TÉCNICO LISBOA

Toward Tool-Independent Summaries for Symbolic Execution

Frederico Duarte Ramos

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Pedro Miguel dos Santos Alves Madeira Adão
Prof. José Faustino Fragoso Femenin dos Santos

November 2021

Acknowledgments

First and foremost I must thank my dissertation supervisors Prof. Pedro Adão and Prof. José Santos for their insight, support and sharing of knowledge that not only made this thesis possible, but also made it an extremely fun project. I would also like to give a special thanks to my colleague Nuno Sabino for his time, patience, and friendship that allowed me to build this thesis on top his previous work.

Last but not least, I must thank my friends and family for always being there for me, specially my girlfriend Cristiana, without whom this project would not be possible.

Abstract

Symbolic execution is a program analysis technique that has been successfully used to find various types of bugs in industrial codebases. Despite being extensively used in practice, this technique suffers from two main limitations when applied to real-world code: *path explosion* and *interactions with the runtime environment*. To address both of these problems, current symbolic execution engines make use of symbolic summaries. These interact with the symbolic state of a given program so as to simulate the behaviour of both external and internal functions without having to symbolically execute them. Symbolic summaries can therefore be used to mitigate the number of paths explored and also allow for the analysis of external calls. Despite their advantages, there is a clear lack of mechanisms for sharing symbolic summaries across different tools and for their uniform validation.

In this thesis we introduce a methodology for implementing tool-independent symbolic summaries for the C programming language. This methodology consists of an API containing a set of symbolic reflection primitives for explicit manipulation of C symbolic states. Symbolic summaries implemented using our API can be shared across different symbolic execution tools, provided that these tools implement the proposed API. Additionally, due to being written directly in C, these summaries can themselves be symbolically executed as standard C code. Hence, we also introduce a summary validation tool that can systematically evaluate the correctness of a symbolic summary with respect to its concrete reference implementation.

Keywords

Symbolic Execution, Runtime Modelling, Symbolic Summaries, Summary Correctness

Resumo

Execução Simbólica é uma técnica de análise de programas que tem sido aplicada com sucesso para encontrar vários tipos de bugs em codebases industriais. Apesar de ser bastante utilizada na prática, esta técnica sofre de duas limitações principais quando aplicada ao código de programas reais: *explosão de caminhos* e *interações com o ambiente*. De forma a tratar ambos os problemas, os motores de execução simbólica atuais fazem uso de sumários simbólicos. Estes interagem com o estado simbólico de um determinado programa de forma a simular o comportamento de funções internas e externas sem ter que executá-las simbolicamente. Os sumários simbólicos podem assim ser usados para mitigar o número de caminhos explorados e permitir a análise de chamadas externas. Apesar das suas vantagens, há uma clara falta de mecanismos para compartilhar sumários simbólicos entre diferentes ferramentas bem como para a sua validação uniforme.

Neste projeto apresentamos uma metodologia para implementar sumários, independentes de ferramentas, para a linguagem de programação C. Esta metodologia consiste numa API que contém um conjunto de primitivas de reflexão simbólica para manipulação explícita de estados simbólicos de C. Os sumários simbólicos implementados utilizando nossa API podem ser compartilhados entre diferentes ferramentas de execução simbólica, assumindo que estas ferramentas implementam a API proposta. Além disso, por serem escritos diretamente em C, estes sumários podem ser executados simbolicamente como qualquer código C. Desta forma propomos também uma ferramenta de validação de sumários que pode avaliar sistematicamente a correção de um sumário de acordo com a respetiva implementação de referência.

Palavras Chave

Execução Simbólica, Modelação de Runtime, Sumários Simbólicos, Correção de Sumários

Contents

List of Figures

List of Tables

List of Listings

Acronyms

CGC	Cyber Grand Challenge
CTF	Capture the Flag
CVE	Common Vulnerabilities and Exposures
I/O	Input/Output
PoV	Proof of Vulnerability
SMT	Satisfiability Modulo Theories

1

Introduction

Contents

1.1 Motivation

The complexity of modern software systems renders the process of bug-finding extremely hard, especially when done manually. This leaves room for undetected security vulnerabilities in production code, which can then be exploited by malicious users and have serious consequences for both organizations and individuals. For instance, the *Heartbleed* bug [1, 2], present in version 1.0.1 of the OpenSSL cryptographic library, allowed for the leakage of sensitive information protected by SSL/TLS encryption, which is used to secure most types of Internet traffic.

Symbolic execution [3, 4] is a program analysis technique that allows for the exploration of all the execution paths of the given program up to a bound by executing that program with symbolic values instead of concrete ones. For each execution path, the symbolic execution engine builds a first order formula, called *path condition*, which accumulates the constraints on the symbolic inputs that cause the execution to take that path. Symbolic execution engines rely on an underlying Satisfiability Modulo Theories (SMT) solver both to check the feasibility of execution paths as well as to check the validity of the assertions supplied by the developer.

Symbolic execution has been successfully used to find a wide variety of bugs and security vulnerabilities in large industrial codebases. For instance, *KLEE* [5] found various fatal bugs in GNU COREUTILS (version 6.10) and a large number of critical bugs in other software systems, such as BUSYBOX [6] and MINIX [7]; *SAGE* [8] found many vulnerabilities in Windows patches and applications, such as MS07-017 patch and Office 2007; *SLAM* [9] found many bugs in Windows Device Drivers; and *Java PathFinder* [10], developed by NASA, was used to test the Orion Spacecraft's control software.

1.2 Problem

Despite being extensively used in practice, symbolic execution suffers from two main limitations when applied to real-world code: *interactions with the runtime environment* and *path explosion*.

Most real-wold programs interact with their runtime environment via complex heterogeneous application programming interfaces (APIs), whose source code is often not available for static analysis. These interactions, which include operations involving the network, the operating system, the file system, and system devices, may have a considerable effect on the execution of the program at hand and therefore must be taken into account by symbolic execution engines. Nevertheless, the symbolic analysis of such interactions is not straightforward as they often step outside the perimeter of the programming language being analysed. For instance, system calls cannot be directly symbolically executed since their implementation is not part of the program being analysed, belonging instead to the underlying operating system. The standard approach to support such interactions is to create summaries of the external runtime functions called by the program to be analysed. These symbolic summaries constrain the symbolic

state of the given program so as to simulate the behaviour of the external functions without having to symbolically execute them.

Let us now consider the path explosion problem. All but the smallest programs have an unmanageable number of possible execution paths, which is exponential in the number of executed conditional instructions. For this reason, a naive symbolic execution engine that attempts to explore all possible paths will never scale to real-world programs. The standard approach to deal with the path explosion problem is to use sophisticated merging algorithms to combine multiple symbolic execution paths into a single path [11], abstracting away the differences between the merged paths. Nevertheless, such general algorithms can be too coarse, forgetting details that would be useful for detecting specific bugs/vulnerabilities, since the optimal merging strategy is oftentimes dependent on the type of bug/vulnerability that one is searching for.

Alternatively, one can leverage symbolic summaries to contain the number of paths explored during symbolic execution. The idea is that instead of symbolically executing the code of a given concrete function on some symbolic inputs, one can choose to implement a symbolic summary that models the behaviour of that function, and then execute the summary instead of the concrete function. Symbolic summaries have two main advantages with respect to concrete implementations. First, they allow developers to choose which execution paths are to be explored by the symbolic execution engine, steering the execution towards the paths that may potentially lead to bugs/vulnerabilities. Second, they allow developers to merge different symbolic execution paths into the same one by explicitly interacting with the current symbolic state (e.g., extending the current path condition and creating new symbolic variables). Hence, symbolic summaries provide an effective merging mechanism, allowing developers to deal with the path explosion problem in an application-specific way.

In general, symbolic summaries can only approximate the behaviour of their corresponding concrete functions. This means that, as a general rule, the symbolic paths captured by a symbolic summary do not exactly coincide with those that would have been generated by the symbolic execution of its corresponding function. When it comes to correctness guarantees, we distinguish two main categories of symbolic summaries: *backward sound* [12] and *forward sound* [13]. We say that a symbolic summary is *backward sound* if all the execution paths modelled by the summary are contained in the set of concrete paths of its corresponding function. In other words, a *backward sound* symbolic summary guarantees that all of its generated paths correspond to concrete paths. Conversely, we say that a symbolic summary is *forward sound* if it models all the concrete paths of its corresponding function; that is, a forward sound symbolic summary must take into account all possible concrete paths, even if that means also including wrong paths. It is often the case that one cannot be at the same time backward and forward sound. The type of property to be aimed at depends on how the summary is going to be used. For instance, security applications often require forward soundness in order to guarantee the absence

of security vulnerabilities, while debugging tools often choose to be backward sound, only reporting the bugs that are guaranteed to exist.

Currently each symbolic execution tool implements its own symbolic summaries in the programming language used to build the tool. For instance, *angr*'s summaries are implemented in Python, *KLEE*'s summaries are implemented in C, and *BINSEC*'s summaries are implemented in OCaml, even though all these tools target C code. Furthermore, symbolic summaries often rely on specific aspects of the tools for which they are implemented. In particular, they interact with the symbolic states of the programs being analysed through the APIs provided by each tool. Hence, it is not only extremely difficult to share symbolic summaries between symbolic execution tools, but also to check whether or not the implemented summaries satisfy the properties that their authors intended them to. Surprisingly, although there is a clear lack of appropriate tool support for developing and sharing symbolic summaries across different symbolic execution tools, the research community has not yet given much attention to this topic.

1.3 Goals

In this thesis we introduce a methodology for implementing tool-independent symbolic summaries for the C programming language. At the core of the proposed methodology is a new API consisting of a set of symbolic reflection primitives [14] for explicit manipulation of C symbolic states in a tool-independent way. Our symbolic primitives include a variety of instructions for: creating symbolic variables and first-order constraints, checking the satisfiability of constraints, and extending the current path condition or symbolic state with a given constraint. Symbolic summaries implemented using our API can be shared across different symbolic execution tools, provided that these tools implement the proposed API. To illustrate the applicability of our methodology, we extended the symbolic execution tools *angr* [15] and *AVD* [16] with support for the proposed API and developed tool-independent symbolic summaries for three classes of libc functions: string manipulation functions, number-parsing functions, and input/output functions.

Given that our symbolic summaries are directly implemented in C, they can themselves be symbolically executed as standard C code. By comparing the execution paths generated by the symbolic execution of a summary against those generated by its corresponding function, we can determine whether or not the summary satisfies the soundness properties discussed above. If a summary is backward sound, its execution paths must be included in those of its corresponding function. Conversely, if a summary is forward sound, its execution paths must include those of the corresponding concrete function.

In this thesis, we extended *AVD* with an infrastructure for automatically checking whether or not a given summary satisfies backward/forward soundness by comparing its execution paths against those of its corresponding function. In a nutshell, our validation tool receives as input a symbolic summary, its corresponding function, and a set of symbolic inputs, checking whether or not the paths generated

by the summary when executed on those inputs are contained in the paths generated by its concrete implementation and vice-versa. Importantly, our validation tool does not provide a verification guarantee with respect to the backward/forward soundness of the given summary; it simply tries to find counterexamples for both properties. It is the job of the developer to provide sufficiently many symbolic inputs to make sure that results are trustworthy/generalizable.

1.4 Evaluation

In order to assess if our symbolic reflection API is expressive enough to allow for the development of useful symbolic summaries, we implement a set of summaries modelling 20 different libc functions and use our validation tool to check their soundness. Additionally, we evaluate the performance of the summaries implemented using our proposed API. To this end, we implement a subset of the native libc summaries originally included in *AVD* directly in C and compare their relative performances on two distinct data sets.

In order to evaluate the effectiveness and scalability of our summary validation tool, we use it to test symbolic summaries included in real-world symbolic execution tools so as to find soundness bugs inadvertently introduced by tool developers. To this end, using our symbolic reflection API, we implement a subset of the libc summaries included in the symbolic execution tools *angr* and *Manticore* [17] and use our summary validation tool to check if the obtained summaries satisfy either backward or forward soundness. Out of a total of 14 analysed summaries, we found two buggy summaries, one in *angr* and one in *Manticore*. Both summaries include spurious paths and exclude correct paths, meaning that they are neither backward nor forward sound.

1.5 Contributions

This thesis focus on the development and porting of symbolic summaries across different symbolic execution tools, making the following contributions:

- An API for the development of reusable summaries for C;
- Library of summaries modelling 20 libc functions, each function with several summaries satisfying different correctness properties;
- Extension of the state-of-the-art symbolic execution tools *angr* and *AVD* with support for the symbolic reflection API;
- Auxiliary tool for automatic summary validation and debugging;

- Experimental evaluation of the performance of platform-independent summaries on both an external dataset of binaries and an HashMap data structure;
- Experimental evaluation of the summary validation tool with external summaries implemented by symbolic execution tools.

1.6 Thesis Outline

The remaining of this document is organized as follows: in Chapter 2 we cover the symbolic execution technique and the common approaches used to mitigate its core limitations, with special focus on symbolic summaries, and how they are used by state-of-the-art symbolic execution tools. Chapter 3 presents our symbolic reflection API for implementing tool independent summaries and how it is supported in the symbolic execution tools *AVD* and *angr*. In Chapter 4 we go over our proposed correctness properties for evaluating symbolic summaries and how we implemented the summary validation tool to systematically check these properties. Chapter 5 covers the evaluation results for all the elements of our solution (symbolic summaries and summary validation tool). Finally, Chapter 6 concludes this document with a final overview of our contributions and possible future work.

2

Related Work

Contents

In this section we start by going over the symbolic execution technique, discussing its core limitations and common optimization strategies employed by symbolic execution tools (Section 2.1). Then, we cover the concept of function modelling and explain how different types of summaries can be implemented to improve the scope and scalability of symbolic execution (Section 2.2). Finally, we conduct a survey to analyse how modern symbolic execution tools utilize summaries to support and model the libc (Section 2.3).

2.1 Symbolic Execution

Modern software testing relies on automatic detection tools that, given a program, can autonomously detect bugs and vulnerabilities with minimal user input. Many of these tools are based on *Symbolic Execution* [11] to allow for going through and analysing all the possible execution paths of a program. This software analysis technique, explores multiple execution paths in a program by abstracting concrete inputs with symbols, consequently allowing to express execution paths in terms of constraint formulas over symbolic inputs, called *path conditions*. Essentially, path conditions accumulate the constraints on the symbolic inputs that cause the execution to take a given path.

2.1.1 Classic Symbolic Execution - Example

Consider the function `test` in Listing 2.1, a function with only 3 execution paths, where a bug is represented by a failed assertion in line 7. Even in this very simple case, it is not immediately obvious what combination of arguments x and y will cause the execution to reach the bug (e.g., $\{x = 1, y = 4\}$). We can see that vulnerability detection quickly becomes a very difficult problem as the software complexity increases, specially considering that in most real cases, the faulty line is not known beforehand.

Figure 2.1 shows the execution tree, produced by symbolically executing the example of Listing 2.1. The symbolic execution engine maintains for each node in the tree a *memory store* that maps variables to their concrete or symbolic values, and the current path condition. The execution starts with a single node containing x and y as symbolic inputs variables, and the path condition is initialized as $True$. For every step the *memory store* is updated according to the declarations of new variables and operations between them. The *path conditions* are updated every time the execution may branch with the formula that forces the execution to take the corresponding branch. At the end of each execution path, the SMT solver can be used to check whether the path is realizable, and what assignment of concrete values to the symbolic arguments satisfies that path's formula.

```
void test(int x, int y){
    i = 2 * x;
    if(y > i){
        if(x < y - 2){
            assert(0 == 1); /*ERROR (Assert Fail)*/
        }
    }
}
int main(){
    x = symbolic_var();
    y = symbolic_var();
    test(x,y);
    return 0;
}
```

Listing 2.1: Example of a *test* function for symbolic execution

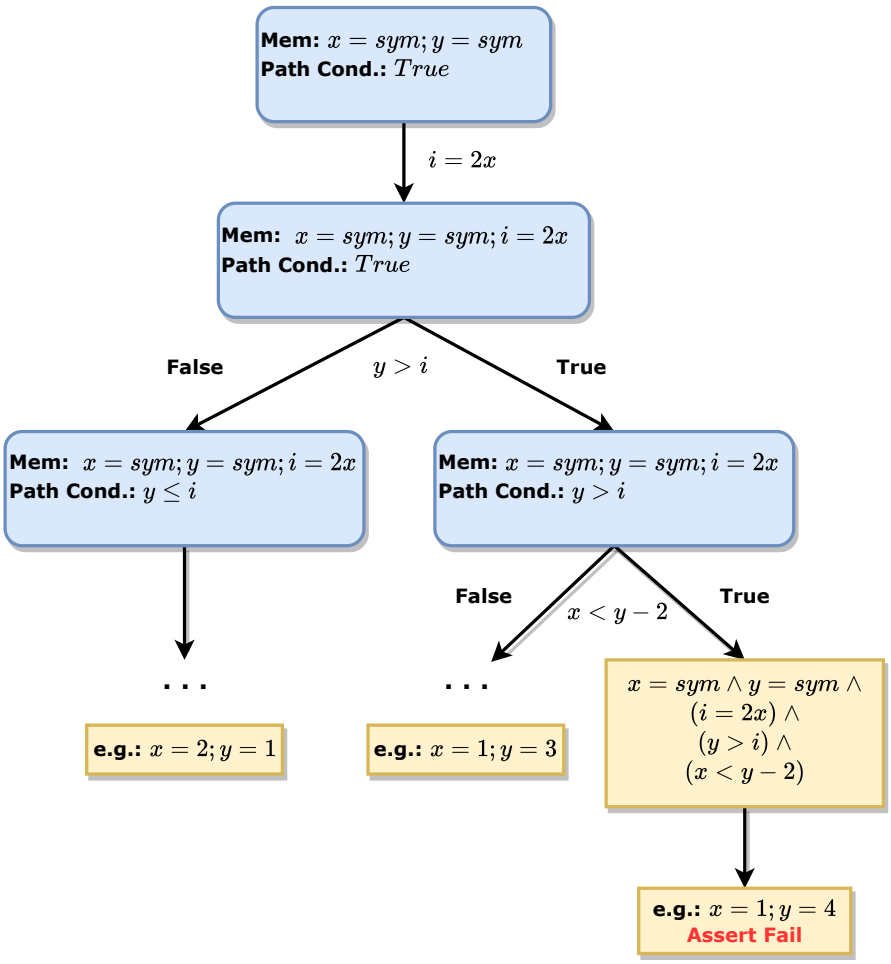


Figure 2.1: Symbolic execution tree of the program in Listing 2.1

2.1.2 Limitations

In recent year, we have seen a great many number of new applications of symbolic execution appearing due to significant advances in constraint solving, which is still one of the technique's main bottlenecks. A constraint solver is needed to reason about the execution paths expressed as logical formulas, and to generate models for those formulas. These interactions can be reduced to the boolean satisfiability problem (SAT) which naturally is an NP-complete problem. Despite that, modern SMT solvers like *Z3* [18] and *CVC4* [19] have pushed the boundaries for what can be attainable in practical applications.

Another major hindrance to symbolic execution is the *path explosion* problem. When performing pure symbolic execution on a target program, such as the example of Figure 2.1, a symbolic engine's executor will fork and create a symbolic state at each conditional statement. We can see that for real programs this can exponentially add up to an enormous number of paths, or even an infinite number of paths when considering loops that branch on symbolic values. For this reason, the standard approach to deal with the path explosion problem is to use sophisticated merging algorithms to combine multiple symbolic execution paths into a single path [11], abstracting away the differences between the merged paths.

Finally, one other key limitation of symbolic execution is the modelling of the runtime environment. Real world code is not self-contained as it interacts with its execution environment via runtime libraries and system calls. This can be an issue not only because the external code may not be available for symbolic execution, such as code dependencies and frameworks, but also because execution can reach elements that are outside the scope of the symbolic engine. Programs frequently interact with system components, for example, the file system, environment variables, devices, or external elements such as the network. Ignoring these interactions altogether, or even executing external calls concretely, can lead to a loss of relevant execution paths.

2.1.3 Advanced Symbolic Execution

Modern symbolic execution techniques use a variety of dynamic approaches to support the analysis of larger scale programs. For example *Concolic Testing* [20] mixes concrete and symbolic execution, using an initial concrete input and collecting the symbolic constraints along the execution path generated by that input. At the end of the execution the SMT solver is used to compute a new input, guiding a new execution towards a different path. Heuristics like DFS, BFS, random search, etc, can also be used to direct path exploration, or even static control-flow graphs (CFG) that allow guiding the exploration towards paths with specific properties [21]. A common technique is to try passing the complexity of reducing the search space to the constraint solver, e.g., *AEG* [22] proposed by Avgerinos et al uses *preconditioned symbolic execution*, a technique that prunes execution paths that do not satisfy a certain

precondition. Most tools concerned with supporting the runtime interactions, implement some variation of symbolic summaries to abstract library code and system calls [15, 16, 23], while some tools even try to model parts of the execution environment, for instance *KLEE* employs a symbolic file system to allow reading and writing from files during an analysis [5]. *AEG* in addition to the file system, also tries to model most of the elements that can be used as user input such as sockets and environment variables. Furthermore, some tools like *S^2E* go one step further and use virtualization to let target programs interact with "real" environments during an analysis [24].

2.2 Function Summaries

In the context of symbolic execution, there are various strategies for modelling runtime functions and system calls. A naive approach is to simply add concrete implementations of the runtime functions to be supported by the program to be analysed. We refer to such concrete implementations as *concrete models*. Concrete models have two main drawbacks: first they promote path explosion by introducing extra code that must be executed symbolically, for example the *strlen* function will create a new execution path for each character of a symbolic string. Second, most of the interactions with the runtime environment cannot be captured by the programming language. In the case of system calls, execution will reach elements that are not under control of the symbolic execution engine. For instance, *fgets* interacts with the kernel to read from the *stdin*.

Given the limitations of concrete models, the standard approach to model the behaviour of runtime functions and system calls is to use *symbolic summaries* [11]. A symbolic *summary* is a model of a function that simulates its behaviour by interacting directly with a symbolic state and the underlying symbolic engine. Symbolic summaries are therefore an excellent device to scale symbolic execution for larger programs, reducing the time spent during the symbolic execution itself, as well as pruning the search space by capturing the outcome paths of a function call in a reduced number of branches compared to a concrete model. Summaries can achieve this by updating the path condition of a symbolic state with the constraints representing the new states that would have been created by a concrete function. Furthermore, with symbolic summaries, one can analyse even the system/runtime calls that cannot be modelled using concrete implementations by simply executing their corresponding summaries with the supplied arguments. Figure 2.2 illustrates the difference between the symbolic execution of a concrete implementation and a symbolic summary for libc's *strlen* function, considering an input string with N symbolic characters.

There are two main approaches for implementing symbolic summaries. The most common approach consists of implementing symbolic summaries in the programming language used to build the symbolic execution tool itself that can directly access and manipulate the symbolic state. For example *angr* [15]

and *KLEE* [5] both implement symbolic summaries in their respective native programming languages as part of the tools themselves. Alternatively some symbolic execution tools implement symbolic summaries in the assembly language used for intermediate representation of the target program, for example *BINSEC* [25] uses OCaml to generate assembly code comprising a symbolic summary, which is then injected in the addresses of external function calls.

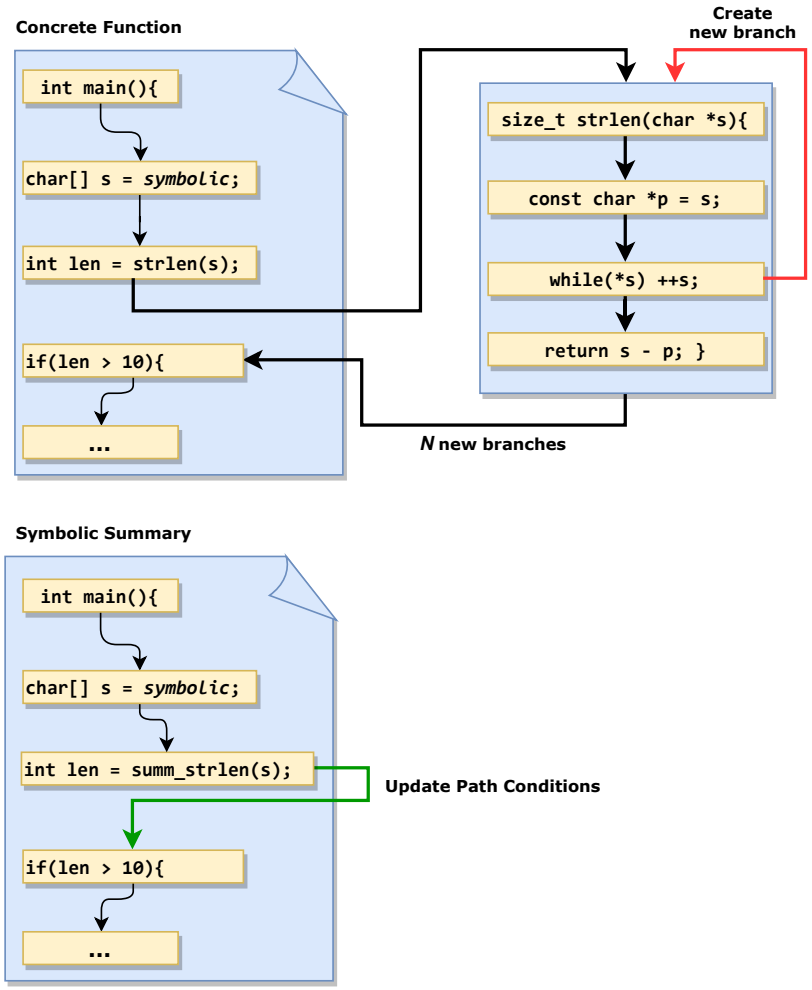


Figure 2.2: Symbolic execution of a concrete function versus a corresponding summary.

2.2.1 Symbolic Reflection

In the context of symbolic execution, symbolic reflection is a mechanism that can be applied by a symbolic engine to infer runtime properties of the symbolic state during an analysis. For example, as illustrated in Figure 2.3, symbolic reflection can be used at a conditional statement to determine if a certain branch is feasible according to the preceding path conditions. Checking branch satisfiability at each conditional statement is a state pruning technique called *eager evaluation* that can be employed to optimize

symbolic execution [11]. Symbolic reflection is not only useful for implementing dynamic approaches to help scale symbolic execution, but it is also very convenient for developing symbolic summaries, as it allows to model the behaviour of a summary according to the specific symbolic runtime properties of its input arguments in a given symbolic state.

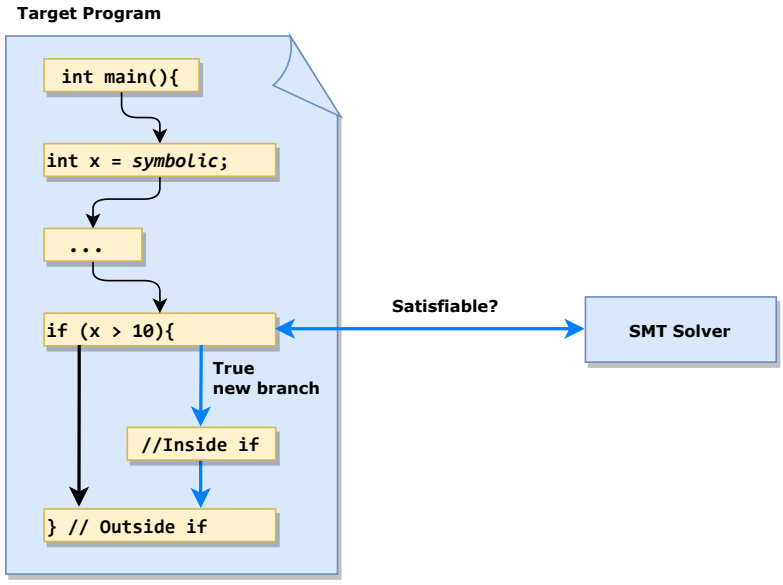


Figure 2.3: Example of Symbolic Reflection for *eager evaluation*

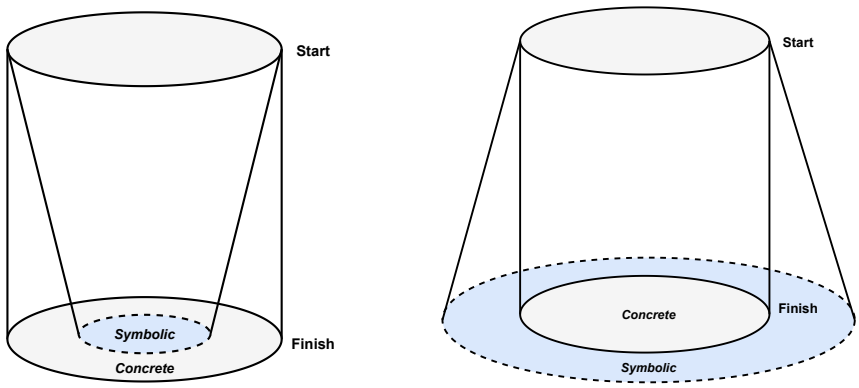
2.2.2 Properties of Summaries

Due to the limitations we have covered in section 2.1.2, usually symbolic execution can only approximate the behaviour of real-world code. This means that when considering larger scale programs, specially those that interact with their runtime environment, as a general rule it is impossible to guarantee that symbolic execution covers all and only the correct execution paths. For example when analysing a program with possible infinite execution paths due a symbolic loop (e.g., `while(n)` where `n` is symbolic), a symbolic execution tool may resolve this by unravelling the loop to a concrete number of iterations, thus potentially losing important paths. On the other hand when considering a program that reads from an external file, a tool may model this interaction by creating symbolic bytes to simulate all the "read" data, possibly leading to invalid execution paths.

The correctness of an analysis is often evaluated according to the properties of *Backward* and *Forward Soundness* [12, 13]. A *backward sound* analysis guarantees that all generated paths are correct with respect to the concrete execution. Conversely, in a *forward sound* analysis all the possible execution paths are taken into account, even if that means covering wrong paths. Figures 2.4(a) and 2.4(b) illustrate the properties of Backward and Forward Soundness respectively. As a device that intrinsi-

cally affects the correctness of an analysis, these properties can also be directly applied to symbolic summaries.

It is often the case that one has to sacrifice backward soundness (precision) for forward soundness and vice-versa. The type of property to be achieved depends on how the summary is going to be used. For instance, security analyses often require sound summaries: if symbolic execution says that there is no security bug, then there is no security bug. In contrast, debugging/testing tools require precise summaries given that developers do not want to waste their time fixing bugs that do not exist: if symbolic execution says that there is a bug, then the bug must exist. Unfortunately, in general, it is not possible to have both.



(a) Backward Soundness **(b)** Forward Soundness

Figure 2.4: Standard Correctness Properties

2.3 *libc* support on Symbolic Execution Tools

In order to understand how current symbolic execution tools for C make use of symbolic summaries, we analyse how such tools model interactions with libc. More concretely, we survey 12 C compatible symbolic execution tools, checking for each tool the number of libc summaries that it implements. Results are shown in Table 2.1, which divides the libc summaries into 7 categories: string manipulation functions, Input/Output (I/O) functions, file handling functions, memory functions, process management functions, socket functions, and other system calls. Below we give a small description of each tool, focusing on their support for libc summaries.

angr *angr* [15] is a multi-architecture binary analysis toolkit, developed by researchers of the Computer Security Lab at UC Santa Barbara and SEFCOM[1] at Arizona State University, with the ability to perform

[1] Laboratory of Security Engineering for Future Computing

Table 2.1: libc summaries implemented by C compatible symbolic execution tools

	angr	AVD	BE-PUM	BINSEC	Kite	KLEE	Manticore	Mayhem	Otter	pysymemu	S2E	Triton
Implementation Language	Python	Python	-	OCaml	-	C	Python	?	C	Python	C++	-
String Manipulation	19	13	0	5	0	16	5	?	0	0	6	0
Input/Output	12	7	0	1	0	2	2	?	3	1	1	0
File Handling	33	4	0	7	0	7	28	?	29	10	1	0
Memory	7	7	0	6	0	8	15	?	18	1	2	0
Process Management	7	1	0	0	0	2	5	?	3	7	0	0
Sockets	9	0	0	0	0	1	11	?	8	2	0	0
Other System Calls	18	4	0	1	0	0	22	?	24	10	0	0
Total	105	36	0	20	0	36	88	≥ 30	85	31	10	0

dynamic symbolic execution, and various static analyses on binaries. *angr* is written in Python and uses VEX [26] as its intermediate representation, supporting various architectures such as x86, x86-64, ARM, MIPS, etc, among many other binaries. To improve the time performance and support for system calls, *angr* implements *SimProcedures*, a dedicated library of function summaries, modelling the behaviour of the most common library functions, including the libc. *angr* is the most comprehensive symbolic execution tool analysed regarding the number of libc functions modelled, implementing a total of 105 libc *SimProcedures* for different categories. The developers intend to extend the *SimProcedures*' list even more for future work. *angr* was used as the backbone of *Team Shellphish*'s Cyber Reasoning System for the DARPA[2] Cyber Grand Challenge (CGC), enabling them to win third place in the final round [27].

AVD *AVD* or Automatic Vulnerability Detection [16] is a symbolic execution tool developed at IST for x86/x64 binary. *AVD* is written in Python and uses BAP [28] for intermediate representation, lifting the binary into an ADT format. The lifted instructions are parsed using a visitor, effectively creating an interpreter for the intermediate language. Symbolic execution is performed using standard DFS search, that can be guided to prioritize paths using an additional heuristic responsible for dynamically directing the symbolic execution along a previously computed execution trace e.g., a known trace that triggers a bug. To further improve scalability and support external calls, *AVD* also implements symbolic summaries for 36 libc functions, including common system calls.

BE-PUM *BE-PUM* or *Binary Emulation for Pushdown Model* [29] is a binary analysis tool with the main

[2]Defense Advanced Research Projects Agency

focus on generating Control Flow Graphs of malware. *BE-PUM* disassembles x86 binary code handling common obfuscation techniques such as *indirect jump*, *self-modification*, *overlapping instructions*, and *structured exception handler*, using concolic execution to generate a model of the binary code. *BE-PUM* does not support any type of external function call, and has mostly been used for research on malware analysis, in particular for malware model generation [29–31].

BINSEC *BINSEC* [25] is a binary analysis tool developed by the BINSEC project. This tool is built based on an extension of the DBA (Dynamic Bit-vector Automata) intermediate representation [32], currently supporting only x86-32/ELF. From the intermediate representation *BINSEC* generates Control Flow Graphs of the executable to be analysed through four different solutions: recursive disassembly, linear disassembly, a combination of both recursive and linear disassembly, and symbolic execution. External functions are supported by *stubs*; a block of DBA instructions generated in OCaml is injected at an address of a function call whose code is not available. These stubs work as function summaries modelling the behaviour of external calls, totalling 24 libc stubs. This tool is responsible for a series of accolades, the most recent ones were the discovery of 30 new bugs in 6 widely used programs (GPAC, GNU Patch, Perl 5, MuPDF, Boolector, fontforge), 7 of which would lead to new CVEs [33], and the analysis of 338 cryptographic primitives (from e.g HACL*, OpenSSL, BearSSL and NaCL) finding 3 new binary-level timing-attack vulnerabilities [34].

Kite *Kite* [35] is a symbolic execution tool built on LLVM, developed as a proof of concept for Conflict-Driven Symbolic Execution (CDSE). CDSE tries to address the path explosion problem during symbolic execution by reducing the search space with the following key insight: if a combination of path conditions makes a specific branch impossible, i.e., raises a conflict, that conflict can be stored in order to prune other execution branches also leading to that conflict. Fruit of being a proof of concept for a novelty technique *Kite* does not support external function calls.

KLEE *KLEE* [5] is a dynamic symbolic execution engine built on LLVM, that can be used for automatic test generation and bug finding even on programs that interact frequently with their execution environment. This tool functions as an *"operating system for symbolic processes"* [5], with each symbolic process having its own symbolic state and path conditions. *KLEE* dynamically guides the symbolic execution by combining two heuristics: *Random Search* that randomly selects an execution path, and *Coverage-Optimized Search* which tries to maximize code coverage. To support calls to external functions, *KLEE* provides a set of runtime libraries to simulate the Linux environment, offering summaries for 36 libc functions. In order to resolve system calls that are not modelled natively, *KLEE* concretizes their arguments and forwards these calls to the real operating system. Additionally, the code for undefined functions can also be provided by the user by linking external libraries. However, in this case, external calls can also be at most executed with concrete arguments. *KLEE* is one of the most popular state-

of-the-art symbolic execution tools, with extensive use from both Industry and Academia, counting over 1700 citations of its introducing paper [5], and responsible for finding a large number of critical bugs in various software systems.

Manticore *Manticore* [17] is a symbolic execution tool developed by the cybersecurity research firm *Trail of Bits*, that can be employed for bug finding and input generation on binaries and smart contracts. This tool aims to support a wide variety of environments while minimizing external dependencies. Currently, *Manticore*'s supports the analysis of ELF binaries (x86, x86_64, aarch64, and ARMv7), Webassembly modules (Wasm) and Ethereum smart contracts (EVM bytecode). To allow for the tool's flexibility, *Manticore*'s authors implement a generic symbolic execution engine that operates over symbolic states whilst making few assumptions over the specific program being analysed. *Manticore* also includes support for the Linux environment modelling a total of 84 system calls. However, in the vast majority of these models, symbolic arguments passed to the system calls are concretized as the authors consider these interactions unreasonable to modelled symbolically [17]. Despite this, the authors intend to expand the list of supported system calls even further. Additionally, *Manticore* also includes 5 symbolic summaries modelling 4 standard libc functions: *strlen*, *strcmp*, *strcpy* and *strncpy*. This tool was used as the foundation of *Trail of Bits*'s Cyber Reasoning System for the CGC, allowing the team to place nineth in qualifying round [36].

Mayhem *Mayhem* [23] is a closed source binary analysis tool, that combines guided fuzzing with symbolic execution for bug finding, generating control flow hijack exploits for every bug found. *Mayhem* is built on the BAP intermediate representation [28], supporting x86_64, x86, and ARM binary. This tool uses an hybrid approach to symbolic execution, combining both concolic and forward (classic) symbolic execution to maximize the effectiveness of each technique. The execution starts in forward mode, switching to concolic when a memory limit is reached. In concolic mode execution no longer branches at conditional statements, instead checkpoints are created that can later be used to resume the forward execution. *Mayhem* also uses dynamic taint analysis to help detect which paths depend on tainted values, e.g. paths where a jump instruction is tainted. According to [23], *Mayhem* models 30 Linux system calls, however due to being closed source, no more information was found about the specific functions that it models and the technique used to implement them. *Mayhem* has found many vulnerabilities in recent years, being responsible for 5 new CVEs in 2020, which were found on 4 different systems/libraries: *OpenWRT RCE* (CVE-2020-7982) [37], *cereal* (CVE-2020-11104 [38] and CVE-2020-11105) [39], *MP3Gain* (CVE-2020-15359) [40], and the *GNU C Library* (CVE-2020-10029) [41].

Otter *Otter* [42] is a symbolic execution tool originally developed to study how configuration options affect the behaviour of a program in practice. This research showed that the actual number of possible

configurations is much smaller than its corresponding mathematical upper bound (which is typically exponential in the number of parameters). Originally *Otter* was implemented as a simplified version of *KLEE* and did not support any type of external calls or function modelling. *Otter* was latter extended with two new symbolic execution techniques [43]: shortest-distance symbolic execution (SDSE) and call-chain-backward symbolic execution (CCBSE). SDSE is a strategy that computes the shortest path from the start of a program to a specific line of code, using an inter-procedural control-flow graph. On the other hand, CCBSE starts at a line of code and symbolically executes the program backwards, trying to find a feasible path from the beginning of the program that reaches that specific line. Additionally, *Otter* also received libc support through the use of *newlib* (a C library implementation intended for use on embedded systems) and *"a partial model of POSIX system calls"* [43] with support for 85 different functions.

pysymemu *pysymemu* [44] is a symbolic execution tool for x86/amd64 used to generate inputs for code coverage. This tool's goal is to be plug-and-play, with simple usage, and using only 3 dependencies: *Capstone* [45] for binary lifting, *pyelftool* [46] to parse ELF files, and *z3* as the SMT solver. Even though it tries to be as simple as possible, this tool still partially simulates the Linux environment, modelling 31 POSIX system calls.

S^{2}E *S^{2}E* [24, 47] is a binary analysis tool for x86, x86-64, or ARM, that uses symbolic execution to analyse the properties and behaviour of software systems. *S^{2}E* is able to scale to large systems by using two key techniques: *selective symbolic execution*, a strategy that minimizes the amount of symbolically executed code during an analysis by automatically switching between symbolic and concrete execution, and *execution consistency models*, *S^{2}E* offers 6 consistency models that specify the accuracy of the analysis in terms of over/under-approximation of paths generated by the symbolic execution. This tool also represents parts of the environment in a virtual machine state that interacts with both the symbolic and concrete executions. This allows *S^{2}E* to support system calls without having to effectively execute them. Additionally, to minimize path explosion, *S^{2}E* offers a plugin containing models for common library functions with summaries for 10 standard C library functions. This tool has been used by security researchers across the world and is responsible for finding 6 new CVEs from 2015 to 2017 [48–53].

Triton *Triton* [54] is a dynamic binary analysis (DBA) framework developed by two MSc students at Bordeaux University and sponsored by the software security company Quarkslab [55]. *Triton* supports x86, x86-64, ARM32 and AArch64 architectures by converting them to a common AST format. This framework combines its concolic execution engine with a taint analysis module in a hybrid way, allowing the symbolic execution to be guided by the taint analysis and vice-versa. *Triton* uses *uClibc* [56], a small C library implementation intended for use on embedded systems for concrete calls, not supporting any type of function modelling, although it is a feature intended for future work.

Summary From this analysis, we can conclude that only three tools, *angr*, *Manticore* and *Otter*, take significant advantage of the scalability offered by symbolic summaries, implementing an already extensive list of summaries for modelling both standard library functions and system calls. The other tools that implement summaries are mostly focused on modelling very common libc functions (e.g., *strlen*), or supporting basic environment interactions through the most used system calls (e.g., *read*/*write*), mostly without much concern about the correctness of the models, but rather focusing on having a simple working environment. Despite this, almost all the teams in charge of developing the analysed symbolic execution tools intend to expand their support for libc functions by implementing appropriate models. This is, in general, a hard task that requires understanding the specific internals of each symbolic execution tool. *angr* is the only tool that comes with an infrastructure for users to write and use their own summaries, albeit with a lot of extra work. Our goal is to render the task of writing new symbolic summaries significantly easier, while at the same time streamlining summary re-use across different tools and summary validation.

3

Symbolic Reflection API

Contents

Symbolic summaries are an effective technique for improving the scalability of symbolic execution tools. The key idea is that instead of symbolically executing the concrete code of a given library function, which can lead to an intractable amount of branching, one executes a symbolic summary instead. Symbolic summaries model the behaviour of their original concrete functions while, at the same time, minimizing the amount of branching. Traditionally, symbolic summaries have been implemented in the programming language of the tool that is using them. For instance, the symbolic summaries of angr are implemented in Python as angr is itself implemented in Python. In order to allow for the reuse of symbolic summaries between different symbolic execution tools, we propose that symbolic summaries be implemented in the analysed language, in our case C. To this end, we introduce a symbolic reflection API consisting of a set of symbolic reflection primitives, which can be used to implement symbolic summaries and which symbolic execution tools need to implement natively in order to execute those summaries. Hence, instead of interpreting these primitives as standard code, symbolic execution tools must have for each primitive an internal algorithm that implements the primitive's expected behaviour by interacting with the current symbolic state. Extending the targeted tools with support for the required API is substantially simpler than designing and implementing the symbolic summaries from scratch. In fact, most of the existing symbolic execution tools for C already provide some of the functions required by our API, albeit with different names.

In this chapter we start by introducing our API for developing symbolic summaries (Section 3.1). Then, we illustrate how the proposed API can be used to develop symbolic summaries for three classes of libc functions (Section 3.2): string manipulation functions, number-parsing functions, and input/output functions. Finally, we describe how we extended the AVD and the angr symbolic execution tools for them to support the proposed API.

3.1 Symbolic Reflection API

This section introduces our symbolic reflection API. We organize the functions of the API into 3 categories: *General Functions*, *Operations with symbolic variables*, and *Operations with restrictions*. Regarding the functions for manipulation of symbolic variables, our symbolic reflection API assumes that symbolic values are internally modelled as bit vectors. This is not an unreasonable assumption since most symbolic execution tools for C represent symbolic values as bit vectors, regardless of the value type (integers, chars, etc. are all modelled as bit vectors). Below we describe the functions corresponding to each of the three categories.

General Functions The primitives in this category provide the functionality required to interact with the symbolic execution engine (Table 3.1). These primitives represent the core behaviour needed to develop symbolic summaries. In fact, all the implemented summaries use at least one of these primitives. For

instance, the primitive *summ_is_symbolic* is used for checking whether or not a given runtime value is symbolic, while the primitive *solver_is_it_possible* checks if its argument denotes a satisfiable formula in the current path condition; e.g., the call `summ_is_symbolic(&var,32)` checks if the 32 bit variable *var* is symbolic, while the call `solver_is_is_possible(restr)` checks whether the formula given as argument is possible, where *restr* is an arbitrary restriction such as $var > 0$. Additionally, this category also includes an API primitive that is specific for the validation tool. The primitive *summ_memory_addr* can be used to specify which memory addresses must be taken into account during the evaluation of a summary that interacts with memory (e.g., a summary for *memcpy*).

Table 3.1: Symbolic Reflection API: General Functions

`summ_not_implemented_error(char *fname)`	Stops the execution and returns a message to the user saying that function <`fname`> must be implemented.
`summ_print_byte(char byte)`	Prints a byte, be it symbolic or concrete.
`summ_maximize(symbolic sym_var, size_t length)`	Takes the symbolic variable <`sym_var`> with <`length`> bits and returns the maximum value that it may denote given the current path condition.
`summ_is_symbolic(symbolic sym_var, size_t length)`	Checks if variable <`sym_var`> with <`length`> bits is symbolic.
`summ_new_sym_var(int length)`	Returns a new symbolic variable denoting a value with <`length`> bits.
`_solver_is_it_possible(restr_t restr)`	Queries the solver to check if the restriction <`restr`> is satisfiable given the current path condition.
`summ_assume(restr_t restr)`	Adds the restriction <`restr`> to the current path condition of the symbolic state.
`summ_memory_addr(void* addr, void* n, size_t length)`	Marks the <`n`> + 1 consecutive memory addresses, starting from <`addr`> inclusive, to be evaluated by the summary validation tool (<`n`> can be symbolic).

Operations with symbolic variables The primitives in this category are responsible for manipulating the bit vectors denoting symbolic variables (Table 3.2). For instance, the primitive *solver_Concat* concatenates two symbolic variables, while the primitive *solver_SignExt* extends a symbolic variable with extra sign bits, which can be used for signed up-casting (e.g., int to long).

Table 3.2: Symbolic Reflection API: Operations with symbolic variables

`_solver_Concat(symbolic sym_var, symbolic sym_var2, int length1, int length2)`	Generates a new symbolic variable denoting the concatenation of the two given symbolic variables.
`_solver_Extract(symbolic sym_var, int start, int end, int length)`	Generates a new symbolic variable denoting the bits from indexes <`start`> to <`end`> (exclusive) from <`sym_var`>.
`_solver_ZeroExt(symbolic sym_var, int to_extend, int length)`	Generates a new symbolic variable denoting the extension of the sequence of bits denoted by <`sym_var`> with <`to_extend`> additional 0 bits (the bits are added to the left).
`_solver_SignExt(symbolic sym_var, int to_extend, int length)`	Generates a new symbolic variable denoting the extension of the sequence of bits denoted by <`sym_var`> with <`to_extend`> additional sign bits (the bits are added to the left).

Operations with symbolic restrictions The primitives in this category are responsible for building restrictions/constraints over symbolic variables (Table 3.3). These restrictions can then be queried for satisfability and/or added to the current symbolic state. For instance the primitive call: `_solver_EQ(&a, &b, 32)`

builds an equality restriction over two 32 bit symbolic variables such that $a = b$.

Table 3.3: Symbolic Reflection API: Operations with symbolic restrictions

`_solver_NOT(restr_t restr)`	Builds the restriction: $\neg restr$
`_solver_Or(restr_t restr1, restr_t restr2)`	Builds the restriction: $restr1 \vee restr2$
`_solver_And(restr_t restr1, restr_t restr2)`	Builds the restriction: $restr1 \wedge restr2$
`_solver_EQ(symbolic symvar, symbolic symvar2, size_t length)`	Builds the restriction: $symvar1 = symvar2$
`_solver_NEQ(symbolic symvar, symbolic symvar2, size_t length)`	Builds the restriction: $symvar1 \neq symvar2$
`_solver_LT(symbolic symvar, symbolic symvar2, size_t length)`	Builds the restriction: $symvar1 < symvar2$ (assumes the values are unsigned)
`_solver_LE(symbolic symvar, symbolic symvar2, size_t length)`	Builds the restriction: $symvar1 \leq symvar2$ (assumes the values are unsigned)
`_solver_SLE(symbolic symvar, symbolic symvar2, size_t length)`	Builds the restriction: $symvar1 < symvar2$ (assumes the values are signed)
`_solver_SLE(symbolic symvar, symbolic symvar2, size_t length)`	Builds the restriction: $symvar1 \leq symvar2$ (assumes the values are signed)
`_solver_IF(restr_t restr, symbolic symvar1, symbolic symvar2, size_t length)`	Returns a value in a similar fashion as the C ternary operator: $restr\ ?\ symvar1\ :\ symvar2$

3.2 Summary Families

In this section we illustrate how the symbolic reflection API can be used to implement symbolic summaries for three different families of libc functions: string manipulation functions, number-parsing functions, and input/output functions. The *C Standard Library* includes a large number of functions, for example the GNU C implementation (glibc), used in most Linux systems, offers over 1600 different functions [57]. This is an unmanageable number of symbolic summaries to implement and evaluate in the time frame of this thesis. Hence, we chose to focus the development of summaries on the most commonly used functions found in the CGC binaries, the main data set that we used to evaluate our approach. Importantly, the CGC binaries [58] provide a good understanding of the common libc usage, since they include a large variety of real-world like programs that must implement their own libc equivalent functions.

3.2.1 String Manipulation

Table 3.4 summarizes the summaries that we have implemented for libc string manipulation functions. For each summary we give the number of lines of the summary and the number of different symbolic primitives that it uses. Furthermore we indicate if the summary is self-contained, i.e., if it does not

makes use of auxiliary functions. Importantly, most functions are associated with several summaries, each satisfying a different property and tailored to a specific type of application. For instance, backward sound summaries are more appropriate for bug finding analyses which should have low false positive rates, while forward sound summaries are more appropriate for security analyses which should not drop execution paths. To distinguish between summary implementations for the same function, all summaries are named using a unique suffix, e.g., *strcpy1* and *strcpy2* are two summaries implemented for the *strcpy* function.

Table 3.4: String manipulation summaries

	Lines of code					API primitives					Self contained					Symbolic return				
(sufix)	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
memchr	17	33	13	86	-	2	6	5	9	-	yes	yes	yes	yes	-	no	no	yes	yes	-
memcmp	21	88	109	18	-	2	10	11	8	-	yes	yes	yes	yes	-	no	yes	yes	yes	-
memcpy	17	20	-	-	-	2	4	-	-	-	yes	yes	-	-	-	no	no	-	-	-
memmove	25	28	-	-	-	2	4	-	-	-	yes	yes	-	-	-	no	no	-	-	-
memset	15	18	-	-	-	2	4	-	-	-	yes	yes	-	-	-	no	no	-	-	-
strcat	12	28	-	-	-	1	3	-	-	-	no	no	-	-	-	no	no	-	-	-
strchr	12	35	12	19	79	1	4	5	7	7	yes	yes	yes	no	no	no	no	yes	yes	yes
strcmp	18	83	119	20	-	1	9	11	7	-	no	no	no	no	-	no	yes	yes	yes	-
strcpy	10	24	-	-	-	1	4	-	-	-	yes	yes	-	-	-	no	no	-	-	-
strlen	10	18	24	39	-	1	4	6	9	-	yes	yes	yes	yes	-	no	no	yes	yes	-
strncat	23	33	37	-	-	2	4	6	-	-	no	no	no	-	-	no	no	no	-	-
strncmp	26	110	124	317	-	2	9	12	8	-	no	no	no	no	-	no	yes	yes	yes	-
strncpy	19	33	38	-	-	2	5	6	-	-	yes	yes	yes	-	-	no	no	no	-	-
strpbrk	10	19	67	-	-	1	5	8	-	-	no	no	no	-	-	no	no	yes	-	-
strrchr	16	30	19	75	-	1	4	8	7	-	no	no	no	no	-	no	no	yes	yes	-

Below we present both a backward sound and a forward sound summary for the function strlen, respectively *strlen2* and *strlen3*. This function receives a string as an argument and returns its length. In C, strings are defined as sequences of characters (char type) terminated by a special *null* character (`'\0'`). For this reason, string manipulation functions will often lead to path explosion when called with symbolic strings. These strings may have symbolic characters, leading the symbolic execution to branch at every index for which the corresponding character may or may not be equal to the null character.

Symbolic summary *strlen2* Listing 3.1 shows the implementation of *strlen2*. This summary iterates over an input string until it finds a concrete null character. During this process, if it finds a symbolic character it tries to prove that the corresponding byte can only be a null character. If it succeeds, the summary returns the current length, otherwise it assumes that the current character is not the null character and continues iterating. In particular, if a character `s[i]` is symbolic, the API primitive *_solver_is_it_possible* queries the solver to check if that symbolic byte can only be the null character, which is translated to the query: " is it possible that `s[i]` $\neq$ `'\0'` ? ". If the answer is negative then that byte must be `'\0'`. On the other hand if the answer is positive, the restriction `s[i]` $\neq$ `'\0'` is built using the API primitive *_solver_NEQ* and added to the current path condition using the primitive *summ_assume*. The path

condition is updated to guarantee that it is consistent with the explored path, making the summary backward sound. For instance, given the symbolic string: "`sym1|a|sym2|\0` ", where `sym1` and `sym2` denote symbolic characters and the character `'|'` is used to separate the different characters occurring in the string, the summary will output the value 3 and add the restrictions: `sym1` $\neq$ `'\0'` and `sym2` $\neq$ `'\0'` to the current path condition.

```
int strlen2(char* s){
    char charZero = '\0';  int i = 0;
    while(1){
        if(summ_is_symbolic(&s[i],CHAR_SIZE)){ //s[i] is symbolic

            //Build restriction: s[i] ≠ '\0'
            restr_t restr = _solver_NEQ(&s[i], &charZero, CHAR_SIZE);

            //Check satisfiability of restr
            if(!_solver_is_it_possible(restr)) break;

            else summ_assume(restr); //Add restr to symbolic state
        }
        else if(s[i] == charZero) break;
        i++;
    }
    return i;
}
```

Listing 3.1: Implementation of summary *strlen2*

Symbolic summary *strlen3* Listing 3.2 shows the implementation of *strlen3*. This summary also iterates over the input string until it finds a concrete null character, while trying to prove that any symbolic byte found can only be a null character. However, unlike the previous summary, if that proof fails, the summary assumes right way that the return value must be greater than or equal to the length of the string analysed so far, returning a symbolic value. In particular, if the solver proves that `s[i]` $\neq$ `'\0'` is possible, a new unconstrained symbolic variable, `retval`, is created with a call to `summ_new_sym_var`. This new variable is then constrained to `retval` $\geq$ `i` using the primitive `_solver_SGE` to build the appropriate restriction. By returning a symbolic variable denoting all the possible return values for the length of the given string, the summary is forward sound. For instance, given the symbolic string: "`a|sym1|sym2|\0`", this summary will return: `retval` $\geq$ `1`.

3.2.2 Parsing of Numbers

Table 3.5 shows the summaries implemented for libc's number parsing functions. Out of these summaries, we explain in detail *atoi2*, a forward summary for libc's *atoi* function. The *atoi* function is a simple parser that converts a string representing a number into an integer value. In abstract terms, a

```
int strlen3(char* s){
    char charZero = '\0';  int i = 0;
    while(1){
        if(summ_is_symbolic(&s[i],CHAR_SIZE)){ //s[i] is symbolic

            //Build restriction: s[i] ≠ '\0'
            restr_t restr = _solver_NEQ(&s[i], &charZero, CHAR_SIZE);

            //Check satisfiability of restr
            if(!_solver_is_it_possible(restr)) break;

            //Assume retval ≥ i
            else{
                symbolic retval = summ_new_sym_var(INT_SIZE);
                restr_t r = _solver_SGE(&val, &i, INT_SIZE);
                summ_assume(r);
                return retval;
            }
        }
        else if(s[i] == charZero) break;
        i++;
    }
    return i;
}
```

Listing 3.2: Implementation of summary *strlen3*

parser is a software element that builds data structures from input data according to well defined rules. For instance, the atoi function builds an integer value from its string representation. In general, even simple parsers are challenging for symbolic execution when given symbolic inputs, as they force the symbolic execution engine to branch on every possible output that can be produced from the symbolic inputs. In the case of atoi, if a symbolic string is received as an argument, returning any single integer would result in the loss of potentially important paths.

Table 3.5: Number parsing summaries

	Lines of code			API primitives			Self contained			Symbolic return		
(sufix)	1	2	3	1	2	3	1	2	3	1	2	3
atoi	118	34	106	10	8	10	no	no	no	yes	yes	yes

Symbolic summary *atoi2* Listing 3.3 shows a shortened implementation of our summary *atoi2*. In particular it shows the part responsible for handling strings containing symbolic characters (the complete implementation can be found in Listing A.1). The key for guaranteeing that summary is forward sound is to return an interval denoting all possible integers that can be parsed from a specific symbolic string. To this end, we determine the maximum conceivable length for the input string and use it to compute the interval of possible return values. For example, considering a symbolic string: "`sym1|sym2|sym3|\0`",

our summary will return a symbolic variable *retval* such that: `retval` $\in [-99, 999]$.

```
1  int atoi2(char *str){
       ...
18         else {
19             symbolic retval = summ_new_sym_var(INT_SIZE);
20             int size = strlen1(str);
21             //Determine bounds
22             int lower_bound = pow(10,size-1) * -1;
23             int upper_bound = pow(10,size);
24
25             //Build interval with restrictions
26             restr_t val_GT_lower = _solver_SGT(&retval, &lower_bound, INT_SIZE);
27             restr_t val_LT_upper = _solver_SLT(&retval, &upper_bound, INT_SIZE);
28             restr_t bounds_restr = _solver_And(val_GT_lower,val_LT_upper);
29             summ_assume(bounds_restr); //Add restrictions to symbolic state
30             return retval;
31         }
32     }
33     return res * sign;
34 }
```

Listing 3.3: Shortened implementation of summary *atoi2*

3.2.3 Input/Output

Table 3.6 shows the summaries implemented for libc's I/O functions. To illustrate how summaries for this family of functions are implemented we present a summary for libc's *fgets* function. I/O functions commonly play an important role during symbolic execution as input functions represent a valid source of symbolic data that must be taken into account in order to maximize the soundness of an analysis. These functions are usually some of the first to be modelled by a symbolic execution tool as all I/O functions interact with, or are itself system calls, which as we have seen, cannot be analysed without some type of symbolic modelling.

Table 3.6: Input/Output summaries

	Lines of code		API primitives		Self contained		Symbolic return	
(sufix)	1	2	1	2	1	2	1	2
fgets	18	24	3	5	yes	yes	no	no
getchar	2	-	1	-	yes	-	yes	-
puthar	2	-	1	-	yes	-	no	-
puts	11	21	2	4	yes	yes	no	no

Symbolic summary *fgets1* Listing 3.4 shows the implementation of our summary *fgets1*. To simulate a call to *fgets* this summary creates a new unconstrained symbolic variable with 8 bits (one byte) for

every character “read”. When the number of characters to be read (*length*) is symbolic, the argument is concretized to its maximum possible value according to the current path condition, through a call to the primitive *summ_maximize*. This is done to prevent the path explosion that would result from branching the symbolic execution on every possible length. We choose the maximum value for the *length* argument as bugs in programs tend to appear at the extreme cases of input. Even though we generate a string with the maximum possible length, this string can actually represent the empty string, as the first symbolic byte can denote a null character.

```
char* fgets(char *str, unsigned int length, void* stream){
    int i = 0;
    //If length is symbolic maximize to a concrete length
    if(summ_is_symbolic(&length, INT_SIZE)){
        length = summ_maximize(&length, INT_SIZE);
    }
    while(length-1 > 0){
        symbolic sym_var = summ_new_sym_var(CHAR_SIZE); //Create a new symbolic variable
        str[i++] = sym_var;
        length--;
    }
    str[i] = '\0';
    return str;
}
```

Listing 3.4: Implementation of summary *fgets1*

3.3 Supporting the Symbolic Reflection API

To demonstrate the portability of our summaries, we extended two symbolic execution tools: *AVD* [16] and *angr* [15], with support for the Symbolic Reflection API. As AVD was implemented at IST, it provides a familiar platform that was easy to extend with support for the required summary API. In addition, *AVD* also serves as the test bed for developing and testing symbolic summaries, as its symbolic engine provides the foundation that our summary validation tool is built upon which we will go over in Section 4.2. On the other hand, *angr* is a tried symbolic execution tool that offers a flexible and well documented toolkit, making it a great candidate for demonstrating the portability of the Symbolic Reflection API in an unfamiliar tool.

3.3.1 Extending AVD

In *AVD* summaries are implemented as subclasses of the abstract *Summary* class. This class provides the two default methods for abstracting the processes of loading the function arguments and returning an appropriate value after the summary is executed. The behaviour of the summary itself is implemented in the *execute* method, that all summaries must override, which receives an instance of a *Memory* class for the summary to interact with. Each *Memory* instance represents the symbolic state associated with a symbolic execution tree node. The API primitives are implemented as standard summaries, added under the *Summaries* directory, and interact with the symbolic execution memory in the same manner as the native summaries. The function names (or symbols) corresponding to the symbolic summaries are listed in a global configuration file (*Config*), so that when the symbolic execution reaches one of these symbols, the execution is stopped, and the *execute* method of corresponding summary is called. To prevent the native libc summaries from being used, the standard libc symbols should also be also be added to the list of symbols to be ignored in the *Config* file. The main execution driver *Sym_exec* can then take the target binary, compiled with the libc summaries, and start the execution. Figure 3.1 shows a simplified model of *AVD*, extended to support the Symbolic Reflection API.

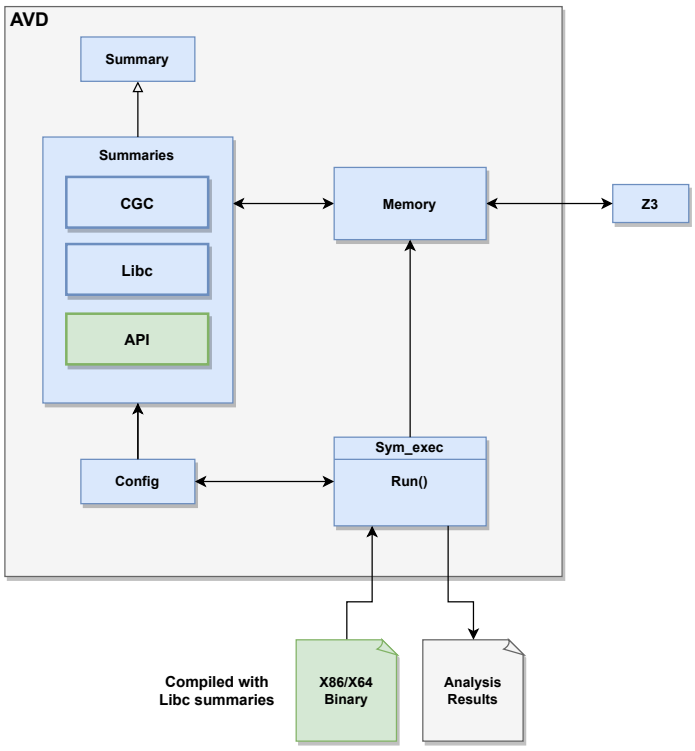


Figure 3.1: AVD extended to support the Symbolic Reflection API.

Listing 3.5 shows the API primitive *summ_is_symbolic* implemented in *AVD*. In the example we can see that *summ_is_symbolic* inherits from the *Summary* class so that it was access to the *load_args* and

ret methods. As mentioned, these are the methods that abstract the interactions with the symbolic memory for loading arguments and returning values. Importantly, this summary also makes use of the function *is_symbolic*, which is implemented by the default in *AVD*'s standard architecture, effectively trivializing this summary's implementation, which only needs to assert that *size* is a multiple of 8 bits, in line with the standard types of C.

```
class summ_is_symbolic(Summary):
    def __init__(self):
        super().__init__('summ_is_symbolic')

    def execute(self, executor, mem):
        sym_var, size = self.load_args(mem, [Pointer, uint])
        assert size % 8 == 0, "Size is in bits but must be divisible by 8"
        sym_val = mem.load(sym_var, size).val
        if is_symbolic(sym_val):
            self.ret(mem, 1)
        else:
            self.ret(mem, 0)
```

Listing 3.5: Implementation of the primitive: *summ_is_symbolic* in AVD.

3.3.2 Extending angr

angr is implemented as a Python framework, hence, to start an analysis it requires a driver program that is responsible for taking the target binary and instantiating a *Project* class, which is *angr*'s main interface. Consequently, and as recommended in the tool's documentation, a user can implement his own symbolic summaries directly in the driver script. Accordingly, we implemented our API primitives as standard user summaries, which *angr* calls *sim procedures*. Similarly to *AVD*, all *sim procedures* inherit from the abstract class *SimProcedure* that abstracts the complexity of interacting with the symbolic state (*SimState* class), for loading the procedure's arguments and returning values. A sim procedure is implemented by overriding the *run* method. However, unlike *AVD*, for each summary call a new *SimProcedure* object is created and automatically tied to an instance of a *SimState*. Every *sim procedure* is stored in a global accessed dictionary that maps a procedure class to its respective symbol or address. In the case of user defined *sim procedures*, such as our API primitives, the *Project* class offers an *hook* method that can be used to store a *sim procedure* with a corresponding symbol (or address) in the procedure's dictionary. Additionally, the *Project* class also allows to specify function symbols to be ignored, thus preventing *angr* from using its native procedures instead of the API. After this initial configuration, the *Project* class can then take the target binary compiled with the libc summaries, instantiate a initial symbolic state (*Entry_state*), and start a normal execution. Figure 3.2 shows a simplified model of *angr* and

a respective driver program with support for the Symbolic Reflection API.

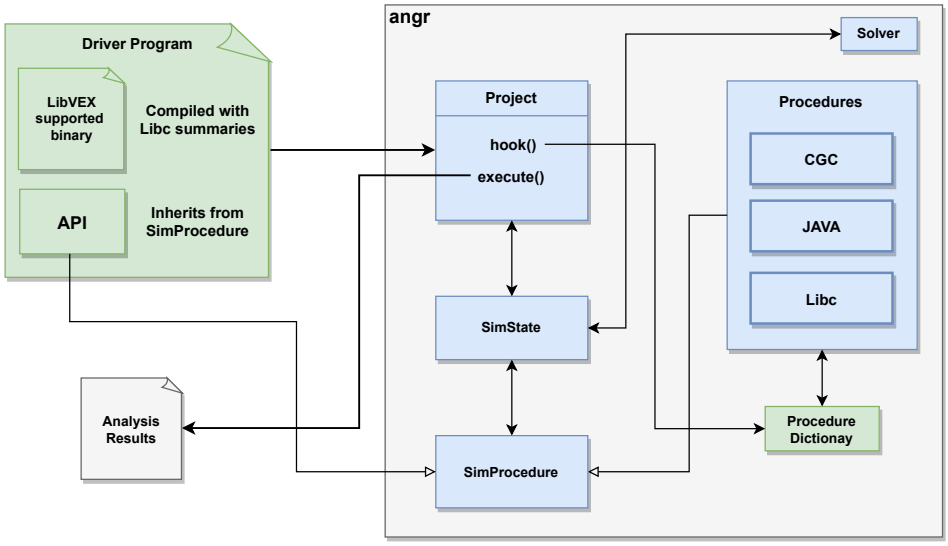


Figure 3.2: Supporting the Symbolic Reflection API in angr.

Listing 3.6 shows a concrete example of an API primitive implemented on a driver program for *angr*. In the example we can see that the primitive *summ_new_sym_var* is implemented as a *sim procedure* extending the *SimProcedure* class and exposing the methods: *run* and *ret*. The *run* method contains the implementation of the primitive and receives its arguments as input, while the *ret* must be used to return a value from the primitive. In addition, the *SimProcedure* class also enables the procedure to manipulate the underlying symbolic state directly, as one of the *SimProcedure*'s properties (*self.state*) is an instance of a *SimState*. The implementation of *summ_is_symbolic* in *AVD* is also greatly simplified through the use a standard function in *angr*'s architecture, namely the *BVS* method, which creates a new symbolic variable of arbitrary size with a unique identifier.

In general, extending angr and AVD with support for our API was almost straightforward as most of the API primitives have a corresponding mechanism in each of the tools that can effectively trivialize the primitive's implementation.

3.3.3 Modelling Symbolic Restrictions in C Code

One technical challenge that arises from implementing symbolic summaries in the target language is modelling restrictions over symbolic values in C code. Different symbolic execution tools have different internal representations for symbolic restrictions. Hence, we chose to represent restrictions through unique number identifiers in the summaries' code. To this end, the restriction type in C (*restr_t*) is defined as an unsigned long. By having each restriction associated with a unique identifier, symbolic execution tools can implement an internal dictionary that translates a number id to the corresponding internal

```python
from angr import Project, SimProcedure
p = Project("binary")

# API primitive
class summ_new_sym_var(SimProcedure):
    def run(self, length):
        length = self.state.solver.eval(length)
        assert length % 8 == 0, "Size is in bits but must be divisible by 8"
        sym_var = self.state.solver.BVS("sym_var", length)
        self.ret(sym_var)

p.hook_symbol('summ_new_sym_var', summ_new_sym_var())
state = p.factory.entry_state()
sm = p.factory.simulation_manager(state)
sm.run()
```

Listing 3.6: Implementation of the primitive: *summ_new_sym_var* in angr's driver program.

restriction object. Figure 3.3 illustrates the pipeline needed for the manipulation of symbolic restrictions in the C code. As shown in the example, when the execution reaches the primitive *_solver_EQ*, the symbolic execution tool creates a new entry in the restrictions dictionary, associating the corresponding restriction object with a unique identifier value (1). In this example the id is stored in the variable *r*, which is then passed as an argument to the primitive *_solver_is_it_possible* to check the satisfability of the restriction. To execute this call, the symbolic execution tool obtains the restriction object corresponding to the id = 1 from the dictionary, and passes it to the SMT solver along with other restrictions in the current path condition.

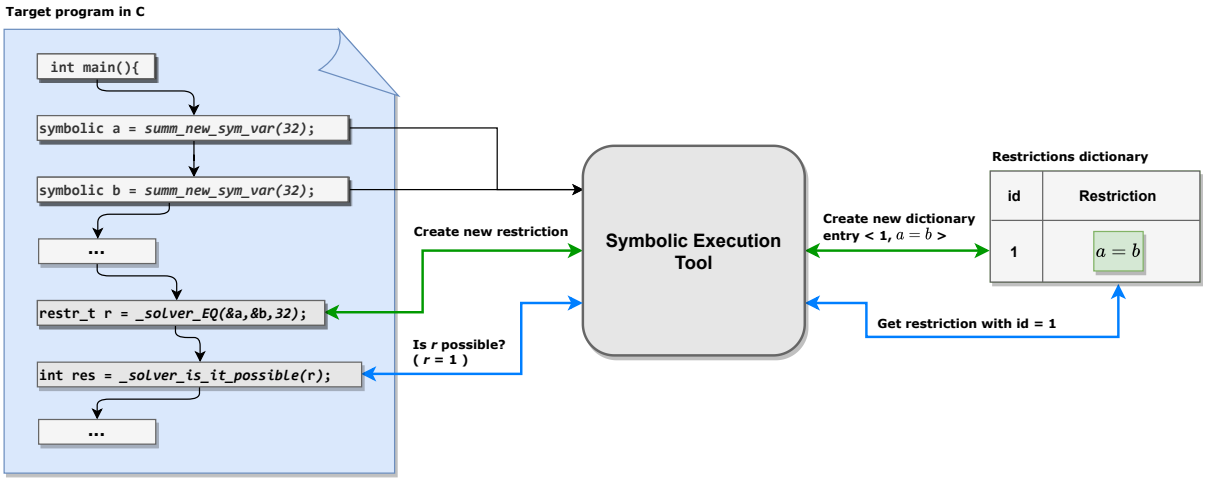


Figure 3.3: Modelling symbolic restrictions in C code

4

Summary Correctness

Contents

In this chapter we start by introducing our proposed correctness properties for evaluating symbolic summaries (Section 4.1). Then, we present our summary validation tool and illustrate how it can be used to systematically validate the correctness of a summary according to the established correctness properties (Section 4.2). Finally, we describe how we implemented the summary validation tool on top of *AVD*'s symbolic execution engine.

4.1 Summary Properties

In this section, we mathematically define the correctness properties required for evaluating our symbolic summaries, some of which were introduced in Section 2.2.2. We start by introducing some notation. We use σ and $\hat{\sigma}$ to denote concrete and symbolic program states, respectively. Program states are composed of program memories; we use μ and $\hat{\mu}$ to range over concrete memories and symbolic memories, respectively. In the following, we assume that concrete states exactly coincide with concrete memories, while symbolic states are composed of a concrete memory and a path condition. Put formally: $\sigma = \langle\mu\rangle$ and $\hat{\sigma} = \langle\hat{\mu}, \pi\rangle$.

We use C and $\hat{C}$ to denote a concrete implementation of a function and its corresponding summaries, respectively. In contrast to concrete implementations, which can be executed both concretely and symbolically, symbolic summaries can only be executed symbolically. In the following, we use:

- $C(\sigma)$ to denote the concrete execution of C on the concrete state σ;
- $C(\hat{\sigma})$ to denote the symbolic execution of C on the symbolic state $\hat{\sigma}$;
- $\hat{C}(\hat{\sigma})$ to denote the symbolic execution of $\hat{C}$ on the symbolic state $\hat{\sigma}$.

While the concrete execution of a program C on a state σ yields a pair consisting of a concrete state σ' and a return value r, the symbolic execution of a program or a summary on a symbolic state $\hat{\sigma}$ yields a set of pairs, each consisting of a symbolic state and a symbolic return value. Put formally: $C(\sigma) = (\sigma', r)$ and $\hat{C}(\hat{\sigma}) = \{(\hat{\sigma}_1, \hat{r}_1), ..., (\hat{\sigma}_n, \hat{r}_n)\}$. To simplify notation, we use: $\hat{C}(\hat{\sigma}) \rightsquigarrow (\hat{\sigma}', \hat{r})$ to mean that $(\hat{\sigma}', \hat{r}) \in \hat{C}(\hat{\sigma})$; informally, this means that the symbolic state $\hat{\sigma}'$ and return value $\hat{r}$ are contained in the set of outcomes resulting from the symbolic execution of $\hat{C}$ on the symbolic state $\hat{\sigma}$;

In the following, we use $\hat{\mathcal{V}}$ to denote a set of symbolic values. Furthermore, we use $\hat{\Sigma}$ to denote any set of pairs of symbolic states and symbolic return values.

Finally, we write $\sigma \in [\![\hat{\sigma}]\!]$ to mean that the concrete state σ is in the interpretation of the symbolic state $\hat{\sigma}$. The interpretation of a symbolic state $\hat{\sigma}$ is the set of concrete states that can be obtained from $\hat{\sigma}$ by mapping the symbolic variables of $\hat{\sigma}$ to concrete values in a way that is consistent with its path condition. For instance, if $\hat{\sigma} = \langle\hat{\mu}, \hat{x} \neq 0\rangle$, then the symbolic variable $\hat{x}$ cannot be replaced by 0 in $\hat{\mu}$. Accordingly,

the interpretation function $[\![.]\!] :: \mathcal{S}ym\mathcal{S}t \to \mathcal{P}(\mathcal{C}onc\mathcal{S}t)$ takes as input a symbolic state and returns a set of concrete states.

We define the interpretation function for symbolic states with the help of an interpretation function for symbolic memories. In contrast to states, the interpretation function for symbolic memories additionally requires a *valuation function*, $\varepsilon : \hat{\mathcal{V}} \rightharpoonup \mathcal{V}$, that maps symbolic values to concrete values. We write $[\![\hat{\mu}]\!]_\varepsilon = \mu$ to mean that the interpretation of $\hat{\mu}$ under ε yields the concrete memory μ. We assume that both concrete and symbolic memories are sequences of memory cells containing byte values, with concrete memories only containing concrete values, and symbolic memories containing both symbolic and concrete values. Formally, we define a concrete memory $\mu : \mathcal{N} \rightharpoonup \mathcal{V}$ to be a partial function from the set of natural numbers, $\mathcal{N}$, to the set of concrete values $\mathcal{V}$. Analogously, we define a symbolic memory $\hat{\mu} : \mathcal{N} \rightharpoonup \hat{\mathcal{V}}$ to be a partial function from the set of natural numbers, $\mathcal{N}$, to the set of symbolic values $\hat{\mathcal{V}}$. For clarity, we assume that the set of symbolic values $\hat{\mathcal{V}}$ contains the set of concrete values $\mathcal{V}$. We interpret symbolic memories point-wise, applying the valuation function to each memory cell; put formally: $[\![\hat{\mu}]\!]_\varepsilon(i) \equiv \varepsilon(\hat{\mu}(i))$. Using memory interpretation, the definition of state interpretation is straightforward:

$$[\![\hat{\sigma}]\!] = [\![\langle \hat{\mu}, \pi \rangle]\!] = \{\, [\![\hat{\mu}]\!]_\varepsilon \mid \varepsilon(\pi) = true \} \tag{4.1}$$

We lift symbolic state interpretation to pairs of symbolic states and return values in the natural way:

$$[\![(\hat{\sigma}, \hat{r})]\!] = [\![\langle \hat{\mu}, \pi, \hat{r} \rangle]\!] = \{\, ([\![\hat{\mu}]\!]_\varepsilon,\ \varepsilon(\hat{r})) \mid \varepsilon(\pi) = true \} \tag{4.2}$$

Symbolic states as boolean formulas In order to reason about the correctness properties of a summary, we introduce a lifting operator $\lceil . \rceil :: \mathcal{P}(\mathcal{S}ym\mathcal{S}t) \to \mathcal{F}ormula$ that transforms a set of symbolic states paired up with return values into a boolean formula: $\lceil \hat{\Sigma} \rceil = \varphi_s$, where we use φ to range over the set of boolean formulas. The lifting operator for symbolic states is formally defined as follows:

$$\lceil \hat{\Sigma} \rceil \equiv \bigvee \left\{ \lceil \hat{\mu} \rceil_m \wedge \pi \wedge (\texttt{ret} = \hat{r}) \mid (\langle \hat{\mu}, \pi \rangle, \hat{r}) \in \hat{\Sigma} \right\} \tag{4.3}$$

Essentially, a set of symbolic states is transformed into a disjunction of boolean formulas, each describing its corresponding symbolic state. The formula created for each state has three components: (1) a memory component $\lceil \hat{\mu} \rceil_m$ describing the content of the symbolic memory, (2) a path condition component π, and (3) a return component $\texttt{ret} = \hat{r}$ describing the return value of the function in the execution path that led to the given state. We use a dedicated variable `ret` to refer to the return value of a function.

Analogously to symbolic state interpretation, the lifting operator for symbolic states is defined with the help of a lifting operator for symbolic memories. More concretely, to transform a symbolic state $\langle \hat{\mu}, \pi \rangle$ into a boolean formula, we also need to transform the symbolic memory $\hat{\mu}$ into a formula. Hence, we define an additional lifting operator $\lceil . \rceil_m :: \mathcal{S}ym\mathcal{M}em \to \mathcal{F}ormula$ that takes a symbolic memory $\hat{\mu}$ and

returns a boolean formula $\lceil\hat{\mu}\rceil_m = \varphi_m$ that describes the contents of the symbolic memory $\hat{\mu}$. The lifting operator for memories is formally defined as follows:

$$\lceil\hat{\mu}\rceil_m \equiv \bigwedge\{\, \mathtt{addr}_\alpha = \hat{\mu}(\alpha) \mid \alpha \in \mathsf{dom}(\hat{\mu})\} \tag{4.4}$$

Essentially, a symbolic memory is transformed into a conjunction of boolean formulas, each describing a single memory cell. To this end, we introduce, for each memory cell α, a symbolic variable $\mathtt{addr}_\alpha$ that denotes its contents.

Naturally, the proposed method for lifting a symbolic memories into formulas may potentially generate extremely large formulas that describe the entire contents of the memory regardless of whether or not those contents are relevant to the behaviour of the function being symbolically executed. Hence, we redefine the lifting operator for memories in order for it to take into account the concrete function or symbolic summary being executed. The new lifting operator $\lceil\hat{\mu}\rceil^f_m$ denotes the boolean formula associated with the memory $\hat{\mu}$ when symbolically executing the function with identifier f:

$$\lceil\hat{\mu}\rceil^f_m \equiv \bigwedge\{\, \mathtt{addr}_\alpha = \hat{\mu}(\alpha) \mid \alpha \in \mathsf{dom}_f(\hat{\mu})\} \tag{4.5}$$

where $\mathsf{dom}_f(\hat{\mu})$ denotes the set of addresses in $\hat{\mu}$ that are relevant for the execution of the function with identifier f.

As seen before, the symbolic execution of either a program or a symbolic summary on a symbolic state may result in a set of symbolic states paired up with return values, $\hat{\Sigma}$, through the application of the lifting operator for states. This set can be, in turn, transformed into a disjunction of boolean formulas of the form $\lceil\hat{\mu}\rceil_m \wedge \pi \wedge (\mathtt{ret} = \hat{r})$. In the following, we will use Φ to range over such disjunctions, writing:

$$\Phi \equiv \begin{cases} \varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_k \\ \varphi_1 \equiv \lceil\hat{\mu}_1\rceil_m \wedge \pi_1 \wedge (\mathtt{ret} = \hat{r}_1) \\ \varphi_2 \equiv \lceil\hat{\mu}_2\rceil_m \wedge \pi_2 \wedge (\mathtt{ret} = \hat{r}_2) \\ \quad\vdots \\ \varphi_k \equiv \lceil\hat{\mu}_k\rceil_m \wedge \pi_k \wedge (\mathtt{ret} = \hat{r}_k) \end{cases}$$

to mean that $\Phi = \varphi_1 \vee \varphi_2 \vee ... \vee \varphi_k$, where each conjunct $\varphi_i \equiv \lceil\hat{\mu}_i\rceil_m \wedge \pi_i \wedge (\mathtt{ret} = \hat{r}_i)$ for $i = 1, ..., k$. Finally, we write $C(\hat{\sigma}) = \Phi$ to mean that there is a set $\hat{\Sigma}$ such that $C(\hat{\sigma}) = \hat{\Sigma}$ and $\lceil\hat{\Sigma}\rceil = \Phi$. Analogously, we write $\hat{C}(\hat{\sigma}) = \hat{\Phi}$ to mean that there is a set $\hat{\Sigma}$ such that $\hat{C}(\hat{\sigma}) = \hat{\Sigma}$ and $\lceil\hat{\Sigma}\rceil = \hat{\Phi}$. For clarity, we use Φ and $\hat{\Phi}$ to represent the boolean formulas that result from symbolically executing a concrete function C and a corresponding summary $\hat{C}$, respectively, in the same symbolic state $\hat{\sigma}$.

4.1.1 Backward Soundness

A symbolic summary $\hat{C}$ is *backward sound* with respect to a concrete implementation C, if and only if it holds that:

$$\forall \hat{\sigma} \,.\, \hat{C}(\hat{\sigma}) \rightsquigarrow (\hat{\sigma}', \hat{r}) \implies \forall (\sigma', r) \in [\![(\hat{\sigma}', \hat{r})]\!] \,.\, \exists\, \sigma \in [\![\hat{\sigma}]\!] \,.\, C(\sigma) = (\sigma', r) \tag{4.6}$$

As illustrated in Figure 4.1, a *backward sound* summary guarantees that, for any symbolic state $\hat{\sigma}$, the interpretation of the set of symbolic execution paths generated by the symbolic execution of $\hat{C}$ in $\hat{\sigma}$ is contained in the set of execution paths produced by the concrete execution of C in the interpretation of $\hat{\sigma}$. In other words, all concretizations of the symbolic execution paths generated by the summary must correspond to execution paths generated by the concrete execution of the original function. Let Φ and $\hat{\Phi}$ be the boolean formulas that result from symbolically executing C and $\hat{C}$, respectively, in a symbolic state $\hat{\sigma}$. For the summary $\hat{C}$ to be *backward sound*, the implication $\hat{\Phi} \Rightarrow \Phi$ must be true. For instance, if $\Phi \equiv \varphi_1 \vee \varphi_2 \vee \varphi_3$ and $\hat{\Phi} \equiv \varphi_1$, the summary is *backward sound*.

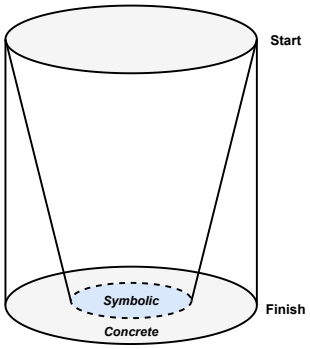


Figure 4.1: Backward Soundness

4.1.2 Forward Soundness

A symbolic summary $\hat{C}$ is *forward sound* with respect to a concrete implementation C, if and only if it holds that:

$$\forall \hat{\sigma}.\, \forall \sigma \in [\![\hat{\sigma}]\!] \;\wedge\; C(\sigma) = (\sigma', r) \;\wedge\; \hat{C}(\hat{\sigma}) \rightsquigarrow (\hat{\sigma}', \hat{r}) \Longrightarrow (\sigma', r) \in [\![(\hat{\sigma}', \hat{r})]\!] \tag{4.7}$$

As illustrated in Figure 4.2, a *forward sound* summary guarantees that, for any symbolic state $\hat{\sigma}$, the set of execution paths produced by the concrete execution of C in the interpretation of $\hat{\sigma}$, is contained in the interpretation of the symbolic execution paths generated by the symbolic execution of $\hat{C}$ in $\hat{\sigma}$. In other words, all the execution paths generated by the concrete execution of the original function correspond to concretizations of the symbolic execution paths generated by the summary. Let Φ and $\hat{\Phi}$ be the boolean formulas that result from symbolically executing C and $\hat{C}$, respectively, in a symbolic state $\hat{\sigma}$. For the summary $\hat{C}$ to be *forward sound*, the implication $\Phi \Rightarrow \hat{\Phi}$ must be true. For instance, if $\Phi \equiv \varphi_1$ and $\hat{\Phi} \equiv \varphi_1 \vee \varphi_2 \vee \varphi_3$, the summary is *forward sound*.

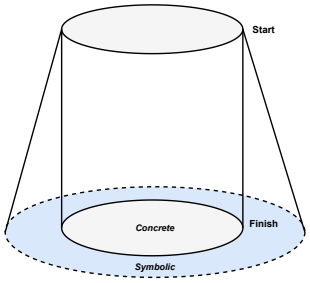


Figure 4.2: Forward Soundness

4.1.3 Completeness

A symbolic summary $\hat{C}$ is *complete* with respect to a concrete implementation C, if and only if it satisfies both (4.7) and (4.6). As illustrated in Figure 4.3, a *complete* summary guarantees that, for any symbolic state $\hat{\sigma}$, the interpretation of the set of symbolic execution paths generated by the symbolic execution of $\hat{C}$ in $\hat{\sigma}$ corresponds to the set of execution paths produced by the concrete execution of C in the interpretation of $\hat{\sigma}$. In other words, this property guarantees that all concretizations of the symbolic execution paths generated by the summary correspond to all, and only to, execution paths generated by the concrete execution of the original function. Let Φ and $\hat{\Phi}$ be the boolean formulas that result from symbolically executing C and $\hat{C}$, respectively, in a symbolic state $\hat{\sigma}$. For the summary $\hat{C}$ to be *complete*, the equivalence $\hat{\Phi} \Leftrightarrow \Phi$ must be true.

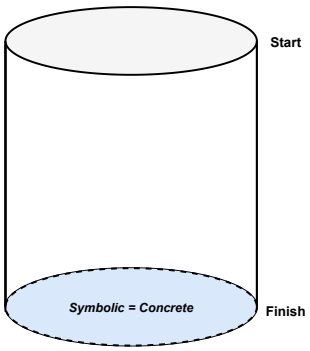


Figure 4.3: Completeness

4.1.4 Generalized Properties

In practice, it is common to have summaries that are neither *backward* nor *forward sound*. This can happen due to the nature of a target function forcing the corresponding symbolic summary to be overly complex in order to satisfy one of the foregoing soundness properties. In such cases, the benefits provided by these properties may be overshadowed by other factors, such as poor time performance and poor maintainability. If this is the case, one can often obtain either soundness or precision if one assumes that the function input satisfies some additional constraints. For instance, it is much easier to write a summary for the *strcmp* function if one assumes that the given strings do not contain the null character in intermediate indexes. To model this type of assumption, we introduce generalized versions of the foregoing soundness properties that allow us to account for additional constraints on the function inputs. In the following, we will assume that the additional constraints on a function's input are expressed as a formula ρ and we write $\sigma \wedge \rho$ to mean $\langle \hat{\mu}, \pi \wedge \rho \rangle$.

4.1.4.A Generalized Backward Soundness

A symbolic summary $\hat{C}$ is considered *generalized backward sound* with respect to a concrete implementation C and a predicate ρ, if and only if it holds:

$$\forall \hat{\sigma} \,.\, \hat{C}(\hat{\sigma} \wedge \rho) \rightsquigarrow (\hat{\sigma}', \hat{r}) \implies \forall (\sigma', r) \in [\![(\hat{\sigma}', \hat{r})]\!] \,,\, \exists\, \sigma \in [\![\hat{\sigma} \wedge \rho]\!] \,.\, C(\sigma) = (\sigma', r) \tag{4.8}$$

As illustrated in Figure 4.4, a summary satisfies this property if the interpretation of the symbolic execution paths generated by the symbolic execution of $\hat{C}$ in $\hat{\sigma} \wedge \rho$ is contained in the set of execution paths produced by the concrete execution of C on the interpretation of $\hat{\sigma}$ filtered by ρ, meaning that we only consider the concrete states in the interpretation of $\hat{\sigma}$ that satisfy the predicate ρ. In other words, the concretizations of the symbolic execution paths generated by the summary must correspond to the execution paths generated by the concrete execution of the original function on states that satisfy the predicate ρ. Let Φ and $\hat{\Phi}$ be the boolean formulas that result from symbolically executing C and $\hat{C}$, respectively, in a symbolic state $\hat{\sigma} \wedge \rho$. For the summary $\hat{C}$ to be *generalized backward sound* with respect to ρ, the implication $\hat{\Phi} \Rightarrow \Phi$ must be true.

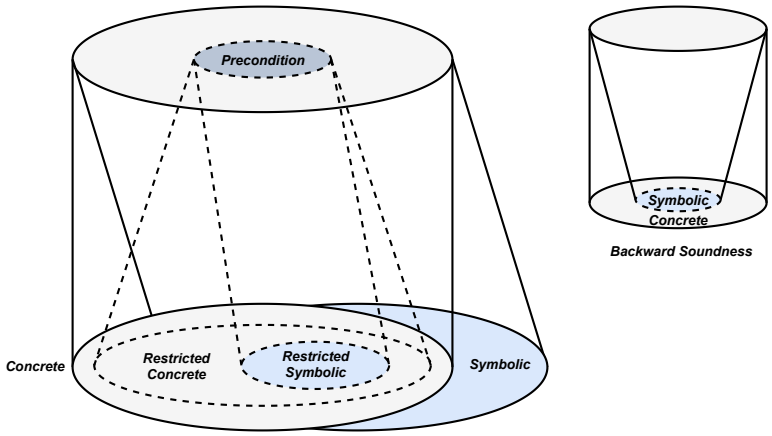


Figure 4.4: Generalized Backward Soundness

To illustrate this property, we can consider a concrete function C and a corresponding symbolic summary $\hat{C}$, that when symbolically executed in an unconstrained symbolic state $\hat{\sigma}$, produce the following boolean formulas Φ and $\hat{\Phi}$:

$$\Phi \equiv \begin{cases} \varphi_1 \vee \varphi_2 \vee \varphi_3 \\ \varphi_1 \equiv \pi_1 \wedge (\texttt{ret} = \hat{r}_1) \\ \varphi_2 \equiv \pi_2 \wedge (\texttt{ret} = \hat{r}_2) \\ \varphi_3 \equiv \pi_3 \wedge (\texttt{ret} = \hat{r}_3) \end{cases} \quad \text{and} \quad \hat{\Phi} \equiv \begin{cases} \hat{\varphi}_1 \vee \hat{\varphi}_2 \\ \hat{\varphi}_1 \equiv \pi_1 \wedge (\texttt{ret} = \hat{r}_1) \\ \hat{\varphi}_2 \equiv \pi_2 \end{cases}$$

In this example, the symbolic summary $\hat{C}$ does not satisfy the standard soundness properties. Not only there are paths of Φ missing, meaning that $\Phi \Rightarrow \hat{\Phi}$ is false, but additionally the path $\hat{\varphi}_2$ in $\hat{\Phi}$ does not imply φ_2 in Φ given that it does not constrain the return value, and so $\hat{\Phi} \Rightarrow \Phi$ is also false. To address this particular case, we can consider a predicate $\rho \equiv \neg\, \pi_2$. With the initial state restricted by ρ, the symbolic

execution of C and $\hat{C}$ in $\hat{\sigma} \wedge \rho$ produce the following formulas:

$$\Phi \equiv \begin{cases} \varphi_1 \vee \varphi_3 \\ \varphi_1 \equiv (\pi_1 \wedge \neg\, \pi_2) \wedge (\texttt{ret} = \hat{r}_1) \\ \varphi_3 \equiv (\pi_3 \wedge \neg\, \pi_2) \wedge (\texttt{ret} = \hat{r}_3) \end{cases} \quad \text{and} \quad \hat{\Phi} \equiv (\pi_1 \wedge \neg\, \pi_2) \wedge (\texttt{ret} = \hat{r}_1)$$

As a consequence, all execution paths of $\hat{\Phi}$ are now contained in Φ, making the implication $\hat{\Phi} \Rightarrow \Phi$ be true.

4.1.4.B Generalized Forward Soundness

A symbolic summary $\hat{C}$ is considered *generalized forward sound* with respect to a concrete implementation C and a predicate ρ, if and only if it holds:

$$\forall \hat{\sigma}.\ \forall \sigma \in [\![\hat{\sigma} \wedge \rho]\!] \ \wedge\ C(\sigma) \rightsquigarrow (\sigma', r) \ \wedge\ \hat{C}(\hat{\sigma} \wedge \rho) = (\hat{\sigma}', \hat{r}) \Longrightarrow (\sigma', r) \in [\![(\hat{\sigma}', \hat{r})]\!] \tag{4.9}$$

As illustrated in Figure 4.5, a summary satisfies this property if the set of execution paths produced by the concrete execution of C on the interpretation of $\hat{\sigma}$ filtered by ρ, is contained in the interpretation of the symbolic execution paths generated by the symbolic execution of $\hat{C}$ in $\hat{\sigma} \wedge \rho$. In other words, all the execution paths generated by the concrete execution of the original function on states that satisfy the predicate ρ correspond to concretizations of the symbolic execution paths generated by the summary. Let Φ and $\hat{\Phi}$ be the boolean formulas that result from symbolically executing C and $\hat{C}$, respectively, in a symbolic state $\hat{\sigma} \wedge \rho$. For the summary $\hat{C}$ to be *generalized backward sound*, the implication $\Phi \Rightarrow \hat{\Phi}$ must be true.

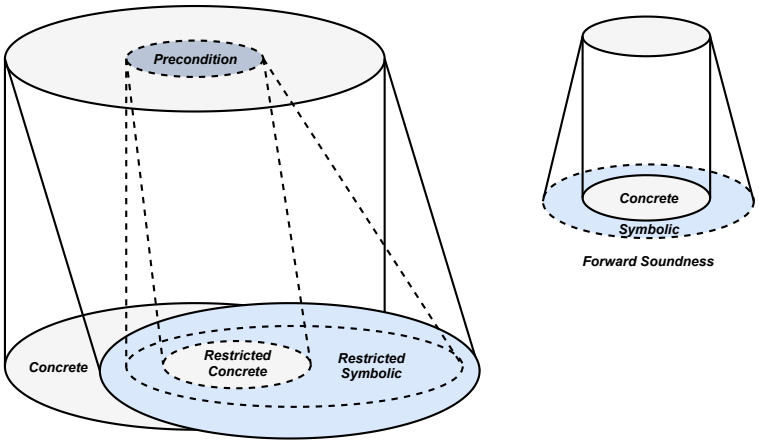


Figure 4.5: Generalized Forward Soundness

To illustrate this property, we can consider a concrete function C and a corresponding symbolic summary $\hat{C}$, that when symbolically executed in an unconstrained symbolic state $\hat{\sigma}$ produce the following

boolean formulas Φ and $\hat{\Phi}$:

$$\Phi \equiv \begin{cases} \varphi_1 \vee \varphi_2 \vee \varphi_3 \\ \varphi_1 \equiv \pi_1 \wedge (\texttt{ret} = \hat{r}_1) \\ \varphi_2 \equiv \pi_2 \wedge (\texttt{ret} = \hat{r}_2) \\ \varphi_3 \equiv \pi_3 \wedge (\texttt{ret} = \hat{r}_3) \end{cases} \quad \text{and} \quad \hat{\Phi} \equiv \begin{cases} \hat{\varphi}_1 \vee \hat{\varphi}_2 \\ \hat{\varphi}_1 \equiv \pi_1 \wedge (\texttt{ret} = \hat{r}_1) \\ \hat{\varphi}_2 \equiv (\texttt{ret} = \hat{r}_2) \end{cases}$$

In this example the symbolic summary $\hat{C}$ does not satisfy any soundness property. Once again there are paths of Φ missing, meaning that $\Phi \Rightarrow \hat{\Phi}$ is false, and the path $\hat{\varphi}_2$ in $\hat{\Phi}$ does not imply φ_2 in Φ given that it does not constrain the input, and so $\hat{\Phi} \Rightarrow \Phi$ is also false. To address this particular case, we can consider a predicate $\rho \equiv \pi_1$. With the initial state restricted by ρ the symbolic execution of C and $\hat{C}$ in $\hat{\sigma} \wedge \rho$ produces the following formulas:

$$\Phi \equiv \pi_1 \wedge (\texttt{ret} = \hat{r}_1) \quad \text{and} \quad \hat{\Phi} \equiv \begin{cases} \hat{\varphi}_1 \vee \hat{\varphi}_2 \\ \hat{\varphi}_1 \equiv \pi_1 \wedge (\texttt{ret} = \hat{r}_1) \\ \hat{\varphi}_2 \equiv \pi_1 \wedge (\texttt{ret} = \hat{r}_2) \end{cases}$$

As a consequence, all execution paths of Φ are contained in $\hat{\Phi}$, making the implication $\Phi \Rightarrow \hat{\Phi}$ be true.

4.1.4.C Generalized Completeness

A symbolic summary $\hat{C}$ is *generalized complete* with respect to a concrete implementation C and predicate ρ, if and only if it satisfies both (4.9) and (4.8). As illustrated in Figure 4.6, a summary satisfies this property if the interpretation of the symbolic execution paths generated by the symbolic execution of $\hat{C}$ in $\hat{\sigma} \wedge \rho$ corresponds to the set of execution paths produced by the concrete execution of C in the interpretation of $\hat{\sigma}$ filtered by ρ. In other words, this property guarantees that the concretizations of the symbolic execution paths generated by the summary correspond to all, and only to, execution paths generated by the concrete execution of the original function on states that satisfy the predicate ρ. Let Φ and $\hat{\Phi}$ be the boolean formulas that result from symbolically executing C and $\hat{C}$, respectively, in a symbolic state $\hat{\sigma}$. For the summary $\hat{C}$ to be *generalized complete*, the equivalence $\hat{\Phi} \Leftrightarrow \Phi$ must be true.

To illustrate this property, we can consider a concrete function C and a corresponding symbolic summary $\hat{C}$ that when symbolically executed in an unconstrained symbolic state $\hat{\sigma}$ produce the following

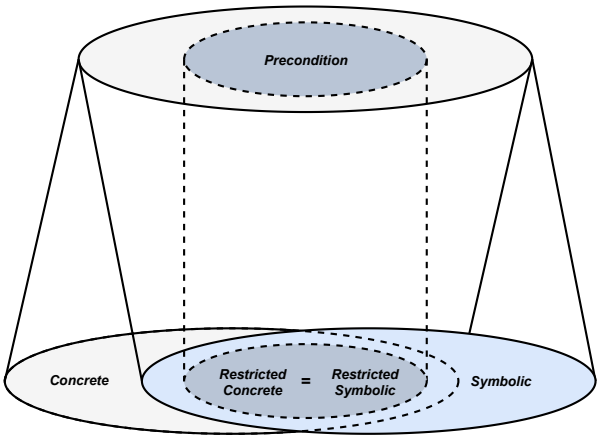


Figure 4.6: Generalized Completeness

boolean formulas Φ and $\hat{\Phi}$:

$$\Phi \equiv \begin{cases} \varphi_1 \vee \varphi_2 \vee \varphi_3 \\ \varphi_1 \equiv \pi_1 \wedge (\texttt{ret} = \hat{r}_1) \\ \varphi_2 \equiv \pi_2 \wedge (\texttt{ret} = \hat{r}_2) \\ \varphi_3 \equiv \pi_3 \wedge (\texttt{ret} = \hat{r}_3) \end{cases} \quad \text{and} \quad \hat{\Phi} \equiv \begin{cases} \hat{\varphi}_1 \vee \hat{\varphi}_2 \vee \hat{\varphi}_3 \\ \hat{\varphi}_1 \equiv \pi_1 \wedge (\texttt{ret} = \hat{r}_1) \\ \hat{\varphi}_2 \equiv \pi_2 \wedge (\texttt{ret} = \hat{r}_2) \\ \hat{\varphi}_3 \equiv \pi_3 \end{cases}$$

In this example the symbolic summary $\hat{C}$ does not satisfy any soundness property due to the execution path $\hat{\varphi}_3$. This path does not imply φ_3 in Φ since it does not constrain the return value, meaning that both implications $\Phi \Rightarrow \hat{\Phi}$ and $\hat{\Phi} \Rightarrow \Phi$ are false. To address this particular case, we can consider a predicate $\rho \equiv \pi_1 \vee \pi_2$. With the initial state restricted by ρ the symbolic execution of C and $\hat{C}$ in $\hat{\sigma} \wedge \rho$ produces the following formulas:

$$\Phi \equiv \begin{cases} \varphi_1 \vee \varphi_2 \\ \varphi_1 \equiv \pi_1 \wedge (\texttt{ret} = \hat{r}_1) \\ \varphi_2 \equiv \pi_2 \wedge (\texttt{ret} = \hat{r}_2) \end{cases} \quad \text{and} \quad \hat{\Phi} \equiv \begin{cases} \hat{\varphi}_1 \vee \hat{\varphi}_2 \\ \hat{\varphi}_1 \equiv \pi_1 \wedge (\texttt{ret} = \hat{r}_1) \\ \hat{\varphi}_2 \equiv \pi_2 \wedge (\texttt{ret} = \hat{r}_2) \end{cases}$$

As a consequence, Φ and $\hat{\Phi}$ denote the same execution paths, making the equivalence $\hat{\Phi} \Leftrightarrow \Phi$ be true.

4.2 Summary Validation Tool

The purpose of symbolic execution is to allow for the exploration and analysis of all possible execution paths in a target program. As a result, this technique will often generate many more paths than what might be expected. For this reason, even for seemingly simple symbolic summaries, the tasks of summary debugging and correctness verification, quickly become infeasible to be carried out manually. Given that our symbolic summaries are implemented in the programming language to be analysed (C), they can themselves be symbolically executed as standard C code. To this end, we implemented an auxiliary infrastructure, illustrated in Figure 4.7, that works on top of the symbolic execution tool *AVD*, to systematically evaluate a summary according to the preceding correctness properties.

This validation tool achieves its goal in two main steps. First, given a binary containing the implementation code for both a target concrete function and the corresponding symbolic summary, *AVD* will compute the boolean formulas Φ and $\hat{\Phi}$ with the selected symbolic input. These formulas are in turn passed to an SMT solver that verifies the satisfiability of the logical implication corresponding to the correctness property one is aiming at; to ensure forward soundness one must check that $\Phi \Rightarrow \hat{\Phi}$ and to ensure backward soundness one must check that $\hat{\Phi} \Rightarrow \Phi$. However, due to the complex nature of symbolic execution and summary debugging, it is often the case that simply knowing whether or not a symbolic summary satisfies a given correctness property is not enough. As a result, the validation tool is also required to generate counterexamples illustrating why a correctness property could not be satisfied. In the case of the *forward soundness* property, a user can even use the validation tool to generate examples for all execution paths that do not satisfy that property.

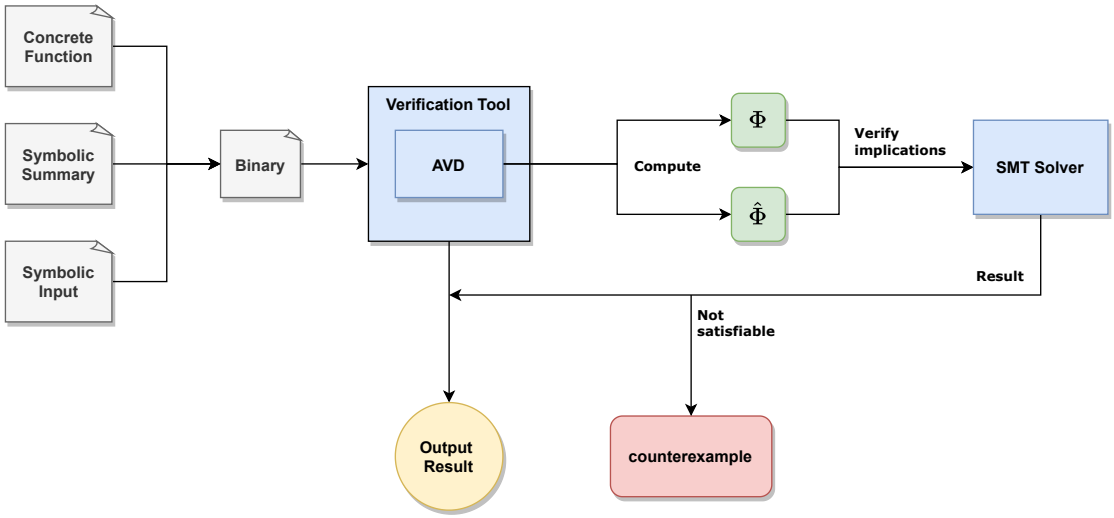


Figure 4.7: Summary Validation Tool

4.2.1 Examples

In this section we will go over the correctness properties of some the summaries we implemented in order to demonstrate how they are tested by our summary validation tool.

4.2.1.A Function - *strlen*

Consider the libc function *strlen* (`size_t strlen(const char *s)`), that given a string as argument, counts the number of characters until a null byte `'\0'` is found. To test the correctness of our summaries, first, we must determine the boolean formula Φ that results from the symbolic execution of *strlen*'s concrete implementation in an input symbolic state $\hat{\sigma}$. In order to test our summaries we have to pick a maximum length for the strings given as input. For presentation purposes, we pick length 3. Hence, we choose $\hat{\sigma} = \langle \hat{\mu}, true \rangle$, such that $s \subset \hat{\mu}$, where $s = `sym1|sym2|sym3|\0'$. In this state, the symbolic execution of *strlen*'s concrete function produces the following formula:

$$\Phi \equiv \begin{cases} \varphi_1 \vee \varphi_2 \vee \varphi_3 \vee \varphi_4 \\ \varphi_1 \equiv (sym1 = \backslash 0) \ \wedge \ (\texttt{ret} = 0) \\ \varphi_2 \equiv (sym1 \neq \backslash 0) \ \wedge \ (sym2 = \backslash 0) \ \wedge \ (\texttt{ret} = 1) \\ \varphi_3 \equiv (sym1 \neq \backslash 0) \ \wedge \ (sym2 \neq \backslash 0) \ \wedge \ (sym3 = \backslash 0) \ \wedge \ (\texttt{ret} = 2) \\ \varphi_4 \equiv (sym1 \neq \backslash 0) \ \wedge \ (sym2 \neq \backslash 0) \ \wedge \ (sym3 \neq \backslash 0) \ \wedge \ (\texttt{ret} = 3) \end{cases}$$

Backward Soundness - *strlen2* Let us recall the summary *strlen2* in Listing 3.1. In the same symbolic state $\hat{\sigma}$, the symbolic execution of this summary produces the boolean formula:

$$\hat{\Phi}_2 \equiv (sym1 \neq \backslash 0) \ \wedge \ (sym2 \neq \backslash 0) \ \wedge \ (sym3 \neq \backslash 0) \ \wedge \ (\texttt{ret} = 3)$$

For this summary we can see that the execution path generated is contained in the concrete function's execution paths, consequently only the implication $\hat{\Phi}_2 \Rightarrow \Phi$ is true, and therefore the summary is exclusively *backward sound*.

Forward Soundness - *strlen3* On the other hand, if we recall the summary *strlen3* given in Listing 3.2, the symbolic execution of this summary in $\hat{\sigma}$ produces the formula: $\hat{\Phi}_3 \equiv (\texttt{ret} \geq 0)$. In contrast to the previous summary, we can observe that $\hat{\Phi}_3$ contains all the execution paths of Φ. The return value of *strlen* is always greater than or equal to zero, hence, $\Phi \Rightarrow \hat{\Phi}_3$ is true. Despite containing all the correct execution paths, $\hat{\Phi}_3$ also contains incorrect paths such as: $(sym1 = \backslash 0) \wedge (\texttt{ret} = 1)$, meaning that the implication $\hat{\Phi}_3 \Rightarrow \Phi$ is false, and therefore the summary is exclusively *forward sound*.

Completeness - *strlen4* As an example of a *complete* summary for *strlen*, consider the summary *strlen4* given in Listing 4.1. Given a symbolic input string, this summary generates the appropriate

execution paths corresponding to all the possible lengths for the given symbolic string. Accordingly, when symbolically executed in $\hat{\sigma}$, this summary produces the same execution paths as the concrete function, and so the equivalence $\hat{\Phi}_4 \Leftrightarrow \Phi$ is true. Although the execution paths covered by the summary exactly coincide with those of the concrete function, the symbolic summary has the advantage of generating a single final symbolic state capturing all those path, while the concrete function will generate a separate symbolic state for each path.

```
int strlen4(char* s){
    int retval = 0, i = 0;
    char char_zero = '\0';

    //Initializes conditions
    restr_t sym_conds = summ_false();
    restr_t different_conds = summ_true();

    while(1){ //Loop through all chars
        if(summ_is_symbolic(&s[i],CHAR_SIZE)){ //Symbolic char

            //s[i] == \0 --> add (s[i] == \0) ∧ (different_conds) ∧ (retval = i) to sym_conds
            if(_solver_is_it_possible(_solver_EQ(&s[i], &char_zero, CHAR_SIZE))){
                if(!summ_is_symbolic(&retval,INT_SIZE)) {retval = summ_new_sym_var(INT_SIZE);}

                restr_t equal_zero =
                 ↪  _solver_And(_solver_EQ(&s[i],&char_zero,CHAR_SIZE),_solver_EQ(&retval,&i,INT_SIZE));
                restr_t aux = _solver_And(equal_zero, different_conds);
                sym_conds = _solver_Or(sym_conds, aux);
            }

            //s[i] ≠ \0 --> add (s[i] ≠ \0) to different_conds
            if(_solver_is_it_possible(_solver_NEQ(&s[i], &char_zero, CHAR_SIZE))){
                restr_t not_equal_zero = _solver_NEQ(&s[i], &char_zero,CHAR_SIZE);
                different_conds = _solver_And(different_conds, not_equal_zero);
            }
            else{ break; }
        }
        else if(s[i] == char_zero){ //Concrete char
            if(!summ_is_symbolic(&retval,INT_SIZE)) {retval = i;}
            else{
                different_conds = _solver_And(different_conds,_solver_EQ(&retval, &i, INT_SIZE))
                sym_conds = _solver_Or(sym_conds, different_conds);
            } break;
        }
        i++;
    }
    if(summ_is_symbolic(&retval,INT_SIZE)) {summ_assume(sym_conds)}; //Add restrictions
    return retval;
}
```

Listing 4.1: Implementation of summary strlen4

Generalized Completeness - *strlen1* Let us consider the summary *strlen1* given in Listing 4.2. This summary simply counts the number of characters (either concrete or symbolic) until a concrete null byte is found. Its symbolic execution in $\hat{\sigma}$ produces the formula: $\hat{\Phi}_1 \equiv (\texttt{ret} = 3)$. Accordingly, none of the soundness properties can be verified, as both implications $\hat{\Phi}_1 \Rightarrow \Phi$ and $\Phi \Rightarrow \hat{\Phi}_1$ are false. Hence, for

this summary, the validation tool will produce the following counterexamples:

$$\text{Missing Path: } [s = \text{`}A|A|\,\backslash 0\,|\,\backslash 0\text{'} \wedge\ ret = 2]$$
$$\text{Wrong Path: } [s = \text{`}A|A|\,\backslash 0\,|\,\backslash 0\text{'} \wedge\ ret = 3]$$

To address the soundness problems of *strlen1*, one can use a predicate that states that the given string has no intermediate null characters; in this case, for strings of length 3, the predicate can be formalized as follows:

$$\rho \equiv (sym1 \neq \backslash 0) \ \wedge\ (sym2 \neq \backslash 0) \ \wedge\ (sym3 \neq \backslash 0)$$

Considering this precondition, we compute a new boolean formula that results from the symbolic execution of *strlen*'s concrete function in $\hat{\sigma} \ \wedge\ \rho$. Given restrictions on the input, the symbolic execution of the concrete function on the initial symbolic state will generate the formula:

$$\Phi \equiv (sym1 \neq \backslash 0) \ \wedge\ (sym2 \neq \backslash 0) \ \wedge\ (sym3 \neq \backslash 0) \ \wedge\ (\texttt{ret} = 3)$$

Analogously, the symbolic execution of the summary *strlen1* in the same symbolic state $\hat{\sigma}_1 \ \wedge\ \rho$ will generate the formula:

$$\hat{\Phi}_1 \equiv (sym1 \neq \backslash 0) \ \wedge\ (sym2 \neq \backslash 0) \ \wedge\ (sym3 \neq \backslash 0) \ \wedge\ (\texttt{ret} = 3)$$

We can see that when restricting the initial symbolic state with the predicate ρ, the two final symbolic states coincide. Hence, we conclude that the symbolic summary is generalized complete with respect to the predicate ρ as it satisfies both implications: $\Phi \Rightarrow \hat{\Phi}_1$ and $\hat{\Phi}_1 \Rightarrow \Phi$.

```
1  int strlen1(char* s){
2      int i = 0;
3      while(1){
4          if(!summ_is_symbolic(&s[i], CHAR_SIZE){ //s[i] is not symbolic
5              if (s[i] == '\0') break;
6          }
7          i++;
8      }
9      return i;
10 }
```

Listing 4.2: Implementation of summary strlen1

4.2.1.B Function - *strcmp*

Consider the libc function *strcmp* (`int strcmp(const char *str1, const char *str2)`), that compares two input strings, character by character according to their ASCII value. As specified by the standard, this function returns the value 0 if the two given strings coincide, a positive integer if the first string is lexicographically greater than the second one, and a negative integer otherwise. In order to check if our summaries for *strcmp* abide by the proposed correctness properties, we first must determine the boolean formula Φ that results from the symbolic execution of *strcmp*'s concrete implementation in an initial symbolic state $\hat{\sigma}$. For this function, we consider an initial symbolic state $\hat{\sigma} = \langle \hat{\mu}, true \rangle$ that contains two symbolic input strings with maximum length two; put formally: $str1, str2 \subset \hat{\mu}$, where $str1 =$ '$sym1|sym2|\setminus 0$' and $str2 =$ '$sym3|sym4|\setminus 0$'. The symbolic execution of *strcmp*'s concrete function in the symbolic state $\hat{\sigma}$ produces the following boolean formula:

$$\Phi \equiv \begin{cases} \varphi_1 \vee \varphi_2 \vee \varphi_3 \vee \varphi_4 \vee \varphi_5 \vee \varphi_6 \vee \varphi_7 \\ \varphi_1 \equiv (sym1 > sym3) \ \wedge \ (\texttt{ret} = 1) \\ \varphi_2 \equiv (sym1 < sym3) \ \wedge \ (\texttt{ret} = -1) \\ \varphi_3 \equiv (sym1 = sym3) \ \wedge \ (sym1 = \setminus 0) \ \wedge \ (\texttt{ret} = 0) \\ \varphi_4 \equiv (sym1 = sym3) \ \wedge \ (sym2 > sym4) \ \wedge \ (sym1 \neq \setminus 0) \ \wedge \ (\texttt{ret} = 1) \\ \varphi_5 \equiv (sym1 = sym3) \ \wedge \ (sym2 < sym4) \ \wedge \ (sym1 \neq \setminus 0) \ \wedge \ (\texttt{ret} = -1) \\ \varphi_6 \equiv (sym1 = sym3) \ \wedge \ (sym2 = sym4) \ \wedge \ (sym1 \neq \setminus 0) \ \wedge \ (sym2 = \setminus 0) \ \wedge \ (\texttt{ret} = 0) \\ \varphi_7 \equiv (sym1 = sym3) \ \wedge \ (sym2 = sym4) \ \wedge \ (sym1 \neq \setminus 0) \ \wedge \ (sym2 \neq \setminus 0) \ \wedge \ (\texttt{ret} = 0) \end{cases}$$

Let us consider the symbolic summary *strcmp2*. Listing 4.3 shows a partial implementation of this summary, focusing on the fragment of the summary responsible for handling input strings of the same size (the complete implementation can be found in A.2). In this case, the summary iterates over the characters of the two given strings. At each index, the summary first checks if the two characters at that index are guaranteed to be different, in which case it simply returns 1. If it cannot prove that the two characters are different, it constructs a formula, *c1_equals_c2*, representing the conditions that must hold for the characters at the current index to coincide and conjuncts that formula with the accumulator formula *equals_conds_restr*. At the end of the loop, the accumulator formula *equals_conds_restr* holds the conditions that must be true for the two given strings to coincide. Hence, the summary creates a new symbolic variable *retval*, which is used as the return value of the summary, and extends the current path condition with the the formula:

$$(equals\_conds\_restr \wedge retval = 0) \vee (\neg equals\_conds\_restr \wedge retval \neq 0)$$

```
1   int strcmp2(char* s1, char* s2){
    ...
28  for(int i = 0; i < size i++){ //Strings can be the same size
29      char c1 = s1[i];
30      char c2 = s2[i];
31
32      //Both chars are concrete and different
33      if(!summ_is_symbolic(&s1[i],CHAR_SIZE) && !summ_is_symbolic(&s2[i],CHAR_SIZE) && c1!=c2){
34          canBeEqual = 0; canBeDifferent = 1;
35          break;
36      }
37      else{
38          restr_t c1_equals_c2 = _solver_EQ(&c1, &c2, CHAR_SIZE);       // c1 == c2
39          restr_t c1_not_equals_c2 = _solver_NEQ(&c1, &c2, CHAR_SIZE);  // c1 != c2
40
41          //c1 must equal c2?
42          if(!_solver_is_it_possible(c1_equals_c2)){
43              canBeEqual = 0; canBeDifferent = 1;
44              break;
45          }
46          else{
47               //can c1 be different than c2?
48              if(_solver_is_it_possible(c1_not_equals_c2)){
49                  canBeDifferent = 1;
50              }
51              canBeEqual = 1;
52              equal_conds_restr = _solver_And(equal_conds_restr, c1_equals_c2);
53          }
54      }
55  }
56
57  if(canBeDifferent && canBeEqual){ //Strings can be both equal and different
58      int zero = 0; symbolic retval = summ_new_sym_var(INT_SIZE);
59      restr_t retval_equals_zero = _solver_EQ(&retval, &zero, INT_SIZE);
60      restr_t retval_diff_zero = _solver_NEQ(&retval, &zero, INT_SIZE);
61
62      diff_conds_restr = _solver_And(_solver_NOT(equal_conds_restr), retval_diff_zero);
63      equal_conds_restr = _solver_And(equal_conds_restr, retval_equals_zero);
64      equal_conds_restr = _solver_And(equal_conds_restr, str_diff_zero(s1));  //s1 has no \0
65
66      final_restr = _solver_Or(equal_conds_restr, diff_conds_restr);
67      summ_assume(final_restr);
68      return retval;
69  }
70
71  else if(canBeEqual){ //Strings can only be equal
72      equal_conds_restr = _solver_And(equal_conds_restr, str_diff_zero(s1));
73      summ_assume(equal_conds_restr);
74      return 0;
75  }
    ...
```

Listing 4.3: Summarized implementation of summary strcmp2

In order to assess whether or not this symbolic summary is *forward/backward sound*, let us consider

the symbolic state resulting from the execution of the summary in the state $\hat{\sigma}$ introduced above:

$$\hat{\Phi} \equiv \begin{cases} \hat{\varphi}_1 \vee \hat{\varphi}_2 \\ \hat{\varphi}_1 \equiv (sym1 = sym3) \ \wedge \ (sym2 = sym4) \ \wedge \ (sym1 \neq \backslash 0) \ \wedge \ (sym2 \neq \backslash 0) \ \wedge \ (\texttt{ret} = 0) \\ \hat{\varphi}_2 \equiv ((sym1 \neq sym3) \ \vee \ (sym2 \neq sym4)) \ \wedge \ (\texttt{ret} \neq 0) \end{cases}$$

With the help of the validation tool, we can conclude that the summary does not satisfy any of the soundness properties, since both implications $\Phi \Rightarrow \hat{\Phi}$ and $\hat{\Phi} \Rightarrow \Phi$ are false. More concretely, for this summary, our validation tool will produce the following counterexamples:

$$\text{Missing Path: } [str1 = \text{`}\backslash 0\,|A|\,\backslash 0\text{'} \ \wedge \ str2 = \text{`}\backslash 0\,|B|\,\backslash 0\text{'} \wedge \ ret = 0]$$
$$\text{Wrong Path: } [str1 = \text{`}A|A|\,\backslash 0\text{'} \ \wedge \ str2 = \text{`}B|B|\,\backslash 0\text{'} \wedge \ ret = 1]$$

The missing path captures the case in which the two strings have a null character in an intermediate position and have mismatching characters after the null character. While the concrete implementation would consider such two strings equal, the summary strcmp2 does not capture this case since it assumes that all intermediate characters are different from $\backslash 0$. The wrong path captures the case in which the first string is lexicographically smaller than the second one. In this case, the summary strcmp2 simply states that $ret \neq 0$, allowing the symbolic return value to include positive integers.

To address the soundness problems of *strcmp2*, we consider two separate predicates over the given input strings. A predicate $\rho_1 \equiv (sym1 \neq \backslash 0) \wedge (sym2 \neq \backslash 0)$ stating that none of the strings contains intermediate null characters and a predicate $\rho_2 \equiv (sym1 = sym3) \wedge (sym2 = sym4)$ stating that the two strings coincide in all characters. In the following, we will demonstrate that the summary strcmp2 is forward sound with respect to ρ_1 and backward sound with respect to ρ_2, meaning that it is complete with respect to the conjunction of the two predicates.

Generalized Forward Soundness - *strcmp2* Let us first consider the predicate ρ_1. The symbolic execution of the concrete function *strcmp* in the symbolic state restricted by this predicate, $\hat{\sigma} \wedge \rho_1$, will result in the following formula:

$$\Phi_{\rho_1} \equiv \begin{cases} \varphi_1 \vee \varphi_2 \vee \varphi_3 \vee \varphi_4 \vee \varphi_5 \\ \varphi_1 \equiv (sym1 > sym3) \ \wedge \ (sym1 \neq \backslash 0) \ \wedge \ (sym2 \neq \backslash 0) \ \wedge \ (\texttt{ret} > 0) \\ \varphi_2 \equiv (sym1 < sym3) \ \wedge \ (sym1 \neq \backslash 0) \ \wedge \ (sym2 \neq \backslash 0) \ \wedge \ (\texttt{ret} < 0) \\ \varphi_3 \equiv (sym1 = sym3) \ \wedge \ (sym2 > sym4) \ \wedge \ (sym1 \neq \backslash 0) \ \wedge \ (sym2 \neq \backslash 0) \ \wedge \ (\texttt{ret} = 1) \\ \varphi_4 \equiv (sym1 = sym3) \ \wedge \ (sym2 < sym4) \ \wedge \ (sym1 \neq \backslash 0) \ \wedge \ (sym2 \neq \backslash 0) \ \wedge \ (\texttt{ret} = -1) \\ \varphi_5 \equiv (sym1 = sym3) \ \wedge \ (sym2 = sym4) \ \wedge \ (sym1 \neq \backslash 0) \ \wedge \ (sym2 \neq \backslash 0) \ \wedge \ (\texttt{ret} = 0) \end{cases}$$

Analogously, the symbolic execution of the summary *strcmp2* in the symbolic state $\hat{\sigma} \wedge \rho_1$, will result in the following formula:

$$\hat{\Phi}_{\rho_1} \equiv \begin{cases} \hat{\varphi}_1 \vee \hat{\varphi}_2 \vee \hat{\varphi}_3 \\ \hat{\varphi}_1 \equiv (sym1 = sym3) \ \wedge \ (sym2 = sym4) \ \wedge \ (sym1 \neq \backslash 0) \ \wedge \ (sym2 \neq \backslash 0) \ \wedge \ (\texttt{ret} = 0) \\ \hat{\varphi}_2 \equiv (sym1 \neq sym3) \ \wedge \ (sym1 \neq \backslash 0) \ \wedge \ (sym2 \neq \backslash 0) \ \wedge \ (\texttt{ret} \neq 0) \\ \hat{\varphi}_3 \equiv (sym2 \neq sym4) \ \wedge \ (sym1 \neq \backslash 0) \ \wedge \ (sym2 \neq \backslash 0) \ \wedge \ (\texttt{ret} \neq 0) \end{cases}$$

One can easily see that the paths generated by the execution of the concrete function are contained in those generated by the summary, meaning that the summary is *generalized forward sound* with respect to predicate ρ_1 ($\Phi_{\rho_1} \Rightarrow \hat{\Phi}_{\rho_1}$).

Generalized Backward Soundness - *strcmp2* Let us now consider predicate ρ_2. The symbolic execution of the concrete function *strcmp* in the symbolic state restricted by this predicate, $\hat{\sigma} \wedge \rho_2$, will result in the following formula:

$$\Phi_{\rho_2} \equiv \begin{cases} \varphi_1 \vee \varphi_2 \vee \varphi_3 \\ \varphi_1 \equiv (sym1 = sym3) \ \wedge \ (sym2 = sym4) \ \wedge \ (sym1 = \backslash 0) \ \wedge \ (\texttt{ret} = 0) \\ \varphi_2 \equiv (sym1 = sym3) \ \wedge \ (sym2 = sym4) \ \wedge \ (sym1 \neq \backslash 0) \ \wedge \ (sym2 = \backslash 0) \ \wedge \ (\texttt{ret} = 0) \\ \varphi_3 \equiv (sym1 = sym3) \ \wedge \ (sym2 = sym4) \ \wedge \ (sym1 \neq \backslash 0) \ \wedge \ (sym2 \neq \backslash 0) \ \wedge \ (\texttt{ret} = 0) \end{cases}$$

Analogously, the symbolic execution of the summary *strcmp2* in the symbolic state $\hat{\sigma} \wedge \rho_2$, will result in the following formula:

$$\hat{\Phi}_{\rho_2} \equiv (sym1 = sym3) \ \wedge \ (sym2 = sym4) \ \wedge \ (sym1 \neq 0) \ \wedge \ (sym2 \neq 0) \ \wedge \ (\texttt{ret} = 0)$$

In contrast to the executions obtained for predicate ρ_1, in this case the paths generated by the symbolic execution of the summary are contained in those generated by concrete function, meaning that the summary is *generalized backward sound* with respect to predicate ρ_2 ($\hat{\Phi}_{\rho_2} \Rightarrow \Phi_{\rho_2}$).

Generalized Completeness - *strcmp2* Finally, consider the predicate $\rho_3 \equiv (\rho_1 \wedge \rho_2)$. The symbolic execution of both the concrete function and the summary in the symbolic state restricted by this predicate, $\hat{\sigma} \wedge \rho_3$, produce the same formula:

$$\Phi_{\rho_3} \equiv \hat{\Phi}_{\rho_3} \equiv (sym1 = sym3) \ \wedge \ (sym2 = sym4) \ \wedge \ (sym1 \neq \backslash 0) \ \wedge \ (sym2 \neq \backslash 0) \ \wedge \ (\texttt{ret} = 0)$$

Hence, the equivalence $\hat{\Phi}_{\rho_3} \Leftrightarrow \Phi_{\rho_3}$ is true, meaning that the summary *strcmp2* is *generalized complete* with respect to the predicate ρ_3.

4.2.1.C Function - memcpy

Consider the libc function *memcpy* (`void* memcpy(void* dst, const void* src, size_t n)`). This function copies n bytes from the memory area pointed to by src to the memory area pointed to by dst, and returns a pointer to dst. To compute the boolean formula Φ for *memcpy*'s concrete implementation, we consider a symbolic state $\hat{\sigma}$ where the input argument n is an unconstrained symbolic variable. In this symbolic state, the symbolic execution of *memcpy*'s concrete implementation will generate the following simplified formula:

$$\Phi \equiv \begin{cases} \varphi_0 \vee \varphi_1 \vee \varphi_2 \vee \ \ldots \ \vee \varphi_{max} \\ \varphi_0 \equiv \ BytesCopied(src, dest, 0) \ \wedge \ (n = 0) \ \wedge \ (\texttt{ret} = dst) \\ \varphi_1 \equiv \ BytesCopied(src, dest, 1) \ \wedge \ (n = 1) \ \wedge \ (\texttt{ret} = dst) \\ \varphi_2 \equiv \ BytesCopied(src, dest, 2) \ \wedge \ (n = 2) \ \wedge \ (\texttt{ret} = dst) \\ \quad \vdots \\ \varphi_{max} \equiv \ BytesCopied(src, dest, max) \ \wedge \ (n = max) \ \wedge \ (\texttt{ret} = dst) \end{cases}$$

where max corresponds to the maximum unsigned integer (*size_t*) that can be represented by the symbolic variable n and the predicate $BytesCopied(src, dest, i)$ is used to signify that i bytes from the memory segment pointed to by src were copied to the memory segment pointed to by dst. In practice, however, it is not feasible to consider all the possible execution paths that can be triggered by a call to memcpy with an unconstrained symbolic argument n. To counter this issue, when testing our summaries for memcpy, we make use of a predicate ρ that bounds the value denoted by the parameter n. Here, to simplify the presentation, we set the maximum value of n to be 3 with the predicate $\rho \equiv (n \leq 3)$. Furthermore, we also constrain the size of the memory segments pointed to by src and dst, picking size 3 for both segments, and assume that they are disjoint. Put formally, we consider the initial symbolic state $\hat{\sigma} = \langle \hat{\mu}, \rho \rangle$, where: $src, dst \subset \hat{\mu}$, $src = \{a, a, a\}$, and $dst = \{b, b, b\}$. The symbolic execution of *memcpy*'s concrete implementation in the state $\hat{\sigma}$ generates the boolean formula:

$$\Phi_\rho \equiv \begin{cases} \varphi_0 \vee \varphi_1 \vee \varphi_2 \vee \varphi_3 \\ \varphi_0 \equiv (src_0 = a \wedge src_1 = a \wedge src_2 = a) \wedge (dst_0 = b \wedge dst_1 = b \wedge dst_2 = b) \wedge (n = 0) \wedge (\texttt{ret} = dst) \\ \varphi_1 \equiv (src_0 = a \wedge src_1 = a \wedge src_2 = a) \wedge (dst_0 = a \wedge dst_1 = b \wedge dst_2 = b) \wedge (n = 1) \wedge (\texttt{ret} = dst) \\ \varphi_2 \equiv (src_0 = a \wedge src_1 = a \wedge src_2 = a) \wedge (dst_0 = a \wedge dst_1 = a \wedge dst_2 = b) \wedge (n = 2) \wedge (\texttt{ret} = dst) \\ \varphi_3 \equiv (src_0 = a \wedge src_1 = a \wedge src_2 = a) \wedge (dst_0 = a \wedge dst_1 = a \wedge dst_2 = a) \wedge (n = 3) \wedge (\texttt{ret} = dst) \end{cases}$$

Backward Soundess - *memcpy2* Now let us consider the summary *memcpy2* given in Listing 4.4. In

the same symbolic state $\hat{\sigma}_\rho$, this summary generates the boolean formula:

$$\hat{\Phi}_\rho \equiv (src_0 = a \wedge src_1 = a \wedge src_2 = a) \wedge (dst_0 = a \wedge dst_1 = a \wedge dst_2 = a) \wedge (n = 3) \wedge (\texttt{ret} = dst)$$

We can see that *memcpy2* generates an execution path that is contained in Φ_ρ. Hence, the implication $\hat{\Phi}_\rho \Rightarrow \Phi_\rho$ is true, meaning that that this summary is *generalized backward sound* with respect to predicate ρ.

In order to implement a summary that takes into account multiple execution paths of a function with memory manipulation side effects, a summary can either fork the symbolic engine's executor to explore several paths, or employ a mechanism that allows to create memory segments of symbolic size [59,60]. Considering that AVD does not support such mechanism, in *memcpy2* we choose to model the execution path corresponding to the maximum value that can be denoted by the input argument n. Hence, we can see that in the symbolic state $\hat{\sigma}_\rho$ *memcpy2* generates the execution path where 3 bytes are copied ($n = 3$). To this end, using the original unconstrained symbolic state $\hat{\sigma}$ the symbolic execution of *memcpy2* would produce the following simplified formula:

$$\hat{\Phi} \equiv BytesCopied(src, dest, max) \ \wedge \ (n = max) \ \wedge \ (\texttt{ret} = dst)$$

We can see that even without the predicate ρ, the implication $\hat{\Phi} \Rightarrow \Phi$ is true, meaning that the summary *memcpy2* is also *backward sound*.

4.3 Supporting the Validation Tool

In this section we will go over the implementation details of our summary validation tool, covering its main features and how it is supported by *AVD*.

AVD is a tool capable of performing unassisted symbolic execution that we can effortlessly extend and modify, making it an ideal platform to support the validation tool. In this context, the role of *AVD* is to provide a symbolic engine that can compute the symbolic outputs Φ and $\hat{\Phi}$ by performing pure symbolic execution over a target concrete function and a respective summary. When computing these formulas it is important to guarantee that the symbolic engine can produce all possible execution paths for Φ and $\hat{\Phi}$ with a given input. This is a crucial requirement for the validation tool, as only by generating every feasible execution path can we evaluate the correctness of a summary.

One of our design goals for the validation tool is to provide an infrastructure that can verify the correctness of a summary in a single joint execution; the user can provide a single binary file containing the code for both the concrete function C and the symbolic summary $\hat{C}$, along with the respective symbolic input. To this end, we have extended *AVD* so that it allows to run the two symbolic executions,

```
void* memcpy2(void *dest, void *src, size_t n){

    summ_memory_addr(src, &n, INT_SIZE);
    summ_memory_addr(dest, &n, INT_SIZE);

    //If length is symbolic maximize and restrict to a concrete length
    if(summ_is_symbolic(&n,INT_SIZE)){

        int max = summ_maximize(&n, INT_SIZE);
        restr_t maximize = _solver_EQ(&n, &max, INT_SIZE);
        summ_assume(maximize);
        n = max;
    }

    unsigned char *str_dest = (unsigned char*) dest;
    unsigned char *str_src = (unsigned char*) src;

    for(int i = 0; i < n; i++){
        unsigned char c = *(str_src + i);
        *(str_dest + i) = c;

    }
    return dest;
}
```

Listing 4.4: Implementation of summary *memcpy2*

on two independent starting *memories*, in a single global execution of the symbolic engine. As previously mentioned, in *AVD*, memories are the nodes of the symbolic execution tree, hence, by separating the starting nodes of the two executions, we can effectively separate the generated execution paths, allowing us to reach the end of the symbolic execution with the boolean formulas Φ and $\hat{\Phi}$ conveniently stored in two variables.

As illustrated in Figure 4.8, a boolean formula Φ is the logical disjunction of all execution paths corresponding to the leaf nodes (memories) that reach the return address of the concrete function. Where as described in Section 4.1, an execution path φ is the conjunction of all the conditional statements over symbolic input variables (π), along with the return value ($\texttt{ret} = \hat{r}$), and a boolean formula denoting the significant memory addresses manipulated ($\lceil \hat{\mu} \rceil_m^f$). The same logic applies to a formula $\hat{\Phi}$, however, as illustrated in our summaries, it is often the case that symbolic summaries for functions that do not interact with memory (e.g., *strlen*) are implemented to prevent the symbolic engine from branching at conditional statements, consequently, only the initial memory containing all the simulated paths will reach the return address of the summary. In order to construct an execution's path formula all its elements are obtained directly from the corresponding memory node. *AVD*'s *Memory* object not only contains the internal representation of symbolic memory, but also maintains the current path condition.

The formulas Φ and $\hat{\Phi}$ are internally represented as Z3 boolean expressions, consequently, the solver

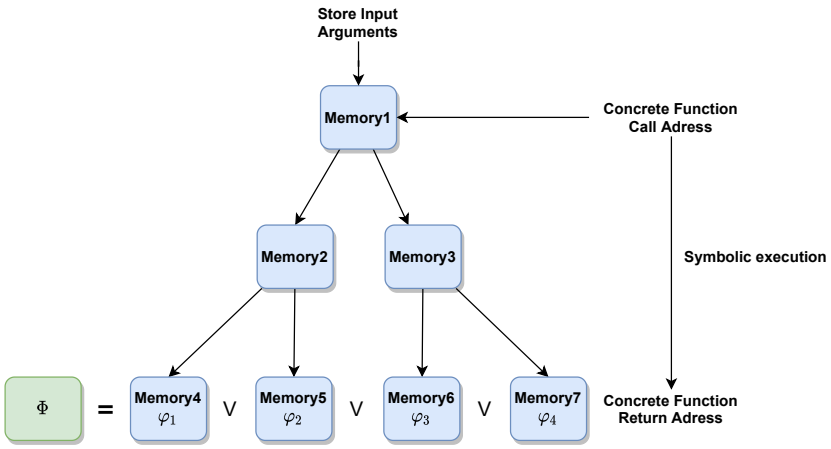


Figure 4.8: Illustration of AVD computing a boolean formula Φ

can prove the satisfiability of the correctness implications without any additional steps. In this manner, implications are effectively verified using Z3 to prove the satisfiability of the corresponding negated boolean formulas. For example, the Backward Soundness implication, $\hat{\Phi} \Rightarrow \Phi$, can be written as the formula: $\neg(\hat{\Phi} \wedge \neg\, \Phi)$, consequently, if the expression $\hat{\Phi} \wedge \neg\, \Phi$ is unsatisfiable the summary is *backward sound*. This also applies to generalized properties. As mentioned in Section 4.2, one of the requirements for this validation tool is to provide counterexamples that illustrate unsatisfiable correctness properties. To this end, when the solver proves a correctness implication false, it also automatically generates a concrete execution path that does not satisfy the given implication. More concretely, given a false implication $A \Rightarrow B$, the validation tool will ask Z3 to produce a model that satisfies $\neg(A \Rightarrow B)$.

4.3.1 Model Simplification

One important feature of the validation tool is model simplification. A Z3 generated model is nothing but an assignment of symbolic variables to concrete values. These models can be hard to interpret in the context of summary debugging, as they represent concretizations of abstract execution paths. To this end, we implement a mechanism that improves the interaction with the validation tool, by displaying the counterexample models prettified according to the arguments and return value of the specific function being tested. For example, suppose we want to generate a concrete model for the following formula:

$$\varphi \equiv (sym1 = sym3) \ \wedge \ (sym2 = sym4) \ \wedge \ (sym1 \neq 0) \ \wedge \ (sym2 \neq 0) \ \wedge \ (\texttt{ret} = 0)$$

where φ corresponds to the path condition generated by the symbolic execution of the *strcmp* function on two unconstrained symbolic input strings: $str1 =$ '$sym1|sym2|\,0$' and $str2 =$ '$sym3|sym4|\,0$'. When

asked for a concrete model for this formula, Z3 will produce the following model:

$$[\texttt{sym1} = 97, \texttt{sym2} = 97, \texttt{sym3} = 97, \texttt{sym4} = 97, \texttt{sym5} = 0]$$

This model assigns the symbolic bytes *sym1* to *sym4* to the ASCII value 97 and the return value *sym5* to the integer 0. Or summary validation tool prettifies the obtained model in order to connect it to the parameters and return value of the summary being tested, which in this case is the function strcmp. Hence, the obtained model would be presented to the user as:

string1 : ”aa\0”

string2 : ”aa\0”

return : 0

As mentioned before, this mechanism moulds the displayed output to conform to the specification of each function tested. Hence, different functions can produce vastly different outputs which implies different code to achieve the desired behaviour. To this end, inside the validation tool, all function dependent behaviour, such as model simplification, is implemented using plugins. Every function has its own plugin that contains all the code for functionalities that depend on that function’s specification. This mechanism allows us to easily extend the validation tool with support for additional functions, besides the default ones, by simply adding new function plugins.

5

Evaluation

Contents

In this chapter we present the results pertaining to the evaluation of our solution.

5.1 Evaluation Questions

This section answers the following three evaluation questions:

EQ1 - What is the overhead of tool independent summaries? We compared the performance of C-implemented symbolic summaries against that of natively implemented summaries. C-implemented summaries are slower than their natively-implemented counterparts since their code must be interpreted by the symbolic execution tool instead of being executed natively.

EQ2 - Is the symbolic reflection API sufficiently expressive to allow for the writing of backward/forward sound summaries? We used our summary validation tool to guarantee that the developed summaries satisfy the correctness properties introduced in Section 4.1.

EQ3 - Can our summary validation tool be used to analyse real-world symbolic summaries developed in the context of other tools? In order to test the applicability and overall scale of our summary validation tool, we used it to find bugs in external summaries developed as part of the *angr* and *Manticore* symbolic execution tools.

5.2 EQ1: Time Performance of Tool Independent Summaries

In this section we measure the difference in performance of C-implemented symbolic summaries against that of natively implemented summaries. For our evaluation, we selected a subset of the libc summaries provided by the *AVD* tool [16], implemented their corresponding C counterparts, and compared their respective performances. *AVD* comes with 36 symbolic summaries for libc functions implemented natively in Python. Out of these 36 summaries, we re-implemented 15 summaries directly in C using the exact same algorithms as their native Python equivalents. Then, we used *AVD* to symbolically execute two data sets using both the natively-implemented and the C-implemented symbolic summaries and measured their respective performances.

In order to carry out our evaluation we require a symbolic test suite in which to measure the difference in performance between Python-implemented and C-implemented summaries. To this end, we have used two distinct symbolic test suites, each with its own benefits. For the first test suite, we used a subset of the challenge binaries from DARPA's Cyber Grand Challenge (CGC) [58]. These challenges were designed for an automated Capture the Flag (CTF) exercise, allowing us to evaluate our summaries with binaries that simulate real world programs. For the second test suite, we have used a real-world HashMap library implemented in C and obtained from github [61]. The HashMap library did not come with symbolic tests, meaning that we had to write our own symbolic test suite. This allowed us to have

complete control over the size and complexity of the symbolic tests, enabling us to use pure symbolic execution as the analysis mode. In contrast, the CGC tests had to be run using an heuristic for trace-driven execution explained below.

Given a symbolic test suite, our goal is to characterize the overhead incurred by executing *AVD* with C-implemented summaries as opposed to natively-implemented summaries. Given a single symbolic test, the overhead for the test is affected by both the number of calls to symbolic summaries and the size of the test itself. Hence, in order to effectively characterize the overall overhead of our summaries considering the test suite as a whole, we cannot simply use an arithmetic average of single-test overheads, as the associated variance would be too high. Instead, we characterize the overhead incurred per executed instruction of C-implemented summaries. To this end, we measure for each symbolic test the number of executed instructions pertaining to C-implemented summaries and the total overhead. We then determine the best linear unbiased estimator for the overhead per executed symbolic summary instruction using a simple linear regression. More concretely, for a given test, the summary overhead O, can be written as:

$$O = \alpha \, . \, I_C \tag{5.1}$$

where the coefficient α is the overhead per executed instruction of a C summary, and I_C is the number of executed instructions of C-implemented summaries ($O = t_P - t_C$ with t_C and t_P being the execution time spent on the C and Python summaries respectively). To compute the best fitting coefficient $\hat{\alpha}$ for all tests, we use a simple linear regression as follows:

$$\hat{\alpha} = \frac{\sum_{i=1}^{n}(I_{C_i} - \overline{I_C})(O_i - \overline{O})}{\sum_{i=1}^{n}(I_{C_i} - \overline{I_C})^2} \tag{5.2}$$

Additionally, for a better understanding of the actual overhead experienced from employing the C-implemented summaries in a given data set, we also measure the global overhead percentage, $G_{\%}$, for a test suit according to the expression:

$$G_{(\%)} = \frac{\sum_{i=1}^{n} T_{C_i}}{\sum_{i=1}^{n} T_{P_i}} \times 100 \tag{5.3}$$

where T_{C_i} and T_{P_i} correspond to the total execution time of a test, i, using C and Python summaries respectively.

In order to streamline the evaluation process, we extended AVD with a simple mechanism to switch between natively-implemented and C-implemented summaries and also instrumented the tool to allow for measuring the execution time and number of instructions for each test and each executed summary.

5.2.1 CGC Data Set

CGC is the world's first all-machine cyber hacking tournament, where teams consisting of some of the top security researchers in the world, competed against each other in an automated CTF exercise. The Challenge Binaries that serve as the test bed for the CTF are tailor-made programs specifically implemented to contain a wide variety of known software vulnerabilities. These challenges do not use the standard libc runtime. However, they rely on various auxiliary functions that can be mapped to standard libc functions without damaging their functionality. For instance the challenge *CGC_Board* uses the function *cgc_strlen* that can be mapped to the standard libc function *strlen*.

The experiments for this data set where conducted using *AVD*'s heuristic for guided symbolic execution, which drives the symbolic execution engine along a specific precomputed path, instead of branching on all possible paths. Due to the large size and complexity of most of the challenge binaries, many of which attempt to simulate real world programs, it would be impractical to conduct this evaluation using pure symbolic execution. In fact, if we were to use pure symbolic execution on the CGC challenges, AVD would not be able complete the analysis of any challenges. Despite this, every CGC challenge has at least one Proof of Vulnerability (PoV), which we use to generate an execution trace that will trigger a vulnerability. Then, we feed these traces to *AVD* to perform guided symbolic execution along the path specified by the PoV.

The CGC dataset includes 246 challenge binaries from which 218 use functions that can be mapped to libc equivalent ones. These 218 challenges correspond to a total of 358 PoVs (recall that a challenge may be associated with more than one PoV). We executed *AVD* on these 358 PoVs using using both summary implementations with a maximum timeout of 1 hour. Some of these PoVs where excluded from the analysis: 88 PoVs timed out in both executions and 59 contained features unsupported by *AVD*, such as floating point operations, and multiple binaries, causing *AVD* to throw an error. From the remaining 211 PoVs, we obtained 100 valid executions, since 111 PoVs could not be analysed as they fall into a current limitation of *AVD*'s trace-driven heuristic for symbolic execution. Out of the remaining 100 valid PoVs, 10 where excluded as they did not call any summaries, and 3 were excluded for causing the symbolic execution to time out, leaving us with 87 selected POVs. We plotted the summary overhead, O, for all the 87 selected PoVs according to the expression (5.1). Results are shown in the scatter plot of Figure 5.1, where the x axis corresponds to the number of executed instructions pertaining to C-implemented summaries, I_C, and the y axis to the measured overhead, $t_C - t_P$. After computing the linear regression to best fit all data, we obtain a coefficient $\hat{\alpha}$ of 0.0005, which translates to an approximate overhead of 0.5 *ms* or half a millisecond per executed instruction of a C summary. As for the global overhead percentage for this data set, we obtain a $G_{\%}$ of 194%.

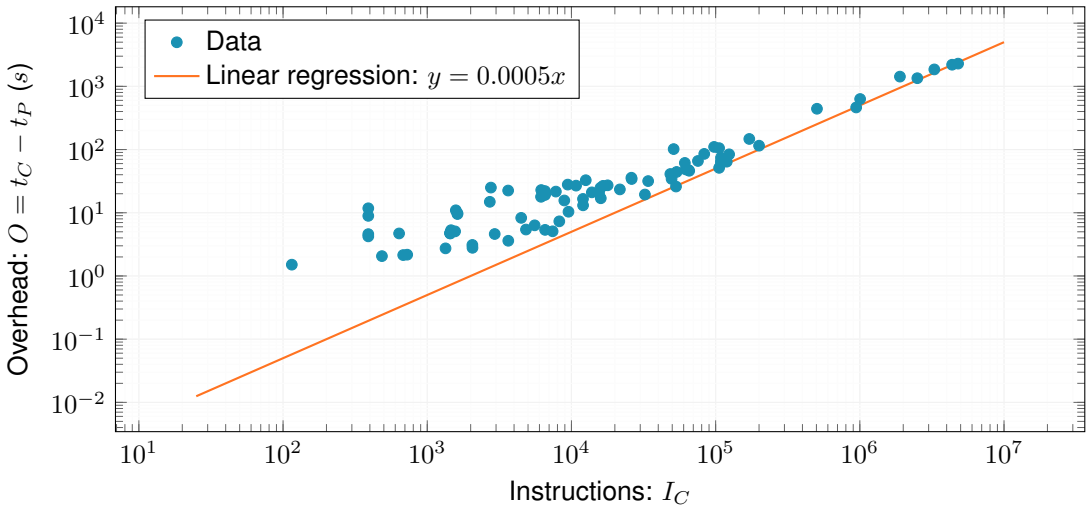


Figure 5.1: Overhead scatter plot for the GCG dataset.

5.2.2 HashMap Library

To complement the results obtained with the CGC binaries, we created our own data set using a C implementation of a chained HashMap data structure library. More concretely, we obtained a real-world HashMap implemented in C from github [61] and wrote a symbolic test suite for that library. This particular data structure and implementation were chosen to maximize the number of libc functions that can be replaced by our libc symbolic summaries. Our test suite consists of 10 symbolic tests, which cover all the functions exposed by the library: *hashmap_new*, *hashmap_free*, *hashmap_get* and *hashmap_set*. We focused on key-manipulating functions, which were tested using symbolic keys instead of concrete ones. Listing 5.1 shows one of the created symbolic tests. As the size and complexity of the targeted library are substantially smaller than those of the CGC benchmarks, our symbolic test suite can be executed without the guided symbolic execution heuristic of *AVD*. This allowed us to compare the level of branching and execution time obtained when using C-implemented symbolic summaries against those obtained when using the concrete implementations of libc functions.

One potential problem that could arise from using an HashMap as the case study is the symbolic execution of the hash function. Hash functions are notoriously problematic in the context of symbolic execution. Not only can they cause the number of paths to explode, but they will likely introduce complex mathematical computations (e.g., non-linear arithmetic) that an SMT solver cannot process. Considering that our goal is to evaluate the performance of the C summaries and not the hash function itself, we circumvent this problem by implementing an additional summary in Python to simulate the hash function. This summary is implemented using modular arithmetic so that it returns bucket values in a circular fashion, guaranteeing consistent hash collisions across executions.

To determine the overhead per instruction for this data set, we executed the 10 symbolic tests using

```c
/*test3-1*/
#define HASHMAP_SIZE 5
#define STRING_SIZE 5
#define MAX_VALUES 5

int main(){
    //Create hashmap
    hashmap_t map = hashmap_new(HASHMAP_SIZE);

    //Insert symbolic key-value pairs
    for(int i = 0; i< MAX_VALUES; i++){
        char key[STRING_SIZE];
        char value[STRING_SIZE];

        fgets(key, STRING_SIZE, 0);
        fgets(value, STRING_SIZE, 0);
        hashmap_set(map, key, &value, sizeof(char)*STRING_SIZE);
    }
    //Get value with concrete key
    char* key = "abcd" ;
    void* ptr = hashmap_get(map, key);

    //Print value
    char* val = (char *) ptr;
    if (val != NULL) puts(val);

    //Free hashmap
    hashmap_free(map);

    return 0;
}
```

Listing 5.1: *test3-1* from the HashMap data set.

both summary implementations with a maximum timeout of 30 minutes per test. Again, we plotted the summary overhead, O, for the 10 test binaries, as none of the executions timed out. The results are shown in the scatter plot of Figure 5.2. As before, we use a simple linear regression to estimate the value of the coefficient $\hat{\alpha}$. Interestingly, the results for the HashMap benchmark are consistent with those of the CGC benchmark in that we obtain the exact same value for $\hat{\alpha}$, 0.0005. This result suggests an overhead of approximately half a millisecond per executed instruction of a C summary that remains consistent across different data sets and heuristics. Regarding the global overhead percentage for this test suite, we obtain a $G_{\%}$ of 511%. As expected this value is substantially larger compared to the CGC one, as this data set is composed by much smaller tests denoting use cases of the HashMap library, consequently spending a much larger portion of the execution time inside summary calls.

For some libc functions we can execute their concrete reference implementations instead of symbolic summaries. However this approach tends to lead to prohibitive performance. To compare the perfor-

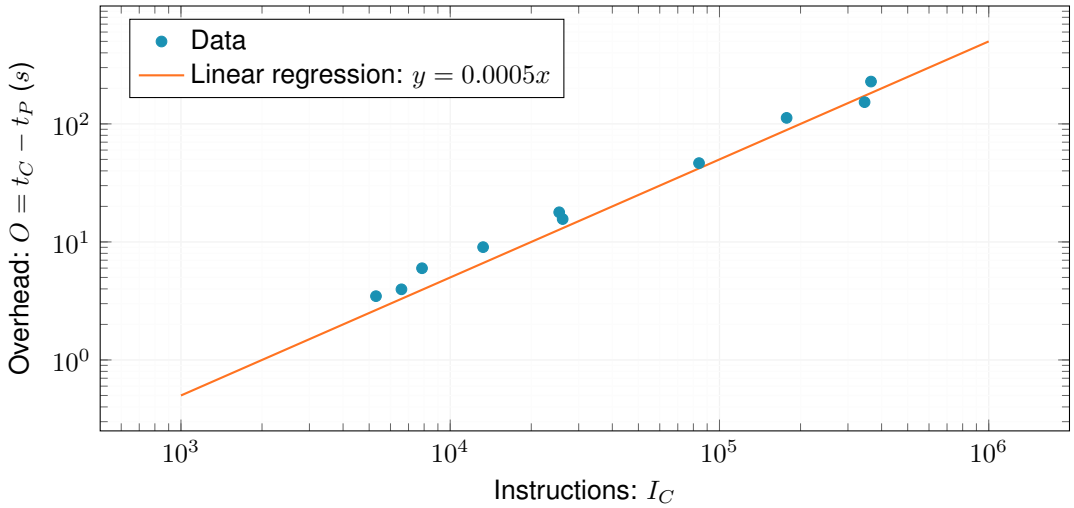


Figure 5.2: Overhead scatter plot for the HashMap dataset.

mance and branching of symbolic summaries against those of concrete reference implementations, we executed our symbolic test suite using both our C symbolic summaries and their respective C reference implementations. Some reference implementations were obtained from their manual page, while others were developed from scratch following their CGC counterparts. Results are summarized in Table 5.1, which shows both the total execution time and branching factor per symbolic test. The branching factor corresponds to the number of generated final symbolic memories. We can see that for all tests, the number of memories produced by the symbolic execution without summaries is always at least one order of magnitude greater than the corresponding execution employing summaries, and can even go as high as four orders of magnitude greater in *test3-1*. Consequently, 6 of the 10 tests executed without summaries reach the maximum time out of 30 minutes, 3 of which are executed in less than one minute with C summaries, which serves as a testament to the usefulness of symbolic summaries even when implemented in C (as opposed to being implemented natively).

5.3 EQ2: Summary Correctness

Our library of summaries models 20 libc functions from 3 different header files (*string.h*, *stdlib.h* and *stdio.h*) for each of which we implemented several summaries satisfying different correctness properties, with a total of 57 summaries. Considering two summaries that satisfy the same correctness property, we say that one summary is more accurate than the other if it is "closer" to satisfy the Completeness property. For example, considering two *backward sound* summaries, the more accurate summary models a larger number of correct paths of the concrete function. In contrast, for two *forward sound* summaries, the more accurate summary models a smaller number of spurious paths. The same logic can be ap-

Table 5.1: Summarized results for the HashMap dataset

	Python		C		No Summaries	
	Time (s)	Memories	Time (s)	Memories	Time (s)	Memories
test1-1	1.12	7	7.53	7	25.18	165
test1-2	4.10	15	19.21	15	124.41	225
test1-3	5.95	27	53.43	27	338.40	325
test2-1	6.53	63	24.02	63	532.55	6249
test2-2	25.42	485	180.59	485	Timeout	17732
test3-1	1.67	3	5.23	3	Timeout	15626
test3-2	32.08	369	152.89	369	Timeout	19524
test4-1	1.85	7	5.99	7	Timeout	9566
test4-2	67.87	919	304.21	919	Timeout	11710
test5	3.36	17	12.75	17	Timeout	9462

plied to the generalized properties, again considering two symbolic summaries that satisfy the same generalized property, the more accurate summary requires a weaker precondition ρ to satisfy the given property. As mentioned in Chapter 3, all summaries are named using a number suffix, which allows to organize the summaries by correctness property and order of accuracy. For example, considering our two *forward sound* summaries for the *atoi* function: *atoi2* and *atoi3*, the latter produces fewer spurious paths, as the summaries are ordered in increasing order of accuracy.

The correctness properties of all implemented summaries, except those modelling I/O functions, are given in Table 5.2, where the N/A column represents the summaries that only satisfy generalized properties. These are further described in Table 5.3, where we detail their corresponding generalized properties. For instance, we have two forward sound summaries (*atoi2* and *atoi3*) for atoi and one generalized forward sound (*atoi1*).

The summaries modelling I/O functions (*getchar*, *putchar fgets* and *puts*) are not included in the tables above since we cannot formally verify that they satisfy any correctness property. I/O functions must interact with their runtime environment through system calls, meaning that they cannot be fully implemented in C, as they have to step outside of the C semantics to execute the system call (in the case of getchar, the read system call). This means that in order to symbolically execute these functions we always have to have a summary to start with. This bootstrapping summary cannot be symbolically verified. Considering the importance of input functions in the context of symbolic execution, specially for security analysis, we implement our summaries for the *getchar* and *fgets* functions in order to model a reasonable number of execution paths that can be produced by the corresponding functions. For example, one of our summaries modelling the *fgets* function generates an unconstrained symbolic string with the maximum size allowed by the current path conditions.

Table 5.2: Correctness properties of the implemented summaries

	N/A	Backward Soundness	Forward Soundness	Completeness
atoi	1	-	2, 3	-
memchr	1	2	3	4
memcmp	1, 2, 3	-	4	-
memcpy	1	2	-	-
memmove	1	2	-	-
memset	1	2	-	-
strcat	1	2	-	-
strchr	1	2	3, 4	5
strcmp	1, 2, 3	-	4	-
strcpy	1	2	-	-
strlen	1	2	3	4
strncat	1, 2	3	-	-
strncmp	1, 2, 3	-	4	-
strncpy	1, 2	3	-	-
strpbrk	1	3	-	4
strrchr	1	2	3	4

5.4 EQ3: Bugs in Symbolic Execution tools

We use our summary validation tool to find bugs in the symbolic summaries included in tools other than *AVD*, more concretely the symbolic execution tools *angr* and *Manticore*. In this context, we consider as a "bug" a summary that does not satisfy any of the standard soundness properties (*backward* or *forward soundness*), and for which there is no additional information about the expected behaviour of the summary regarding missing/incorrect paths. To this end, we implemented a total of 14 summaries from both tools directly in C, using our reflection API and following their original Python code. We then used our validation tool to evaluate these summaries by comparing them against their corresponding concrete implementations. Out of the analysed 14 summaries, we found two buggy summaries, one in *angr* and one in *Manticore*. Both summaries include spurious paths and exclude correct paths, meaning that they are neither backward nor forward sound. Importantly, the code of these summaries is not annotated with any comments clarifying the preconditions that must hold for the summary to be applied; hence, we cannot say whether or not the authors were aiming at a specific generalized property.

5.4.1 Bug in *angr*

The first detected bug occurs in *angr*'s summary for libc's *strncmp* function. The *strncmp* function is a variation of the standard *strcmp*, whose specification we described in Section 4.2.1.B, which receives an additional argument n specifying the maximum number of bytes to be compared from the two strings. In

Table 5.3: Generalized correctness properties of the implemented summaries

	Generalized Backward Soundness	Generalized Forward Soundness	Generalized Completeness
atoi	-	1	-
memchr	-	-	1
memcmp	2, 3	2, 3	1, 2, 3
memcpy	-	-	1
memmove	-	-	1
memset	-	-	1
strcat	-	-	1
strchr	-	-	1
strcmp	2, 3	2, 3	1, 2, 3
strcpy	-	-	1
strlen	-	-	1
strncat	-	-	1, 2
strncmp	2, 3	2, 3	1, 2, 3
strncpy	-	-	1, 2
strpbrk	-	-	1
strrchr	-	-	1

angr's architecture the *strncmp* summary also provides the core functionality for other string comparison summaries, as the implementations for the *strcmp*, *strstr* and *strcasecmp* summaries will call the parent *strncmp* summary. For instance, *strcmp*'s summary is implemented as a particular case of *strncmp*, where the argument n is equal to the length of the larger string.

All the possible execution paths for the *strncmp* function can be divided into two main groups according to the returned value: the execution paths where the return value is equal to zero; and in contrast, the execution paths where the return value is different from zero. *angr* correctly models all the execution paths that return the value zero, accounting for the cases where both strings are equal, both strings are empty, or the argument n is equal to zero. Regarding the execution paths with a return value different that zero, i.e, the cases where the input strings are different, using our notation, *angr* will generate the following execution path formula:

$$\varphi = [\pi \wedge (\texttt{ret} = 1)]$$

where π represents all the possible combinations for two strings to be different. However, by having a fixed return value of 1, the summary does not satisfy any of the standard soundness properties, as this formula produces both missing and incorrect executions paths. If we recall *strncmp*'s specification, this function should also return a negative value when the first string is lower than the second. Assuming for example two symbolic input strings of size 3, *str1* and *str2*, our validation tool will produce the following

counterexamples:

$$\text{Missing Path: } [str1 = \text{`}aaa\text{'} \wedge\ str2 = \text{`}bbb\text{'} \wedge\ n = 3 \wedge\ ret = -1]$$

$$\text{Wrong Path: } [str1 = \text{`}aaa\text{'} \wedge\ str2 = \text{`}bbb\text{'} \wedge\ n = 3 \wedge\ ret = 1]$$

5.4.2 Bug in *Manticore*

The second bug was found in *Manticore*'s summary for libc's *strcmp* function. In this tool's architecture *strcmp* is modelled using an if-then-else formula. The summary iterates over the two strings to build a recursive if-then-else formula over pairs of symbolic bytes. This formula expresses that if two symbolic bytes are different then the summary must return the difference of those bytes, else, if they are equal, the summary must return the value 0 when they are the last two bytes of the string or continue iterating otherwise. For instance, considering two symbolic input strings: $str1 = \text{`}sym1|sym2|\,\backslash 0\text{'}$ and $str2 = \text{`}sym3|sym4|\,\backslash 0\text{'}$, this summary will create an if-then-else tree as illustrated in Figure 5.3.

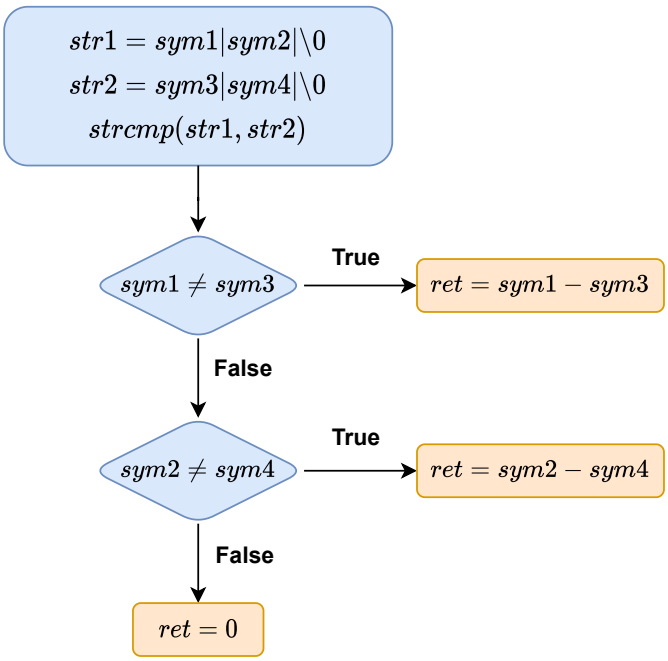


Figure 5.3: Illustration of an if-then-else tree produced by *Manticore*'s strcmp summary.

Considering a symbolic state $\hat{\sigma} = \langle \hat{\mu}, true \rangle$ that contains the two symbolic input strings, such that $str1, str2 \subset \hat{\mu}$. The symbolic execution of this summary in $\hat{\sigma}$ produces the following boolean formula:

$$\hat{\Phi} \equiv \begin{cases} \varphi_1 \vee \varphi_2 \vee \ \varphi_3 \vee \varphi_4 \vee \ \varphi_5 \\ \varphi_1 \equiv (sym1 > sym3) \ \wedge \ (\texttt{ret} = 1) \\ \varphi_2 \equiv (sym1 < sym3) \ \wedge \ (\texttt{ret} = -1) \\ \varphi_3 \equiv (sym1 = sym3) \ \wedge \ (sym2 > sym4) \ \wedge \ (\texttt{ret} = 1) \\ \varphi_4 \equiv (sym1 = sym3) \ \wedge \ (sym2 < sym4) \ \wedge \ (\texttt{ret} = -1) \\ \varphi_5 \equiv (sym1 = sym3) \ \wedge \ (sym2 = sym4) \ \wedge \ (\texttt{ret} = 0) \end{cases}$$

However, if we recall *strcmp*'s specification, the symbolic execution of a concrete function in the same symbolic state $\hat{\sigma}$ produces the formula:

$$\Phi \equiv \begin{cases} \varphi_1 \vee \varphi_2 \vee \varphi_3 \vee \varphi_4 \vee \varphi_5 \vee \varphi_6 \vee \varphi_7 \\ \varphi_1 \equiv (sym1 > sym3) \ \wedge \ (\texttt{ret} = 1) \\ \varphi_2 \equiv (sym1 < sym3) \ \wedge \ (\texttt{ret} = -1) \\ \varphi_3 \equiv (sym1 = sym3) \ \wedge \ (sym1 = \backslash 0) \ \wedge \ (\texttt{ret} = 0) \\ \varphi_4 \equiv (sym1 = sym3) \ \wedge \ (sym2 > sym4) \ \wedge \ (sym1 \neq \backslash 0) \ \wedge \ (\texttt{ret} = 1) \\ \varphi_5 \equiv (sym1 = sym3) \ \wedge \ (sym2 < sym4) \ \wedge \ (sym1 \neq \backslash 0) \ \wedge \ (\texttt{ret} = -1) \\ \varphi_6 \equiv (sym1 = sym3) \ \wedge \ (sym2 = sym4) \ \wedge \ (sym1 \neq \backslash 0) \ \wedge \ (sym2 = \backslash 0) \ \wedge \ (\texttt{ret} = 0) \\ \varphi_7 \equiv (sym1 = sym3) \ \wedge \ (sym2 = sym4) \ \wedge \ (sym1 \neq \backslash 0) \ \wedge \ (sym2 \neq \backslash 0) \ \wedge \ (\texttt{ret} = 0) \end{cases}$$

With the two formulas side by side, we can see that *Manticore*'s summary does not take into account that the intermediate symbolic bytes can also be null characters. Consequently, it does not satisfy any of the standard soundness properties, as $\hat{\Phi}$ leads to both missing and incorrect executions paths. Considering the same symbolic state $\hat{\sigma}$, our validation tool will produce the following counterexamples:

$$\text{Missing Path: } [str1 = \text{`}\backslash 0\,|A|\,\backslash 0\text{'} \ \wedge \ str2 = \text{`}\backslash 0\,|B|\,\backslash 0\text{'} \wedge \ ret = 0]$$

$$\text{Wrong Path: } [str1 = \text{`}\backslash 0\,|B|\,\backslash 0\text{'} \ \wedge \ str2 = \text{`}\backslash 0\,|A|\,\backslash 0\text{'} \wedge \ ret = 1]$$

6

Conclusion

Contents

6.1 Conclusions

Symbolic summaries are a key element of modern symbolic execution engines. They are an essential tool for both containing the path explosion problem and modelling interactions with the runtime environment. Even though the implementation of symbolic summaries is time-consuming and error-prone, there is still a clear lack of mechanisms and methodologies for sharing symbolic summaries across different tools and for their uniform validation.

This thesis proposes a new methodology for developing tool-independent summaries and semi-automatically validating them, which has at its core a new symbolic reflection API for explicit manipulation of C symbolic states in a tool-independent way. Using the proposed API, symbolic summaries can be directly implemented in C and shared across different symbolic execution tools, provided that these tools implement the API. To demonstrate the expressiveness of our API, we extended the symbolic execution tools *angr* and *AVD* in order to support it and developed tool-independent symbolic summaries for 20 different libc functions, comprising string manipulation functions, number-parsing functions, and input/output functions. Furthermore, we develop an infrastructure for the semi-automatic validation of the summaries written with our API and apply this infrastructure to the validation of 57 libc summaries written by us and 14 summaries obtained from state-of-the-art symbolic execution tools. Our validation tool flagged two of the third-party analysed summaries as being incorrect in that they both exclude correct execution paths and include spurious ones.

6.2 Future Work

This work streamlines the implementation and evaluation of summaries that can be employed by different symbolic execution tools. Despite this, there is still work to be done before our solution can be used to develop summaries for all types of C functions. Currently, our symbolic reflection API does not support the implementation of summaries for some system calls such as heap manipulation functions (e.g., *malloc*). To this end, we would like to extend our API with primitives that allow to interact with the execution environment of program, starting with a tool's internal representation of the *heap*. This is not a trivial task as symbolic execution tools can have vastly different mechanisms to model the C heap. Consequently, it is a significant challenge to design refined primitives that allow for the creation of expressive summaries for interaction with the heap

Still on the subject of system calls, we also would like to introduce new correctness properties for evaluating symbolic summaries that model system calls. All of our correctness properties evaluate summary models according to the results produced from the symbolic execution of concrete functions. Hence, these properties cannot be used to evaluate system calls as they represent code that can not be symbolically executed. Designing these new properties is also a considerable challenge, as it would

require defining the behaviour of system call functions in the context of symbolic execution; functions whose code is outside the scope of the programming language.

Finally, we also believe that the interaction with the validation tool can be improved. Currently, this tool must be run with one symbolic input at a time. In the future, we plan to extend the tool for it to accept a range of symbolic inputs on which to evaluate the given summary.

Bibliography

[1] M. Carvalho, J. Demott, R. Ford, and D. Wheeler, “Heartbleed 101,” *Security & Privacy, IEEE*, vol. 12, pp. 63–67, 07 2014.

[2] Z. Durumeric, M. Payer, V. Paxson, J. Kasten, D. Adrian, J. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, and J. Beekman, “The matter of heartbleed,” in *Proceedings of the 2014 Conference on Internet Measurement Conference*, 11 2014, pp. 475–488.

[3] R. S. Boyer, B. Elspas, and K. N. Levitt, “Select—a formal system for testing and debugging programs by symbolic execution,” in *Proceedings of the International Conference on Reliable Software*. New York, NY, USA: Association for Computing Machinery, 1975, p. 234–245. [Online]. Available: https://doi.org/10.1145/800027.808445

[4] J. C. King, “A new approach to program testing,” in *Proceedings of the International Conference on Reliable Software*. New York, NY, USA: Association for Computing Machinery, 1975, p. 228–233. [Online]. Available: https://doi.org/10.1145/800027.808444

[5] C. Cadar, D. Dunbar, and D. 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*, ser. OSDI’08. USA: USENIX Association, 2008, p. 209–224.

[6] N. Wells, “Busybox: A swiss army knife for linux,” *Linux Journal*, vol. 2000, p. 10, 01 2000.

[7] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems Design and Implementation (3rd Edition)*. USA: Prentice-Hall, Inc., 2005.

[8] P. Godefroid, M. Y. Levin, and D. A. Molnar, “Automated whitebox fuzz testing,” in *Network Distributed Security Symposium (NDSS)*. Internet Society, 2008.

[9] T. Ball, B. Cook, V. Levin, and S. K. Rajamani, “Slam and static driver verifier: Technology transfer of formal methods inside microsoft,” in *Integrated Formal Methods*, E. A. Boiten, J. Derrick, and G. Smith, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 1–20.

[10] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, “Model checking programs,” *Autom. Softw. Eng.*, vol. 10, pp. 203–232, 04 2003.

[11] R. Baldoni, E. Coppa, D. C. D’elia, C. Demetrescu, and I. Finocchi, “A survey of symbolic execution techniques,” *ACM Comput. Surv.*, vol. 51, no. 3, May 2018.

[12] P. O’Hearn, “Incorrectness logic,” *Proceedings of the ACM on Programming Languages*, vol. 4, pp. 1–32, 12 2019.

[13] K. R. Apt, “Ten years of hoare’s logic: A survey—part i,” *ACM Trans. Program. Lang. Syst.*, vol. 3, no. 4, p. 431–483, Oct. 1981. [Online]. Available: https://doi.org/10.1145/357146.357150

[14] E. Torlak and R. Bodik, “Growing solver-aided languages with rosette,” in *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ser. Onward! 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 135–152. [Online]. Available: https://doi.org/10.1145/2509578.2509586

[15] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, “Sok: (state of) the art of war: Offensive techniques in binary analysis,” in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 138–157.

[16] N. Sabino, “Automatic vulnerability detection: Using compressed execution traces to guide symbolic execution,” Master’s thesis, Instituto Superior Técnico, November 2019.

[17] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, “Manticore: A user-friendly symbolic execution framework for binaries and smart contracts,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1186–1189.

[18] L. de Moura and N. Bjørner, “Z3: An efficient smt solver,” in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[19] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, “CVC4,” in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 171–177. [Online]. Available: https://doi.org/10.1007/978-3-642-22110-1_14

[20] K. Sen, D. Marinov, and G. Agha, “Cute: A concolic unit testing engine for c,” in *SIGSOFT Software Engineering Notes*, vol. 30, 09 2005, pp. 263–272.

[21] J. Burnim and K. Sen, “Heuristics for scalable dynamic test generation,” in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 443–446.

[22] T. Avgerinos, S. Cha, B. Hao, and D. Brumley, “Aeg: Automatic exploit generation.” in *Communications of the ACM*, vol. 57, 01 2011.

[23] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, “Unleashing mayhem on binary code,” in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP ’12. USA: IEEE Computer Society, 2012, p. 380–394. [Online]. Available: https://doi.org/10.1109/SP.2012.31

[24] V. Chipounov, V. Kuznetsov, and G. Candea, “S2e: A platform for in-vivo multi-path analysis of software systems,” *Computer Architecture News*, vol. 39, 06 2012.

[25] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M. Potet, and J. Marion, “Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 653–656.

[26] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, “Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware,” in *22nd Annual Network and Distributed System Security Symposium*, 01 2015.

[27] Y. Shoshitaishvili, A. Bianchi, K. Borgolte, A. Cama, J. Corbetta, F. Disperati, A. Dutcher, J. Grosen, P. Grosen, A. Machiry, C. Salls, N. Stephens, R. Wang, and G. Vigna, “Mechanical phish: Resilient autonomous hacking,” *IEEE Security Privacy*, vol. 16, no. 2, pp. 12–22, 2018.

[28] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, “Bap: A binary analysis platform,” in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 463–469.

[29] N. M. Hai, M. Ogawa, and Q. T. Tho, “Obfuscation code localization based on CFG generation of malware,” in *Foundations and Practice of Security - 8th International Symposium, FPS 2015, Clermont-Ferrand, France, October 26-28, 2015, Revised Selected Papers*, ser. Lecture Notes in Computer Science, J. García-Alfaro, E. Kranakis, and G. Bonfante, Eds., vol. 9482. Springer, 2015, pp. 229–247. [Online]. Available: https://doi.org/10.1007/978-3-319-30303-1_14

[30] F. Song and T. Touili, “Pushdown model checking for malware detection,” in *International Journal on Software Tools for Technology Transfer*, vol. 16, 03 2012, pp. 110–125.

[31] N. Minh Hai, B. Nguyen, T. Quan, and M. Ogawa, “A hybrid approach for control flow graph construction from binary code,” in *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, 12 2013, pp. 159–164.

[32] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent, “The bincoa framework for binary code analysis,” in *Proceedings of the 23rd International Conference on Computer Aided Verification*, ser. CAV’11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 165–170.

[33] M.-D. Nguyen, S. Bardin, R. Bonichon, R. Groz, and M. Lemerre, “Binary-level directed fuzzing for use-after-free vulnerabilities,” *ArXiv*, vol. abs/2002.10751, 2020.

[34] L. Daniel, S. Bardin, and T. Rezk, “Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level,” in *2020 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2020, pp. 1021–1038. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP40000.2020.00074

[35] C. G. D. Val, “Conflict-driven symbolic execution : how to learn to get better,” Master’s thesis, University of British Columbia, 2014.

[36] Trail of Bits. (2015) How We Fared in the Cyber Grand Challenge. Accessed: September 7, 2022. [Online]. Available: https://blog.trailofbits.com/2015/07/15/how-we-fared-in-the-cyber-grand-challenge

[37] Common Vulnerabilities and Exposures. (2020) CVE-2020-7982. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-7982

[38] ——. (2020) CVE-2020-11104. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-11104

[39] ——. (2020) CVE-2020-11105. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-11105

[40] ——. (2020) CVE-2020-15359. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-15359

[41] ——. (2020) CVE-2020-10029. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-10029

[42] E. Reisner, C. Song, K. Ma, J. S. Foster, and A. Porter, “Using symbolic evaluation to understand behavior in configurable software systems,” in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1, 2010, pp. 445–454.

[43] K.-K. Ma, K. Yit Phang, J. S. Foster, and M. Hicks, “Directed symbolic execution,” in *Static Analysis*, E. Yahav, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 95–111.

[44] F. A. Manzano. Pysymemu. Accessed: September 7, 2022. [Online]. Available: https://github.com/feliam/pysymemu

[45] Capstone: The ultimate disassembler. Accessed: September 7, 2022. [Online]. Available: https://uclibc.org/

[46] pyelftools: Pure-python library for parsing elf and dwarf. Accessed: September 7, 2022. [Online]. Available: https://github.com/eliben/pyelftools

[47] V. Chipounov, V. Kuznetsov, and G. Candea, "The s2e platform: Design, implementation, and applications," *ACM Transactions on Computer Systems - TOCS*, vol. 30, pp. 1–49, 02 2012.

[48] Common Vulnerabilities and Exposures. (2015) CVE-2015-1536. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1536

[49] ——. (2015) CVE-2015-6098. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6098

[50] ——. (2016) CVE-2016-0040. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0040

[51] ——. (2016) CVE-2016-5400. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5400

[52] ——. (2017) CVE-2016-7219. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-7219

[53] ——. (2017) CVE-2017-15102. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-15102

[54] F. Saudel and J. Salwan, "Triton: A dynamic symbolic execution framework," in *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015*. SSTIC, 2015, pp. 31–54.

[55] Quarkslab. Accessed: September 7, 2022. [Online]. Available: https://quarkslab.com/

[56] uclibc. Accessed: September 7, 2022. [Online]. Available: https://uclibc.org/

[57] The GNU C library. Accessed: September 7, 2022. [Online]. Available: https://www.gnu.org/software/libc/

[58] DARPA. (2015) The Cyber Grand Challenge. Accessed: September 7, 2022. [Online]. Available: https://www.darpa.mil/program/cyber-grand-challenge

[59] J. Fragoso Santos, P. Maksimović, S.-E. Ayoun, and P. Gardner, "Gillian, part i: A multi-language platform for symbolic execution," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New

York, NY, USA: Association for Computing Machinery, 2020, p. 927–942. [Online]. Available: https://doi.org/10.1145/3385412.3386014

[60] L. Borzacchiello, E. Coppa, D. C. D'Elia, and C. Demetrescu, "Memory models in symbolic execution: key ideas and new thoughts," *Software Testing, Verification and Reliability*, vol. 29, 12 2019.

[61] R. Wiedenhöft. (2014) C Hash map. Accessed: September 7, 2022. [Online]. Available: https://gist.github.com/Richard-W/9568649

AppendixA

Listing A.1: Complete implementation of summary atoi2

```
int atoi2(char *str){
    int i = 0, sign = 1, res = 0;
    while(1){
        // Concrete char
        if(!summ_is_symbolic(&str[i],CHAR_SIZE)){
            if(is_numeric(str[i])){
                res = res * 10 + str[i] - '0';
                i++;
            }
            else if(str[i] == '-' && i == 0){
                sign = -1;
                i++;
            }
            else if(str[i] == '\0') break;
            else return 0;
        }
        //Symbolic char
        else{
```

```
            symbolic val = summ_new_sym_var(INT_SIZE);
            int size = strlen1(str);
            //Determine bounds
            int lower_bound = pow(10,size-1) * -1;
            int upper_bound = pow(10,size);

            //Build interval with restrictions
            restr_t val_GT_lower = _solver_SGT(&val, &lower_bound, INT_SIZE);
            restr_t val_LT_upper = _solver_SLT(&val, &upper_bound, INT_SIZE);
            restr_t bounds_restr = _solver_And(val_GT_lower,val_LT_upper);
            summ_assume(bounds_restr);
            return retval;
        }
    }
    return res * sign;
}
```

Listing A.2: Complete implementation of summary strcmp2

```
int strcmp2(char* s1, char* s2){

    int canBeDifferent = 0;
    int canBeEqual = 1;

    //Initializes condition with true
    restr_t equal_conds_restr = summ_true();

    char char_zero = '\0';
    int size1 = strlen1(s1);
    int size2 = strlen1(s2);

    //Strings must have different sizes
    if(size1 < size2 && (!summ_is_symbolic(&s2[size1],CHAR_SIZE) ||
    !_solver_is_it_possible(_solver_EQ(&s2[size1], &char_zero, CHAR_SIZE)))){
        canBeDifferent = 1;
        canBeEqual = 0;
    }
    else if(size1 > size2 && (!summ_is_symbolic(&s1[size2],CHAR_SIZE) ||
    !_solver_is_it_possible(_solver_EQ(&s1[size2], &char_zero, CHAR_SIZE)))){
        canBeDifferent = 1;
        canBeEqual = 0;
    }

    //Strings can be the same size
    else{
        int size = MIN(size1,size2);
```

```
28          for(int i = 0; i < size i++){
29              char c1 = s1[i];
30              char c2 = s2[i];
31
32              //Both chars are concrete and different
33              if(!summ_is_symbolic(&s1[i],CHAR_SIZE) && !summ_is_symbolic(&s2[i],CHAR_SIZE) &&
                ↪  c1!=c2){
34                  canBeEqual = 0; canBeDifferent = 1;
35                  break;
36              }
37              else{
38                  restr_t c1_equals_c2 = _solver_EQ(&c1, &c2, CHAR_SIZE);       // c1 == c2
39                  restr_t c1_not_equals_c2 = _solver_NEQ(&c1, &c2, CHAR_SIZE);  // c1 != c2
40
41                   //c1 must equal c2?
42                  if(!_solver_is_it_possible(c1_equals_c2)){
43                      canBeEqual = 0; canBeDifferent = 1;
44                      break;
45                  }
46                  else{
47                      //can c1 be different than c2?
48                      if(_solver_is_it_possible(c1_not_equals_c2)){
49                          canBeDifferent = 1;
50                      }
51                      canBeEqual = 1;
52                      equal_conds_restr = _solver_And(equal_conds_restr, c1_equals_c2);
53                  }
54              }
55          }
56      }
57      //Strings can be both equal and different
58      if(canBeDifferent && canBeEqual){
59          int zero = 0; symbolic retval = summ_new_sym_var(INT_SIZE);
60          restr_t retval_equals_zero = _solver_EQ(&retval, &zero, INT_SIZE);
61          restr_t retval_diff_zero = _solver_NEQ(&retval, &zero, INT_SIZE);
62
63          diff_conds_restr = _solver_And(_solver_NOT(equal_conds_restr), retval_diff_zero);
64          equal_conds_restr = _solver_And(equal_conds_restr, retval_equals_zero);
65          equal_conds_restr = _solver_And(equal_conds_restr, str_diff_zero(s1));  //s1 has no \0
66
67          final_restr = _solver_Or(equal_conds_restr, diff_conds_restr);
68          summ_assume(final_restr);
69          return retval;
70      }
71
72      //Strings can only be equal
73      else if(canBeEqual){
74          equal_conds_restr = _solver_And(equal_conds_restr, str_diff_zero(s1));
```

```
75          summ_assume(equal_conds_restr);
76          return 0;
77      }
78
79      //Strings can only be different
80      else{
81          return 1;
82      }
83  }
```