Toward Tool-Independent Summaries for Symbolic Execution

Frederico Ramos
Instituto Superior Técnico, University of Lisbon, Portugal
INESC-ID Lisbon, Portugal

Nuno Sabino
Instituto Superior Técnico, University of Lisbon, Portugal
Carnegie Mellon University, Pittsburgh, PA, USA
Institute of Telecommunications, Campus de Santiago, Aveiro, Portugal

Pedro Adão
Instituto Superior Técnico, University of Lisbon, Portugal
Institute of Telecommunications, Campus de Santiago, Aveiro, Portugal

David A. Naumann
Stevens Institute of Technology, Hoboken, NJ, USA

José Fragoso Santos
Instituto Superior Técnico, University of Lisbon, Portugal
INESC-ID Lisbon, Portugal

Abstract

We introduce a new symbolic reflection API for implementing tool-independent summaries for the symbolic execution of C programs. We formalise the proposed API as a symbolic semantics and extend two state-of-the-art symbolic execution tools with support for it. Using the proposed API, we implement 67 tool-independent symbolic summaries for a total of 26 LIBC functions. Furthermore, we present SUMBOUNDVERIFY, a fully automatic summary validation tool for checking the bounded correctness of the symbolic summaries written using our symbolic reflection API. We use SUMBOUNDVERIFY to validate 37 symbolic summaries taken from 3 state-of-the-art symbolic execution tools, *angr*, *Binsec* and *Manticore*, detecting a total of 24 buggy summaries.

2012 ACM Subject Classification Software and its engineering → Software verification and validation; Security and privacy → Formal methods and theory of security

Keywords and phrases Symbolic Execution, Runtime Modelling, Symbolic Summaries

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.24

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.9.2.7`

Funding The authors were supported by Fundação para a Ciência e a Tecnologia (UIDB/50008/2020, Instituto de Telecomunicações, and UIDB/50021/2020, INESC-ID multi-annual funding, and PhD grant SFRH/BD/150692/2020), project DIVINA (CMU/TIC/0053/2021), the SmartRetail project (C6632206063-00466847) financed by IAPMEI, the European Commission under grant agreement number 830892 (SPARTA), and the NSF award CNS-1718713.

1 Introduction

Symbolic execution [14, 34] 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 the 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 follow that path. Symbolic execution engines rely on an underlying SMT solver [20, 9] to check the feasibility of execution paths

and the validity of any assertions supplied by the developer. Despite being extensively used in practice [15, 26, 7, 54], symbolic execution suffers from two main limitations when applied to real-world code: interactions with the runtime environment (e.g. file system, network, operating system) and path explosion. An effective approach to deal with these two issues is to use symbolic summaries to model the behaviour of both external runtime functions as well as internal functions with a high level of branching [6].

Symbolic summaries constrain the symbolic state of the given program so as to simulate the behaviour of the modelled functions without having to symbolically execute them. The idea is that instead of symbolically executing the code of a given concrete function on some symbolic inputs, one implements a symbolic summary that models the behaviour of that function, and then executes the summary instead of the concrete function. Importantly, symbolic summaries allow developers to merge different symbolic execution paths into a single path by explicitly interacting with the current symbolic state [6, 52]. Hence, they provide an effective mechanism for containing the number of paths to be explored during symbolic execution, allowing developers to mitigate the effect of the path explosion problem.

When writing a symbolic summary, tool developers must carefully construct the symbolic state that properly captures the outcome of all the execution paths that the summary is supposed to model. They do this by directly interacting with the various elements of the given symbolic state using symbolic reflection mechanisms [6, 52]. This is often a challenging task that is both error prone and difficult to validate. For this reason, most symbolic execution tools for C have very limited support for external functions and commonly used library functions, such as those of the Standard C Library (LIBC).

State-of-the-art symbolic execution tools [50, 15, 38] come with their own symbolic summaries implemented in the programming languages used to build each tool. For instance, *angr*'s [50] summaries are implemented in Python, *KLEE*'s [15] summaries in C, and *BINSEC*'s [19] summaries in OCaml, even though all these tools target C code. These summaries often rely on specific aspects of the tools for which they were implemented, making it extremely difficult to share summaries between different symbolic execution tools. Surprisingly, and 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. The current state of affairs is, however, dire: even though the Standard C Library (LIBC) includes more than one thousand functions, the symbolic execution tool with the broadest support for LIBC is *angr* [50], with only 128 unverified symbolic summaries. This situation is made considerably worse by the fact that even the few existing summaries are written manually and not verified, potentially compromising the correctness and coverage guarantees of their corresponding symbolic execution tools.

In this paper, we introduce a new symbolic reflection API for the implementation of tool-independent symbolic summaries for the C programming language. The proposed API consists of a set of symbolic reflection primitives [52] for the 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 path condition of the current symbolic state with a given constraint. Symbolic summaries implemented using our API are written directly in C and can therefore be shared across different symbolic execution tools, provided that these tools implement the proposed API. Importantly, the goal of our API is not to make symbolic summaries simpler or easier to write, but rather to establish a symbolic reflection interface shared by all symbolic execution tools, allowing for the decoupling of symbolic summaries from

the internal details of each tool. To illustrate the applicability of our API, we have extended the symbolic execution tools *angr* [50] and *AVD* [45] with support for it and developed 67 tool-independent symbolic summaries for a total of 27 LIBC functions, including string-manipulating, number-parsing, input/output, and heap-manipulating functions. Furthermore, we formalised our symbolic reflection API as a symbolic semantics [52, 21] and used this semantics to formally characterise the correctness properties that symbolic summaries are expected to observe, specifically, over- and/or under-approximation.

Leveraging our symbolic reflection API, we developed SUMBOUNDVERIFY, a new fully-automated tool for the bounded verification of symbolic summaries. SUMBOUNDVERIFY works by comparing the execution paths modelled by the given summary against those generated by symbolically executing its corresponding concrete function up to the chosen bound. In order to assess the effectiveness of SUMBOUNDVERIFY, we used it to verify summaries belonging to three state-of-the-art symbolic execution tools: *angr* [50], *Binsec* [19, 39, 18], and *Manticore* [38]. Out of the 37 analysed summaries, 24 were flagged as buggy, clearly demonstrating the need for tool support when it comes to designing and implementing symbolic summaries. This need is further confirmed by our own experience in the development of symbolic summaries, which we typically found to be highly complex and error-prone. This paper bridges this gap by providing the first verification tool specifically aimed at the development of correct symbolic summaries.

In summary, the contributions of this paper are the following: **(1)** a formally defined API for developing symbolic summaries for C; **(2)** a library of 67 symbolic summaries modelling 26 LIBC functions; and **(3)** SUMBOUNDVERIFY, an automatic bounded verification tool for validating symbolic summaries against their corresponding concrete implementations.

2 Overview

In this section, we first contrast the existing methodology for implementing symbolic summaries with our proposed approach (§2.1) and then give a high-level overview of SUMBOUNDVERIFY (§2.2), illustrating how it can be used to verify symbolic summaries.

2.1 Tool-Specific vs. Tool-Independent Symbolic Summaries

A symbolic summary is an operational model of a function that simulates its behaviour by interacting directly with the underlying symbolic state. So far, each symbolic execution tool for C comes with its own summaries directly implemented in the programming language used to build the tool. Existing symbolic summaries are therefore tightly connected to the architecture of their corresponding tools, preventing summaries from being shared between different tools. In order to cater for the reuse of symbolic summaries, we propose an alternative approach: *Symbolic summaries are to be directly implemented in C using a shared symbolic reflection API for direct manipulation of symbolic states at the programming language level.*

To illustrate the difference between tool-specific and tool-independent summaries, we compare Manticore's [38] symbolic summary for `strlen` (Figure 1) with the equivalent summary written directly in C using our API (Figure 2).

Manticore's Tool-Specific Summary

Figure 1 shows Manticore's summary for the function `strlen`. This summary first checks if the given string pointer is itself symbolic, in which case it throws an error (lines 3-4). Then, the summary uses the Manticore's internal function `find_zero` to determine the index of the

```python
def strlen_approx(state: State, s: Union[int, BitVec]) -> Union[int, BitVec]:

    if issymbolic(s):
        raise ConcretizeArgument(state.cpu, 1)

    #Find max string length
    cpu = state.cpu
    zero_idx = _find_zero(cpu, state, s)
    ret = zero_idx

    #Build nested ITE formula
    for offset in range(zero_idx - 1, -1, -1):
        byt = cpu.read_int(s + offset, 8)
        if issymbolic(byt):
            ret = ITEBV(cpu.address_bit_size, byt == 0, offset, ret)

    return ret
```

Figure 1 Implementation of Manticore's `strlen` summary.

first concrete null character in the input string, `zero_idx`. Note that if the string does not contain any symbolic character, `zero_idx` coincides with the length of the string. In the final for-loop, the summary iterates over the characters of the given string to construct a symbolic expression denoting its length. For instance, given the symbolic string `[c0, c1, c2, \0]`, the loop will generate the expression:

$$\texttt{ret} = \texttt{ITE}(\texttt{c0} == \texttt{\textbackslash 0}, 0, \overline{\texttt{ITE}(\texttt{c1} == \texttt{\textbackslash 0}, 1, \underline{\texttt{ITE}(\texttt{c2} == \texttt{\textbackslash 0}, 2, 3)})})$$

signifying that: if the first character (`c0`) is null, then the return value is 0; if the second character (`c1`) is null, then the return value is 1; if the third character (`c2`) is null, then the return value is 2; otherwise, the return value is 3. Note that the overlines have no semantic meaning, being only there to facilitate the reading.

Tool-Independent Summary

Figure 2 shows our equivalent C implementation of Manticore's summary for `strlen`. Although both summaries implement approximately the same logic,[1] our summary is written directly in C using our symbolic reflection API. It uses the following primitives: **(1)** `is_symbolic(x)` to check if variable `x` denotes a symbolic value; **(2)** `new_sym_var(size)` to create a new symbolic variable to represent a value of size `size`; **(3)** `_solver_EQ(a, b)` to build a constraint stating that the two given values are equal; **(4)** `assume(c)` to add the constraint `c` to the path condition of the current symbolic state; and **(5)** the primitive `_solver_IF(c, a, b)` to build an if-then-else symbolic expression of the form $\texttt{ITE}(\texttt{c}, \texttt{a}, \texttt{b})$.

Why use symbolic summaries?

To better understand the benefits of symbolic summaries, let us consider the symbolic execution of the concrete implementation of `strlen` on the symbolic string `[c0, c1, c2, \0]`. That execution would generate four execution paths, each corresponding to one of the possible outputs. In contrast, the execution of either of the symbolic summaries described above generates a single execution path representing all four outputs. Both symbolic summaries

[1] The C summary assumes that it is never given a symbolic pointer as input.

```
size_t strlen(char* s){
    int i = 0;
    char char_zero = '\0';

    //Calculate max string length
    while(is_symbolic(&s[i]) || s[i] != '\0'){
        i++;
    }
    int len = i;
    symbolic ret = new_sym_var(INT_SIZE);
    cnstr_t ret_cnstr = _solver_EQ(&ret, &len, INT_SIZE);

    //Build nested ITE constraint
    for(i = len-1; i >= 0; i--){

        if(is_symbolic(&s[i])){

            cnstr_t c_eq_zero = _solver_EQ(&s[i], &char_zero, CHAR_SIZE);
            cnstr_t ret_eq_i = _solver_EQ(&ret, &i, INT_SIZE);

            ret_cnstr = _solver_IF(c_eq_zero, ret_eq_i, ret_cnstr);
        }
    }
    assume(ret_cnstr);
    return ret;
}
```

Figure 2 Implementation of *Manticore*'s `strlen` in C.

achieve this by directly extending the path condition of the calling state with a formula that constrains the return value appropriately, depending on the symbolic characters appearing in the given string.

2.2 Bounded Verification of Symbolic Summaries

In this section we show how SumBoundVerify can be used to verify the symbolic summary of `strlen` given above. We support two flavours of correctness properties: *under-approximating* [40] and *over-approximating* [5]. A summary is *under-approximating* if all the execution paths modelled by the summary are contained in the set of concrete paths of its corresponding function. In other words, an *under-approximating* summary guarantees that all its generated paths have corresponding concrete paths. Conversely, a symbolic summary is *over-approximating* if it models all the concrete paths of its corresponding function; that is, an over-approximating summary must take into account all possible concrete paths. If a summary is both under- and over-approximating, we say that it is *exact*, following recent terminology in the context of separation-logic-based verification [37]. Naturally, the type of correctness property to be aimed at depends on how the summary is going to be used. For instance, over-approximating summaries are essential for security applications that must guarantee the absence of security vulnerabilities; in contrast, under-approximating summaries may be a better fit for debugging tools that aim at reporting only real bugs.

Bounded Verification

Let us now take a closer look at the inner workings of SumBoundVerify. In a nutshell, SumBoundVerify requires the developer to provide the summary to be verified, its corresponding concrete implementation, and an integer bound on the size of its parameters; for instance, for inputs of array type, this bound corresponds to the maximum length of

```
1  int main(){
2
3   char s[4];
4   for (int i = 0; i < 2; i++){
5    s[i] = new_sym_var_array("c", i, CHAR_SIZE);
6   }
7   s[3] = '\0';
8
9   state_t fresh_state = save_current_state();
10
11  int ret1 = concrete_strlen(s);
12  cnstr_t c1 = get_cnstr(&ret1, INT_SIZE);
13  store_cnstr("cncrt", c1);
14
15  switch_state(fresh_state);
16
17  int ret2 = summ_strlen(s);
18  cnstr_t c2 = get_cnstr(&ret2, INT_SIZE);
19  store_cnstr("summ", c2);
20
21  result_t res = check_implications("summ", "cncrt");
22  print_counterexamples(res);
23  return 0;
24 }
```

(a) Test code.

Reference Implementation Formula:

$(c_0 = \backslash 0) \wedge (ret = 0) \vee$
$(c_0 \neq \backslash 0) \wedge (c_1 = \backslash 0) \wedge (ret = 1) \vee$
$(c_0 \neq \backslash 0) \wedge (c_1 \neq \backslash 0) \wedge (c_2 = \backslash 0) \wedge (ret = 2) \vee$
$(c_0 \neq \backslash 0) \wedge (c_1 \neq \backslash 0) \wedge (c_2 \neq \backslash 0) \wedge (c_3 = \backslash 0)$
$\wedge (ret = 3)$

Symbolic Summary Formula:

$ITE(c_0 = \backslash 0, r = 0,$
$\quad ITE(c_1 = \backslash 0, r = 1,$
$\quad\quad ITE(c_2 = \backslash 0, r = 2, 3)) \wedge ret = r$

(b) Generated formulas.

Figure 3 Bounded Verification of the `strlen` summary given in Figure 2.

the array. Given the signature of the summarised function, SumBoundVerify synthesises a set of symbolic tests to check the correctness of the given summary. These tests can be executed by any symbolic execution tool that implements our reflection API. If the summary passes all the generated tests, then it is correct up to the pre-established bound. If it does not, then SumBoundVerify generates a concrete input that is not correctly modelled by the summary.

Figure 3 illustrates one of the tests generated for the `strlen` summary discussed in §2.1, assuming that the developer specified bound 3. The test first creates an array of size 4, initialises the first 3 characters to new symbolic characters, and sets the fourth element of the array to be the null character (lines 3-7). Then, the test uses the API function `save_current_state` to save the current symbolic state (line 9). Next, the test calls the concrete `strlen` function on the created symbolic string and stores the generated return values and final path conditions for future reference (lines 12-14). Then, the test re-establishes the symbolic state saved in line 10 by calling the API function `switch_state` (line 15), which simply replaces the current symbolic state with the given symbolic state. Having re-established the original symbolic state, the test calls the summary on the input string and stores the generated return values and final path conditions (lines 17-19). Finally, the test compares the return values and path conditions generated by the summary against those generated by the concrete function.

A summary can be classified as being: under-approximating correct, over-approximating correct, or incorrect. In a nutshell, the two correct cases are checked as follows:

- *Under-approximating:* the formula describing the final state resulting from the symbolic execution of the summary implies the formula describing the final state resulting from the symbolic execution of its reference implementation;
- *Over-approximating:* the formula describing the final state resulting from the symbolic execution of the reference implementation implies the formula describing the final state resulting from the symbolic execution of the summary.

The `strlen` summary given in Figure 2 is exact (i.e., both under- and over-apaproximating). Figure 3b shows the formulas generated by the execution of both the reference implementation and the summary. As the summary is exact, the solver can check both implications.

3 Symbolic Reflection API

We formally define the semantics of our API for developing symbolic summaries (§3.1) and use this semantics to characterise the correctness properties that symbolic summaries are expected to observe (§3.2). Then, we illustrate how our reflection API can be used to develop symbolic summaries for string-manipulating and number-parsing LIBC functions (§3.3).

3.1 Formal Semantics

We define the formal semantics of our Symbolic Reflection API on top of a core C-like language in the style of the core language used in [41], which we extend with our symbolic reflection primitives. Importantly, and in order not to clutter the formalism with unnecessary technical details, the formal model is a simplified version of our proposed API. The syntax of the language is given in the table below.

Syntax

e	$:=$	$n \mid x \mid \ominus(e) \mid \oplus(e_1, e_2) \mid \otimes(e_1, e_2, e_3)$
$\hat{s}$	$:=$	$x \coloneqq e \mid \texttt{skip} \mid \hat{s}_1; \hat{s}_2 \mid \texttt{if}(e)\{\hat{s}_1\}\texttt{else}\{\hat{s}_2\} \mid \texttt{while}(e)\{\hat{s}\} \mid \texttt{return}\ e$ $\mid x \coloneqq e_1[e_2] \mid e_1[e_2] \coloneqq e_3 \mid x \coloneqq \texttt{new}(e) \mid rs$
rs	$:=$	$\texttt{assert}(e) \mid \texttt{assume}(e) \mid x \coloneqq \texttt{is\_symbolic}(e) \mid x \coloneqq \texttt{symb()} \mid x \coloneqq \texttt{is\_sat}(e)$ $\mid x \coloneqq \texttt{maximize}(e) \mid x \coloneqq \texttt{minimize}(e) \mid x \coloneqq \texttt{cur\_pc()} \mid x \coloneqq \texttt{eval(e)}$ $\mid x \coloneqq \texttt{block\_size}(e) \mid x \coloneqq \texttt{constr}_{uop}(e) \mid x \coloneqq \texttt{constr}_{bop}(e_1, e_2)$
bop	$:=$	$\texttt{or} \mid \texttt{and} \mid \texttt{eq} \mid \texttt{neq} \mid \texttt{lt} \mid \texttt{le}$
uop	$:=$	$\texttt{not}$

Expressions $e \in Expr$ include integers n, program variables x, unary, binary, and ternary operators. Statements $\hat{s} \in Stmt$ include: **(i)** the typical imperative statements, i.e. variable assignment, skip, sequence, if, while, and return statements; **(ii)** statements for interaction with a linear memory, and **(iii)** symbolic reflection primitives $rs \in \mathcal{RS}$. In the following we use $\hat{s}$ for statements that may include reflection primitives and s for statements that do not. Accordingly, we use $\hat{s}$ for symbolic summaries and s for reference implementations.

The statements for memory interaction are the following: **(1)** the statement $x \coloneqq e_1[e_2]$ assigns to x the value stored in the memory block denoted by e_1 at the offset denoted by e_2; **(2)** the statement $e_1[e_2] \coloneqq e_3$ stores the value denoted by e_3 in the memory block denoted by e_1 at the offset denoted by e_2; and **(3)** the statement $x \coloneqq \texttt{new}(e)$ creates a memory block with the size denoted by e and assigns the obtained pointer to x.

The symbolic reflection primitives $rs \in \mathcal{RS}$ are the following: **(1)** $\texttt{assert}(e)$ to check if the current path condition implies the constraint denoted by e; **(2)** $\texttt{assume}(e)$ to extend the current path condition with the constraint denoted by e; **(3)** $x \coloneqq \texttt{is\_symbolic}(e)$ to assign to variable x a boolean value indicating if e denotes a symbolic expression; **(4)** $x \coloneqq \texttt{symb()}$ to assign a fresh symbolic value to x; **(5)** $x \coloneqq \texttt{is\_sat}(e)$ to check if the constraint denoted by e is satisfiable when conjoined with the current path condition; **(6)** $x \coloneqq \texttt{maximize}(e)$ to assign the largest possible value that may be denoted by e to x; **(7)** $x \coloneqq \texttt{minimize}(e)$ to assign the smallest possible value that may be denoted by e to x; **(8)** $x \coloneqq \texttt{cur\_pc()}$ to assign the formula denoting the current path condition to x; **(9)** $x \coloneqq \texttt{eval}(e)$ to assign one of the concrete values denoted by e to x; **(10)** $x \coloneqq \texttt{block\_size}(e)$ to assign the size of the memory block pointed to by e to x; **(11)** $x \coloneqq \texttt{constr}_{uop}(e)$ to assign the constraint resulting from the application of the logical unary operator uop to the symbolic value denoted by e to x; and **(12)** $x \coloneqq \texttt{constr}_{bop}(e_1, e_2)$ to assign the constraint resulting from the application of the logical binary operator bop to the symbolic values denoted by e_1 and e_2.

$$\textsc{New}\quad \frac{[\![e]\!]_{\hat\rho} = n \quad \texttt{alloc}(\hat\mu, n) = \langle \hat\mu', l\rangle \quad \hat\rho' = \hat\rho[x \mapsto l]}{\hat\mu, \hat\rho,\ x := \texttt{new}(e) \rightsquigarrow \hat\mu', \hat\rho', \mathbb{C}\langle\cdot\rangle}$$

$$\textsc{Load}\quad \frac{[\![e_1]\!]_{\hat\rho} = l \quad [\![e_2]\!]_{\hat\rho} = \hat o \quad \texttt{load}(\hat\mu, l, \hat o) \rightsquigarrow_s \mathbb{S}\langle \hat v, \pi'\rangle \quad \hat\rho' = \hat\rho[x \mapsto \hat v] \quad \pi'' = \pi \wedge \pi'}{\hat\mu, \hat\rho, \pi,\ x := e_1[e_2] \rightsquigarrow \hat\mu, \hat\rho', \pi'', \mathbb{C}\langle\cdot\rangle}$$

$$\textsc{Store}\quad \frac{[\![e_1]\!]_{\hat\rho} = l \quad [\![e_2]\!]_{\hat\rho} = \hat o \quad [\![e_3]\!]_{\hat\rho} = \hat v \quad \texttt{store}(\hat\mu, l, \hat o, \hat v) \rightsquigarrow \mathbb{S}\langle \hat\mu', \pi'\rangle}{\hat\mu, \hat\rho, \pi,\ e_1[e_2] := e_3 \rightsquigarrow \hat\mu', \hat\rho, \pi \wedge \pi', \mathbb{C}\langle\cdot\rangle}$$

$$\textsc{Skip}\quad \frac{}{\texttt{skip} \rightsquigarrow \mathbb{C}\langle\cdot\rangle}$$

$$\textsc{Return}\quad \frac{[\![e]\!]_{\hat\rho} = \hat v}{\hat\rho, \texttt{return}\ e \rightsquigarrow \hat\rho, \mathbb{R}\langle\hat v\rangle}$$

$$\textsc{If-True}\quad \frac{[\![e]\!]_{\hat\rho} = \hat v \quad \pi' = \pi \wedge (\hat v = true)}{\hat\rho, \pi, \texttt{if}(e)\,\{\hat s_1\}\,\texttt{else}\,\{\hat s_2\} \rightsquigarrow \hat\rho, \pi', \mathbb{C}\langle\hat s_1\rangle}$$

$$\textsc{If-False}\quad \frac{[\![e]\!]_{\hat\rho} = \hat v \quad \pi' = \pi \wedge (\hat v = false)}{\hat\rho, \pi, \texttt{if}(e)\,\{\hat s_1\}\,\texttt{else}\,\{\hat s_2\} \rightsquigarrow \hat\rho, \pi', \mathbb{C}\langle\hat s_2\rangle}$$

$$\textsc{Assignment}\quad \frac{[\![e]\!]_{\hat\rho} = \hat v}{\hat\rho,\ x := e \rightsquigarrow \hat\rho[x \mapsto \hat v], \mathbb{C}\langle\cdot\rangle}$$

$$\textsc{Sequence - Cont}\quad \frac{\hat\mu, \hat\rho, \pi, \hat s_1 \rightsquigarrow \hat\mu', \hat\rho', \pi', \mathbb{C}\langle\hat s_1'\rangle}{\hat\mu, \hat\rho, \pi,\ \hat s_1;\ \hat s_2 \rightsquigarrow \hat\mu', \hat\rho', \pi',\ \mathbb{C}\langle\hat s_1';\ \hat s_2\rangle}$$

$$\textsc{Sequence - Empty}\quad \frac{\hat\mu, \hat\rho, \pi, \hat s_1 \rightsquigarrow \hat\mu', \hat\rho', \pi', \mathbb{C}\langle\cdot\rangle}{\hat\mu, \hat\rho, \pi,\ \hat s_1;\ \hat s_2 \rightsquigarrow \hat\mu', \hat\rho', \pi',\ \mathbb{C}\langle\hat s_2'\rangle}$$

$$\textsc{While}\quad \texttt{while}(e)\,\{\hat s\} \rightsquigarrow \mathbb{C}\langle\texttt{if}(e)\,\{\hat s, \texttt{while}(e)\,\{\hat s\}\}\,\texttt{else}\,\{\texttt{skip}\}\rangle$$

Figure 4 Core Semantics: Imperative Fragment.

Symbolic Execution – Trace Semantics

The symbolic semantics of our core language operates on symbolic states, which store symbolic values given by the grammar: $\hat v \in \hat{\mathcal{V}} ::= n \mid \hat x \mid \ominus(\hat v) \mid \oplus(\hat v, \hat v) \mid \otimes(\hat v, \hat v, \hat v)$. Symbolic values include: integers n, symbolic variables $\hat x \in \hat{\mathcal{X}}$, and unary, binary, and ternary operators, respectively ranged by $\ominus$, $\oplus$, and $\otimes$. Furthermore, we use $\pi \in \Pi$ to range over symbolic values of type Boolean. Symbolic states $\hat\sigma \in \mathcal{SymSt}$ are composed of:

- a *symbolic heap*, $\hat\mu : \mathbb{N} \rightharpoonup \hat{\mathcal{V}} \times \mathbb{N} \times \mathbb{N}$, mapping integer pointers $l \in \mathbb{N}$ to triples of the form $(\hat v, k_l, k_r)$, where $\hat v$ denotes the symbolic value stored at location l and k_l and k_r respectively denote the number of cells that can be accessed to the left and to the right of l using l as the accessing pointer;
- a *symbolic store*, $\hat\rho : \mathcal{X} \rightharpoonup \hat{\mathcal{V}}$, mapping program variables to symbolic values; and
- a *path condition*, $\pi \in \Pi$, keeping track of the constraints on which the current symbolic execution branched so far.

Note that our symbolic execution model requires heap locations to always be concrete; hence, we take the domain of symbolic heaps to be the set of naturals, $\mathbb{N}$, rather than that of symbolic values, $\hat{\mathcal{V}}$. Symbolic heaps, as concrete heaps, are organised in *blocks*, with each block being a sequence of memory locations; blocks can be univocally identified by their first location, which is referred to as the *head of the block* (i.e., l is the head of a block in $\hat\mu$, if $\hat\mu(l) = (-, 0, -)$, meaning that one cannot access any location before l within its block). Furthermore, we assume heaps to be *well-formed*, meaning that ranges of adjacent locations are consistent with each other; put formally, a heap $\hat\mu$ is said to be *well-formed* if and only if:

$$\forall l \in \texttt{dom}(\hat\mu).\ \hat\mu(l) = (-, 0, k_r) \implies \forall 0 \le i < k_r.\ \hat\mu(l{+}i) = (-, i, k_r{-}i)$$
$$\forall l \in \texttt{dom}(\hat\mu).\ \hat\mu(l) = (-, k_l, k_r) \implies \hat\mu(l - k_l) = (-, 0, k_l + k_r)$$

$$\textsc{Alloc}\quad \frac{l = |\mathtt{dom}(\hat{\mu})| \qquad \hat{\mu}' = \hat{\mu}[l{+}i \mapsto (0, n{-}i, i) \mid 0 \leq i < n]}{\mathtt{alloc}(\hat{\mu}, n) \triangleq \langle \hat{\mu}', l \rangle}$$

$$\textsc{Block-size}\quad \frac{\hat{\mu}(l) = (-, -, k_r)}{\mathtt{b\_size}(\hat{\mu}, l) \triangleq k_r}$$

$$\textsc{Load - In Bounds}\quad \frac{\hat{\mu}(l) = (-, k_l, k_r) \qquad -k_l \leq i < k_r \qquad \pi' = \hat{o} = i \qquad \hat{\mu}(l+i) = (\hat{v}, -, -)}{\mathtt{load}(\hat{\mu}, l, \hat{o}) \rightsquigarrow_s \mathbb{S}\langle \hat{v}, \pi' \rangle}$$

$$\textsc{Store - In Bounds}\quad \frac{\hat{\mu}(l) = (-, k_l, k_r) \qquad -k_l \leq i < k_r \qquad \hat{\mu}(l+i) = (-, k_l', k_r') \qquad \hat{\mu}' = \hat{\mu}[l{+}i \mapsto (\hat{v}, k_l', k_r')]}{\mathtt{store}(\hat{\mu}, l, \hat{o}, \hat{v}) \rightsquigarrow_s \mathbb{S}\langle \hat{\mu}', \pi' \rangle}$$

Figure 5 Core Semantics: Memory Actions.

In order to define the symbolic semantics of our core language, we make use of computation *outcomes* [21, 36], which capture the flow of execution and are generated by the following grammar: $\hat{o} \in \hat{\mathcal{O}} ::= \mathbb{C}\langle \hat{s} \rangle \mid \mathbb{C}\langle \cdot \rangle \mid \mathbb{R}\langle \hat{v} \rangle \mid \mathbb{E}\langle \pi \rangle$. We make use of four types of outcomes: **(1)** the non-empty continuation outcome $\mathbb{C}\langle \hat{s} \rangle$, signifying that the execution of the current statement generated a new statement to be executed next; **(2)** the empty continuation outcome $\mathbb{C}\langle \cdot \rangle$, signifying that the execution may proceed to the next instruction; **(3)** the return continuation outcome $\mathbb{R}\langle \hat{v} \rangle$, signifying that the current execution terminated with return value $\hat{v}$; and **(4)** the error outcome $\mathbb{E}\langle \pi \rangle$, signifying that the current execution generates an error.

We define the symbolic semantics of our core language using a semantic judgement of the form: $\hat{\sigma}, s \rightsquigarrow \hat{\sigma}', \hat{o}$, meaning that the symbolic evaluation of statement s in the state $\hat{\sigma}$ generates the state $\hat{\sigma}'$ and outcome $\hat{o}$. We splice the components of the state into the semantic transition, simply writing $\langle \hat{\mu}, \hat{\rho}, \pi, s \rangle \rightsquigarrow \langle \hat{\mu}', \hat{\rho}', \pi', \hat{o} \rangle$ when $\hat{\sigma} = (\hat{\mu}, \hat{\rho}, \pi)$ and $\hat{\sigma}' = (\hat{\mu}', \hat{\rho}', \pi')$. The semantic rules are given in Figures 4 and 6, where the former focus on the core language and the latter on symbolic reflection API. Due to space constraints, we omit all error-generating transitions with the exception of those describing errors generated by API primitives. We further omit the elements of the configuration that are neither updated nor inspected by the current rule, writing, for instance, $\hat{\rho}, s \rightsquigarrow \hat{\rho}', \hat{o}$ to mean $\langle \hat{\mu}, \hat{\rho}, \pi, s \rangle \rightsquigarrow \langle \hat{\mu}, \hat{\rho}', \pi, \hat{o} \rangle$. Note that the semantics is non-deterministic, meaning the symbolic execution of a statement on a given state may generate multiple states and continuations.

The symbolic semantics of the imperative fragment is straightforward. It makes use of the auxiliary function **alloc** and relations **load** and **store** for interacting with the linear memory; their meanings are as follows:

- $\mathtt{alloc}(\hat{\mu}, n)$ allocates a new memory block of size n;
- $\mathtt{load}(\hat{\mu}, l, \hat{o}) \rightsquigarrow_s \mathbb{S}\langle \hat{v}, \pi' \rangle$ successfully loads the value $\hat{v}$ stored at the offset $\hat{o}$ of location l in memory $\hat{\mu}$; as the loading operation may cause the execution to branch, it additionally generates a new formula π' with the conditions that must hold for the symbolic value $\hat{v}$ to be returned;
- $\mathtt{store}(\hat{\mu}, l, \hat{o}, \hat{v}) \rightsquigarrow \mathbb{S}\langle \hat{\mu}', \pi' \rangle$ successfully stores the symbolic value $\hat{v}$ at the offset $\hat{o}$ of location l in memory $\hat{\mu}$ under the path condition π, returning a new memory $\hat{\mu}'$; as for the loading operation, the storing operation may cause the execution to branch, hence the returned constraint π'.

In the above, we use the symbol $\mathbb{S}$ to distinguish the successful transitions of **load** and **store** from their error-leading transitions, which are labelled with $\mathbb{E}$.

The definitions of **alloc**, **load**, and **store** are given in Figure 5. In contrast to **load** and **store** which may cause the current execution to branch, **alloc** is assumed to be deterministic. Hence, it is modelled as a function. Load and store operations *fail* if they attempt to read/update the contents of a memory cell beyond the bounds of the inspected

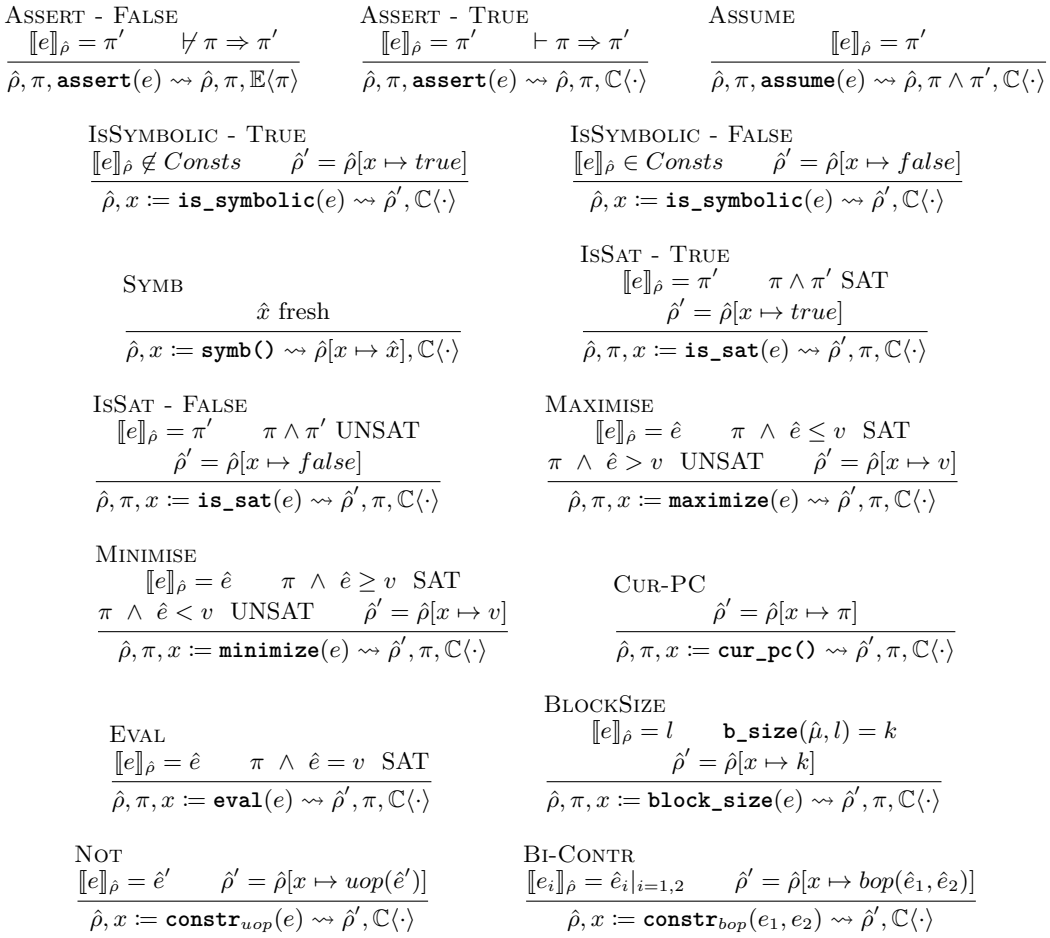

Figure 6 Core Semantics: Symbolic Reflection API.

location (the failing rules are omitted due to space constraints). In the success case, both load and store operations branch on the value of all legal offsets. This means that these operations may generate unsatisfiable path conditions (for instance, when the given offset is concrete), in such cases the symbolic execution path is unfeasible and will be filtered out by symbolic execution.

Finally, Figure 6 gives the rules that describe the semantics of our proposed API. The rules are straightforward. Note that constraints are simply symbolic values of boolean type; hence, various rules either assign constraints to variables (e.g. Cur-Pc) or obtain a constraint as the result of symbolically evaluating an expression (e.g. Assert, Assume, IsSat).

Symbolic Execution – Collecting Semantics

So far, we have defined the semantics of a single symbolic execution trace. In the following, we extend this definition to account for multiple traces. We use $\hat{\phi}$ and $\hat{\omega}$ to range over input and output symbolic configurations, respectively.[2] We further use $\hat{\Omega}$ to range over sets of

[2] Input configurations differ from output configurations in that former are composed of a symbolic state and a statement whereas the latter are composed of a symbolic state and an outcome.

output configurations. We capture the semantics of multiple-trace symbolic execution using a big-step relation of the form $\hat{\phi} \Downarrow \hat{\Omega}$, meaning that if we "run" the symbolic semantics on the input configuration $\hat{\phi}$, we obtain the set of output configurations $\hat{\Omega}$. As symbolic execution often diverges, we additionally introduce a bounded version of the collecting semantics, writing $\hat{\phi} \Downarrow_k \hat{\Omega}$ to mean that if we "run" the symbolic semantics on the input configuration $\hat{\phi}$ and ignore symbolic traces with more than k steps, we end up with the set of output configurations $\hat{\Omega}$. In order to formalise these relations, we make use of collecting one-step transitions, writing $\hat{\phi} \searrow \hat{\Omega}$ to mean $\hat{\Omega}$ contains all the configurations resulting from the application of a single symbolic step on the input configuration $\hat{\phi}$ and no other; put formally:

$$\frac{\hat{\Omega} = \{\langle \hat{\mu}', \hat{\rho}', \pi', \hat{o}\rangle \mid \langle \hat{\mu}, \hat{\rho}, \pi, \hat{s}\rangle \rightsquigarrow \langle \hat{\mu}', \hat{\rho}', \pi', \hat{o}\rangle\}}{\langle \hat{\mu}, \hat{\rho}, \pi, \hat{s}\rangle \searrow \hat{\Omega}}$$

Definition 1 formalises the collecting semantics for bounded symbolic execution (the unbounded version can be easily obtained from the bounded one by dropping the constraints on k). The definition makes use of the following auxiliary predicates and functions:

- The predicates $\mathsf{Final}(\hat{\omega})$ and $\mathsf{NonFinal}(\hat{\omega})$ respectively hold if the output configuration $\hat{\omega}$ is, respectively, final and non-final, with a configuration being final if it contains either a return or an error outcome.
- The function Next can only be applied to non-final output configurations, turning the given output configuration into an input configuration by unwrapping the statement contained in its non-empty continuation output.

We say that $\mathsf{Final}/\mathsf{NonFinal}$ holds for a set of configurations if it holds for all of them.

▶ **Definition 1** (Symbolic Execution – Collecting Semantics).

$$\frac{\langle \hat{\mu}, \hat{\rho}, \pi, \hat{s}\rangle \searrow \hat{\Omega}_1 \cup \hat{\Omega}_2 \qquad \pi \ SAT \qquad \mathsf{Final}(\hat{\Omega}_1) \qquad \mathsf{NonFinal}(\hat{\Omega}_2)}{\langle \hat{\mu}, \hat{\rho}, \pi, \hat{s}\rangle \Downarrow_1 \hat{\Omega}_1} \textsc{Bounded - Base}$$

$$\frac{\pi \ UNSAT}{\langle \hat{\mu}, \hat{\rho}, \pi, \hat{s}\rangle \Downarrow_k \emptyset} \textsc{Bounded - False}$$

$$\frac{\langle \hat{\mu}, \hat{\rho}, \pi, \hat{s}\rangle \searrow \hat{\Omega} \cup \{\hat{\omega}_i \mid_{i=1}^{n}\} \qquad \mathsf{Final}(\hat{\Omega}) \qquad k > 1 \qquad \pi \ SAT \qquad \mathsf{Next}(\hat{\omega}_i) \Downarrow_{k-1} \hat{\Omega}_i \mid_{i=1}^{n}}{\langle \hat{\mu}, \hat{\rho}, \pi, \hat{s}\rangle \Downarrow_k \cup_{i=1}^{n} \hat{\Omega}_i \ \cup \ \hat{\Omega}} \textsc{Bounded - Rec}$$

3.2 Summary Correctness Properties

In contrast to most works on verification in which a concrete program is proven correct with respect to a specification [5], here we take the concrete function associated with the given summary to be the ground truth. Given a summary $\hat{s}$ and its associated concrete implementation s, we say that:

- $\hat{s}$ is an *over-approximation* of s if all the concrete executions of s are contained in the set of the executions modelled by $\hat{s}$;
- $\hat{s}$ is an *under-approximation* of s if all the executions modelled by $\hat{s}$ are contained in the set of concrete executions of s.

When a summary satisfies both properties, we say that it is *exact*. In the following, we make use of the concrete and symbolic semantics of our core language to establish the rigorous definitions underpinning these concepts.

Symbolic State Interpretation

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 values of $\hat{\sigma}$ to concrete values in a way that is

consistent with its path condition. For instance, if $\hat{\sigma} = \langle \hat{\mu}, \hat{\rho}, \hat{x} \neq 0 \rangle$, then the symbolic variable $\hat{x}$ cannot be replaced by 0 in $\hat{\mu}$ and $\hat{\rho}$. Accordingly, the interpretation function $[\![.]\!] :: \mathcal{S}ym\mathcal{S}t \to \wp(\mathcal{C}onc\mathcal{S}t)$ 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 two auxiliary interpretation functions, one for symbolic memories and one for symbolic stores. These interpretation functions require a *valuation function*, $\varepsilon : \hat{\mathcal{X}} \rightharpoonup \mathcal{V}$, that maps symbolic variables to concrete variables. We write $[\![\hat{\mu}]\!]_\varepsilon = \mu$ to mean that the interpretation of $\hat{\mu}$ under ε yields the concrete memory μ (analogously for stores). We interpret symbolic memories and stores point-wise, applying the valuation function to each memory/store cell. Put formally:

$$\textsc{Heap-Interp.}\ \frac{\hat{\mu}(l) = (\hat{v}, k_l, k_r)}{[\![\hat{\mu}]\!]_\varepsilon(l) \triangleq ([\![\hat{v}]\!]_\varepsilon, k_l, k_r)} \qquad \textsc{Store-Interp.}\ \frac{\hat{\rho}(x) = \hat{v}}{[\![\hat{\rho}]\!]_\varepsilon(x) \triangleq [\![\hat{v}]\!]_\varepsilon} \qquad \textsc{State-Interp.}\ \frac{\hat{\sigma} = \langle \hat{\mu}, \hat{\rho}, \pi \rangle}{[\![\hat{\sigma}]\!] \triangleq \{\, ([\![\hat{\mu}]\!]_\varepsilon, [\![\rho]\!]_\varepsilon) \mid [\![\pi]\!]_\varepsilon = true \}}$$

In the definitions that follow, we characterise the correctness of a given summary with respect to its corresponding reference implementation. In this context, the contents of the store are not relevant as they are discarded after the execution of the function. To account for this, we make use of *truncated state interpretations*, which ignore the store component. We use $[\![\![\hat{\sigma}]\!]\!]$ to refer to the truncated interpretation of the symbolic state $\hat{\sigma}$, which is defined as follows: $[\![\![\langle \hat{\mu}, \hat{\rho}, \pi \rangle]\!]\!] \triangleq \{\, [\![\hat{\mu}]\!]_\varepsilon \mid [\![\pi]\!]_\varepsilon = true \}$ For convenience, we lift truncated symbolic state interpretation to pairs of symbolic states and symbolic outcomes as follows: $[\![\![(\langle \hat{\mu}, \hat{\rho}, \pi \rangle, \hat{o})]\!]\!] \triangleq \{\, ([\![\hat{\mu}]\!]_\varepsilon,\ [\![\hat{o}]\!]_\varepsilon) \mid [\![\pi]\!]_\varepsilon = true \}$.

Correctness Properties

We are now at a position to formally define the correctness properties of symbolic summaries. Definitions 2 and 3 respectively define over- and under-approximating summaries. We omit the definition of *exactness*, which is simply the conjunction of the first two. Both definitions rely on the concrete semantics of our core language, using $\langle \mu, \rho, s \rangle \to_k \langle \mu', \rho', o \rangle$ to state that the concrete execution of s in the concrete memory μ and store ρ, finishes in k steps and generates the concrete memory μ', store ρ', and outcome o.

▶ **Definition 2** (Bounded Over-Approximating Summary). *A symbolic summary $\hat{s}$ is said to be a k-bound* over-approximation *of a concrete implementation s with respect to a symbolic memory $\hat{\mu}$ and symbolic store $\hat{\rho}$, if and only if it holds that:*

$$\begin{array}{l} \langle \hat{\mu}, \hat{\rho}, true, \hat{s} \rangle \rightsquigarrow^* \langle \hat{\mu}', \hat{\rho}', \pi, \hat{o} \rangle \ \wedge\ \mathsf{Final}(\hat{o}) \\ \quad \implies \forall \mu, \mu', \rho, \rho', o, k'.\ (\mu, \rho) \in [\![\langle \hat{\mu}, \hat{\rho}, \pi \rangle]\!] \ \wedge\ \langle \mu, \rho, s \rangle \to_{k'} \langle \mu', \rho', o \rangle \ \wedge\ k' \leq k \wedge \mathsf{Final}(o) \\ \quad\quad \implies (\mu', o) \in [\![\![\langle \hat{\mu}', \hat{\rho}', \pi \rangle, \hat{o}]\!]\!] \end{array}$$

▶ **Definition 3** (Under-Approximating Summary). *A symbolic summary $\hat{s}$ is said to be an* under-approximation *of a concrete implementation s with respect to a symbolic memory $\hat{\mu}$ and symbolic store $\hat{\rho}$, if and only if it holds that:*

$$\begin{array}{l} \langle \hat{\mu}, \hat{\rho}, true, \hat{s} \rangle \rightsquigarrow^* \langle \hat{\mu}', \hat{\rho}', \pi, \hat{o} \rangle \ \wedge\ \mathsf{Final}(\hat{o}) \\ \quad \implies \forall \mu', o.\ (\mu', o) \in [\![\![\langle \hat{\mu}', \hat{\rho}', \pi \rangle, \hat{o}]\!]\!] \\ \quad\quad \implies \exists \mu, \rho, \rho'.\ (\mu, \rho) \in [\![\langle \hat{\mu}, \hat{\rho}, \pi \rangle]\!] \ \wedge\ \langle \mu, \rho, s \rangle \to^* \langle \mu', \rho', o \rangle \end{array}$$

The proposed definitions are unusual in that they relate the symbolic execution of the summary $\hat{s}$ with the concrete executions of its reference implementation s. Typical definitions of this type [21, 36, 48, 47] relate symbolic execution of a given program with its concrete execution. In our setting, we have two different programs being related: a symbolic summary and its reference implementation.

Table 1 C-Implemented summaries.

Class	N	Lines of code	API calls	Memory Effects ✗	Memory Effects ✓	Symbolic Return ✗	Symbolic Return ✓
Strings	34	1639	194	24	10	18	16
Number Parsing	6	563	56	6	0	0	6
I/O	6	79	16	4	2	5	1
Memory	14	508	71	8	6	9	5
Heap	7	174	29	5	2	7	0
Total	67	2814	344	47	20	39	28

Finally, we note that the proposed definitions require that over/under-approximating summaries allocate memory in the exact same order of their corresponding reference implementations. This requirement could be relaxed by modifying our definition of symbolic heap interpretation for it to relate each symbolic heap with all its valid concrete reshufflings instead of simply those that follow its allocation order. We opted, however, for the more restrictive definitions in order to simplify the presentation.

3.3 Modelling LIBC Functions

We illustrate how the symbolic reflection API can be used to implement symbolic summaries for two families of LIBC functions: string manipulation functions and number-parsing functions. In total, we have implemented 67 summaries targeting 24 LIBC functions. Table 1 gives an overview of the 67 implemented summaries, showcasing for each category of summaries: **(1)** the total number of lines of code; **(2)** the total number of calls to the symbolic reflection API; **(3)** the number of summaries that update the heap memory; and **(4)** the number of summaries that return symbolic values. In the following, we give an example of a summary from each category.

Example – String Summaries

Figure 7 shows the implementation of an under-approximating symbolic summary for the `strlen` function (recall that the summary given in Figure 2 is exact). 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, then the summary returns the current length, otherwise it assumes that the current character is not the null character and continues iterating. More concretely, if a character `s[i]` is symbolic, the summary builds the constraint `cnstr` $\equiv$ (`s[i]` $\neq$ `\0`) and uses the primitive `is_sat` to check if `s[i]` can only be the null character (i.e., `!is_sat(cnstr)`), in which case the summary simply returns the value of `i`. Otherwise, `cnstr` is added to the current path condition using the primitive `assume`, making the summary under-approximating. For example, given the symbolic string `[c0, 'a', c2, \0]`, where `c0` and `c2` denote unconstrained symbolic characters, the summary outputs the value 3 and adds the constraints `c0` $\neq$ `\0` and `c2` $\neq$ `\0` to the path condition.

```
int strlen2(char* s){
    char charZero = '\0';  int i = 0;
    while(1){
        if(is_symbolic(&s[i])){ //s[i] is symbolic
            cnstr_t cnstr = _solver_NEQ(&s[i], &charZero, CHAR_SIZE); // s[i] ≠ '\0'
            if(!is_sat(cnstr)) break;
            else assume(cnstr); //Add cnstr to symbolic state
        }
        else if(s[i] == charZero) break;
        i++;
    }
    return i;
}
```

Figure 7 Under-approximating Summary of `strlen`.

```
int atoi2(char *str){
    ...
        else {
            symbolic retval = new_sym_var(INT_SIZE);
            int size = strlen(str); //Max possible length

            //Determine bounds
            int lower_bound = pow(10,size-1) * -1;
            int upper_bound = pow(10,size);

            //Build interval with constraints
            cnstr_t val_GT_lower = _solver_SGT(&retval, &lower_bound, INT_SIZE);
            cnstr_t val_LT_upper = _solver_SLT(&retval, &upper_bound, INT_SIZE);
            cnstr_t bounds_cnstr = _solver_And(val_GT_lower,val_LT_upper);

            //Add constraints to symbolic state
            assume(bounds_cnstr);
            return retval;
        }
    }
    return res * sign;
}
```

Figure 8 Fragment of Over-approximating Summary of `atoi`.

Example – Number summaries

Figure 8 shows a fragment of an over-approximating summary for `atoi`. The `atoi` function is used to parse strings denoting integer values. The key for guaranteeing that the summary is over-approximating is to return a fresh symbolic value and constrain this value to be: **(i)** greater than or equal to the smallest possible value that may be denoted by the given string; and **(ii)** smaller than or equal to the largest possible value that may be denoted by the given string. To this end, the summary determines the maximum possible length of the given string and uses it to compute the interval of possible return values. For example, considering a symbolic string `[c0, c1, c2, \0]`, the summary constrains the returned symbolic value, `retval` to lie within the interval $\texttt{retval} \in [-99, 999]$.

3.4 Reflection API Implementation

The proposed API can be implemented on top of any symbolic execution tool whose representation of symbolic states includes a symbolic memory and a path condition. We have found this to be the case for all the symbolic execution tools that we have analysed so

far [50, 19, 8, 15, 38, 43, 17, 16, 31]. In the formalism we have chosen to include a symbolic store component, to simplify the presentation, but this component is not essential for implementing the API.

In order to illustrate the effort involved in implementing the API, we have extended *AVD* [45] and *angr* [50], two current symbolic execution tools, with support for it. *AVD* is a novel symbolic execution tool developed by the authors, whereas *angr* is a widely used binary analysis toolkit. In both cases, the API implementation was straightforward, with the API code totalling 330 LoC for *angr* and 529 LoC for *AVD*. Table 2 shows the number of LoC of both implementations per type of API primitive. API primitives were implemented differently in *AVD* and *angr*. In *AVD*, we have implemented each reflection primitive as if it were a native symbolic summary, interacting directly with our own internal representation of symbolic states. In *angr*, we have used an internal API provided by the tool for developers to implement their own *angr* summaries directly in Python, associating each API primitive to an *angr simprocedure*.

Table 2 Lines of code per type of API primitive implemented in *AVD* and *angr*.

	Reflection Primitives				Total
	Core	Memory	Symbolic Values	Constraints	
AVD	206	54	56	213	529
angr	100	23	53	154	330

We believe that extending other symbolic execution tools with support for the proposed API should be as straightforward as extending *angr* and *AVD* since most tools already possess internally, albeit with variations, the reflection mechanisms captured by our API. The main difficulty in the implementation of the API is that it requires a thorough understanding of the inner-workings of the targeted tool and therefore should be done by the tools' own engineering teams rather than by external users. This effort should, however, pay off as implementing our API requires considerably less code than implementing a comprehensive library of symbolic summaries for LIBC from scratch, while also being conceptually simpler.

4 SumBoundVerify: Bounded Verification of Symbolic Summaries

In this section, we introduce our proposed methodology for the bounded verification of symbolic summaries. We first introduce our main summary verification algorithm (§4.1) and then explain how we leverage this algorithm to build SUMBOUNDVERIFY by automatically generating symbolic tests for the summaries to be verified (§4.2).

4.1 Bounded Summary Verification Algorithm

Symbolic State Lifting

In order to verify the correctness properties of a summary, we introduce a lifting operator $\lceil . \rceil :: \mathcal{P}(\mathcal{S}ym\mathcal{S}t \times \hat{\mathcal{O}}) \to \Pi$ that transforms a set of output configurations (i.e. symbolic states paired up with symbolic outcomes) into a boolean formula. We write $\lceil \hat{\Omega} \rceil = \pi$ to denote that the lifting of the output configurations in $\hat{\Omega}$ generates the formula π. The lifting operator is formally defined as follows:

$$\lceil \hat{\Omega} \rceil \triangleq \bigvee \left\{ \lceil \hat{\mu} \rceil_{\mathsf{m}} \wedge \pi \wedge (\mathtt{ret} = \hat{v}) \;\mid\; (\langle \hat{\mu}, \hat{\rho}, \pi \rangle, \mathbb{R}\langle \hat{v} \rangle) \in \hat{\Omega} \right\}$$

Essentially, a set of output configurations is transformed into a disjunction of boolean formulas, each describing its corresponding configuration. The formula created for each configuration has three components: **(i)** a memory component $\lceil \hat{\mu} \rceil_{\mathsf{m}}$ describing the content of the symbolic memory (defined below); **(ii)** a path condition component π; and **(iii)** a return component $\texttt{ret} = \hat{v}$, describing the return value of the function in the execution path that led to the given configuration. We use two dedicated variable `ret` and `count` to refer to the return value of a function and the number of cells in the current heap, respectively. Importantly, the lifting of a set of output configurations is only defined if all configurations are associated with a return outcome. The lifting operator for symbolic memories $\lceil . \rceil_{\mathsf{m}} :: \mathcal{S}ym\mathcal{M}em \rightarrow \Pi$, which takes a symbolic memory $\hat{\mu}$ and returns a boolean formula $\lceil \hat{\mu} \rceil_{\mathsf{m}}$ describing its contents, is defined as follows:

$$\textsc{Memory Lifting}\quad \frac{\pi_{blocks} = \wedge_{l \in \texttt{blocks}(\hat{\mu})} \lceil \hat{\mu} \rceil^{l}_{\mathsf{b}} \qquad \pi_{count} = \texttt{count} = |\,\texttt{dom}(\hat{\mu})\,|}{\lceil \hat{\mu} \rceil_{\mathsf{m}} \triangleq \pi_{blocks} \wedge \pi_{count}}$$

$$\textsc{Block Lifting}\quad \frac{\hat{\mu}(l) = (-, 0, k_r)}{\lceil \hat{\mu} \rceil^{l}_{\mathsf{b}} \triangleq \wedge_{0 \leq i < k_r} \{\, \texttt{cell}(l{+}i, \hat{\mu}(l+i)) \}}$$

A symbolic memory $\hat{\mu}$ is encoded into the conjunction of its blocks, which are, in turn, encoded using an auxiliary encoding function $\lceil . \rceil^{.}_{\mathsf{b}} : \mathcal{S}ym\mathcal{M}em \times \mathbb{N} \rightarrow \Pi$ for lifting memory blocks to formulas. The memory encoding function makes use of an auxiliary function $\texttt{blocks} : \mathcal{S}ym\mathcal{M}em \rightarrow \wp(\mathbb{N})$ to obtain the locations in the given memory corresponding to the beginning of blocks. The block-lifting function transforms each memory block into a formula describing the contents of each cell in the given block. To this end, we make use of an uninterpreted predicate `cell` to denote that the cell at a given address has the given content.

Bounded Verification Algorithm

Algorithm 1 describes our procedure for verifying if a summary $\hat{s}$ is under/over-approximating with respect to a concrete implementation s, symbolic memory $\hat{\mu}$, and symbolic store $\hat{\rho}$. In a nutshell, this algorithm compares the symbolic state(s) resulting from the execution of the summary, $\hat{\Omega}_{sum}$, against those generated by its reference implementation, $\hat{\Omega}_{ref}$. We do not bound the symbolic execution of the summary given that summaries should be designed to be convergent. In contrast, we bound the execution of their reference implementations as they often diverge. In a nutshell, we conclude that:

- a summary is *over-approximating* if $\lceil \hat{\Omega}_{ref} \rceil \implies \lceil \hat{\Omega}_{sum} \rceil$, i.e. the set of symbolic states generated by the reference implementation are contained in those generated by the summary;
- a summary is *under-approximating* if $\lceil \hat{\Omega}_{sum} \rceil \implies \lceil \hat{\Omega}_{ref} \rceil$, i.e. the set of symbolic states generated by the summary are contained in those generated by the corresponding reference implementation.

These implications are, however, too strong as they do not account for the creation of new symbolic values within the execution of the summary. To account for these, we have to existentially quantify the symbolic variables created during the execution of the summary. Algorithm 1 makes use of an auxiliary function $\mathsf{existentials}$ for computing the variables to be existentially quantified according to the following equation:

$$\mathsf{existentials}(\pi, \hat{\mu}, \hat{\rho}) \triangleq \mathsf{lvars}(\pi) \backslash (\mathsf{lvars}(\hat{\mu}) \cup \mathsf{lvars}(\hat{\rho}))$$

Essentially, all the symbolic variables created during the execution of the summary must be existentially quantified; these correspond to the symbolic variables that exist in the final symbolic states obtained from the summary execution but do not in the initial state.

Algorithm 1 Bounded Summary Verification.

```
1  function VerifySummary(prop, ŝ, s, μ̂, ρ̂, k)
2      ⟨μ̂, ρ̂, true, ŝ⟩ ⇓ Ω̂_sum
3      ⟨μ̂, ρ̂, true, s⟩ ⇓_k Ω̂_ref
4      π_sum ← ⌈Ω̂_sum⌉
5      π_ref ← ⌈Ω̂_ref⌉
6      xs_sum ← existentials(π_sum, μ̂, ρ̂)
7      if prop = over then
8          out ← isValid(π_ref ⟹ ∃xs_sum. π_sum)
9      else
10         out ← isValid(∃xs_sum. π_sum ⟹ π_ref)
11     return out
```

Correctness Result

Theorem 4 is our main correctness result. Essentially, it states that if we apply VerifySummary in over-approximation mode and it returns *true*, then the given summary is a bounded over-approximation of the given concrete implementation and analogously for under-approximation mode.

▶ **Theorem 4** (Summary Correctness). *Let $\hat{s}$ be a symbolic summary, s its reference implementation, $\hat{\mu}$ a symbolic memory, $\hat{\rho}$ a symbolic store, and k and a positive integer; then, it holds that:*

- *If* VerifySummary$(\texttt{over}, \hat{s}, s, \hat{\mu}, \hat{\rho}, k) = true$*, then $\hat{s}$ is a k-bound over-approximation of s with respect to $\hat{\mu}$ and $\hat{\rho}$;*
- *If* VerifySummary$(\texttt{under}, \hat{s}, s, \hat{\mu}, \hat{\rho}, k) = true$*, then $\hat{s}$ is an under-approximation of s with respect to $\hat{\mu}$ and $\hat{\rho}$.*

4.2 Automatic Symbolic Test Generation

In §4.1, we introduced a method for checking whether or not a given summary $\hat{s}$ is correct with respect to a reference implementation s, a symbolic memory $\hat{\mu}$, and a symbolic store $\hat{\rho}$. Naturally, one would like to prove that a summary is correct with respect to all symbolic memories and stores consistent with the signature of the summarised function instead of only a particular symbolic memory and store. SumBoundVerify solves this problem partially by bounding the size of memories and stores to be explored and using the type information in the signature of the summarised function to generate the initial symbolic states for which to check the summary up to the pre-established bound. In this section, we explain the procedure by example, leaving its formalisation for future work.

Instead of directly creating the symbolic states on which to run the summaries to be evaluated, we generate the initialisation code that creates such states from the signature of the function to be summarised. In general, the generated initialisation code depends on the type of the arguments given to the summary:

- For *non-character arrays*, we generate one fully symbolic array for each array size under the specified bound;
- For *character arrays*, we generate a single fully symbolic array terminated with a concrete null character with size given by the specified bound (note that this single character array models all strings of size lower than the specified bound since the intermediate symbolic characters of the array may denote the null character);

- For *non-recursive structures*, we generate a single fully symbolic test with the elements of the structure being mapped to fresh symbolic values;
- For *recursive structures*, we generate one fully symbolic test for each unfolding of the recursive structure up to the specified bound.

The test generation algorithm has two important limitations. First, it does not cover cases in which there are mutual dependencies between the parameters of the function to be summarised (e.g. a function with two array parameters with shared elements). Second, when it comes to recursive structures, the algorithm does not consider structures with loops, such as doubly-linked lists. If the arguments of the summarised function may exhibit one of these features, then the corresponding tests should be created manually.

Examples - Non-Character vs. Character Arrays

We now illustrate the test generation algorithm with two simple examples, covering non-character and character arrays. Suppose we want to validate a summary for a function with signature `int f(int* arr)`; in this case, SumBoundVerify would generate a set of tests, each with an initial section in charge of creating the symbolic integer array associated with the parameter `arr`. An example of one such initialisation code is given below.

```
1 int arr[4];
2 for (int i = 0; i < 4; i++) { arr[i] = new_sym_var_array("i", i, INT_SIZE); }
```

Essentially, the initialisation code allocates an integer array of size 4 in the stack and fills the four elements of the array with fresh symbolic integers. Suppose now, we want to validate a summary for a function with signature `int g(char* s)`; in this case, SumBoundVerify would generate a single test with the initialisation code given below.

```
1 char s[BOUND];
2 for (int i = 0; i < BOUND-1; i++) { s[i] = new_sym_var_array("c", i, CHAR_SIZE); }
3 s[BOUND-1] = '\0';
```

The initialisation code for character arrays has two main differences with respect respect to the code generated for non-character arrays: **(1)** the size of the character array allocated in the stack is always set to the specified bound and **(2)** the last element of the allocated character array is always set to the concrete null character.

5 Evaluation

This section answers the following evaluation questions:

EQ1 - Is the proposed symbolic reflection API sufficiently expressive to allow for the implementation of under/over-approximating summaries?

EQ2 - Can tool independent symbolic summaries be used to contain path explosion in symbolic execution?

EQ3 - Can SumBoundVerify be used to analyse real-world symbolic summaries developed in the context of state-of-the-art symbolic execution tools?

5.1 EQ1: API Expressivity

In order to illustrate the expressivity of our symbolic reflection API, we implement a library of symbolic summaries consisting of 67 summaries covering 26 libc functions from three different header files: *string.h*, *stdlib.h* and *stdio.h*. For most of these functions, we have implemented

at least two summaries, each illustrating a different correctness property. We then used SumBoundVerify to check the correctness properties of the implemented summaries by comparing them against their corresponding reference implementations. The fact that we were able to implement both under/over-approximating summaries for a large number of libc functions gives us a strong confidence that our library is expressive enough to model a wide range of behaviours.

Table 3 shows, for each function category, the number of implemented summaries and their corresponding correctness properties. In the table, *N* represents the total number of implemented summaries; *N/A* represents the number of implemented summaries that do not satisfy any correctness property; and the remaining columns represent the number of implemented summaries that satisfy the corresponding property (Under, Over, or Exact).

Table 3 Correctness properties for the C-implemented summaries.

Category	*N*	*N/A*	*Under-aprox.*	*Over-aprox.*	*Exact*
Strings	34	14	7	6	7
Memory	14	6	4	2	2
Number Parsing	6	2	–	4	–
System Calls	13	13	–	–	–
Total	67	35	11	12	9

Note that the summaries modelling functions that use system calls (e.g., `malloc` and `fgets`) cannot be verified against their respective reference implementations. The reason is that the symbolic execution of the reference implementations would necessarily step outside the perimeter of the language and, therefore, of the symbolic execution engine. For instance, the function `fgets` uses the `read` system call to obtain the string with the contents of the given file. In order to symbolically execute `fgets`, we always need to have a summary of `read` to start with, and this bootstrapping summary cannot be symbolically checked. Additionally, some summaries are neither over- nor under-approximating for performance reasons. These summaries assume that the function inputs satisfy certain preconditions, which we explicitly specify as comments in the summary code. The formal characterisation and verification of the properties satisfied by these summaries is, however, left for future work.

5.2 EQ2: Performance of Tool Independent Summaries

We measure the performance gains that can be obtained through the use of symbolic summaries implemented with our API and compare the performance of tool-independent summaries against that of native summaries. In order to carry out this experiment we set up a symbolic test suite focusing on libc function usage. To the best of our knowledge, no such test suite exists in the literature. In particular, we have analysed both the TestComp [11, 12] and SVComp [10, 13] test suites, counting for each test suite the number of calls to libc functions. We concluded that both these test suites make scarce use of libc functions, each calling fewer than 3 functions per test on average. Furthermore, the libc functions used in these test suites are mostly called with concrete arguments, rendering the use of symbolic summaries pointless.

Table 4 Summarized results in *angr* and *AVD* for both datasets.

			Memory Out ✗	Timeout ✗	Success ✓	Avg. N_{Paths}	Avg. N_{Libc}	Avg. N_{API}	Avg. $Time\,(s)$
Hash Map	*angr*	Concrete	7	0	3	2.2k	6.3k	3.6k	–
		C-Summaries	0	0	10	80	419	7.7k	199.66
		Native Summ	0	0	10	72	390	222	97.99
	AVD	Concrete	0	7	3	6.2k	9.6k	1.7k	–
		C-Summaries	0	0	10	95	483	8.4k	61.55
		Native Summ	0	0	10	96	487	244	26.66
Dynamic Strings	*angr*	Concrete	6	2	4	2.3k	1.0k	4.4k	–
		C-Summaries	1	0	11	431	397	4.0k	235.58
		Native Summ	1	0	11	353	237	97	143.96
	AVD	Concrete	0	7	5	3.5k	5.0k	105	–
		C-Summaries	0	1	11	499	1.9k	1.8k	14.20
		Native Summ	0	1	11	564	1.4k	97	18.94

Experimental Set-up and Symbolic Test Suites

As the test bed for this experiment, we used *angr* [50] and *AVD* [45] extended with our symbolic reflection API (see §3.4). All tests were run on a Ubuntu server (18.04.5 LTS) with an Intel Xeon E5–2620 CPU and 32GB of RAM. Additionally, each test was given 16GB of RAM with a maximum timeout of 30 minutes (1800 seconds).

We searched GitHub for open source C libraries that make intensive use of LIBC string-processing functions. We found two such libraries: **(1)** the *HashMap* library [55], which provides an implementation of a standard array-based hash table, and **(2)** the *Dynamic Strings* library [46], which provides an implementation of heap-allocated strings that extend the functionality offered by LIBC strings. Neither of these libraries came with concrete test suites. We created a symbolic test suite for each library: 10 symbolic tests for *HashMap* and 12 symbolic tests for *Dynamic Strings*. The symbolic test suites cover all the functions exposed by two libraries that interact with LIBC functions.

Results

We ran both symbolic test suites on *angr* and *AVD* using: **(1)** our tool-independent symbolic summaries implemented in C (C Summaries); **(2)** the native symbolic summaries originally included in each tool (Native Summaries); and **(3)** the corresponding C reference implementations. Part of these implementations were obtained from *Verifiable C* [4], a toolset for proving the functional correctness of C programs which comes with verified LIBC implementations [3]. The remaining functions were obtained from *glibc* [23] and *libiberty* [22] LIBC implementations.

Results are summarised in Table 4, which shows for each test suite run: **(1)** the number of tests that failed due to lack of memory (Memory Out); **(2)** the number of tests that failed for exceeding the time limit (Time Out); **(3)** the number of tests that finished executing within the given time limit (Success); **(4)** the average number of explored paths per test (Avg. N_{Paths}); **(5)** the average number of calls to LIBC functions per test (Avg. N_{Libc}); **(6)** the average number of calls to API primitives per test (Avg. N_{API}); and **(7)** the average execution time per test (Avg. $Time$). Note that we do not include the average execution time for the concrete summaries as the majority of the corresponding executions do not finish within the time limit. In contrast, we do include the average execution time for both C summaries and native summaries given that they execute successfully the exact same set of tests.

Unsurprisingly, results clearly show that symbolic summaries substantially improve the performance of symbolic execution tools. For the *HashMap* test suite, we observe that, for both tools, 7 out of the 10 symbolic tests fail to execute without summaries. Using reference implementations, all but the three smallest tests, either exhausted all the available memory or hit the timeout of 30 minutes. The results for the *Dynamic Strings* test suite are analogous. When using reference implementations: 8 out of 12 tests fail to execute with *angr* and 7 out of 12 tests fail to execute with *AVD*. When it comes to the path explosion problem, we observe that, for both libraries, the number of execution paths generated by symbolic execution with reference implementations is always at least one order of magnitude greater than that generated by symbolic execution with summaries. Table 4 also shows the average number of calls to LIBC functions and API functions. The number of calls to LIBC functions is lower for symbolic execution with summaries than with reference implementations; this is expected as symbolic execution with reference implementations generates a much larger number of paths. Note that, there are calls to the symbolic reflection API even when using reference implementations; this is due to the fact that we always model system calls with summaries.

Finally, we observe that tool-independent summaries are generally less performant than native summaries implemented directly within the code base of the corresponding tools. This slowdown is expected since tool-independent summaries are interpreted, whereas tool-specific summaries are executed natively. Furthermore, our test suites were specifically created to make heavy use of LIBC, making the slowdown more significant than for typical codebases, which interact less frequently with LIBC functions. Importantly, the proposed reflection API was not designed to obtain either performance or expressivity gains with respect to tool-specific summaries, but rather to allow for the implementation of verified, tool-independent summaries that can be shared across multiple SE tools. If the performance of a given tool-independent summary becomes an execution bottleneck for a specific application, then that summary should be implemented natively for the job at hand. However, we believe that will not be the case for the majority of summaries.

5.3 EQ3: Bugs in Symbolic Execution tools

In order to test the applicability of SUMBOUNDVERIFY, we used it to find bugs in summaries used in three high-profile symbolic execution tools: *angr* [50], *Binsec* [19] and *Manticore* [38]. Additionally, we also used SUMBOUNDVERIFY to verify the summaries that came with the AVD tool. In the following, we classify a summary as *buggy* if it is neither *under-* nor *over-approximating and* if there is no additional information about the expected behaviour of the summary regarding missing/incorrect paths (for instance, in the form of a code comment clarifying how the inputs should be constrained). As we implemented our API in both *AVD* and *angr*, we were able to use their summaries directly. In contrast, we had to manually re-write *Binsec*'s and *Manticore*'s summaries using our reflection API, following the original algorithms line-by-line. Using this methodology, we were able to validate a total of 52 LIBC symbolic summaries against their corresponding reference implementations.[3]

The results for all the validated summaries are shown in Table 5. Out of the analysed summaries, we found 14 buggy summaries in *angr*, 9 in *Binsec*, 1 in *Manticore*, and 13 in *AVD*. These summaries include spurious paths and exclude correct paths, meaning that they are neither under- nor over-approximating. Importantly, only a few summaries included

[3] As in §5.2, we use as reference the LIBC implementations from the *Verifiable C* tool chain, and the *glibc* and *libiberty* libraries

comments restricting the conditions under which they could be soundly applied. However, even these summaries contained bugs that were not ruled out by their authors' comments, which clearly demonstrates that the development of sound symbolic summaries is a difficult and error prone task that requires automated assistance.

Table 5 Summary bugs found in state-of-the-art symbolic execution tools.

	$N_{Evaluated}$	N_{Bugs}	Bugs Found				
angr	24	**14**	`atoi` `strncat` `strtol`	`atol` `strncmp` `strtoul`	`strcasecmp` `strncpy` `wcscasecmp`	`strchr` `strstr` `wcscmp`	`strcmp` `strtok_r`
Binsec	9	**9**	`memcmp` `strcpy`	`memcpy` `strncmp`	`memset` `strncpy`	`strchr` `strrchr`	`strcmp`
Manticore	4	**1**	`strcmp`				
Total	37	**24**					
AVD	15	**13**	`atoi` `strcat` `strtol`	`memcmp` `strchr` `tolower`	`memcpy` `strcmp` `toupper`	`memmove` `strncmp`	`memset` `strncpy`
Total (incl. *AVD*)	52	**37**					

To illustrate the type of bug uncovered by SumBoundVerify, we present three bugs, each corresponding to a different tool and all three to the function `strcmp`. This function is used to compare two strings lexicographically, returning: **(i)** a positive integer if the first string is greater than the second one, **(ii)** a negative integer if it is lower, and **(iii)** 0 if the two strings coincide. Even though the reference implementation of this function is very compact with less than 10 LoC, its corresponding summaries can be extremely complex (for instance, *angr*'s summary has 160 LoC).

Bug in *angr*

The possible execution paths for the `strcmp` function can be divided into two main sets according to the returned value: the execution paths where the return value is equal to zero and those where it is different. *angr* correctly models all the execution paths that return the value zero, i.e., the cases where the two symbolic input strings are equal. Regarding the execution paths with a return value different from zero, i.e, the cases where the input strings are different, *angr* always constrains the return value to 1. Hence, by returning a fixed positive integer for all paths where the two strings differ, the summary does not satisfy any of the correctness properties. Assuming, for example, two symbolic input strings of size 3, *str1* and *str2*, SumBoundVerify produces the following counterexample for *angr*'s summary:

$$\text{Missing Path: } [str1 = [\texttt{A}, \texttt{A}, \texttt{A}, \texttt{\textbackslash 0}] \wedge \ str2 = [\texttt{B}, \texttt{B}, \texttt{B}, \texttt{\textbackslash 0}] \wedge \ ret = -1]$$
$$\text{Wrong Path: } [str1 = [\texttt{A}, \texttt{A}, \texttt{A}, \texttt{\textbackslash 0}] \wedge \ str2 = [\texttt{B}, \texttt{B}, \texttt{B}, \texttt{\textbackslash 0}] \wedge \ ret = 1]$$

Essentially, the missing path describes a concrete execution that is not covered by the summary, whereas the wrong path describes a behaviour covered by the summary that is not generated by the execution of the concrete function.

Bug in *Manticore*

Manticore's `strcmp` summary iterates over the two strings to build a nested if-then-else formula over pairs of symbolic bytes. This formula means that if two symbolic bytes are different, then the summary must return the difference of those bytes; if they are equal, the summary must return the value 0 when they are the last two bytes of the strings or continue iterating otherwise. For instance, considering two symbolic input strings: $str1 = [\texttt{c1}, \texttt{c2}, \texttt{\textbackslash 0}]$ and $str2 = [\texttt{c3}, \texttt{c4}, \texttt{\textbackslash 0}]$, this summary will create the following if-then-else formula:

$$\texttt{ret} = \texttt{ITE}(\texttt{c1} \neq \texttt{c3},\ \texttt{c1} - \texttt{c3},\ \underline{\texttt{ITE}(\texttt{c2} \neq \texttt{c4},\ \texttt{c2} - \texttt{c4},\ 0)})$$

Even though it appears to be correct, *Manticore*'s summary does not take into account the fact that intermediate symbolic bytes may also be the null character (`\0`). When validating this summary on the input strings $str1$ and $str2$, SumBoundVerify produces the following counterexample:

$$\text{Missing Path: } [str1 = [\texttt{\textbackslash 0}, \texttt{A}, \texttt{\textbackslash 0}] \wedge str2 = [\texttt{\textbackslash 0}, \texttt{B}, \texttt{\textbackslash 0}] \wedge\ ret = 0]$$
$$\text{Wrong Path: } [str1 = [\texttt{\textbackslash 0}, \texttt{B}, \texttt{\textbackslash 0}] \wedge str2 = [\texttt{\textbackslash 0}, \texttt{A}, \texttt{\textbackslash 0}] \wedge\ ret = 1]$$

Bug in *Binsec*

Unlike the previous summary, *Binsec*'s `strcmp` summary takes into account the case where the intermediate symbolic bytes are null characters, but still gets it wrong. For instance, given the same two symbolic input strings: $str1 = [\texttt{c1}, \texttt{c2}, \texttt{\textbackslash 0}]$ and $str2 = [\texttt{c3}, \texttt{c4}, \texttt{\textbackslash 0}]$, this summary generates the following formula:

$$\begin{aligned}
\texttt{ret} = \texttt{ITE}(&\texttt{c1} = \texttt{\textbackslash 0},\\
&\texttt{ITE}(\texttt{c3} = \texttt{\textbackslash 0},\ 0,\ 1),\\
&\texttt{ITE}(\texttt{c1} = \texttt{c3},\\
&\quad \texttt{ITE}(\texttt{c2} = \texttt{\textbackslash 0},\\
&\quad\quad \texttt{ITE}(\texttt{c4} = \texttt{\textbackslash 0},\ 0,\ 1),\\
&\quad\quad \texttt{ITE}(\texttt{c2} = \texttt{c4},\ 0,\ \texttt{ITE}(\texttt{c2} > \texttt{c4},\ 1, -1))),\\
&\quad \texttt{ITE}(\texttt{c1} > \texttt{c3},\ 1,\ -1)))
\end{aligned}$$

Note that, when comparing each pair of symbolic bytes, this formula first checks if the current byte of $str1$ (e.g., `c1`) is equal to the null character, in which case it then checks if the corresponding byte of $str2$ (e.g., `c3`) is also equal to null; if it is, it evaluates to 0; otherwise, it evaluates to 1 and here lies the problem. When validating this summary on the input strings $str1$ and $str2$, SumBoundVerify produces the following counterexample:

$$\text{Missing Path: } [str1 = [\texttt{A}, \texttt{\textbackslash 0}, \texttt{\textbackslash 0}] \wedge str2 = [\texttt{A}, \texttt{B}, \texttt{\textbackslash 0}] \wedge\ ret = -1]$$
$$\text{Wrong Path: } [str1 = [\texttt{A}, \texttt{\textbackslash 0}, \texttt{\textbackslash 0}] \wedge str2 = [\texttt{A}, \texttt{B}, \texttt{\textbackslash 0}] \wedge\ ret = 1]$$

When the first string is shorter than the second one, *Manticore*'s summary assumes that `strcmp` returns 1 when it should instead return -1.

Bug in Verifiable C

During our validation experiments we saw unexpected results when using the `strcmp` implementation of *Verifiable C* to validate the corresponding symbolic summaries. In particular, we observed different results for the same `strcmp` summaries when using as reference implementation that of *Verifiable C* and those of the *glibc* and *libiberty* libraries. We found

a bug in the `strcmp` implementation of *Verifiable C* [53]: it compares characters as signed values instead unsigned ones as mandated by the POSIX specification of LIBC. The code had been proven correct, but for a specification too weak to bring the bug to light. This illustrates yet another application of SUMBOUNDVERIFY: it can be used to validate reference implementations against each other.

6 Related Work

There is a vast body of work on summaries for different types of program analysis, such as program logics [30, 44], abstract interpretation [56], and symbolic execution [25, 29]. However, in contrast to program logics, which are typically designed to be compositional and therefore place a heavy emphasis on summaries, in the form of function specifications and their usage, the study of summaries in symbolic execution literature is much more uneven. In particular, while there is a large number of symbolic execution tools that make use of operational summaries in the style of those described in this paper [50, 19, 8, 15, 38, 43, 17, 16, 31], we believe we are the first to address the issue of their formalisation and verification. The existing work on the use of summaries in symbolic execution can broadly be divided into two main groups: **(1)** *first-order summaries* that either do not reason about the heap memory or do so in a very limited way; and **(2)** *structural summaries* that rely on various types of representations to model the effects of heap-manipulating functions. In the following, we give a brief outline of research in both categories of summaries, focusing on the work that is closest to ours.

Godefroid et al. were the first to explore the use of first-order summaries in symbolic execution [24, 2, 28]. The first compositional tool in this line of work was SMART [24], a dynamic symbolic execution tool with support for summaries. SMART analyses functions in isolation in a bottom-up manner, encoding the testing results of each function as a first-order summary to be re-used in the analysis of other functions. Then, the authors proposed a variation of their original algorithm to allow for the lazy exploration of the search space in a top-down manner [2]. Later, the authors presented SMASH [28], a framework for testing and verifying C programs. Analogously to SMART, SMASH is incremental, analysing one function at a time and generating summaries that can be used in the analysis of other functions. The novelty of SMASH is that it allows for the combined use of both under- and over-approximating summaries in a demand-driven way. Importantly, the summaries of all three tools consist only of first-order formulas that cannot describe heap effects.

First-order summaries have also been used in the context of loop-summarisation [27, 51, 35]. These works typically combine symbolic execution with a custom-made static analysis component for detecting the induction variables of the loops to be summarised and constructing partial invariants describing their behaviour. Along this line of research, Kapus et al. [32] have recently proposed a new algorithm for inferring loop invariants for string-manipulating C programs using counter-example guided synthesis [1, 49]. These works are, however, complementary to ours since our goal is not to automatically generate summaries but rather to validate manually-written ones.

When it comes to structural summaries, Qiu et al. [42] proposed a new approach for compositional symbolic execution where function summaries are expressed as *memoization trees*. A *memoization tree* is a tree-like data structure that describes the various paths taken by the summarised function, *including* their effects on the heap. To this end, memoization tree nodes describe both the variable store as well as the contents of the heap of the summarised function. The proposed tool is compositional, analysing one function at a time and expanding the contents of symbolic objects by need, following the lazy initialisation approach [33].

Recently, Fragoso Santos et al. [48] proposed JaVerT 2.0, a new compositional symbolic execution tool for JavaScript. JaVerT 2.0 combines separation-logic-based summaries with static symbolic execution. More concretely, it allows for the incremental analysis of the given program, generating separation-logic-based specifications that can later be used during symbolic execution. The use of separation-logic summaries during symbolic execution has also been explored in the design of the Gillian framework [21, 36], which resulted from the generalisation of JaVerT 2.0 to a multi-language setting.

7 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. The implementation of symbolic summaries is time-consuming and error-prone, but until now there was a lack of mechanisms and methodologies for sharing symbolic summaries across different tools and for their validation.

This paper proposes a new API for developing and verifying tool-independent summaries. 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* with support for it and developed tool-independent symbolic summaries for 26 different LIBC functions, comprising string-manipulating functions, number-parsing functions, input/output functions, and heap functions. Furthermore, we presented SumBoundVerify, a new tool for the bounded verification of the summaries written with our symbolic reflection API. We applied SumBoundVerify to 37 symbolic summaries taken from 3 state-of-the-art symbolic execution tools, *angr*, *Binsec* and *Manticore*, detecting a total of 24 buggy summaries.

As future work, we intend to design a tool for automating the creation of symbolic summaries by synthesising them from declarative specifications, such as separation logic triples. To this end, we plan to leverage recent results on code synthesis from separation logic specifications [41], with the key difference being that our goal is to synthesise symbolic summaries instead of reference implementations.

References

1 Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Counterexample guided inductive synthesis modulo theories. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 270–288, Cham, 2018. Springer International Publishing.

2 Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 367–381, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

3 Andrew Appel. The Verifiable C string library, 2021. Software distribution that accompanies [4]. URL: `https://softwarefoundations.cis.upenn.edu/vc-current/index.html`.

4 Andrew W. Appel, Lennart Beringer, and Qinxiang Cao. *Verifiable C*, volume 5 of *Software Foundations*. Electronic textbook, 2021. URL: `http://softwarefoundations.cis.upenn.edu`.

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

6 Roberto Baldoni, Emilaio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), May 2018. `doi:10.1145/3182657`.

7 Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *Integrated Formal Methods*, pages 1–20, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

8 Zuzana Baranová, Jiří Barnat, Katarína Kejstová, Tadeáš Kučera, Henrich Lauko, Jan Mrázek, Petr Ročkai, and Vladimír Štill. Model checking of C and C++ with DIVINE 4. In *Automated Technology for Verification and Analysis*, pages 201–207, Cham, 2017. Springer International Publishing.

9 Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification CAV 2011*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

10 Dirk Beyer. Software Verification: 10th Comparative Evaluation (SV-COMP 2021). In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 401–422, Cham, 2021. Springer International Publishing.

11 Dirk Beyer. Status Report on Software Testing: Test-Comp 2021. In *Fundamental Approaches to Software Engineering*, pages 341–357, Cham, 2021. Springer International Publishing.

12 Dirk Beyer. Advances in Automatic Software Testing: Test-Comp 2022. In *Fundamental Approaches to Software Engineering - 25th International Conference, FASE 2022*, volume 13241 of *Lecture Notes in Computer Science*, pages 321–335. Springer, 2022.

13 Dirk Beyer. Progress on software verification: SV-COMP 2022. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022*, volume 13244 of *Lecture Notes in Computer Science*, pages 375–402. Springer, 2022.

14 Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT—a Formal System for Testing and Debugging Programs by Symbolic Execution. *SIGPLAN Not.*, 10(6):234–245, April 1975. doi:10.1145/390016.808445.

15 Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, USA, 2008. USENIX Association.

16 Marek Chalupa, Vincent Mihalkovič, Anna Řechtáčková, Lukáš Zaoral, and Jan Strejček. Symbiotic 9: String analysis and backward symbolic execution with loop folding. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 462–467, Cham, 2022. Springer International Publishing.

17 Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The S2E Platform: Design, Implementation, and Applications. *ACM Transactions on Computer Systems - TOCS*, 30:1–49, February 2012. doi:10.1145/2110356.2110358.

18 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)*, pages 1021–1038, Los Alamitos, CA, USA, May 2020. IEEE Computer Society. doi:10.1109/SP40000.2020.00074.

19 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)*, volume 1, pages 653–656, 2016. doi:10.1109/SANER.2016.43.

20 Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

21 José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa 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*, PLDI 2020. Association for Computing Machinery, 2020.

22 GNU. GNU libiberty, 2022. Accessed: 6th July 2023. URL: https://gcc.gnu.org/onlinedocs/libiberty/.

23 GNU. The GNU C library, 2022. Accessed: 6th July 2023. URL: https://www.gnu.org/software/libc/.

24 Patrice Godefroid. Compositional Dynamic Test Generation. *SIGPLAN Not.*, 42(1):47–54, January 2007. doi:10.1145/1190215.1190226.

25 Patrice Godefroid, Shuvendu K. Lahiri, and Cindy Rubio-González. Statically validating must summaries for incremental compositional dynamic test generation. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, volume 6887 of *Lecture Notes in Computer Science*, pages 112–128. Springer, 2011. doi:10.1007/978-3-642-23702-7_12.

26 Patrice Godefroid, Michael Y. Levin, and David A Molnar. Automated whitebox fuzz testing. In *Network Distributed Security Symposium (NDSS)*. Internet Society, 2008.

27 Patrice Godefroid and Daniel Luchaup. Automatic Partial Loop Summarization in Dynamic Test Generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 23–33, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2001420.2001424.

28 Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and SaiDeep Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*, pages 43–56. ACM, 2010. doi:10.1145/1706299.1706307.

29 Denis Gopan and Thomas Reps. Low-level library analysis and summarization. In *Computer Aided Verification*, pages 68–81, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

30 Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK*, pages 14–26. ACM, 2001. doi:10.1145/360204.375719.

31 Joxan Jaffar, Rasool Maghareh, Sangharatna Godboley, and Xuan-Linh Ha. TracerX: Dynamic Symbolic Execution with Interpolation (Competition Contribution). In *Fundamental Approaches to Software Engineering*, pages 530–534, Cham, 2020. Springer International Publishing.

32 Timotej Kapus, Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Cristian Cadar. Computing summaries of string loops in c for better testing and refactoring. In *PLDI 2019: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 874–888, New York, NY, USA, 2019. Association for Computing Machinery.

33 Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2003*, volume 2619 of *Lecture Notes in Computer Science*. Springer, 2003.

34 James C. King. A new approach to program testing. In *Proceedings of the International Conference on Reliable Software*, pages 228–233, New York, NY, USA, 1975. Association for Computing Machinery. doi:10.1145/800027.808444.

35 Yude Lin, Tim Miller, and Harald Søndergaard. Compositional symbolic execution using fine-grained summaries. In *2015 24th Australasian Software Engineering Conference*, pages 213–222, 2015. doi:10.1109/ASWEC.2015.32.

36 Petar Maksimovic, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. Gillian, part II: real-world verification for javascript and C. In *Computer Aided Verification - 33rd International Conference, CAV 2021*, volume 12760 of *Lecture Notes in Computer Science*, pages 827–850. Springer, 2021.

37 Petar Maksimovic, Caroline Cronjäger, Julian Sutherland, Andreas Lööw, Sacha-Élie Ayoun, and Philippa Gardner. Exact separation logic. *CoRR*, abs/2208.07200, 2022. arXiv:2208.07200.

38 Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem 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)*, pages 1186–1189, 2019. doi:10.1109/ASE.2019.00133.

39 Manh-Dung Nguyen, S'ebastien Bardin, Richard Bonichon, R. Groz, and Matthieu Lemerre. Binary-level directed fuzzing for use-after-free vulnerabilities. *ArXiv*, abs/2002.10751, 2020. arXiv:2002.10751.

40 Peter O'Hearn. Incorrectness logic. *Proceedings of the ACM on Programming Languages*, 4:1–32, December 2019. doi:10.1145/3371078.

41 Nadia Polikarpova and Ilya Sergey. Structuring the synthesis of heap-manipulating programs. *Proc. ACM Program. Lang.*, 3(POPL):72:1–72:30, 2019.

42 Rui Qiu, Guowei Yang, Corina S. Pasareanu, and Sarfraz Khurshid. Compositional symbolic execution with memoized replay. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 632–642, 2015. doi:10.1109/ICSE.2015.79.

43 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*, volume 1, pages 445–454, 2010. doi:10.1145/1806799.1806864.

44 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science LICS 2002*, pages 55–74. IEEE Computer Society, 2002.

45 Nuno Sabino. Automatic vulnerability detection: Using compressed execution traces to guide symbolic execution. Master's thesis, Instituto Superior Técnico, November 2019.

46 Salvatore Sanfilippo. Simple dynamic strings, 2015. Accessed: 6th July 2023. URL: https://github.com/antirez/sds.

47 José Fragoso Santos, Petar Maksimovic, Théotime Grohens, Julian Dolby, and Philippa Gardner. Symbolic Execution for JavaScript. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018*, pages 11:1–11:14. ACM, 2018.

48 José Fragoso Santos, Petar Maksimovic, Gabriela Sampaio, and Philippa Gardner. JaVerT 2.0: compositional symbolic execution for JavaScript. *Proc. ACM Program. Lang.*, 3(POPL):66:1–66:31, 2019.

49 Vaibhav Sharma, Kesha Hietala, and Stephen McCamant. Finding substitutable binary code by synthesizing adapters. *IEEE Transactions on Software Engineering*, PP:1–1, July 2019. doi:10.1109/TSE.2019.2931000.

50 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)*, pages 138–157, 2016. doi:10.1109/SP.2016.17.

51 Jan Strejček and Marek Trtík. Abstracting Path Conditions. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 155–165, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2338965.2336772.

52 Emina Torlak and Rastislav 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*, Onward! 2013, pages 135–152, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2509578.2509586.

53 Verifiable C – Verif_strlib. Bug in strcmp function. Accessed: 6th July 2023. URL: https://softwarefoundations.cis.upenn.edu/vc-current/Verif_strlib.html.

54 Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Autom. Softw. Eng.*, 10:203–232, April 2003. doi:10.1023/A:1022920129859.

55 Richard Wiedenhöft. C Hash map, 2014. Accessed: 6th July 2023. URL: https://gist.github.com/Richard-W/9568649.

56 Greta Yorsh, Eran Yahav, and Satish Chandra. Generating precise and concise procedure summaries. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 221–234. ACM, 2008.