Specification-Driven Generation of Summaries for Symbolic Execution

Rafael Gonçalves[1,2,3] (✉), Frederico Ramos[2,3], Pedro Adão[2,4], and José Fragoso Santos[2,3]

[1]Carnegie Mellon University, Pittsburgh, PA, USA
rgoncalv@andrew.cmu.edu
[2]Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal
{frederico.ramos,pedro.adao,jose.fragoso}@tecnico.ulisboa.pt
[3]INESC-ID, Lisbon, Portugal
[4]Instituto de Telecomunicações, Aveiro, Portugal

Abstract. Symbolic execution is a popular program analysis technique that has been successfully used for bug-finding and bounded verification in various modern programming languages. Despite its popularity, however, symbolic execution suffers from two main limitations when applied to real-world code: interactions with the runtime environment and path explosion. Symbolic summaries are the standard solution to tackle these challenges. Yet, the development of summaries remains to this day a manual task that is known to be highly error-prone. To address this, we propose SumGen, a new tool for automatically generating correct-by-construction summaries from function specifications. With SumGen, we were able to generate a total of 131 summaries for 47 libc functions, demonstrating the effectiveness of our methodology in producing correct summaries for real-world, highly complex code.

Keywords: Symbolic Execution, Symbolic Summaries, Summary Generation, Separation Logic.

1 Introduction

Symbolic execution (SE) [8,35] is a program analysis technique that bridges the gap between program testing and verification. It has been used for bug detection and bounded verification in multiple programming languages, such as C [9,59], JavaScript [18,19,37], and WebAssembly [43,44]. By allowing the execution of programs with symbolic inputs instead of concrete ones, SE engines are able to explore all execution paths of a given program up to a bound. The idea is to maintain for each execution path a *symbolic state* with: **(i)** a *symbolic store and heap* holding the contents of the variable store and heap memory of the program along that path; and **(ii)** a first order formula, called *path condition*, describing the constraints on the symbolic inputs that led the execution towards that path. SE engines rely on SMT solvers such as Z3 [45] or cvc5 [5] to check the feasibility of explored paths as well as the validity of any assertions provided by the developers.

© The Author(s) 2026

R. Krebbers (Ed.): ESOP 2026, LNCS 16501, pp. 283–313, 2026.
https://doi.org/10.1007/978-3-032-22720-1_11

Despite its widespread use, symbolic execution suffers from two main limitations when applied to real-world code: interactions with the runtime environment and path explosion [3]. A common solution employed by modern SE engines to tackle these challenges is to use *symbolic summaries* [9,12,15,44,55,59] to model both external functions and internal functions with a high degree of branching. A symbolic summary is an operational model of a function that simulates its behavior by interacting directly with the underlying symbolic state. In general, summaries extend the current path condition with constraints that capture the behavior of their corresponding functions without having to actually symbolically execute them. By acting directly on the symbolic state, summaries are an effective mechanism to both model the execution of external functions, whose code is not available for analysis, and contain the number of explored paths by avoiding unnecessary branching [3,54].

Modern summaries, however, are typically handwritten, often in a tool-specific manner and without being verified. This manual approach is far from ideal, as it is highly error-prone and can introduce subtle bugs that are difficult to detect. Such bugs can undermine the soundness and/or completeness guarantees of the tools that rely on these summaries, potentially leading to the incorrect exclusion of valid execution paths [54]. The consequences can be dire; for instance, *angr* [59], a state-of-the-art binary analysis tool, may fail to identify vulnerable paths due to its reliance on buggy summaries, resulting in missed vulnerabilities. Furthermore, this issue is widespread; Ramos et al. [54] automatically analyzed 37 summaries sourced from three popular SE engines (*angr* [59], *Binsec* [15] and *Manticore* [44]), and detected bugs in 24 of those. The significant percentage of summaries with bugs clearly shows that the manual approach to writing summaries is inadequate. Instead, symbolic summaries should be generated using automated methods.

To fill this gap, we propose a new methodology for automatically generating symbolic summaries from function specifications. The declarative nature of specifications makes the behaviors they model much easier to understand compared to those captured by summaries, which are often procedural, lengthy, and dependent on complex symbolic formulas. We advocate the philosophy that tool developers should focus on writing specifications for the library functions they wish to support, rather than writing highly complex summaries that lack any correctness guarantees. Examples of LIBC function specifications are available [2,21,47,48] and can be adapted for this purpose.

Prior work on program synthesis has mostly relied on SMT solvers to drive code generation [16,33,51]. This paper examines a different approach; namely, that we can represent the specifications themselves in a format that directly translates into executable code, bypassing the need for an SMT solver-guided search. To this end, we introduce *matching trees*, which coalesce the various cases of a specification into a single tree-like data structure. Matching trees are the cornerstone of our methodology, as they provide a convenient format for the internal representation of assertions. While approaches similar to matching trees have been used for frame inference calculation in Separation Logic

engines [17,19,39,40], we are the first to employ them to drive code synthesis. Moreover, while we use them here to generate symbolic summaries, matching trees can also be applied to produce other forms of executable code, including regular programs.

We implement our methodology in SumGen, a new tool for the automatic generation of summaries for modern SE engines. To produce a summary, SumGen constructs a matching tree from the corresponding specification, which it then leverages to: **(i)** resolve the existential bindings in the function's precondition; and **(ii)** use the computed bindings to update the symbolic state according to the function's postcondition. SumGen produces code in an intermediate language, allowing for summaries to be generated for multiple tools. Currently, we support the generation of summaries in C and Python, respectively using the symbolic reflection API proposed by Ramos et al. [54] and the internal API of *angr* [59]. Extending SumGen with support for other backends is straightforward (*cf.* §3). To evaluate SumGen, we used it to generate 131 symbolic summaries for 47 libc functions. Our evaluation shows that these summaries are both correct by construction, easier to obtain, and as performant as handcrafted summaries.

In short, we make the following contributions:

- A methodology for generating summaries from function specifications (§3);
- SumGen, a tool for automatically generating symbolic summaries in C and Python (§3);
- A library of 131 summaries modeling 47 libc functions, shown to be correct, easier to obtain, and as performant as handcrafted summaries (§5).

2 Motivation and Overview

In this section, we present our motivation and approach for summary generation. We begin by discussing summary support in state-of-the-art tools (§2.1) and the challenges posed by buggy summaries (§2.2), and then give a high-level overview of SumGen (§2.3).

2.1 Summaries in State-of-the-Art Tools

To motivate our approach, we survey summary support for libc functions in nine state-of-the-art symbolic execution tools: *angr* [59], *Binsec* [15], *Divine* [4], *Klee* [9], *Manticore* [44], *otter* [55], *PySymEmu* [42], *S2E* [12], and *Symbiotic* [10]. We have excluded tools that do not include any libc summary or that are not open source. We focus on libc functions because they are widespread, foundational to many C programs, and have a significant security impact, making robust summaries essential for effectively detecting vulnerabilities. Furthermore, their well-defined behaviors and specifications make them amenable to precise modeling for symbolic execution. The results of our survey are shown in Figure 1.

The first notable observation is that symbolic summaries are pervasive in modern symbolic execution tools. Figure 1a shows, for each selected tool, the

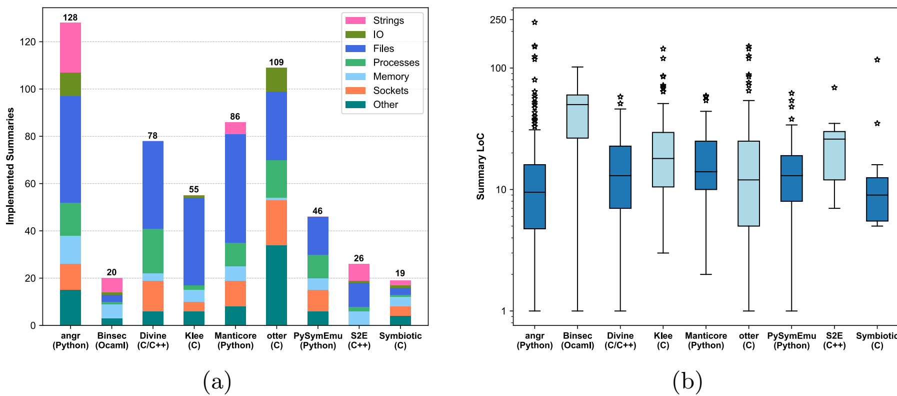

Fig. 1. Tool statistics: (a) number of implemented summaries, and (b) complexity of summaries.

total number of supported LIBC summaries and the programming language in which those summaries are implemented. We divide LIBC summaries into seven categories: string manipulation, I/O, file handling, process management, memory, sockets, and other system calls (*e.g.*, `time` and `sysinfo`). All tools implement multiple summaries from different categories; for instance, the three tools with the highest support, *angr*, *otter* and *Manticore*, implement 128, 109, and 86 summaries, respectively. In contrast, the tool with the least support, *Symbiotic*, implements 19 summaries from all seven categories.

Another notable observation is the significant variation in summary complexity. The box plot given in Figure 1b shows, for each of the tools, the complexity of the implemented summaries in terms of lines of code (LoC). Despite the large variety of summaries, the average number of LoC per summary is relatively stable across different tools, ranging between 10 and 20. This stems from the fact that many summaries are simple stubs (*e.g.*, those in the file handling and I/O categories) rather than fully-fledged models of the corresponding concrete functions. However, the complexity of individual summaries varies greatly; for example, *angr*'s summaries range from 1 to 238 LoC, with a significant number of outliers on the longer end. Large summaries, in particular, present a significant challenge for tool developers, as their reliance on complex handcrafted symbolic formulas makes them highly error-prone. We elaborate on this issue in §2.2 through a motivating example.

2.2 Bugs in Summaries

In the context of symbolic summaries, we define a *bug* as a defect that causes the summary's behavior to deviate from the reference implementation of the function it models in an irregular manner (*cf.* §2.3 for the formal definition). Buggy summaries are a common occurrence in symbolic execution tools. In a study analyzing 37 summaries from *angr* [59], *Binsec* [15] and *Manticore* [44], Ramos

```
1  def sum_strcmp(s1, s2):
2      min_zero_idx = min(api.find_zero(s1), api.find_zero(s2))
3      ret = s1[min_zero_idx] - s2[min_zero_idx]
4      for i in range(min_zero_idx - 1, -1, -1):
5          c1, c2 = s1[i], s2[i]
6          if api.is_symb(c1) or api.is_symb(c2):
7              ret = api.mk_ite(c1 != c2, c1 - c2, ret)
8          elif c1 != c2:
9              ret = c1 - c2
10     return ret
```

Fig. 2. Buggy strcmp summary in *Manticore*.

et al. [54] found bugs in 24, including in summaries for popular LIBC functions such as strcmp, strcpy and atoi. These bugs are often subtle and linked to corner cases that can cause tools to miss potential vulnerabilities when verifying complex codebases.

Bug in Manticore. In the following, we examine a buggy summary for the LIBC function strcmp sourced from *Manticore* [44]. The strcmp function compares two null-terminated strings, s1 and s2, returning 0 if they are equal or the difference between the first non-matching pair of characters otherwise.

Figure 2 shows a simplified but functionally equivalent implementation of the summary. This version improves clarity by removing unnecessary details and standardizing notation; the original summary is substantially more complex, totaling 24 LoC.[1] Nevertheless, even in this simplified form, the summary remains difficult to debug, as it manipulates complex symbolic formulas and depends on several symbolic reflection primitives [54]: **(i)** find_zero, which returns the index of the *first* byte whose value (concrete or symbolic) is guaranteed to be zero; **(ii)** is_symb, which checks if a variable is symbolic; and **(iii)** mk_ite, which creates an if-then-else (ITE) expression. In addition, arithmetic and logical operators are overloaded to operate on symbolic expressions; for example, the formula c1 != c2 is interpreted as a not-equal constraint between c1 and c2 whenever at least one of them is symbolic.

The summary begins by locating the first byte guaranteed to be zero (line 2) and setting the initial return value to be the difference between the two strings at that position (line 3). It then iterates backwards through the strings, considering two cases at each position: **(1)** if at least one byte is symbolic, it builds an if-then-else expression that evaluates to the difference between the bytes if they differ, or to the previous return value otherwise (line 7); or **(2)** if both bytes are concrete, it updates the return value only if they differ (line 9).

As an example, consider an execution with symbolic strings s1 $\mapsto [\hat{c}_1, \hat{c}_2, \text{`\textbackslash 0'}]$ and s2 $\mapsto [\hat{c}_3, \hat{c}_4, \text{`\textbackslash 0'}]$, where $\hat{c}_1, \ldots, \hat{c}_4$ are unconstrained symbolic bytes. Since

[1] https://github.com/trailofbits/manticore/blob/master/manticore/native/models.py#L99

$\hat{c}_1, \ldots, \hat{c}_4$ are unconstrained, the return value is initially set to 0, and the summary subsequently builds a nested if-then-else expression to compare the remaining pairs of symbolic bytes, producing the formula:

$$\mathsf{ITE}(\hat{c}_1 \neq \hat{c}_3, \hat{c}_1 - \hat{c}_3, \mathsf{ITE}(\hat{c}_2 \neq \hat{c}_4, \hat{c}_2 - \hat{c}_4, 0))$$

This summary, however, contains a subtle bug: it overlooks the possibility that intermediate symbolic bytes may be the null terminator. A counterexample is given by constraining $\hat{c}_1, \ldots, \hat{c}_4$ in the formula above such that $\mathsf{s1} \mapsto [\text{`\textbackslash 0'}, \text{`}a\text{'}, \text{`\textbackslash 0'}]$ and $\mathsf{s2} \mapsto [\text{`\textbackslash 0'}, \text{`}b\text{'}, \text{`\textbackslash 0'}]$, for which the if-then-else expression incorrectly evaluates to -1, while the strcmp function returns 0.

2.3 Summary Generation

The example given above shows that the manual approach to summary writing is highly error-prone. In contrast, our methodology allows for the automatic generation of correct-by-construction summaries from function specifications. With our approach, rather than writing a potentially buggy summary from scratch, users need only to provide a specification for the target function, and our tool automatically generates a summary that captures its behavior.

Specifications SumGen supports three types of specifications: **(1)** *under-approximating* (UX) specifications [49,53] that model a subset of the paths of the function; **(2)** *over-approximating* (OX) specifications [50,56] that model a superset of the paths of the function; and **(3)** *exact* (EX) specifications [41] that model the same set of paths as the function.

Consider, for example, the strcmp function. To write a specification for strcmp, we first define a predicate $\mathrm{strd}(s1, s2, \delta)$, which asserts that the difference between the values of the first non-matching pair of characters in $s1$ and $s2$ is δ. The predicate $\mathrm{strd}(s1, s2, \delta)$ has two cases: **(1)** $s1$ or $s2$ point to '\0', or to two distinct non-null characters, and δ is the difference between those characters; or **(2)** $s1$ and $s2$ point to equal non-null characters and δ is the difference between $s1 + 1$ and $s2 + 1$. We assume that all character arrays are eventually null-terminated by a concrete '\0'. Put formally:

$$\begin{array}{l}\mathrm{strd}(s1, s2, \delta) \triangleq \\ \quad s1 \mapsto c1 \uplus s2 \mapsto c2 \uplus (c1 = \text{`\textbackslash 0'} \vee c2 = \text{`\textbackslash 0'} \vee c1 \neq c2) \uplus \delta = c1 - c2 \\ \quad \vee \\ \quad s1 \mapsto c1 \uplus s2 \mapsto c2 \uplus (c1 \neq \text{`\textbackslash 0'} \wedge c2 \neq \text{`\textbackslash 0'} \wedge c1 = c2) \uplus \mathrm{strd}(s1 + 1, s2 + 1, \delta)\end{array}$$

where $\uplus$ denotes the *overlapping conjunction* [20,31]. The overlapping conjunction $P \uplus Q$ states that the heap may be split into two subheaps, one satisfying P and the other Q, but these subheaps need not be disjoint. Unlike the more traditional separating conjunction $P \star Q$, the overlapping conjunction does not enforce resource non-duplication. This is particularly useful in our setting, as

the functions we target may have overlapping arguments, making it counterproductive to enforce strict separation. Given this predicate, we can easily write an exact specification for `strcmp`:

$$\{\mathrm{strd}(s1, s2, \delta)\}\ \texttt{int strcmp(char * s1, char * s2)}\ \{\mathsf{ret} = \delta\}$$

where ret denotes the return value of the function.

The core challenge in specification-based synthesis lies in how one goes about generating heap-manipulating code from *declarative* specifications such as the one above. Prior work addresses this by encoding specifications as a set of constraints and letting an SMT solver find a program that satisfies them [16,33,51]. We advocate a different approach; namely, that we can leverage matching plans [17,19,39,40] for code synthesis by encoding specifications in a new intermediate representation that we call *matching trees*. A matching tree is a tree-like data structure that coalesces the various cases of a specification/predicate in a format that more closely resembles procedural code. Matching trees are the cornerstone of our methodology; we elaborate on them in §3.1.

From Specifications to Summaries Like specifications, there are three types of symbolic summaries: **(1)** *under-approximating* (UX) summaries that model a subset of the paths of the function; **(2)** *over-approximating* (OX) summaries that model a superset of the paths of the function; and **(3)** *exact* (EX) summaries that model the same set of paths as the function. A summary is considered *buggy* if it is neither UX, OX, nor EX, meaning that it excludes correct paths (*i.e.*, it does not over-approximate) and includes spurious paths (*i.e.*, it does not under-approximate). For example, the *Manticore* summary for `strcmp` (Figure 2) is buggy because it may produce an ITE expression that evaluates to -1 when $\mathsf{s1} \mapsto$ [‘\0’, ‘a’, ‘\0’] and $\mathsf{s2} \mapsto$ [‘\0’, ‘b’, ‘\0’], excluding a correct path (where the return value should be 0) and including a spurious one (where the return value is -1).

Table 1. Specification types and the summaries they produce.

		Summary		
		UX	OX	EX
Spec	UX	✓	×	×
	OX	×	✓	×
	EX	✓	✓	✓

SumGen supports the generation of the three types of summaries (UX, OX and EX). Which type of summary is more useful depends on the application: UX summaries are generally better suited for bug finding, whereas verification typically relies on OX summaries [49]. Table 1 shows which summary types can be generated from each specification type. Note that not all summary types can be generated from every specification type. For example, attempting to generate an OX summary from a UX specification could lead to both missing correct paths that the UX specification does not cover, and including spurious paths introduced by the OX generation procedure, rendering the summary buggy. Conversely, we cannot obtain a UX summary from an OX specification, nor an EX summary from UX and OX specifications.

```
1  def sum_strcmp(s1, s2):
2      c1, c2 = s1[0], s2[0]
3      if api.is_certain(c1 == '\0' or c2 == '\0' or c1 != c2):
4          return c1 - c2
5      else if api.is_certain(c1 != '\0' and c2 != '\0' and c1 == c2):
6          return sum_strcmp(s1[1:], s2[1:])
7      else:
8          d1 = api.scoped_call(sum_strcmp, [s1, s2], \
9                               c1 == '\0' or c2 == '\0' or c1 != c2)
10         d2 = api.scoped_call(sum_strcmp, [s1, s2], \
11                              c1 != '\0' and c2 != '\0' and c1 == c2)
12         return api.mk_ite(c1 == '\0' or c2 == '\0' or c1 != c2, d1, d2)
```

Fig. 3. Exact SumGen summary for strcmp.

Broadly speaking, the task of generating a summary can be understood as producing code that simulates the behavior of the corresponding function. To achieve this, SumGen constructs a matching tree from the given specification and compiles it into a program that: **(i)** resolves the existential bindings in the function's precondition; and **(ii)** uses the computed bindings to update the symbolic state according to the function's postcondition. As part of step **(ii)**, the summary computes a symbolic expression denoting the return value of the target function and updates the current path condition with the appropriate constraints on that value.

Figure 3 shows the exact strcmp summary generated by SumGen. This summary relies on two new symbolic reflection primitives: **(i)** is_certain, which checks if a given formula provably holds in the current state; and **(ii)** scoped_call, which performs a scoped function call. Scoped calls are central to our approach. In a nutshell, they augment the path condition with some logical constraint and call a function under that scope, restoring the original path condition once the call returns. We delve into the mechanics of scoped calls further in §3.2.

Given the two symbolic strings $\mathsf{s1} \mapsto [\hat{c}_1, \hat{c}_2, \text{‘}\backslash 0\text{’}]$ and $\mathsf{s2} \mapsto [\hat{c}_3, \hat{c}_4, \text{‘}\backslash 0\text{’}]$, where $\hat{c}_1, \ldots, \hat{c}_4$ are unconstrained symbolic bytes, this summary produces the formula:

$$\begin{aligned}&\mathsf{ITE}(\hat{c}_1 = \text{‘}\backslash 0\text{’} \vee \hat{c}_3 = \text{‘}\backslash 0\text{’} \vee \hat{c}_1 \neq \hat{c}_3, \hat{c}_1 - \hat{c}_3,\\ &\qquad \mathsf{ITE}(\hat{c}_2 = \text{‘}\backslash 0\text{’} \vee \hat{c}_4 = \text{‘}\backslash 0\text{’} \vee \hat{c}_2 \neq \hat{c}_4, \hat{c}_2 - \hat{c}_4, 0))\end{aligned}$$

In general, SumGen summaries can be seen as a disjunction of the various cases of a specification. When the summary knows precisely which case to apply, it computes the return value directly (lines 4 and 6). When it does not, the summary guides execution towards a *default branch* modeling the possible cases of the call (lines 8–12). For instance, if we do not know whether the constraint $\mathsf{c1} = \text{‘}\backslash 0\text{’} \vee \mathsf{c2} = \text{‘}\backslash 0\text{’} \vee \mathsf{c1} \neq \mathsf{c2}$ or its negation holds, the summary computes the return values d1 and d2 based on each assumption (lines 8–9 and 10–11, respectively) and builds an if-then-else expression that evaluates to d1 if the first constraint holds and to d2 otherwise (line 12). In our example, since $\hat{c}_1, \ldots, \hat{c}_4$ are unconstrained symbolic bytes, the execution is guided towards the default

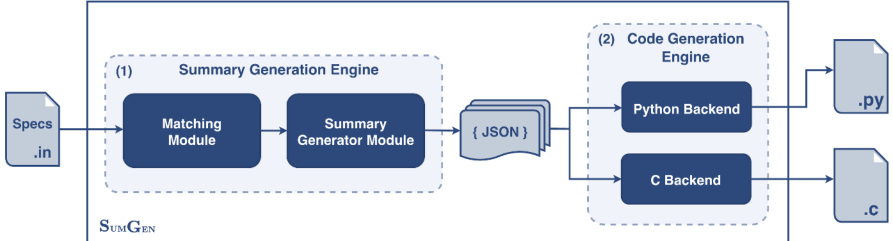

Fig. 4. Architecture of SumGen.

branch, building a nested if-then-else expression that exactly models the possible outcomes of the call to `strcmp`.

Unlike the `strcmp` summary implemented by *Manticore*, this summary is correct; in particular, the if-then-else expression derived above correctly evaluates to 0 if both $\hat{c}_1$ and $\hat{c}_3$ are equal to '\0'.

3 Summary Generation

Figure 4 gives a high-level overview of the architecture of SumGen. The implementation consists of two main components: **(1)** a *summary generation engine* implemented in Haskell (≈3.5k LoC) that, given a specification and desired correctness property (UX/OX/EX), generates a summary in our intermediate language; and **(2)** a *code generation engine* implemented in Python (≈700 LoC) that translates the generated summaries into executable code. SumGen is open source [28] and available online as a web application [27].

The summary generation engine consists of two modules: **(i)** the *matching module*, which parses a function specification written in our assertion language together with its associated predicate definitions and constructs the appropriate matching trees (*cf.* §3.1); and **(ii)** the *summary generator module*, which produces symbolic summaries in our intermediate language and outputs their ASTs in JSON format (*cf.* §3.2).

Our choice to produce summaries first in an intermediate language allows for the modularity of the code generation engine, which can be adapted to support multiple backends with minimal development effort. Currently, we support two backends: a C backend, which generates C summaries that use Ramos et al.'s [54] tool-independent symbolic reflection API, and a Python backend, which generates Python summaries that use *angr*'s [59] symbolic API. In order to extend SumGen with support for other languages, one needs only to implement the corresponding backend and add it to the code generation engine.

3.1 Syntax and Matching Trees

SumGen is built on top of a simple assertion language whose syntax is given below. Simple assertions, $p \in \mathcal{SA}$, are either pure assertions (*e.g.*, relational and

logical operations over expressions $e \in \mathcal{E}xpr$), directed equalities, cell assertions, or predicate assertions. Directed equalities, $x \ominus e$, differ from regular equalities in that they can be interpreted as an assignment; for example, if x is existentially quantified, the directed equality $x \ominus e$ gives us the binding of x. The cell assertion, $x \mapsto_\tau y$, states that x points to a memory cell holding y with type τ. Predicate assertions are of the form $\alpha(\bar{e}; y)$, where $\bar{e}$ is a list of arguments and y is a special return argument to be used by the generation procedure. Assertions, P, $Q \in \mathcal{A}srt$, are either the empty assertion, a simple assertion, or two assertions conjoined by the overlapping conjunction $P \uplus Q$.

The syntax of assertions

$\pi \in \Pi \triangleq e_1 \otimes e_2 \mid \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \mid \neg\pi$	$\otimes \in \{>, <, \geq, \leq, =, \neq\}$
$p, q \in \mathcal{SA} \triangleq \pi \mid x \ominus e \mid x \mapsto_\tau y \mid \alpha(\bar{e}; y)$	$P, Q \in \mathcal{A}srt \triangleq \mathsf{emp} \mid p \mid P \uplus Q$
$\Phi \in \mathcal{P}red \triangleq \mathsf{pred}\ \alpha(\bar{x}; y)\,\{\bar{P}\}$	$\Sigma \in \mathcal{S}pec \triangleq \{P\}\,\mathsf{fn}\,f(\bar{x})\,\{Q; y; \pi\}$

Predicate definitions, $\Phi \in \mathcal{P}red$, are of the form $\mathsf{pred}\ \alpha(\bar{x}; y)\,\{\bar{P}\}$, where α is the predicate name, $\bar{x}$ a list of the formal parameters, y a special return parameter to be used by the generation procedure, and $\bar{P}$ a sequence of assertions, each corresponding to a predicate case (*i.e.*, the full predicate definition is a disjunction of the elements of $\bar{P}$). Specifications, $\Sigma \in \mathcal{S}pec$, are of the form $\{P\}\,\mathsf{fn}\,f(\bar{x})\,\{Q; y; \pi\}$, where P is the precondition, Q the postcondition, y the return variable, and π a set of restrictions on the value of y. A specification is said to be *non-mutating* if it does not modify heap memory (*i.e.*, if the postcondition Q equals the precondition P); otherwise, it is said to be *mutating*. For simplicity, we omit the postcondition in non-mutating specifications, writing them as $\{P\}\,\mathsf{fn}\,f(\bar{x})\,\{y; \pi\}$. Moreover, we assume any variables that appear in an assertion and are not among the formal parameters of the corresponding specification/predicate to be existentially quantified; for instance, in the $\mathrm{strd}(s1, s2; \delta)$ predicate, the variables $c1$ and $c2$ are existentially quantified. Following this syntax, we write the strcmp specification as:

$$\{\mathrm{strd}(s1, s2; \delta)\}\,\mathsf{fn}\ \mathsf{strcmp}(s1, s2)\,\{\delta\}$$

and the predicate $\mathrm{strd}(s1, s2; \delta)$ as:

$$\begin{array}{l} \mathsf{pred}\ \mathrm{strd}(s1, s2; \delta)\,\{ \\ \quad s1 \mapsto c1 \uplus s2 \mapsto c2 \uplus (c1 = \text{`\textbackslash 0'} \vee c2 = \text{`\textbackslash 0'} \vee c1 \neq c2) \uplus \delta \ominus c1 - c2; \\ \quad s1 \mapsto c1 \uplus s2 \mapsto c2 \uplus (c1 \neq \text{`\textbackslash 0'} \wedge c2 \neq \text{`\textbackslash 0'} \wedge c1 = c2) \uplus \mathrm{strd}(s1+1, s2+1; \delta) \\ \} \end{array}$$

Note that, in the strcmp specification, both Q and π are omitted, as the function is non-mutating and does not place any constraints on the return value (*i.e.*, δ).

SumGen implements its *specification parser* on top of *Parsec* [36]. For simplicity, we omit some implementation details; in particular, while most types are implicit in the theory, all types are explicit in practice to support statically typed languages such as C.

Matching Generating a summary requires representing assertions in a format that can be more directly translated into executable code. To this end, we build on matching plans [19,17,40,39] with a new structure called *matching trees.*

In/Out-Parameters. Matching trees leverage *in/out-parameters* [19,46] to describe how the existentially quantified variables of a given specification/predicate can be determined from its formal parameters. Let us consider the case of simple assertions, which may contain any number of variables. Intuitively, an in-parameter of a simple assertion can be understood as a variable that we "know" and an out-parameter as a variable that we can "learn" from the assertion by inspecting the current state. For instance, the cell assertion $s1 \mapsto c1$ appearing in the predicate $\mathrm{strd}(s1, s2; \delta)$ has one in-parameter, $s1$, and one out-parameter, $c1$. Note that, given the value of $s1$ and the current state, we can learn the value of $c1$ by inspecting the contents of $s1$, but the opposite does not hold. The definition also extends to specifications and predicates, where the in-parameters are the formal parameters $\bar{x}$ and the out-parameter is the return variable/parameter y.

Matching Trees. Matching trees describe how the out-parameters of specifications/predicates can be computed from their in-parameters. In short, a *matching tree* is a tree-like data structure that coalesces the various cases of a specification/predicate by representing a disjunction of compound assertions as a single tree of simple assertions. We formalize matching trees in Definition 1.

Definition 1 (Matching Tree). *A matching tree $t \in \mathcal{MT}$ is defined as:*

$$t \in \mathcal{MT} \triangleq \bullet \mid \langle p, t' \rangle \mid \langle \pi, t_1, t_2 \rangle$$

The leaf node $\bullet$ is simply emp. The single node $\langle p, t' \rangle$ represents a simple assertion. The double node $\langle \pi, t_1, t_2 \rangle$ represents a condition that discriminates between two disjuncts. In Figure 5, we show a matching tree for the predicate $\mathrm{strd}(s1, s2; \delta)$. Notice the double node labeled $c1 = \text{`\textbackslash 0'} \vee c2 = \text{`\textbackslash 0'} \vee c1 \neq c2$, which discriminates between the two cases of the predicate, and how we compute the out-parameter δ accordingly.

A matching tree t is said to be *valid* w.r.t. to a set of variables X *iff*, for every node, each of its in-parameters is either an out-parameter of one of its ancestors or an element of X. A matching tree representing a specification/predicate should always be valid w.r.t. the set containing its formal parameters, ensuring that the out-parameters can be computed from the in-parameters. The matching tree we derived for $\mathrm{strd}(s1, s2; \delta)$, for instance, is valid w.r.t. the set $\{s1, s2\}$ containing the in-parameters of the predicate. However, if we were to remove the simple assertion $s1 \mapsto c1$, the tree would no longer

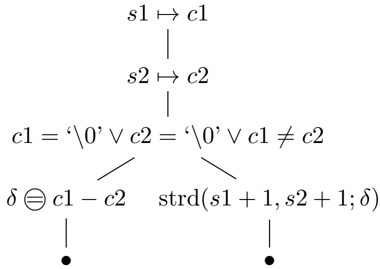

Fig. 5. Matching tree for $\mathrm{strd}(s1, s2; \delta)$.

be valid: in the node labeled $\delta \ominus c1 - c2$, the in-parameter $c1$ would be neither an out-parameter of its ancestors nor an element of $\{s1, s2\}$.

To construct valid matching trees from the given specifications and predicate definitions, SumGen implements a *matching module*. At a high level, it performs a search over the space of candidate matching trees and selects one that satisfies the validity constraints defined above.

3.2 Summary Generation

At the center of SumGen is a *summary generator module* that compiles matching trees into our intermediate summary language. This language can be easily transpiled to more complex programming languages, provided they expose the symbolic primitives required by our summaries. The syntax is given below.

The syntax of statements

$$e \in \mathcal{E}xpr \triangleq v \in \mathcal{V} \mid x \in \mathcal{X} \mid \ominus e \mid e_1 \oplus e_2 \qquad b \in \mathcal{BE} \triangleq \pi \mid \mathsf{isCertain}(\pi)$$

$$\begin{aligned} s \in \mathcal{S}tmt \triangleq\ & \mathsf{skip} \mid x \leftarrow (\tau)\ e \mid x \leftarrow (\tau)\ \mathsf{symvar}() \mid x \leftarrow (\tau) * e \mid * e_1 \leftarrow e_2 \\ & \mid * e_1 \overset{\pi}{\leftarrow} e_2 \mid s_1; s_2 \mid \mathsf{assume}\ \pi \mid \mathsf{assert}\ \pi \mid \mathsf{if}\ (b)\ \{s_1\}\ \mathsf{else}\ \{s_2\} \\ & \mid x \leftarrow (\tau)\ f(\bar{e}) \mid x \leftarrow (\tau)\ f(\bar{e})\ \mathsf{with}\ \pi \mid \mathsf{return}\ e \end{aligned}$$

Summary Language Expressions, $e \in \mathcal{E}xpr$, include literals, program variables, and unary and binary operators. Boolean expressions, $b \in \mathcal{BE}$, include pure assertions and isCertain expressions, which check if the given formula provably holds in the current state. Statements, $s \in \mathcal{S}tmt$, include: **(i)** the skip primitive; **(ii)** typed operations: the standard variable assignment, $x \leftarrow (\tau)\ e$, symbolic variable generation, $x \leftarrow (\tau)\ \mathsf{symvar}()$, and the memory read operation, $x \leftarrow (\tau) * e$; **(iii)** the standard and conditional memory update operations, $*e_1 \leftarrow e_2$ and $*e_1 \overset{\pi}{\leftarrow} e_2$; **(iv)** sequenced statements, $s_1; s_2$; **(v)** the assume and assert statements; **(vi)** if-then-else conditionals; **(vii)** non-scoped and scoped function calls, $x \leftarrow (\tau)\ f(\bar{e})$ and $x \leftarrow (\tau)\ f(\bar{e})\ \mathsf{with}\ \pi$; and **(viii)** return statements. Function declarations are of the form $\mathsf{fn}\ f(\bar{x})\ \{s\}$, where f is the function identifier, $\bar{x}$ the list of parameters, and s the function body. With the exception of scoped calls and conditional memory updates, the semantics of our language is standard (*cf.* [54] for a similar one). A scoped call, $x \leftarrow (\tau)\ f(\bar{e})\ \mathsf{with}\ \pi$, augments the path condition with π and calls f under that scope, restoring the original path condition once the call returns. A conditional memory update, $*e_1 \overset{\pi}{\leftarrow} e_2$, stores an ITE expression at address e_1 that evaluates to e_2 if π holds and to its previous content otherwise.

To support SumGen summaries, the target runtime must expose a set of symbolic reflection primitives that implement these statements. With the exception of scoped calls, modern SE engines offer most of these primitives by default. Scoped calls can be easily implemented by storing the path condition before the function call, augmenting it with the appropriate constraints, and

restoring it afterwards. Additionally, executing a statement in a state that violates its semantic precondition results in an error state. For instance, we do not allow reads/writes that access uninitialized memory or are incorrectly typed.

Specification and Predicate Definition Compilation The goal of the summary generation procedure is to transform the specification of the target function into a symbolic summary. To achieve this, we must also transform predicate definitions into executable code. Below, we give an overview of both procedures; for a more detailed discussion and formalization, see [29].

Specifications. The task of compiling a specification can be understood as producing code that: **(i)** resolves the existential bindings in the function's precondition; and **(ii)** uses the computed bindings to update the symbolic state according to the function's postcondition. We define a *specification compiler* $\mathcal{S}^{\beta}(\Sigma)$, parameterized by a flag $\beta \in \{\mathrm{UX}, \mathrm{OX}, \mathrm{EX}\}$, that, given a specification:

$$\Sigma = \{P\}\, \mathsf{fn}\, f(\bar{x})\, \{Q; y; \pi\}$$

generates a summary observing the specified correctness property. Informally, $\mathcal{S}^{\beta}(\Sigma)$ proceeds as follows: **(1)** it compiles the matching tree of the precondition P into a statement that computes the existentially quantified variables in P; **(2)** it compiles the matching tree of the postcondition Q into a statement that updates the heap memory according to the semantics of Q; and **(3)** it generates the summary's return value and extends the current path condition with π. Note that the returned expression may either be a fresh symbolic variable constrained by π or the result of a series of operations applied to the variables of the precondition.

Figure 6 shows the generated summary for strcmp. Recall that the precondition of strcmp has a single existentially quantified variable, δ, which denotes the difference between the values of the first non-matching pair of characters in $s1$ and $s2$. To obtain the symbolic value of δ, the summary calls the function $\mathit{fold}^{\beta}_{\mathrm{strd}}$ with the in-parameters of $\mathrm{strd}(s1, s2; \delta)$ (*i.e.*, $s1$ and $s2$) as its arguments. The function $\mathit{fold}^{\beta}_{\mathrm{strd}}$ computes the out-parameter δ by *folding* [11] the predicate $\mathrm{strd}(s1, s2; \delta)$ using its in-parameters. We delve into the specifics of folding/unfolding in our discussion of predicate definition compilation. Note that the summary works for all correctness properties; one needs only to substitute β with the desired property.

```
fn strcmp(s1, s2) {
    δ ← fold^β_strd(s1, s2);
    return δ
}
```

Fig. 6. Full strcmp summary.

Predicate Definitions. SumGen uses *fold/unfold reasoning* [11] to handle predicate definitions. In brief, *folding* a predicate computes its out-parameter from its in-parameters, while *unfolding* a predicate updates the resources reachable from its in-parameters using the value of its out-parameter. Hence, predicates in the precondition should be compiled into fold functions that resolve existential bindings in the specification, whereas predicates in the postcondition should be

Pure Assertion
$$\mathcal{A}^{\beta}(\pi) \triangleq \mathsf{assert}\ \pi$$

Directed Equality
$$\mathcal{A}^{\beta}(x \ominus e) \triangleq x \leftarrow e$$

Cell Assertion
$$\mathcal{A}^{\beta}(x \mapsto_{\tau} y) \triangleq y \leftarrow (\tau) * x$$

Predicate Assertion
$$\mathcal{A}^{\beta}(\alpha(\bar{e}; y)) \triangleq y \leftarrow \mathit{fold}^{\beta}_{\alpha}(\bar{e})$$

(a)

Leaf Node
$$\frac{t = \bullet}{\mathcal{C}^{\mathrm{EX}}_{\Gamma}(t) \triangleq \mathsf{skip}}$$

Single Node
$$\frac{t = \langle p,\, t' \rangle}{\mathcal{C}^{\mathrm{EX}}_{\Gamma}(t) \triangleq \mathcal{A}^{\mathrm{EX}}(p);\, \mathcal{C}^{\mathrm{EX}}_{\Gamma}(t')}$$

Double Node
$$\frac{t = \langle \pi,\, t_1,\, t_2 \rangle \qquad \Gamma = \alpha(\bar{x}; y)}{\begin{array}{l} \mathcal{C}^{\mathrm{EX}}_{\Gamma}(t) \triangleq \mathsf{if}\ (\mathsf{isCertain}(\pi))\ \{\mathcal{C}^{\mathrm{EX}}_{\Gamma}(t_1)\} \\ \quad \mathsf{elif}\ (\mathsf{isCertain}(\neg\pi))\ \{\mathcal{C}^{\mathrm{EX}}_{\Gamma}(t_2)\} \\ \quad \mathsf{else}\ \{y_1 \leftarrow \mathit{fold}^{\mathrm{EX}}_{\alpha}(\bar{x})\ \mathsf{with}\ \pi; \\ \qquad y_2 \leftarrow \mathit{fold}^{\mathrm{EX}}_{\alpha}(\bar{x})\ \mathsf{with}\ \neg\pi; \\ \qquad y \leftarrow \mathsf{ITE}(\pi, y_1, y_2)\} \end{array}}$$

(b)

Fig. 7. Compilers: (a) simple assertion, $\mathcal{A}^{\beta}$; and (b) exact matching tree, $\mathcal{C}^{\mathrm{EX}}_{\Gamma}$.

compiled into unfold functions that use these bindings to update the symbolic state. In what follows, we focus on the former, noting that the latter is its dual. We expand on unfold compilation when discussing the generation of summaries for mutating functions.

To compile predicates appearing in the precondition, we define a *predicate definition compiler* $\mathcal{P}^{\beta}(\Phi)$, again parameterized by a flag $\beta \in \{\mathrm{UX}, \mathrm{OX}, \mathrm{EX}\}$, that, given a predicate definition:

$$\Phi = \mathsf{pred}\ \alpha(\bar{x}; y)\,\{\bar{P}\}$$

generates a fold function observing the specified correctness property. Informally, $\mathcal{P}^{\beta}(\Phi)$ compiles the matching tree derived for the sequence of predicate cases $\bar{P}$ into a statement that computes the predicate's out-parameter from its in-parameters and returns it. The compilation of matching trees is at the core of the summary generation procedure, and depends on which correctness property the generated summary should observe.

Matching Tree Compilation The compilation of matching trees depends on whether we want to generate a UX, OX or EX summary. We define for each case a compilation function (henceforth referred to as *matching tree compiler*) that, given a matching tree t, generates a statement that computes the out-parameters of t. Here, we focus on the exact compiler, and then briefly outline how the under- and over-approximating variants differ from it.

Simple Assertion Compilation. The matching tree compilers make use of an auxiliary compiler $\mathcal{A}^{\beta} : \mathcal{SA} \rightarrow \mathcal{S}tmt$, formalized in Figure 7a, which compiles a simple assertion p into a statement that computes the out-parameters of p from its in-parameters. Broadly, $\mathcal{A}^{\beta}$ has four cases:

- **Pure Assertions:** A pure assertion π is compiled to the statement $\mathsf{assert}\ \pi$ that checks whether π is implied by the current path condition.

- **Directed Equalities:** A directed equality $x \circleddash e$ is compiled to the variable assignment $x \leftarrow e$, which assigns to the out-parameter x the in-expression e.
- **Cell Assertions:** A cell assertion $x \mapsto_\tau y$ is compiled to the statement $y \leftarrow (\tau) *x$, which assigns to the out-parameter y the value pointed to by x.
- **Predicate Assertions:** A predicate assertion $\alpha(\bar{e}; y)$ is compiled to the function call $y \leftarrow \mathit{fold}_\alpha^\beta(\bar{e})$, which assigns to the out-parameter y its corresponding value by folding the predicate α using its in-parameters.

Exact Matching Tree Compilation. We define a compiler $\mathcal{C}_\Gamma^{\mathrm{EX}} : \mathcal{MT} \to \mathcal{S}tmt$, formalized in Figure 7b, which, for some context Γ, transforms a valid matching tree into an EX summary $s \in \mathcal{S}tmt$. The context Γ contains the signature of the target predicate, or $\varnothing$ when compiling a specification. $\mathcal{C}_\Gamma^{\mathrm{EX}}$ has three cases, each corresponding to a matching tree constructor:

- **Leaf Nodes:** The leaf node is simply compiled to the skip instruction.
- **Single Nodes:** A single node $\langle p,\, t' \rangle$ is compiled by sequencing the compilation of p with that of t'.
- **Double Nodes:** A double node $\langle \pi,\, t_1,\, t_2 \rangle$ is compiled to an if-then-else conditional with three outcomes: **(1)** if π is certain, we follow the branch t_1; **(2)** if $\neg\pi$ is certain, we follow t_2; or **(3)** if neither is certain, we follow the *default* (exact) case. In the latter, the function builds an ITE expression capturing the value of the out-parameter y for both cases **(1)** and **(2)**. To this end, it calls itself under a path condition augmented with π, which will guide execution towards the left branch and compute y_1, and under a path condition augmented with $\neg\pi$, which will guide execution towards the right branch and compute y_2. It then assigns to the out-parameter y the expression $\mathsf{ITE}(\pi, y_1, y_2)$ modeling the two branches.

The strcmp summary shown in Figure 3 roughly mirrors the fold function $\mathit{fold}_{\mathrm{strd}}^{\mathrm{EX}}$ generated for the predicate $\mathrm{strd}(s1, s2; \delta)$. Notice how it builds the ITE out-expression in the exact case. For example, given the strings $\mathsf{s1} \mapsto [\hat{c}_1, \text{`\textbackslash 0'}]$ and $\mathsf{s2} \mapsto [\hat{c}_2, \text{`\textbackslash 0'}]$ (where $\hat{c}_1, \hat{c}_2$ are unconstrained symbolic characters), the first scoped call yields $\delta_1 = \hat{c}_1 - \hat{c}_2$, as the constraint implies that δ is the difference between the first pair of characters. Conversely, the second scoped call yields $\delta_2 = 0$, since in that case δ is the difference between the second pair of characters. The function then builds the out-expression:

$$\delta = \mathsf{ITE}(\hat{c}_1 = \text{`\textbackslash 0'} \vee \hat{c}_2 = \text{`\textbackslash 0'} \vee \hat{c}_1 \neq \hat{c}_2, \hat{c}_1 - \hat{c}_2, 0)$$

which exactly models the two possible values of δ.

Under-Approximating Matching Tree Compilation. The compilation of UX summaries is analogous to the EX case for leaf and single nodes. For double nodes, the compilation differs in the default case: instead of building an ITE expression modeling both branches, one path is chosen as a default path that will never be dropped by the engine. A double node is then compiled to an if-then-else conditional with two outcomes: **(1)** if the non-default case is implied by the path

```
fn fold^UX_strd(s1, s2) {
    c1 ← ∗s1;
    c2 ← ∗s2;
    π ← c1 = '\0' ∨ c2 = '\0' ∨ c1 ≠ c2;
    if (isCertain(π)) { δ ← c1 − c2 }
    else {
        assume ¬π;
        δ ← fold^UX_strd(s1 + 1, s2 + 1)
    }; return δ
}
```

(a)

```
fn fold^OX_strd(s1, s2) {
    c1 ← ∗s1;
    c2 ← ∗s2;
    π ← c1 = '\0' ∨ c2 = '\0' ∨ c1 ≠ c2;
    if (isCertain(π)) { δ ← c1 − c2 }
    elif (isCertain(¬π)) {
        δ ← fold^OX_strd(s1 + 1, s2 + 1)
    } else { δ ← symvar() };
    return δ
}
```

(b)

Fig. 8. Fold functions for $\mathrm{strd}(s1, s2; \delta)$: (a) $\mathit{fold}^{\mathrm{UX}}_{\mathrm{strd}}$ (UX), and (b) $\mathit{fold}^{\mathrm{OX}}_{\mathrm{strd}}$ (OX).

condition, we follow that path; **(2)** otherwise, we assume the constraints of the default case and follow that path.

Figure 8a shows the generated function $\mathit{fold}^{\mathrm{UX}}_{\mathrm{strd}}$. Notice how, unlike its EX counterpart, this function has a default case, $\neg\pi$:

$$\neg(c1 = \text{`\textbackslash 0'} \vee c2 = \text{`\textbackslash 0'} \vee c1 \neq c2) \;\equiv\; c1 \neq \text{`\textbackslash 0'} \wedge c2 \neq \text{`\textbackslash 0'} \wedge c1 = c2$$

towards which the engine is guided if π is not implied by the path condition. Given the strings $\mathsf{s1} \mapsto [\hat{c}_1, \text{`\textbackslash 0'}]$ and $\mathsf{s2} \mapsto [\hat{c}_2, \text{`\textbackslash 0'}]$, the fold function now returns $\delta = 0$ and adds $c1 \neq \text{`\textbackslash 0'} \wedge c2 \neq \text{`\textbackslash 0'} \wedge c1 = c2$ to the path condition, as the default case of the predicate assumes that $\hat{c}_1$ and $\hat{c}_2$ are equal non-null characters.

Over-Approximating Matching Tree Compilation. The compilation of OX summaries is similarly analogous to the EX case for leaf and single nodes. Conversely, a double node $\langle \pi, t_1, t_2 \rangle$ is compiled to an if-then-else conditional with three outcomes: **(1)** if π is certain, we follow the branch t_1; **(2)** if $\neg\pi$ is certain, we follow t_2; or **(3)** if neither is certain, we follow an over-approximating case. In the latter, the summary creates a fresh symbolic variable representing the out-parameter y and constrains it with the simple assertions shared by t_1 and t_2, if any.

Figure 8b shows the generated function $\mathit{fold}^{\mathrm{OX}}_{\mathrm{strd}}$. Notice that, instead of guiding execution towards a default case, the function guides it towards an OX case that generates a new symbolic variable representing the out-parameter. Given the strings $\mathsf{s1} \mapsto [\hat{c}_1, \text{`\textbackslash 0'}]$ and $\mathsf{s2} \mapsto [\hat{c}_2, \text{`\textbackslash 0'}]$, the fold function now returns a fresh unconstrained symbolic variable δ.

Mutating Summary Generation We give a brief overview of the summary generation procedure for mutating functions by appealing to the example of the LIBC function strcpy. Given two pointers, *dest* and *src*, strcpy copies the string pointed to by *src*, up to and including the first null byte, to the address specified

(a)

src	$\hat{c}_1$	$\hat{c}_2$	'\0'
$dest$	$\hat{d}_1$	$\hat{d}_2$	$\hat{d}_3$

(b)

src	$\hat{c}_1$	$\hat{c}_2$	'\0'
$dest$	$\hat{c}_1$	$ITE(\hat{c}_1 = \text{'\textbackslash 0'}, \hat{d}_2, \hat{c}_2)$	$ITE(\hat{c}_1 = \text{'\textbackslash 0'} \vee \hat{c}_2 = \text{'\textbackslash 0'}, \hat{d}_3, \text{'\textbackslash 0'})$

Fig. 9. Visualizing the memory: (a) before and (b) after running strcpy$(dest, src)$ (EX).

by $dest$. We write the strcpy specification as:

$$\{\text{cstr}(src; \sigma) \uplus \text{len}(\sigma; n) \uplus \text{allocd}(dest, n+1)\}$$
$$\textsf{fn strcpy}(dest, src)$$
$$\{\text{cstr}(src; \sigma) \uplus \text{cstr}(dest; \sigma);\ dest\}$$

where: **(i)** $\text{cstr}(s; \sigma)$ states that the string s can be mathematically represented as the list of characters σ; **(ii)** $\text{len}(\sigma; n)$ states that the list σ has length n; and **(iii)** $\text{allocd}(l, n)$ asserts that n consecutive cells are allocated starting at l. The predicate $\text{cstr}(s; \sigma)$ has two cases: **(1)** s points to '\0' and σ is empty; or **(2)** s points to a non-null character c and σ is constructed by prepending c to the list σ' representing the rest of the string. Put formally:

$$\textsf{pred}\ \text{cstr}(s; \sigma)\ \{ s \mapsto \text{'\textbackslash 0'} \uplus \sigma \doteq [\,];$$
$$s \mapsto c \uplus c \neq \text{'\textbackslash 0'} \uplus \sigma \doteq c : \sigma' \uplus \text{cstr}(s+1;\ \sigma')\}$$

Figure 10 shows the generated summary for strcpy. The summary starts by obtaining the symbolic values of σ and n by calling the fold functions $fold^{\beta}_{\text{cstr}}$ and $fold^{\beta}_{\text{len}}$. It then checks via the SumGen builtin allocd whether there exist n consecutive allocated cells starting at address $dest$. If the check succeeds, the summary calls $unfold^{\beta}_{\text{cstr}}$ with $dest$ and σ as arguments and returns $dest$. The function $unfold^{\beta}_{\text{cstr}}$ updates the memory segment pointed to by $dest$ using the character list σ by *unfolding* [11] the predicate $\text{cstr}(s; \sigma)$. Note that the summary does not update the resource src, as it remains unchanged from the precondition.

```
fn strcpy(dest, src) {
    σ ← fold^β_cstr(src);
    n ← fold^β_len(σ);
    allocd(dest, n + 1);
    unfold^β_cstr(dest, σ);
    return dest
}
```

Fig. 10. Full strcpy summary.

Figure 9 shows the symbolic memory during an execution of the EX strcpy summary. We assume that a sufficient number of consecutive memory cells (three, in this case) are allocated starting at address $dest$. Figure 9a shows the memory segments pointed to by src and $dest$ before calling the summary, where $\hat{c}_1, \hat{c}_2$ are unconstrained symbolic characters and $\hat{d}_1, \ldots, \hat{d}_3$ are the previous contents of $dest$, symbolic or otherwise. For simplicity, we assume that the path condition is initially set to true.

When executing the EX strcpy summary, the call $fold^{\text{EX}}_{\text{cstr}}(src)$ produces a list given by the symbolic expression:

$$\sigma = \textsf{ITE}(\hat{c}_1 = \text{'\textbackslash 0'}, [\,], \textsf{ITE}(\hat{c}_2 = \text{'\textbackslash 0'}, [\hat{c}_1], [\hat{c}_1, \hat{c}_2]))$$

```
1 void vuln1() {
2     char dest[5];
3     char src[20] = "aaaabbbbcccc";
4     strcpy(dest, src);
5 }
```

(a)

```
1 void vuln2() {
2     char src[20] = "aaaabbbbcccc";
3     strcpy(src + 1, src);
4 }
```

(b)

Fig. 11. Issues: (a) out-of-bounds write (incorrect behavior), and (b) overlapping buffers (undefined behavior).

which exactly models the possible values of *src*. Given the list σ, the call $unfold^{\mathrm{EX}}_{\mathrm{cstr}}(dest, \sigma)$ results in the memory given in Figure 9b. Let us analyze each memory cell from *dest* at a time. Regardless of the value of $\hat{c}_1$, the *first cell* has content $\hat{c}_1$: if $\hat{c}_1 \neq$ '\0', it is copied to *dest*; if $\hat{c}_1 =$ '\0', the summary stores '\0' at *dest*, making it also equal to $\hat{c}_1$. Next, if $\hat{c}_1 =$ '\0', the content of the *second cell* is left unchanged; otherwise, it is set to $\hat{c}_2$. In the latter case, $\hat{c}_2$ is copied to *dest* + 1 even if $\hat{c}_2 =$ '\0', similarly to what happens in the first cell. Finally, if $\hat{c}_1 =$ '\0' or $\hat{c}_2 =$ '\0', the content of the *third cell* is left unchanged; otherwise, it is set to '\0'. One may easily confirm that the generated output memory is correct by replacing the symbolic characters with concrete values. For instance, executing the summary with $src \mapsto$ ['a', 'b', '\0'] yields $dest \mapsto$ ['a', 'b', '\0'], while executing it with $src \mapsto$ ['a', '\0', '\0'] yields $dest \mapsto$ ['a', '\0', $\hat{d}_3$] and executing it with $src \mapsto$ ['\0', '\0', '\0'] yields $dest \mapsto$ ['\0', $\hat{d}_2$, $\hat{d}_3$].

The compilation of unfold functions is dual to that of fold functions. Whereas in the latter we compile the corresponding matching tree to compute its out-parameter from its in-parameters, in the former we compile it to update the resources reachable from its in-parameters using the value of its out-parameter.

3.3 Usage in Symbolic Execution Tools

We discuss how SumGen summaries can be used by symbolic execution tools to analyze C programs. We focus on two main issues: *bugs and incorrect behavior* and *undefined behavior*.

Bugs and Incorrect Behavior One of the main applications of symbolic execution is to find bugs in programs. For example, a symbolic execution engine should detect accesses to unallocated memory, as well as those that are incorrectly typed. To that end, both our specification language and SumGen summaries should also capture and prevent incorrect behavior.

Consider, for instance, the C function vuln1 shown in Figure 11a, which copies a larger string src to a smaller buffer dest via the libc function strcpy. Since only five bytes are allocated for dest, the copy operation writes past the end of the buffer, resulting in a segmentation fault. Our specification for strcpy disallows this behavior by requiring that the destination buffer be large enough to hold the source string via the predicate $\mathrm{allocd}(l, n)$. As a consequence, the

generated summary will also prevent out-of-bounds writes through the SumGen builtin allocd (*cf.* Figure 10, line 4). In this example, dest is only allocated five bytes, while the src string requires at least 13, so the engine will detect that the precondition is violated and report an error.

Undefined Behavior Another common issue in C programs is undefined behavior, which occurs when a program executes operations that are not well-defined by the standard. For example, consider the function vuln2 shown in Figure 11b, which copies a string from src to src + 1, causing source and destination buffers to overlap. According to the ISO/IEC 9899:2024 C standard [32], calling strcpy on overlapping buffers results in undefined behavior. The risk is that if a strcpy implementation writes to the destination buffer before reading the full source, it may overwrite data that is yet to be copied and enter an infinite loop. While modern libc implementations like *glibc* [23] avoid this by caching the source buffer before writing it to the destination, such behavior is not enforced by the standard. By default, SumGen summaries do not disallow undefined behavior; for instance, our strcpy summary assumes the caching approach. If the developer wishes to exclude such behavior, or enforce a different solution to handle it, they must express this explicitly in the function's precondition.

4 Soundness of Summary Generation

In this section, we establish the soundness of the summary generation procedure. We first define the formal semantics of specifications and symbolic summaries (Definition 2). Essentially, both semantics are defined in terms of sets of tuples of the form (μ_i, ρ, μ_o, v), where μ_i and ρ are the input memory and variable store, μ_o is the output memory, and v is the returned value. Note that we do not take into account the output variable store, as it cannot be accessed by the calling function.

Definition 2 (Formal Semantics of Specifications and Summaries).

Specification Semantics

$$[\![\{P\}\,\mathsf{fn}\,f(\bar{x})\,\{Q; y; \pi\}]\!] \triangleq \{(\mu, \rho, \mu', \varepsilon(y)) \mid \mu, \rho, \varepsilon \vDash P \wedge \mu', \rho, \varepsilon \vDash Q \wedge \varepsilon \vDash \pi\}$$

Summary Semantics

$$[\![\mathsf{fn}\,f(\bar{x})\,\{s\}]\!] \triangleq \{(\mu, \rho, \mu', \varepsilon(\hat{v})) \mid [\![\hat{\mu}, \hat{\rho}]\!]_\varepsilon = (\mu, \rho) \wedge [\![\hat{\mu}']\!]_\varepsilon = \mu' \\ \wedge \langle \hat{\mu}, \hat{\rho}, s\rangle \Downarrow \langle \hat{\mu}', \pi, \hat{v}\rangle \wedge \varepsilon \vDash \pi\}$$

In the definition, $\mu, \rho, \varepsilon \vDash P$ denotes the satisfiability relation for spatial formulas introduced by Hobor and Villard [31], which differs from the standard Separation Logic satisfiability relation [50,56] in that it accounts for the overlapping conjunction. Essentially, we say that a memory μ, variable store ρ, and logical environment ε satisfy an assertion P if μ exactly matches the resource described by P after replacing program and logical variables with the mappings given by ρ

and ε, respectively. Furthermore, we write: **(i)** $\varepsilon \models \pi$ for standard first-order satisfiability; **(ii)** $[\![\hat{\mu}, \hat{\rho}]\!]_\varepsilon = (\mu, \rho)$ to mean that the concrete state (μ, ρ) corresponds to the interpretation of the symbolic state $(\hat{\mu}, \hat{\rho})$ under the logical environment $\varepsilon : \mathcal{X} \rightharpoonup \mathcal{V}$ that maps symbolic variables to concrete values; **(iii)** $[\![\hat{\mu}']\!]_\varepsilon = \mu'$ to mean that the concrete memory μ' corresponds to the interpretation of the symbolic memory $\hat{\mu}'$ under ε; and **(iv)** $\langle \hat{\mu}, \hat{\rho}, s \rangle \Downarrow \langle \hat{\mu}', \pi, \hat{v} \rangle$ to capture the symbolic execution of s that starts in the input state $(\hat{\mu}, \hat{\rho})$ and finishes in an output state with memory $\hat{\mu}'$, path condition π and return value $\hat{v}$. Both semantics are defined using set comprehension style, with variables appearing on the right-hand side of the comprehension but not on the left assumed to be existentially quantified.

Next, we define what it means for a specification/summary to be UX, OX, and EX, using Ξ to range over both specifications and summaries. In a nutshell, a specification/summary is: **(1)** UX if all the return values/memories it models are return values/memories of the corresponding function; **(2)** OX if it models all the return values/memories of the corresponding function; and **(3)** EX if it models all the return values/memories of the corresponding function and only those, *i.e.*, if it is both UX and OX. Definition 3 captures these intuitions. In the definition, we use $\langle \mu, \rho, s \rangle \Downarrow \langle \mu', v \rangle$ to denote the concrete execution that starts in state (μ, ρ) and generates the output memory μ' and return value v.

Definition 3 (Correctness Properties). *Let* $\mathsf{fn}\, f(\bar{x})\, \{s\}$ *be a concrete function. A specification or symbolic summary* Ξ *for* f *is said to be:*

- *UX iff* $\forall \mu, \rho, \mu', v. (\mu, \rho, \mu', v) \in [\![\Xi]\!] \implies \langle \mu, \rho, s \rangle \Downarrow \langle \mu', v \rangle$.
- *OX iff* $\forall \mu, \rho, \mu', v. \langle \mu, \rho, s \rangle \Downarrow \langle \mu', v \rangle \implies (\mu, \rho, \mu', v) \in [\![\Xi]\!]$.
- *EX iff it is both UX and OX.*

Theorem 1 captures the soundness of the summary generation procedure. In a nutshell, it guarantees that: **(i)** $\mathcal{S}^{\text{UX}}$ generates summaries whose semantics is included in that of the given specification; **(ii)** $\mathcal{S}^{\text{OX}}$ generates summaries whose semantics includes that of the given specification; and **(iii)** $\mathcal{S}^{\text{EX}}$ generates summaries whose semantics coincides with that of the given specification.

Theorem 1 (Summary Generation: Soundness). *Suppose that:*

$$\mathcal{S}^{\beta}(\{P\}\, \mathsf{fn}\, f(\bar{x})\, \{Q; y; \pi\}) = \mathsf{fn}\, f(\bar{x})\, \{s\}$$

Then it holds that:

- *If* $\beta = \text{UX}$, *then* $[\![\{P\}\, \mathsf{fn}\, f(\bar{x})\, \{Q; y; \pi\}]\!] \supseteq [\![\mathsf{fn}\, f(\bar{x})\, \{s\}]\!]$.
- *If* $\beta = \text{OX}$, *then* $[\![\{P\}\, \mathsf{fn}\, f(\bar{x})\, \{Q; y; \pi\}]\!] \subseteq [\![\mathsf{fn}\, f(\bar{x})\, \{s\}]\!]$.
- *If* $\beta = \text{EX}$, *then* $[\![\{P\}\, \mathsf{fn}\, f(\bar{x})\, \{Q; y; \pi\}]\!] = [\![\mathsf{fn}\, f(\bar{x})\, \{s\}]\!]$.

An immediate corollary of the theorem is the set of properties summarized in Table 1, which we state below as a theorem.

Theorem 2 (Correctness of Generated Summaries). *Suppose that:*

$$\mathcal{S}^{\beta}(\{P\}\, \mathsf{fn}\, f(\bar{x})\, \{Q; y; \pi\}) = \mathsf{fn}\, f(\bar{x})\, \{s\}$$

and $\{P\}\, \mathsf{fn}\, f(\bar{x})\, \{Q; y; \pi\}$ *is a* β*-specification. Then,* $\mathsf{fn}\, f(\bar{x})\, \{s\}$ *is a* β*-summary.*

Table 2. Correctness of generated summaries.

Function Categories			Specifications				Summary Correctness			
			UX	OX	EX	N_{Total}	UX	OX	EX	N_{Total}
Non-Mutating	*Characters*	13	13	13	13	39	13	13	13	39
	I/O	3	2	3	2	7	n/a	n/a	n/a	n/a
	Numbers	5	1	5	1	7	1	4	1	6
	Strings	13	13	13	13	39	13	13	13	39
Mutating	*Memory*	7	7	7	7	21	5	5	5	15
	Strings	6	6	6	6	18	6	6	6	18
Total		47	42	47	42	131	38	41	38	117

5 Evaluation

In this section, we answer the following evaluation questions:

EQ1: Is SumGen capable of generating correct UX, OX and EX summaries?
EQ2: How does the complexity of SumGen specifications compare to that of handcrafted summaries?
EQ3: How does the performance of SumGen summaries compare to that of handcrafted summaries?

5.1 EQ1: Correctness of Generated Summaries

To evaluate the expressivity of our assertion language and the soundness of the subsequent summary generation, we developed: **(i)** a set of 92 specifications encompassing 34 non-mutating libc functions from four different categories (*Characters*, *I/O*, *Numbers* and *Strings*); and **(ii)** a set of 39 specifications covering 13 mutating libc functions from two different categories (*Memory* and *Strings*).

For most of these functions, we defined one specification of each type (UX, OX and EX), generating a total of 131 summaries out of a possible 141. While the theoretical results in §4 guarantee the soundness of our approach, it is still possible to introduce soundness errors at the implementation level. To demonstrate that this does not occur, we used SumBoundVerify [54] to check if each generated summary adheres to its specified property. SumBoundVerify validates the bounded correctness of a summary by comparing the paths it models with those generated by symbolically executing the corresponding function.

The validation results are shown in Table 2. Out of 131 generated summaries, we were able to confirm the correctness of 117. All 14 non-validated summaries correspond to functions which SumBoundVerify does not support, as they step outside the bounds of the engine (*e.g.*, `putchar` and `malloc`).

> **Takeaway EQ1:** All SumGen-generated summaries supported by SumBoundVerify were successfully validated (117/131).

Table 3. Average LoC of SumGen specifications vs. handcrafted summaries (C/Py).

		Non-Mutating				Mutating		All
		Characters	*I/O*	*Numbers*	*Strings*	*Memory*	*Strings*	
UX	*Specs*	9	9	–	12	14	14	12
	Handcrafted	18 / –	20 / –	– / –	28 / –	21 / –	27 / –	25 / –
OX	*Specs*	9	–	9	11	–	–	10
	Handcrafted	19 / –	– / –	42 / –	17 / –	– / –	– / –	22 / –
EX	*Specs*	9	5	–	10	14	13	11
	Handcrafted	21 / 5	5 / 7	– / –	38 / 126	– / 42	– / 108	28 / 75

5.2 EQ2: Complexity of Specifications vs. Summaries

We assessed the complexity of SumGen specifications, measured in terms of lines of code (LoC), against a baseline of: **(1)** native *angr* [59] summaries implemented in Python; and **(2)** tool-independent summaries developed by Ramos et al. [54] directly in C. We note that the LoC criterion is not exhaustive, and may not always reflect the true complexity of a specification/summary. However, it serves as a quantitative metric to assess the complexity of specifications versus that of summaries, and, in our own experience, provides a good estimate of the required implementation effort.

SumGen allows for the generation of UX, OX and EX summaries. However, our baseline of handcrafted summaries does not provide every kind of summary for all of the analyzed libc functions; for instance, *angr* offers a single, usually exact, summary per function. In contrast, we typically generate three summaries, one per correctness property. To address this discrepancy, for this experiment we consider only the functions for which we have both an input specification and a handcrafted summary in C and/or Python.

Results. Table 3 shows the average number of LoC, computed with *cloc* [14], across the six function categories. We group specifications and summaries by correctness property, showing, for each category, the average LoC of our specifications versus that of handcrafted summaries. For the latter, we give in each cell the LoC for both the C and Python summaries.

Results show that, across all correctness properties, SumGen specifications are approximately 56% and 85% shorter than handcrafted summaries implemented in C and Python, respectively. These results are expected, given that specifications are typically simpler than summaries.

Takeaway EQ2: On average, SumGen specifications are 56% shorter than handcrafted C summaries and 85% shorter than handcrafted Python ones.

5.3 EQ3: Performance of Generated Summaries

We measured the execution time and code coverage of generated and handcrafted summaries on two symbolic test suites.

Test Suites. To compare the performance of SumGen summaries against that of handcrafted ones in the wild, we evaluated them on a symbolic test suite based on real-world codebases. Popular test suites for symbolic execution such as *TestComp* [6] and *SVComp* [7] make very limited use of libc functions, as they were designed to evaluate the core performance of symbolic execution engines rather than support for libc. To evaluate SumGen summaries, we required a symbolic test suite with emphasis on libc usage. We built on the symbolic test suite developed by Ramos et al. [54], based on two open-source C libraries that make heavy use of libc functions: **(i)** the *HashMap* [61] library, which offers an implementation of a standard hash table; and **(ii)** the *Dynamic Strings* [57] library, which extends the string handling functionality of libc by introducing support for dynamic-size heap-allocated strings. To maximize interactions with libc and improve code coverage, we implemented 63 new symbolic tests across the two libraries. In particular, we extended the *HashMap* library from 10 original tests to 20, and the *Dynamic Strings* library from 12 to 65.

Experimental Setup. As the test bed for our experiments, we extended *angr*'s symbolic engine with support for our generated summaries. *angr* [59] is a widely used binary analysis toolkit developed at UC Santa Barbara that has the ability to perform symbolic execution on C-compiled binaries. All tests were run on an Ubuntu server (18.04.5 LTS) with an Intel Xeon E5–2620 CPU and 32 GB of RAM. Each test was allowed to consume a maximum of 16 GB of RAM with a timeout of 1800 seconds (30 minutes).

Results. We ran *angr* on our symbolic test suite using four categories of summaries: **(1)** our generated C summaries (*Generated-C*); **(2)** our generated Python summaries (*Generated-Python*); **(3)** Ramos et al.'s C-implemented summaries (*Handcrafted-C*); and **(4)** *angr*'s native summaries (*Handcrafted-Python*). For reference, we also executed the test suites using concrete implementations of the libc functions (*Concrete*), which we sourced from the *glibc* [23] and *libiberty* [22] libc implementations. Note that not all test runs include categories **(3)** and **(4)**. The UX and OX runs do not include *Handcrafted-Python* summaries, given that *angr*'s native summaries are all EX. Conversely, Ramos et al.'s mutating summaries are exclusively UX, and so are excluded from the OX and EX runs.

The combined results for the *HashMap* and *Dynamic Strings* libraries are given in Table 4. We show for each summary type: **(i)** the number of tests that failed due to exceeding the memory limit (Memout); **(ii)** the number of tests that failed due to exceeding the time limit (Timeout); **(iii)** the number of tests that executed successfully (Success); **(iv)** the average number of explored paths per test (Avg. N_{Paths}); **(v)** the average execution time per test (Avg. $Time$); and **(vi)** the line coverage achieved using the inputs computed by the symbolic engine, expressed as a percentage of the total number of lines (Total $Cov.$).

Results show that the performance of the generated summaries is comparable to that of their handcrafted counterparts. In the UX case, *Generated-C* summaries perform slightly better than *Handcrafted-C* summaries in terms of

Table 4. Performance of SumGen summaries on the *HashMap* and *DStrings* libraries.

Summaries			Memout ✗	Timeout ✗	Success ✓	Avg. N_{Paths}	Avg. *Time* (*s*)	Total *Cov.* (%)
UX	*Handcrafted*	C	2	10	73	47	187.75	79%
	Generated	C	2	9	74	65	194.2	81%
		Python	4	7	74	70	121.21	81%
OX	*Generated*	C	40	23	22	226	921.27	65%
		Python	3	55	27	60	887.25	64%
EX	*Handcrafted*	Python	6	1	78	90	104.28	82%
	Generated	C	11	18	56	87	630.36	82%
		Python	10	10	65	78	247.08	82%
Concrete			19	14	52	1.20k	521.85	80%

completed tests (74 vs. 73) and line coverage (81% vs. 79%). In the EX case, results are more mixed: *Generated-Python* summaries underperform compared to *Handcrafted-Python* summaries in completed tests (65 vs. 78), while performing identically in terms of coverage (82% for both). Finally, in the OX case, we have no baseline to compare to. However, we note that the performance of OX summaries is considerably worse than that of its UX and EX counterparts.

The comparison with *angr* warrants a more careful examination. First, as we have previously noted, *angr*'s summaries are often buggy, excluding feasible paths and including spurious ones. This skews the results in their favor, as these summaries may ignore relevant paths without being adequately penalized. Second, *angr*'s summaries are tailored to the specificities of its underlying engine. In contrast, SumGen summaries are designed to be compatible with any symbolic execution tool that implements our symbolic reflection API, and therefore do not exploit the internal details of *angr*'s implementation. Moreover, with our methodology, a developer can easily create native summaries in Python for *angr*, which consistently execute faster than summaries written in C (since C summaries must be interpreted), without needing any knowledge of *angr*'s architecture and implementation details.

Takeaway EQ3: The performance of SumGen summaries is comparable to the performance of handcrafted summaries.

6 Discussion

We discuss the implications of our work, focusing on two main points: the *recommended summary development workflow* and the *limitations of our approach.*

Summary Development Workflow Developers often write summaries without much guidance. SumGen helps streamline summary development by automatically generating summaries from function specifications. However, this

approach alone is not foolproof, as writing specifications remains error-prone, particularly for non-experts.

To help avoid mistakes, we recommend using SumGen in combination with SumBoundVerify [54]. The workflow is straightforward: the developer writes a specification, uses SumGen to generate a summary, and then validates it against a reference implementation with SumBoundVerify. If the specification is incorrect, the generated summary will also be incorrect, and SumBoundVerify will produce a counterexample. Thus, validating the summary also indirectly validates the specification, reducing the need to trust it blindly.

Limitations While general, our approach has both theoretical and practical limitations. First, we assume that all input specifications are well-formed. Second, our theoretical framework is restricted to modeling functions whose behavior can be expressed through a finite number of recursive predicates where out-parameters can be deterministically computed from in-parameters. As a result, we do not support, among others: **(i)** non-deterministic functions; **(ii)** higher-order functions; and **(iii)** concurrency. Additionally, while theoretically allowed by our approach, the current implementation does not support: **(i)** structures; and **(ii)** interactions with the file system.

7 Related Work

Summaries in Symbolic Execution There is a vast body of work on the use of summaries in symbolic execution. Existing approaches can be roughly divided into three main types of summaries: *first-order summaries* [24,30,38], *structured summaries* [19,52] and *operational summaries* [9,12,15,44,54,55,59].

A first-order summary is a simple first-order formula with either limited support for reasoning about heap memory or no support at all. Early work in the area is attributed to Godefroid et al. [1,24,25,26], who leverage *compositionality* by symbolically executing functions in isolation and producing first-order summaries that can later be reused to analyze code that relies on those functions. In contrast, structured summaries are able to reason about the heap. Qiu et al. [52] introduce *memoization trees*, a tree-like data structure that captures the various paths in a function and their respective path conditions, including constraints on the heap memory. Fragoso Santos et al. [19] propose JaVerT 2.0, a compositional symbolic execution tool for JavaScript that allows for the generation of Separation Logic-based specifications, which can be used as function summaries by its symbolic execution engine.

Operational summaries are a loose grouping of summaries developed for the express purpose of usage in symbolic execution tools. Interestingly, despite their widespread use in practice [9,12,15,44,55,59], existing work on operational summaries is sparse. To the best of our knowledge, Ramos et al. [54] were the first to address their formalization and verification. Their work further includes a symbolic reflection API for the implementation of tool-independent summaries,

which SumGen supports, and SumBoundVerify, which we use to validate our summaries.

Specification-Based Synthesis Prior work in specification-based synthesis includes *test synthesis* [13,18,58], *program synthesis* [33,34,51,60] and *wrapper synthesis* [46]. Cosette [18] allows for the generation of symbolic tests for JavaScript from Separation Logic specifications. Similarly to our own use of matching trees, Cosette employs *unification* to find the bindings of existentially quantified variables and generate executable code. SuSLik [51] is a program syntheziser that works by reducing the problem of deriving heap-manipulating programs to a proof search under Synthetic Separation Logic (SSL) [33,51]. The authors have since extended their work to *cyclic program synthesis* [33], *synthesis certification* [60] and *synthesis in Rust* [16]. SLICK [46] is a runtime checker for Java programs that ensures that the pre- and postconditions of a function are met before and after its execution, respectively. SLICK makes use of a partial ordering of Separation Logic formulas by means of a topological sort similar to matching trees. However, to the best of our knowledge, SumGen is the first tool that is able to generate *symbolic summaries* from function specifications.

8 Conclusions

Symbolic summaries are an essential tool for modern symbolic execution engines to tackle the challenges of modeling interactions with the runtime environment and countering path explosion. However, the development of summaries remains to this day a manual task that is known to be highly error-prone. In this paper, we propose a novel methodology for generating correct-by-construction summaries from function specifications, which we realize as the tool SumGen. We used SumGen to generate a total of 131 summaries for 47 libc functions. Our evaluation shows that SumGen summaries are not only correct, but also easier to obtain and as performant as their handcrafted counterparts, demonstrating the effectiveness of our methodology in producing symbolic summaries for real-world, highly complex code.

Acknowledgments. We thank the anonymous reviewers for their comments and insightful feedback. This work was supported by national funds through Fundação para a Ciência e a Tecnologia, I.P. (FCT) via a CMU Portugal Dual Degree PhD fellowship (ref. 2024.12581.PRT), as well as projects UID/50021/2025 (doi: 10.54499/UID/50021/2025), UID/PRR/50021/2025 (doi: 10.54499/UID/PRR/50021/2025), and WebCAP (ref. 2024.07393.IACDC, doi: 10.54499/2024.07393.IACDC), and by IAPMEI under grant ref. C6632206063-00466847 (SmartRetail).

Data Availability Statement. SumGen, including its source code, benchmarks, and experimental data, is open source [28] and publicly available online as a web application [27].

References

1. Anand, S., Godefroid, P., Tillmann, N.: Demand-driven compositional symbolic execution. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 367–381. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
2. Appel, A.W., Beringer, L., Cao, Q.: Verifiable C, Software Foundations, vol. 5. Electronic textbook (2022), version 1.2.1. http://softwarefoundations.cis.upenn.edu
3. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Comput. Surv. **51**(3) (May 2018). https://doi.org/10.1145/3182657
4. Baranová, Z., Barnat, J., Kejstová, K., Kučera, T., Lauko, H., Mrázek, J., Ročkai, P., Štill, V.: Model checking of C and C++ with DIVINE 4. In: Automated Technology for Verification and Analysis. pp. 201–207. Springer International Publishing, Cham (2017)
5. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: CVC5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 415–442. Springer International Publishing, Cham (2022)
6. Beyer, D.: Advances in automatic software testing: Test-Comp 2022. In: Fundamental Approaches to Software Engineering - 25th International Conference, FASE 2022. Lecture Notes in Computer Science, vol. 13241, pp. 321–335. Springer (2022)
7. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022. Lecture Notes in Computer Science, vol. 13244, pp. 375–402. Springer (2022)
8. Boyer, R.S., Elspas, B., Levitt, K.N.: SELECT: A formal system for testing and debugging programs by symbolic execution. In: Proceedings of the International Conference on Reliable Software. p. 234–245. Association for Computing Machinery, New York, NY, USA (1975). https://doi.org/10.1145/800027.808445
9. Cadar, C., Dunbar, D., Engler, D.: 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. p. 209–224. OSDI'08, USENIX Association, USA (2008)
10. Chalupa, M., Mihalkovič, V., Řechtáčková, A., Zaoral, L., Strejček, J.: Symbiotic 9: String analysis and backward symbolic execution with loop folding. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 462–467. Springer International Publishing, Cham (2022)
11. Chin, W.N., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Science of Computer Programming **77**(9), 1006–1036 (2012). https://doi.org/10.1016/j.scico.2010.07.004, the Programming Languages track at the 24th ACM Symposium on Applied Computing (SAC'09)
12. Chipounov, V., Kuznetsov, V., Candea, G.: The S2E platform: Design, implementation, and applications. ACM Trans. Comput. Syst. **30**(1) (Feb 2012). https://doi.org/10.1145/2110356.2110358

13. Claessen, K., Hughes, J.: QuickCheck: A lightweight tool for random testing of Haskell programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming. p. 268–279. ICFP '00, Association for Computing Machinery, New York, NY, USA (2000). https://doi.org/10.1145/351240.351266
14. Danial, A.: cloc: v1.92 (Dec 2021). https://doi.org/10.5281/zenodo.5760077
15. David, R., Bardin, S., Ta, T.D., Mounier, L., Feist, J., Potet, M.L., Marion, J.Y.: Binsec/SE: A dynamic symbolic execution toolkit for binary-level analysis. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). vol. 1, pp. 653–656 (2016). https://doi.org/10.1109/SANER.2016.43
16. Fiala, J., Itzhaky, S., Müller, P., Polikarpova, N., Sergey, I.: Leveraging Rust types for program synthesis. Proc. ACM Program. Lang. **7**(PLDI) (Jun 2023). https://doi.org/10.1145/3591278
17. Fragoso Santos, J., Maksimović, P., Ayoun, S.E., Gardner, P.: Gillian, Part I: A multi-language platform for symbolic execution. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 927–942. PLDI 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3385412.3386014
18. Fragoso Santos, J., Maksimović, P., Grohens, T., Dolby, J., Gardner, P.: Symbolic execution for JavaScript. In: Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming. PPDP '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3236950.3236956
19. Fragoso Santos, J., Maksimović, P., Sampaio, G., Gardner, P.: JaVerT 2.0: Compositional symbolic execution for JavaScript. Proc. ACM Program. Lang. **3**(POPL) (Jan 2019). https://doi.org/10.1145/3290379
20. Gardner, P., Maffeis, S., Smith, G.D.: Towards a program logic for JavaScript. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 31–44. POPL '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2103656.2103663
21. Gardner, P., Ntzik, G., Wright, A.: Local reasoning for the POSIX file system. In: Shao, Z. (ed.) Programming Languages and Systems. pp. 169–188. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
22. GNU: GNU libiberty (2022), https://gcc.gnu.org/onlinedocs/libiberty/, accessed: January 22, 2026
23. GNU: The GNU C library (2022), https://www.gnu.org/software/libc/, accessed: January 22, 2026
24. Godefroid, P.: Compositional dynamic test generation. SIGPLAN Not. **42**(1), 47–54 (Jan 2007). https://doi.org/10.1145/1190215.1190226
25. Godefroid, P., Luchaup, D.: Automatic partial loop summarization in dynamic test generation. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. p. 23–33. ISSTA '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/2001420.2001424
26. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.D.: Compositional may-must program analysis: Unleashing the power of alternation. In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 43–56. POPL '10, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1706299.1706307
27. Gonçalves, R., Ramos, F., Adão, P., Fragoso Santos, J.: SumGen Web Interface (2025), https://sumsynth.duckdns.org/

28. Gonçalves, R., Ramos, F., Adão, P., Fragoso Santos, J.: Specification-Driven Generation of Summaries for Symbolic Execution (Artifact) (2026). https://doi.org/10.6084/m9.figshare.30992707
29. Gonçalves, R., Ramos, F., Adão, P., Fragoso Santos, J.: Specification-Driven Generation of Summaries for Symbolic Execution (Extended Version) (2026). https://doi.org/10.5281/zenodo.18296457
30. Gopan, D., Reps, T.: Low-level library analysis and summarization. In: Damm, W., Hermanns, H. (eds.) Computer Aided Verification. pp. 68–81. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
31. Hobor, A., Villard, J.: The ramifications of sharing in data structures. In: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 523–536. POPL '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2429069.2429131
32. ISO/IEC 9899:2024: Information technology – Programming languages – C. International Standard, International Organization for Standardization and International Electrotechnical Commission, Geneva, CH (Oct 2024)
33. Itzhaky, S., Peleg, H., Polikarpova, N., Rowe, R.N.S., Sergey, I.: Cyclic program synthesis. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 944–959. PLDI 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3453483.3454087
34. Itzhaky, S., Peleg, H., Polikarpova, N., Rowe, R.N.S., Sergey, I.: Deductive synthesis of programs with pointers: Techniques, challenges, opportunities. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification. pp. 110–134. Springer International Publishing, Cham (2021)
35. King, J.C.: A new approach to program testing. In: Proceedings of the International Conference on Reliable Software. p. 228–233. Association for Computing Machinery, New York, NY, USA (1975). https://doi.org/10.1145/800027.808444
36. Leijen, D., Meijer, E.: Parsec: A practical parser library. Electronic Notes in Theoretical Computer Science **41**(1), 1–20 (2001)
37. Li, G., Andreasen, E., Ghosh, I.: SymJS: Automatic symbolic testing of JavaScript web applications. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. p. 449–459. FSE 2014, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2635868.2635913
38. Lin, Y., Miller, T., Søndergaard, H.: Compositional symbolic execution using fine-grained summaries. In: 2015 24th Australasian Software Engineering Conference. pp. 213–222 (2015). https://doi.org/10.1109/ASWEC.2015.32
39. Lööw, A., Nantes-Sobrinho, D., Ayoun, S.E., Maksimović, P., Gardner, P.: Matching plans for frame inference in compositional reasoning. In: Aldrich, J., Salvaneschi, G. (eds.) 38th European Conference on Object-Oriented Programming (ECOOP 2024). Leibniz International Proceedings in Informatics (LIPIcs), vol. 313, pp. 26:1–26:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2024). https://doi.org/10.4230/LIPIcs.ECOOP.2024.26
40. Maksimović, P., Ayoun, S.É., Santos, J.F., Gardner, P.: Gillian, Part II: Real-world verification for JavaScript and C. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification. pp. 827–850. Springer International Publishing, Cham (2021)
41. Maksimović, P., Cronjäger, C., Lööw, A., Sutherland, J., Gardner, P.: Exact separation logic: Towards bridging the gap between verification and bug-finding. In: Ali, K., Salvaneschi, G. (eds.) 37th European Conference on Object-Oriented Programming (ECOOP 2023). Leibniz International Proceedings in Informatics (LIPIcs),

vol. 263, pp. 19:1–19:27. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2023). https://doi.org/10.4230/LIPIcs.ECOOP.2023.19

42. Manzano, F.A.: Pysymemu, https://github.com/feliam/pysymemu, accessed: January 22, 2026
43. Marques, F., Fragoso Santos, J., Santos, N., Adão, P.: Concolic execution for WebAssembly. In: Ali, K., Vitek, J. (eds.) 36th European Conference on Object-Oriented Programming (ECOOP 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 222, pp. 11:1–11:29. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022). https://doi.org/10.4230/LIPIcs.ECOOP.2022.11
44. Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T., Dinaburg, A.: Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 1186–1189 (2019). https://doi.org/10.1109/ASE.2019.00133
45. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
46. Nguyen, H.H., Kuncak, V., Chin, W.N.: Runtime checking for separation logic. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 203–217. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
47. Ntzik, G., Gardner, P.: Reasoning about the POSIX file system: local update and global pathnames. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. p. 201–220. OOPSLA 2015, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2814270.2814306
48. Ntzik, G., da Rocha Pinto, P., Sutherland, J., Gardner, P.: A concurrent specification of POSIX file systems. In: Millstein, T. (ed.) 32nd European Conference on Object-Oriented Programming (ECOOP 2018). Leibniz International Proceedings in Informatics (LIPIcs), vol. 109, pp. 4:1–4:28. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2018). https://doi.org/10.4230/LIPIcs.ECOOP.2018.4
49. O'Hearn, P.W.: Incorrectness logic. Proc. ACM Program. Lang. **4**(POPL) (Dec 2019). https://doi.org/10.1145/3371078
50. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Proceedings of the 15th International Workshop on Computer Science Logic. p. 1–19. CSL '01, Springer-Verlag, Berlin, Heidelberg (2001)
51. Polikarpova, N., Sergey, I.: Structuring the synthesis of heap-manipulating programs. Proc. ACM Program. Lang. **3**(POPL) (Jan 2019). https://doi.org/10.1145/3290385
52. Qiu, R., Yang, G., Pasareanu, C.S., Khurshid, S.: Compositional symbolic execution with memoized replay. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. vol. 1, pp. 632–642 (2015). https://doi.org/10.1109/ICSE.2015.79
53. Raad, A., Berdine, J., Dang, H.H., Dreyer, D., O'Hearn, P., Villard, J.: Local reasoning about the presence of bugs: Incorrectness separation logic. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification. pp. 225–252. Springer International Publishing, Cham (2020), https://doi.org/10.1007/978-3-030-53291-8_14

54. Ramos, F., Sabino, N., Adão, P., Naumann, D.A., Fragoso Santos, J.: Toward tool-independent summaries for symbolic execution. In: Ali, K., Salvaneschi, G. (eds.) 37th European Conference on Object-Oriented Programming (ECOOP 2023). Leibniz International Proceedings in Informatics (LIPIcs), vol. 263, pp. 24:1–24:29. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2023). https://doi.org/10.4230/LIPIcs.ECOOP.2023.24
55. Reisner, E., Song, C., Ma, K.K., Foster, J.S., Porter, A.: Using symbolic evaluation to understand behavior in configurable software systems. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1. p. 445–454. ICSE '10, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1806799.1806864
56. Reynolds, J.: Separation logic: a logic for shared mutable data structures. In: Proceedings 17th Annual IEEE Symposium on Logic in Computer Science. pp. 55–74 (2002). https://doi.org/10.1109/LICS.2002.1029817
57. Sanfilippo, S.: Simple dynamic strings (2015), https://github.com/antirez/sds, accessed: January 22, 2026
58. Seidel, E.L., Vazou, N., Jhala, R.: Type targeted testing. In: Vitek, J. (ed.) Programming Languages and Systems. pp. 812–836. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
59. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: SOK: (State of) The art of war: Offensive techniques in binary analysis. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 138–157 (2016). https://doi.org/10.1109/SP.2016.17
60. Watanabe, Y., Gopinathan, K., Pîrlea, G., Polikarpova, N., Sergey, I.: Certifying the synthesis of heap-manipulating programs. Proc. ACM Program. Lang. **5**(ICFP) (Aug 2021). https://doi.org/10.1145/3473589
61. Wiedenhöft, R.: C hashmap (2014), https://gist.github.com/Richard-W/9568649, accessed: January 22, 2026

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.