of C and C++ with DIVINE 4. In *International Symposium on Automated Technology for Verification and Analysis (ATVA) (to appear)*, volume 10482 of *Lecture Notes in Computer Science*, 2017. https://divine.fi.muni.cz/2017/divine4/.

[36] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. *Introduction to Runtime Verification*, pages 1–33. Springer International Publishing, Cham, 2018.

[37] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *International Journal on Software Tools for Technology Transfer*, 9(5–6):505–525, 2007.

[38] Fraser Brown, Deian Stefan, and Dawson Engler. Sys: a static/symbolic tool for finding good bugs in good (browser) code. In *Proceedings of the 29th USENIX Security Symposium*, pages 199–216, 2020.

[39] David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. Native x86 decompilation using Semantics-Preserving structural analysis and iterative Control-Flow structuring. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 353–368, Washington, D.C., August 2013. USENIX Association.

[40] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[41] Cristian Cadar and Koushik Sen. Symbolic execution for software testing. *Communications of the ACM*, 56(2):82–90, 2013.

[42] Hao Chen and David Wagner. Mops : an infrastructure for examining security properties of software. *Proceedings of the 9th ACM conference on Computer and communications security*, 2002.

[43] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. Selective symbolic execution. In *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*, number CONF, 2009.

[44] E. Clarke. Counterexample-guided abstraction refinement. *10th International Symposium on Temporal Representation and Reasoning, 2003 and Fourth International Conference on Temporal Logic. Proceedings.*, 2000.

[45] Edmund M. Clarke and Jeannette M. Wing. Formal methods. *ACM Computing Surveys*, 28(4):626–643, 1996.

[46] Luke Craig, Andrew Fasano, Tiemoko Ballo, Tim Leek, Brendan Dolan-Gavitt, and William Robertson. Pypanda: Taming the Pandamonium of Whole System Dynamic Analysis. *Proceedings 2021 Workshop on Binary Analysis Research*, Feb 2021.

[47] Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang. Verifying systems rules using rule-directed symbolic execution. *ACM SIGPLAN Notices*, 48(4):329–342, 2013.

[48] R.S. de Oliveira D. Bristot de Oliveira, T. Cucinotta. Efficient formal verification for the linux kernel. Technical report, Pisa, Italy, 2019.

[49] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp. *Proceedings of the ACM SIGPLAN 2002 conference on Programming language design and implementation*, 2002.

[50] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[51] Michael Donohue. Meta-compilation, 2007. `http://web.stanford.edu/class/archive/cs/cs295/cs295.1086/lectures/donohue-slides.pdf`.

[52] Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. Efficient algorithms for model checking pushdown systems. *Computer Aided Verification*, page 232–247, 2000.

[53] Curtis Franks. Propositional Logic. In Edward N. Zalta and Uri Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2023 edition, 2023.

[54] Valentin Goranko and Antje Rumberg. Temporal Logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2022 edition, 2022.

[55] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The seahorn verification framework. *Computer Aided Verification*, page 343–361, 2015.

[56] Dwight Guth, Chris Hathhorn, Manasvi Saxena, and Grigore Roşu. Rv-match: Practical semantics-based program analysis. *Computer Aided Verification*, page 447–453, 2016.

[57] Rachid Hadjidj, Xiaochun Yang, Syrine Tlili, and Mourad Debbabi. Model-checking for software vulnerabilities detection with multi-language support. *2008 Sixth Annual Conference on Privacy, Security and Trust*, 2008.

[58] Christian Heitman and Iván Arce. Barf: a multiplatform open source binary analysis and reverse engineering framework. 2014.

[59] Ori Hollander and Shachar Menashe. Watch our for dos when using rust's popular hyper package, 2023. `https://jfrog.com/blog/watch-out-for-dos-when-using-rusts-popular-hyper-package/`.

[60] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[61] Suman Jana. Data flow analysis, 2006. `https://www.cs.columbia.edu/~suman/secure_sw_devel/Basic_Program_Analysis_DF.pdf`.

[62] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):1–54, 2009.

[63] Katarína Kejstová, Petr Ročkai, and Jiří Barnat. From model checking to runtime verification and back. *Runtime Verification*, page 225–240, 2017.

[64] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. FirmAE: Towards large-scale emulation of iot firmware for dynamic analysis. In *Annual Computer Security Applications Conference (ACSAC)*, Online, December 2020.

[65] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

[66] Martin Leucker. Sliding between model checking and runtime verification. *Runtime Verification*, page 82–87, 2013.

[67] Georgy Lukyanov, Andrey Mokhov, and Jakob Lechner. Formal verification of spacecraft control programs. *ACM Transactions on Embedded Computing Systems*, 19(5):1–18, 2020.

[68] Victor Cutillas Matthieu Barjole. Security advisory: Sudoedit bypass in sudo ¡= 1.9.12p1 cve-2023-22809. Technical report, 2023. `https://www.synacktiv.com/sites/default/files/2023-01/sudo-CVE-2023-22809.pdf`.

[69] Madanlal Musuvathi, David YW Park, Andy Chou, Dawson R Engler, and David L Dill. Cmc: A pragmatic approach to model checking real code. *ACM SIGOPS Operating Systems Review*, 36(SI):75–88, 2002.

[70] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. *Lecture Notes in Computer Science*, page 213–228, 2002.

[71] Madhusudan Parthasarathy. Lecture 4: The product construction: Closure under intersection and union, 2010. `https://courses.engr.illinois.edu/cs373/sp2010/lectures/lect_04.pdf`.

[72] Maxime Peterlin, Philip Petterson, Alexandre Adamski, and Alex Radocea. Exploiting a single instruction race condition in binder, 2020. `https://www.longterm.io/cve-2020-0423.html`.

[73] Van-Thuan Pham, Marcel Bohme, and Abhik Roychoudhury. Aflnet: A greybox fuzzer for network protocols. *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020.

[74] The QEMU Project. Qemu disk image utility - qemu 8.0.0 documentation, 2022. `https://qemu.readthedocs.io/en/latest/tools/qemu-img.html`.

[75] radare. radare/radare2: Unix-like reverse engineering framework and commandline toolset. `https://github.com/radareorg/radare2`.

[76] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. *NDSS*, 2015.

[77] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, December 2008.

[78] Johannes Späth, Karim Ali, and Eric Bodden. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[79] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. *Proceedings 2016 Network and Distributed System Security Symposium*, 2016.

[80] Robert Tarjan. Testing flow graph reducibility. *Proceedings of the fifth annual ACM symposium on Theory of computing - STOC '73*, 1973.

[81] TEC. Virtualization vs. emulation vs. simulation: What's the difference? `https://www3.technologyevaluation.com/research/article/virtualization-vs-emulation-vs-simulation-whats-the-difference.html`.

[82] Douglas Thain. *Intermediate Representation.* Douglas Thain, 2020.

[83] Liz White. Control flow analysis - comp 621 - program analysis and transformations. `https://www.sable.mcgill.ca/~hendren/621/ControlFlowAnalysis_Handouts.pdf`.

[84] Ding Zhanzhao. Ghidra to the next level, 2021. `https://conference.hitb.org/hitbsecconf2021sin/materials/D1T2%20-%20Taking%20Ghidra%20to%20The%20Next%20Level%20-%20Zhanzhao%20Ding.pdf`.

[85] Shunfan Zhou, Zhemin Yang, Dan Qiao, Peng Liu, Min Yang, Zhe Wang, and Chenggang Wu. Ferry: State-Aware symbolic execution for exploring State-Dependent program paths. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4365–4382, Boston, MA, August 2022. USENIX Association.