Toward Tool-Independent Summaries for Symbolic Execution

Frederico Ramos
frederico.ramos@tecnico.ulisboa.pt
Instituto Superior Técnico
Lisboa, Portugal

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

1 Introduction

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 [5, 9], 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, 10] 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 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* [4] found various fatal bugs in GNU COREUTILS (version 6.10) and a large number of critical bugs in other software systems, such as in BUSYBOX [16] and MINIX [14].

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 heterogeneous 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 [2]. 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. Hence, symbolic summaries provide an effective merging mechanism, allowing developers to deal with the path explosion problem in an application-specific way.

Despite being an essential tool for modelling interactions with the environment and containing the path explosion problem, symbolic summaries are extremely hard to design, with their implementation being both error-prone and time-consuming [6]. For this reason, developers of symbolic execution tools often make it possible for users of the tool to write their own summaries and even to add their summaries to the codebase of the tool. 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 and *KLEE*'s summaries are implemented in C. 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 [15] 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* [13] and *AVD* [12] 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. To this end, we also introduce a summary validation tool, built on top of *AVD*, that can systematically evaluate the correctness of C-implemented summaries. The validation tool employs symbolic execution to compare the execution paths modelled by a summary with respect to the execution paths produced by the corresponding concrete reference implementation.

2 Related Work

2.1 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* [2]. 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 restrictions 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.

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* [13] and *KLEE* [4] 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* [8] uses OCaml to generate assembly code comprising a symbolic summary, which is then injected in the addresses of external function calls.

2.1.1 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 instance, symbolic reflection can be used at a conditional statement to determine if a certain branch is feasible according to the preceding path conditions (*eager evaluation* [2]). 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.

2.1.2 Properties of summaries.

Due to the limitations covered in section 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 outputs all and only the correct execution paths. For example when analysing a program with 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* [1, 11]. 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 producing wrong paths. As a device that intrinsically 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 precision for 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 forward sound summaries: if symbolic execution says that there is no security bug, then there is no security bug. In contrast, debugging/testing tools require backward sound 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.

2.2 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 1, which divides the libc summaries into 7 categories.

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.

3 Symbolic Reflection API

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

Table 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	-	Assembly	-	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

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. 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. Most of these tools already provide some of the functions required by our API, albeit with different names.

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. 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 (Figure 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, e.g., the call `summ_is_symbolic(&var,32)` checks if the 32 bit variable *var* is symbolic. 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*).

```
summ_not_implemented_error(char *fname)
summ_maximize(symbolic sym_var, size_t length)
summ_is_symbolic(symbolic sym_var, size_t length)
summ_new_sym_var(int length)
summ_assume(restr_t restr)
_solver_is_it_possible(restr_t restr)
summ_memory_addr(void* addr, void* n, size_t length)
```

Figure 1. Symbolic Reflection API: General Functions

Operations with symbolic variables. The primitives in this category are responsible for manipulating the bit vectors denoting symbolic variables (Figure 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).

```
_solver_Concat(symbolic sym_var, symbolic sym_var2, int length1,
↪  int length2)
_solver_Extract(symbolic sym_var, int start, int end, int length)
_solver_ZeroExt(symbolics sym_var, int to_extend, int length)
_solver_SignExt(symbolic sym_var, int to_extend, int length)
```

Figure 2. Symbolic Reflection API: Operations with symbolic variables

Operations with symbolic restrictions. The primitives in this category are responsible for building restrictions over symbolic variables (Figure 3 shows a partial list of the available operations). These restrictions can then be queried for satisfiability and/or added to the current symbolic state. For instance the primitive call: `_solve_EQ(&a, &b, 32)` builds an equality restriction over two 32 bit symbolic variables such that $a = b$.

```
_solver_NOT(restr_t restr)
_solver_Or(restr_t restr1, restr_t restr2)
_solver_And(restr_t restr1, restr_t restr2)
_solver_EQ(symbolic sym_var, symbolic sym_var2, size_t length)
...
```

Figure 3. Symbolic Reflection API: Operations with symbolic restrictions

To demonstrate the portability of our summaries, we extended two symbolic execution tools: *AVD* [12] and *angr* [13], 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 4.3. 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.

3.1 Summary Example

Figure 4 shows a backward sound summary implemented for libc's *strlen* function. 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.

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] ≠ '\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] ≠ '\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 ≠ '\0'` and `sym2 ≠ '\0'` to the current path condition.

Figure 4. Implementation of summary *strlen2*

```
1  int strlen2(char* s){
2    char charZero = '\0';  int i = 0;
3    while(1){
4      //s[i] is symbolic
5      if(summ_is_symbolic(&s[i],CHAR_SIZE)){
6
7        //Build restriction: s[i] ≠ '\0'
8        restr_t restr = _solver_NEQ(&s[i], &charZero,
         ↪  CHAR_SIZE);
9
10       //Query satisfiability of restr
11       if(!_solver_is_it_possible(restr)) break;
12
13       //Add restr to symbolic state
14       else summ_assume(restr);
15     }
16     else if(s[i] == charZero) break;
17     i++;
18   }
19   return i;
20 }
```

4 Summary Correctness

4.1 Correctness Properties

In this section, we mathematically define the correctness properties required for evaluating our symbolic summaries, some of which were introduced in Section 2.1.2. 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 the set of symbolic values. Furthermore, we use $\hat{\Sigma}$ to denote an any set of pairs of symbolic states and symbolic return values.

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 $[\![.]\!] :: \mathcal{S}ym\mathcal{S}t \rightarrow \mathcal{P}(Conc\mathcal{S}t)$ takes as input a symbolic state and returns a set of concrete states.

Symbolic states as boolean formulas. In order to reason about the correctness properties of a summary, we introduce a lifting operator $\lceil . \rceil :: \mathcal{P}(\mathcal{S}ym\mathcal{S}t) \rightarrow \mathcal{F}ormula$ 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:

$$\lceil \hat{\Sigma} \rceil \equiv \bigvee \left\{ \lceil \hat{\mu} \rceil_m \wedge \pi \wedge (\texttt{ret} = \hat{r}) \;\mid\; (\langle \hat{\mu}, \pi \rangle, \hat{r}) \in \hat{\Sigma} \right\} \quad (1)$$

Essentially, a set of symbolic states is transformed into a disjunction of boolean formulas, each describing the execution path of 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.

Finally, 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:

$$\forall \hat{\sigma} \,.\, \hat{C}(\hat{\sigma}) \rightsquigarrow (\hat{\sigma}', \hat{r}) \implies \forall (\sigma', r) \in (\hat{\sigma}', \hat{r}) \,.\, \exists\, \sigma \in \hat{\sigma} \,.\, C(\sigma) = (\sigma', r) \quad (2)$$

A *backward sound* summary guarantees that, for any symbolic state $\hat{\sigma}$, the interpretation of the 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 on the interpretation of $\hat{\sigma}$. Consequently, for the summary $\hat{C}$ to be *backward sound*, the implication $\hat{\Phi} \Rightarrow \Phi$ must be true.

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:

$$\forall \hat{\sigma}.\, \forall \sigma \in [\![\hat{\sigma}]\!] \;\wedge\; C(\sigma) = (\sigma', r) \;\wedge\; \hat{C}(\hat{\sigma}) \rightsquigarrow (\hat{\sigma}', \hat{r}) \Longrightarrow (\sigma', r) \in [\![(\hat{\sigma}', \hat{r})]\!] \quad (3)$$

A *forward sound* summary guarantees that, for any symbolic state $\hat{\sigma}$, the set of execution paths produced by the concrete execution of C on the interpretation of $\hat{\sigma}$, is contained in the interpretation of the symbolic execution paths generated by the symbolic execution of $\hat{C}$ in $\hat{\sigma}$. Consequently, for the summary $\hat{C}$ to be *forward sound*, the implication $\Phi \Rightarrow \hat{\Phi}$ must be true.

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 (3) and (2). 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 on the interpretation of $\hat{\sigma}$. Consequently, for the summary $\hat{C}$ to be *complete*, the equivalence $\hat{\Phi} \Leftrightarrow \Phi$ must be true.

4.2 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. If this is the case, one can often obtain either soundness or precision if one assumes that the function input satisfies some additional constraints. 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 $\langle \hat{\mu}, \pi \wedge \rho \rangle$.

4.2.1 Generalized Backward Soundness.

A 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:

$$\forall \hat{\sigma} \,.\, \hat{C}(\hat{\sigma} \wedge \rho) \rightsquigarrow (\hat{\sigma}', \hat{r}) \implies \forall (\sigma', r) \in [\![(\hat{\sigma}', \hat{r})]\!],\ \exists\, \sigma \in [\![\hat{\sigma} \wedge \rho]\!] \,.\, C(\sigma) = (\sigma', r) \quad (4)$$

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.

4.2.2 Generalized Forward Soundness.

A 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:

$$\forall \hat{\sigma}. \forall \sigma \in [\![\hat{\sigma} \wedge \rho]\!] \wedge C(\sigma) \rightsquigarrow (\sigma', r) \wedge \hat{C}(\hat{\sigma} \wedge \rho) = (\hat{\sigma}', \hat{r}) \implies (\sigma', r) \in [\![(\hat{\sigma}', \hat{r})]\!] \quad (5)$$

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 $\hat{\sigma} \wedge \rho$.

4.2.3 Generalized Completeness.

A symbolic summary $\hat{C}$ is *complete* with respect to a concrete implementation C and a predicate ρ, if and only if it satisfies both (5) and (4). A summary satisfies this property if the interpretation of the symbolic execution paths generated by the symbolic execution of $\hat{C}$ in $\hat{\sigma} \wedge \rho$ corresponds to the set of execution paths produced by the concrete execution of C in the interpretation of $\hat{\sigma}$ filtered by ρ.

4.3 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 that what might be expected. For this reason, even for seemingly simple symbolic summaries, the tasks of summary debugging and correctness verification, quickly become unfeasible to be carried out manually. Given that our symbolic summaries are implemented in the programming language of analysis (C), they can themselves be symbolically executed as standard C code. To this end, we implemented an auxiliary infrastructure, illustrated in Figure 5, that works on top of the symbolic execution tool *AVD* [12], 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 implications denoting correctness properties (e.g, $\hat{\Phi} \Rightarrow \Phi$). 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.

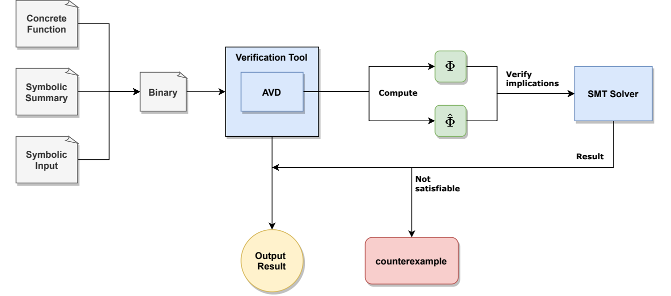

Figure 5. Summary Validation Tool

5 Evaluation

This section answers the following three evaluation questions:

EQ1 What is the time performance overhead of tool independent summaries?
EQ2 Is the symbolic reflection API sufficiently expressive to allow for the writing of backward/forward sound summaries?
EQ3 Can our summary validation tool be used to analyse real-world symbolic summaries developed in the context of other tools?

5.1 EQ1: Time Performance of Tool independent Summaries

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 [12], 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) [7]. These challenges were designed for an automated 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 [17]. 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 sing 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. 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:

$$O = \alpha \, . \, I_C \tag{6}$$

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.

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:

$$G_{(\%)} = \frac{\sum_{i=1}^{n} T_{C_i}}{\sum_{i=1}^{n} T_{P_i}} \times 100 \tag{7}$$

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

5.1.1 CGC Data set. The Challenge Binaries that serve as the test bed for the CTF are tailor-made programs 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.

Due to the large size and complexity of most of the challenge binaries, 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. 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 (6). Results are shown in the scatter plot of Figure 6, 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%.

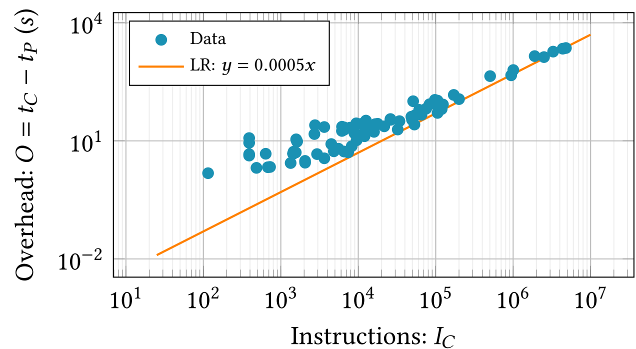

Figure 6. Overhead scatter plot for the GCG dataset.

5.1.2 HashMap Library. To complement the results obtained with the CGC binaries, we obtained a real-world HashMap implemented in C from github [17] 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. We focused on key-manipulating functions, which were tested using symbolic keys instead of concrete ones.

To determine the overhead per instruction for this data set, we executed the 10 symbolic tests using 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 7. 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.

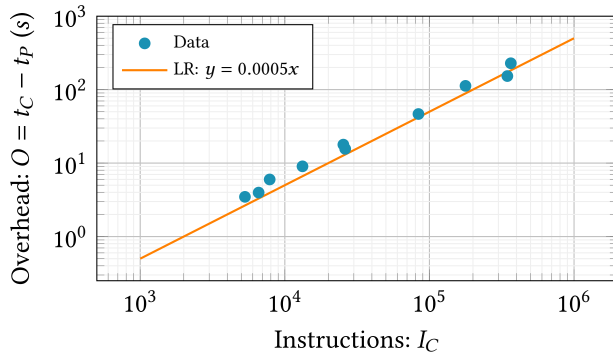

Figure 7. Overhead scatter plot for the Hashmap dataset.

5.2 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. All summaries are named using a number suffix, which allows to organize the summaries by correctness property and order of accuracy.

The correctness properties of all implemented summaries, except those modelling I/O functions, are given in Table 2, where the *N/A* column represents the summaries that only satisfy generalized properties. These are further described in Table 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*).

Table 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

Table 3. Generalized correctness properties of the implemented summaries

	Gen. Backward Soundness	Gen. Forward Soundness	Gen. 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

5.3 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.

5.3.1 Bug in *angr*. The first detected bug occurs in *angr*'s summary for libc's *strncmp* function. In *angr*'s architecture the *strncmp* summary 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.

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:

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

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. According to *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 2, *str1* and *str2*, our validation tool will produce the following counterexamples:

$$\begin{aligned} \text{Missing}: \quad & [str1 = \text{`}aa\text{'} \land\ str2 = \text{`}bb\text{'} \land\ n = 2\ \land\ ret = -1] \\ \text{Wrong}: \quad & [str1 = \text{`}aa\text{'} \land\ str2 = \text{`}bb\text{'} \land\ n = 2\ \land\ ret = 1] \end{aligned}$$

5.3.2 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. Consequently, it does not satisfy any of the standard soundness properties since it does not take into account that the symbolic bytes can also be null characters. For instance, considering two symbolic input strings: $str1$ = '$sym1|sym2|\backslash 0$' and $str2$ = '$sym3|sym4|\backslash 0$', our validation tool will produce the following counterexamples:

$$\begin{aligned} \text{Missing}: & [str1 = \text{`}\backslash 0\,|A|\,\backslash 0\text{'}\ \land\ str2 = \text{`}\backslash 0\,|B|\,\backslash 0\text{'} \land\ ret = 0] \\ \text{Wrong}: & [str1 = \text{`}\backslash 0\,|B|\,\backslash 0\text{'}\ \land\ str2 = \text{`}\backslash 0\,|A|\,\backslash 0\text{'} \land\ ret = 1] \end{aligned}$$

6 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.

6.1 Future Work

For future work we would like to extend our API with primitives to interact with the execution environment of program. This would allow for the implementation of summaries that model system calls such as heap manipulation functions (e.g., *malloc*). Accordingly, we also would need to introduce new correctness properties for evaluating such summaries.

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 summary validation tool with support for it to accept a range of symbolic inputs on which to evaluate the given summary.

References

[1] Krzysztof R. Apt. 1981. Ten Years of Hoare's Logic: A Survey—Part I. *ACM Trans. Program. Lang. Syst.* 3, 4 (Oct. 1981), 431–483. https://doi.org/10.1145/357146.357150

[2] Roberto Baldoni, Emilaio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (May 2018), 39 pages. https://doi.org/10.1145/3182657

[3] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT—a Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software* (Los Angeles, California). Association for Computing Machinery, New York, NY, USA, 234–245. https://doi.org/10.1145/800027.808445

[4] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. 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* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[5] Marco Carvalho, Jared Demott, Richard Ford, and David Wheeler. 2014. Heartbleed 101. *Security & Privacy, IEEE* 12 (07 2014), 63–67. https://doi.org/10.1109/MSP.2014.66

[6] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E Platform: Design, Implementation, and Applications. *ACM Transactions on Computer Systems - TOCS* 30 (02 2012), 1–49. https://doi.org/10.1145/2110356.2110358

[7] DARPA. 2015. The Cyber Grand Challenge. https://www.darpa.mil/program/cyber-grand-challenge Accessed: November 27, 2021.

[8] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M. Potet, and J. Marion. 2016. BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis, In 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* 1, 653–656. https://doi.org/10.1109/SANER.2016.43

[9] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. 2014. The Matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (Vancouver, BC, Canada) *(IMC '14)*. Association for Computing Machinery, New York, NY, USA, 475–488. https://doi.org/10.1145/2663716.2663755

[10] James C. King. 1975. A New Approach to Program Testing. *SIGPLAN Not.* 10, 6 (April 1975), 228–233. https://doi.org/10.1145/390016.808444

[11] Peter O'Hearn. 2019. Incorrectness logic. *Proceedings of the ACM on Programming Languages* 4 (12 2019), 1–32. https://doi.org/10.1145/3371078

[12] Nuno Sabino. 2019. *Automatic Vulnerability Detection: Using Compressed Execution Traces to Guide Symbolic Execution.* Master's thesis. Instituto Superior Técnico.

[13] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. 138–157. https://doi.org/10.1109/SP.2016.17

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

[15] Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 135–152. https://doi.org/10.1145/2509578.2509586

[16] Nicholas Wells. 2000. BusyBox: A Swiss Army Knife for Linux. *Linux Journal* 2000 (01 2000), 10.

[17] Richard Wiedenhöft. 2014. C Hash map. https://gist.github.com/Richard-W/9568649 Accessed: November 27, 2021.