Imperial College of Science, Technology and Medicine

Department of Computing

Enhancing Dynamic Symbolic Execution via Loop Summarisation, Segmented Memory and Pending Constraints

Timotej Kapus

supervised by

Dr. Cristian Cadar

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy in Computing of Imperial College London

Abstract

Software has become ubiquitous and its impact is still increasing. The more software is created, the more bugs get introduced into it. With software's increasing omnipresence, these bugs have a high probability of negative impact on everyday life. There are many efforts aimed at improving software correctness, among which symbolic execution, a program analysis technique that aims to systematically explore all program paths. In this thesis we present three techniques for enhancing symbolic execution.

We first present a counterexample-guided inductive synthesis approach to summarise a class of loops, called memoryless loops using standard library functions. Our approach can summarize two thirds of memoryless loops we gathered on a set of open-source programs. These loop summaries can be used to: 1) enhance symbolic execution, 2) optimise native code and 3) refactor code.

We then propose a technique that avoids expensive forking by using a segmented memory model. In this model, we split memory into segments using pointer alias analysis, so that each symbolic pointer refers to objects in a single segment. This results in a memory model where forking due to symbolic pointer dereferences is reduced. We evaluate our segmented memory model on benchmarks such as SQLite, m4 and make and observe significant decreases in execution time and memory usage.

Finally, we present pending constraints, which can enhance scalability of symbolic execution by aggressively prioritising execution paths that are already known to be feasible either via cached solver solutions or seeds. The execution of other paths is deferred until no paths are known to be feasible without using the constraint solver. We evaluate our technique on nine applications, including SQLite3, make and tcpdump, and show it can achieve higher coverage for both seeded and non-seeded exploration.

Statement of Originality

I declare that all work shown in this thesis is my own. Information derived from the work of others has been appropriately acknowledged.

Copyright Declaration

Acknowledgements

I would like to thank my supervisor, Dr. Cristian Cadar. Without him this work would not have been even attempted nor completed.

I would also like to thank Frank Busse and Martin Nowack for the numerous insightful discussions we have had during my PhD.

Finally I would like to thank Max Falkenberg McGillivray and Eva Menari for supporting me during this time.

Contents

List of Figures

List of Tables

List of Publications

This thesis is based on the following papers:

- Timotej Kapus, Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Cristian Cadar. Computing summaries of string loops in C for better testing and refactoring. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'19)*, June 2019
- Timotej Kapus and Cristian Cadar. A segmented memory model for symbolic execution. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*, August 2019
- Timotej Kapus, Frank Busse, and Cristian Cadar. Pending constraints in symbolic execution for better exploration and seeding. In *Proc. of the 35th IEEE International Conference on Automated Software Engineering (ASE'20)*, September 2020

Papers produced as part of the PhD, but not included in the thesis:

- Timotej Kapus, Martin Nowack, and Cristian Cadar. Constraints in dynamic symbolic execution: Bitvectors or integers? In *Proc. of the 13th International Conference on Tests and Proofs (TAP'19)*, October 2019
- David Trabish, Timotej Kapus, Noam Rinetzky, and Cristian Cadar. Past-sensitive pointer analysis for symbolic execution. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*, November 2020

Chapter 1

Introduction

Software development is hard. It also seems necessary and increasingly pervasive. The difficulty of software development means it is expensive to write correct or good software, and therefore few software developers can afford to do so. The pervasiveness of software means mistakes or bugs can cause loss of life, equipment or widespread disruption. There are many examples of software bugs resulting in deaths such as Therac-25 [67], loss of expensive hardware as with Ariane-5 in 1996 [74] or Baresheet in 2019 [82]. Software bugs can also cause sensitive data leakage such as the notorious Heartbleed bug [53], whose full impact is unknown, but the Canadian tax agency is known to have leaked personal information due to Heartbleed [36]. Even benign cases such as the recent Garmin hack [56] or the unicode handling bug in iOS [109] caused a denial of service to many consumers.

Making writing good software easier is a multipronged industry and research effort. There are many practices and tools that improve the quality of software. For example, writing test suites and running them via continuous integration tries to ensure future changes do not introduce unexpected new behaviour. Refactoring engines ensure bugs are not introduced in semantic preserving modifications because they automate them away and linters encourage safer use of a programming language.

Compilers are perhaps the most widely deployed tool in software engineering practice. While their main stated purpose is to generate correct and fast machine code, they are

also an important tool in improving software quality. For example compiler warnings are usually a good indication of a possible bug hiding in the background. However while compiler warnings are common and widely available, they are not sophisticated enough to catch more resilient programming faults.

More complex techniques such as verification and formal methods in general are able to guarantee software correctness. But wide adoption of formal methods is held back both by the lack of skills as well as the laborious process required. Thus they are still delegated to highly specialised areas such as aerospace.

Fuzzers are a recent example of tools that have research roots [81] and have gained large adoption through initiatives like OSS-Fuzz.[1] One of the reasons for wide adoption of fuzzers is that they are easy to use and widely applicable. That means it does not require a specialist to run and use fuzzers. In addition, fuzzers are likely to work on the programs you would want to test. In other words the ease of use of fuzzers is similar to ease of use of compilers. The aim of this thesis is to push symbolic execution further down the road of wider adoption that fuzzers have already taken.

1.1 Scope

The body of software engineering is large, with many different programs being written for many different purposes, using a variety of methods. That means there is a huge variety of problems or bugs that can arise from them. Unfortunately, there is no silver bullet that would eliminate them all in one stroke. This section therefore presents the kind of programs and corresponding bugs this thesis will consider.

We will only consider programs written in C-like languages, like C or C++ and to some extent assembly code. However, assembly or binary analysis presents additional challenges which we will not consider, such as control flow graph reconstruction.

C-like languages present many challenges not present in higher-level languages such as Java or Python. C does not have built-in memory safety and allows for arbitrary pointer

[1] `https://google.github.io/oss-fuzz/`

arithmetic. That means it is possible to address memory that is "not there" or it is not the portion of memory the programmer meant to address. This can occur for several reasons: The address might point a couple of bytes beyond the end of the allocated buffer, which is called a buffer overflow. At that address there might be nothing or another object, which the programmer did not intend to reference. This is the root of the previously mentioned Heartbleed bug [53].

Dangling pointers are another manifestation of the lack of memory safety. In this case, the address has pointed to the right object at some point in the execution, but that object has since been freed. This again means the address points to either nothing or a new unrelated object that has been placed in the freed space in the meantime.

Memory leaks are another challenge of C-like languages. In C, it is up to the programmer to explicitly free all the memory allocated. Failure to do so can lead to memory leaks, where all references to a portion of memory are forgotten and thus cannot be accessed any more, but they were not freed. This is a wasted resource and can cripple the program if it happens frequently.

There are also challenges in C unrelated to memory. Example of those are: division-by-zero errors, signed integers overflows, over-shifts and arbitrary control-flow. C allows for diverting the control-flow outside the normal branching structures via the `goto` statements, which can make programs harder to understand and lead to bugs such as Apple's goto fail bug.[2]

The final relevant challenge in the C family of languages are strings. Unlike higher level languages, strings in C are just chunks of memory that end with a zero byte, that is sometimes called the null character ("`\0`") in this context. That means strings are liable to all the memory bugs described above and thus a common source of errors.

While it could be argued that the problems from these languages are better solved by changing the programming language, C and C++ remain hugely popular. There are many essential projects written in C or C++ such as the Linux kernel and most other operating systems, even the very new Fuschia[3]. Web infrastructure such as Apache Web server, Nginx,

[2] `https://www.imperialviolet.org/2014/02/22/applebug.html`
[3] `fuchsia.dev`

```
1 const char *key = "asdfghjklzxcvbnmqwertyuiop";
2
3 // assumes input is all lower case
4 char* caesar_cipher(char* input) {
5     int len = 0;
6     char* output = calloc(N, sizeof(char));
7     while(input[len] != '\0') len++;
8
9     if(len > N) {
10      printf("Input␣text␣too␣long.␣Aborting");
11      exit(0);
12    }
13
14    for(int i = 0; i < len; i++) {
15        output[i] = key[input[i] - 'a'];
16    }
17
18    assert(output[N - 1] == '\0' && "Output␣string␣is␣not␣a␣C␣string");
19    return output;
20 }
```

Figure 1.1: A sample program written in C.

databases like PostgreSQL and MySQL are also written in either C or C++. Even tools that enable higher level languages are all written in C or C++, like JVM implementations, Python interpreters and compiler infrastructure such as GCC and LLVM.

Therefore, we believe there is still a huge legacy code base, momentum and continuing need to be close to the hardware, that make research in improving the software quality of C-like languages relevant and directly applicable.

To make some of the bugs described above more concrete, consider a program written in C in Figure 1.1. It shows an implementation of a function that encodes an input string. It is using a simple Caesar cipher method, where each letter is mapped to another one using a simple map. For example, character b would get mapped to character s.

The `caesar_cipher` function first allocates a space of N bytes that will contain the output string on line 6. It then computes the length of the string on line 7 and checks that the output string has enough space to contain the encoded message on lines 9–12. Finally, it performs the encoding on lines 14–16, by iterating over the input string and then performing a lookup into the key on line 15.

Notice that there is an off-by-one bug present on line 9. If the length of the input string is exactly N, the output string is not large enough for the terminating null-character. In this

case the programmer was careful enough to check the output string is null terminated with the assertion on line 18. That ensures that this case would be caught early. If uncaught, an unterminated string could cause memory corruption later in program execution. At best, the program would abort, due to accessing unallocated memory (one byte past the end of the output string). In the worst case it could be exploited to leak or even change program memory, which could lead to serious vulnerabilities.

A buffer overflow could also occur on line 15 if someone violeted the precondition on line 3. For example, if the character "}" would be present in the `input` string, `key[input[i] - a]` would access memory two bytes beyond the end of the `key` memory object. In this case, there is nothing guarding against it and would go unnoticed, potentially leaking some unintended information.

Finally, a possible memory leak might occur due to memory allocated on line 6, which is then passed to the caller of the `caesar_cipher` function. This caller then needs to free the memory. However, there is little indication that the `output` string needs to be freed so it is easy to forget it.

1.2 Symbolic Execution

Now that we have defined the kind of programs we are interested in and the type of bugs these can have, we present symbolic execution. Symbolic execution is a dynamic program analysis technique that has established itself as an effective approach for many software engineering problems such as test case generation [16, 47], bug finding [18, 48], equivalence checking [25, 26], vulnerability analysis [21, 123] and debugging [83, 58]. While symbolic execution is applicable to a broader scope, with many higher-level languages having tools performing symbolic execution [4, 95, 113, 22], the techniques presented in this thesis are mostly targeted towards symbolic execution of C-like languages.

In brief, symbolically executing a program means running it on a *symbolic* input, i.e. an input which is initially allowed to have any value. As the program runs, the operations on symbolic values are evaluated as symbolic expressions, while non-symbolic (concrete) values

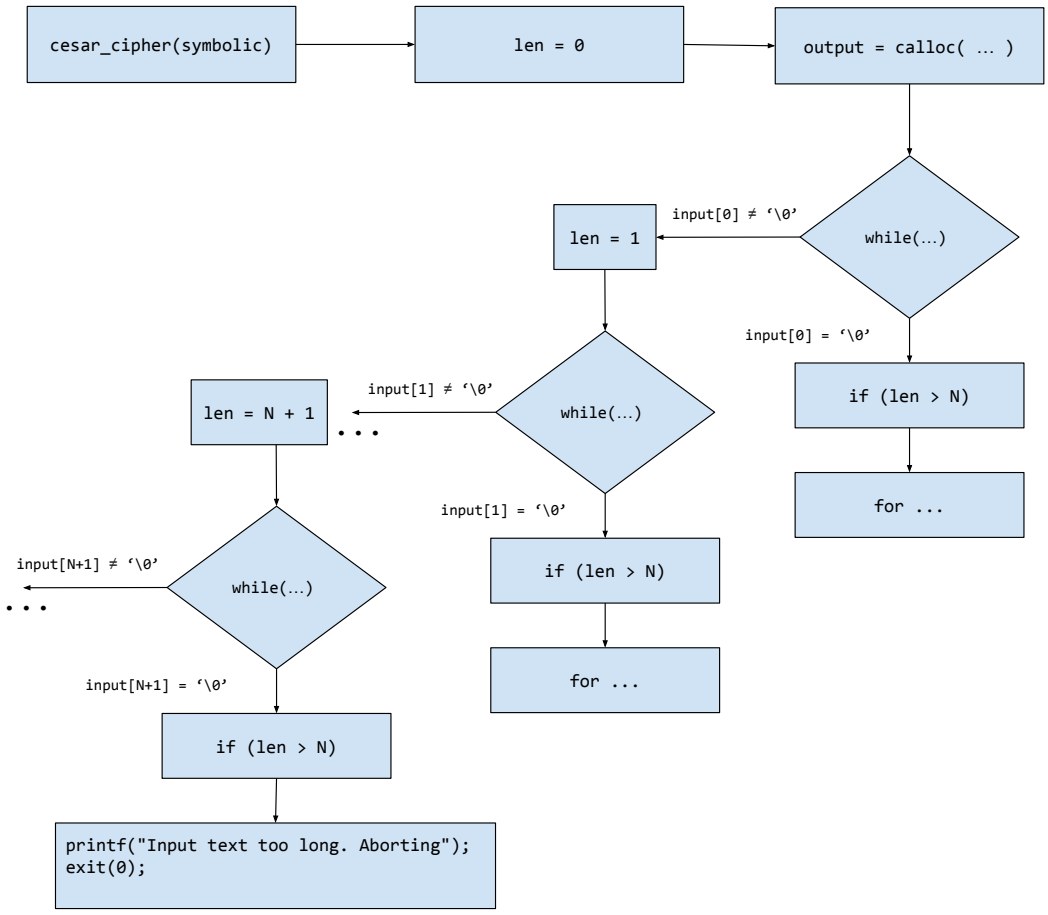


Figure 1.2: Symbolic exploration tree of the `caesar_cipher` function.

are computed normally. When the program branches on one of these symbolic expressions, symbolic execution explores both branches if they are feasible, adding corresponding symbolic constraints to the respective path conditions. Upon path termination, symbolic execution can ask the solver for a solution to the current path conditions, thus obtaining a concrete test input that will follow the same path when executed concretely.

To make things more concrete, consider the example from before in Figure 1.1. Figure 1.2 shows the symbolic exploration of `caesar_cipher` function up to line 14. The symbolic execution starts on the top left of the image with a symbolic string as the argument of the `caesar_cipher`. In this case a symbolic string means that each character of an arbitrarily

sized[4] string can initially have any value. The execution then proceeds normally, with rectangles indicating concrete (or normal) execution. Zero is first assigned to variable `len` and a N-byte memory region is allocated.

We then hit our first branch point, which is the exit condition of the `while` loop. At this point symbolic execution forks the execution and tries to follow both paths if they are feasible. A path is feasible if there is an assignment to symbolic values, that makes all the *path constraints* evaluate to `true`. The check is performed by an SMT solver, which we briefly introduce in Section 1.3. In Figure 1.2, path constraints are the union of all the logical formulae next to the arrows, while tracing them back to the root of the graph. In this case, both paths are feasible so the execution is forked and the branch condition and its negation added to the path constraints of respective paths. At this point a *search heuristic* (sometimes called a searcher and discussed in Section 1.2.2) decides which path to explore next. Let's say the searcher picks the branch (going down) that exits the loop. On this path the `if (len > N)` branch is encountered, which is not a symbolic branch, because it does not depend on any symbolic value, therefore the condition is evaluated to false as it normally would, because `len` has value `0` on this path. This path then continues to the `for` loop at which point we consider the path completed, for the purpose of this simple introduction.

Symbolic execution then picks the other path where the body of the loop is entered and increments `len`. The execution then hits the loop exit branch again, where the process largely repeats itself. The notable difference is that `len` is now `1`, so the second byte of the string is now part of the branch condition. The path condition also becomes larger:

$$(input[0] \neq 0)$$
$$\wedge\ (input[1] = 0)$$

It includes the path condition of the previous branch too. The process then repeats itself for the next N iterations in the loop. In the $N+1$ iteration and all iterations afterwards the condition of `if (len > N)` evaluates to `true` and the *then* branch is followed. The program thus prints a message and terminates normally and the path is completed.

[4]In further sections we adopt KLEE's assumption that symbolic strings have a fixed maximum length

1.2.1 Challenges

While symbolic execution is already a useful and powerful technique, it faces two major classes of challenges: constraint solving and path explosion. Most symbolic execution research can be viewed as tackling either or both of these problems.

Even for medium (Coreutils [40]) to large (but not huge) sized real programs (*SQLite3* [106]) path constraints can get very large, such that they can be time consuming to simply traverse in memory, let alone solve. The number of branches on a path in real programs is huge, so there can be many symbolic branches on a path, each symbolic branch adding a new conjunct to the path condition. In addition, each branch might depend on a large area of memory (e.g. big arrays). This memory needs to be represented in the constraints and it can hugely inflate the size of constraints. Large constraints combined with unfavourable complexity of feasibility checking means that even excellent contemporary SMT solvers quickly take seconds (or more) to solve a constraint.

This might not sound too bad, until we consider the second problem: path explosion. Path explosion means there are a lot of constraints to be solved. Path explosion arises due to the exponential relationship between branches and paths. Each branch for which both directions are feasible spawns two new paths that both split again at the next feasible branch and so on. This quickly leads to an explosion of paths. While it is very hard to compute or estimate the number of paths a program can have, we can put an upper limit on the number of paths. The limit is simply the number of different inputs a program can have. A program cannot follow more than one path per concrete input. In the context of symbolic execution, the number of different inputs a program can have is $2^{number\ of\ symbolic\ bits}$.

As an example, consider *SQLite3* and a 20-byte symbolic input, in which we can barely put a small SQL statement. There are $2^{8*20} = 1.46 * 10^{48}$ possible inputs and thus the upper limit on the number of paths. If only a second of constraint solving time is spent on each one, the constraint solving time for this symbolic execution is 31 orders of magnitude larger then the age of the universe! This is of course a vast exaggeration as an average path in a program usually covers a large number of inputs. However, it is still useful to illustrate just how large of a problem path explosion can be.

1.2.2 Search Strategies

An important way of tackling the path explosion outlined above is search strategies. Because path explosion prevents us from exploring all paths, we can instead try to focus on a small subset of "interesting" paths. An interesting path might be one that covers previously uncovered code, has a potential for a bug [78] or a code region of interest such as a patch [79].

In our outline of symbolic execution we glossed over search strategies by simply stating symbolic execution picks the next state to explore. A search strategy or a search heuristic is an algorithm that chooses the next state for symbolic execution to explore. It is invoked often, usually after every instruction that is symbolically executed.

Another way of seeing search strategies is as the order in which a symbolic exploration tree (as shown in Figure 1.2) is traversed. We can use the standard breadth or depth first traversal strategies (*DFS* and *BFS*). These are good baseline search heuristics that show two contrasting ways of traversing the exploration tree. First, represented by DFS, is to go deep — this has the advantage of low memory usage and favourable use of the constraint solver, because the states it picks have similar constraints due to a common prefix. However, DFS is likely to explore the same code over and over again thus achieving low coverage. On the other hand, BFS represents search strategies that go wide, which is good for achieving high coverage, but is very prone to path explosion, therefore usually infeasible. Most search strategies fall somewhere in-between the two.

Random path search strategy [16] is popular because it tries to go wide, but somewhat limits the path explosion. It does so by performing a random walk of the symbolic exploration tree each time it is picking a state. It starts from the root and flips a fair coin at each branch until it ends up at a leaf state, which is the state it picks next.

Search heuristics can also pick states randomly. *Random state search* picks each state with equal probability, which is somewhere between DFS and BFS, but tends to be more similar to BFS and suffers from the same problem. The probability of picking the state can be biased by some metric; for example KLEE [16] can bias them by distance to uncovered code. Similarly, the probability of picking a state can be biased by its depth, that is, the number of symbolic branches through which the state has passed already.

The search heuristics above can also be combined with so-called meta-search heuristics, for example by interleaving search hearustics. That is the default mode of KLEE, which uses random path and coverage-biased search alternatingly.

There are also more complex search strategies such as ZESTI's search strategy [78], which aims to explore paths around sensitive instructions, where it defines sensitive instructions as instructions that are likely to contain a bug, such as memory accesses. KATCH [79] uses a search strategy that guides symbolic execution towards patched code. MoKLEE [15] prioritizes states that have not been explored before.

In Chapter 4 we propose a pending meta search strategy that picks states based on them already having an available solver solution. This both makes symbolic execution use constraint solving more efficiently and can modify a broad search strategy to also explore deeper parts of the program.

1.2.3 Symbolic Executors

There are many implementations of the symbolic execution algorithm. Outside our scope of C programs there is JPF-SE [4] for Java, PEX [113] for .NET and KALUZA [95] for JavaScript, to name a few. For programs in our scope we could look at the large number of binary symbolic executors such as Angr [102], FuzzBALL [80] or MAYHEM [21]. But these operate at a binary level, which provides a whole new set of very hard challenges that are unique to binaries due to useful information — such as control flow — being stripped out.

There are also concolic executors such as DART [47], SAGE [48], CREST [28] and CUTE [99] which implement a slight variation of symbolic execution called concolic execution or white-box fuzzing. In the concolic variant of symbolic execution execution, a path (an input) is executed from start to finish, while still gathering path constraints. A path constraint is then negated and solved to get a new input, which is executed start to finish again and the process repeats. [20] This approach is easier to implement and can be more scalable. However it is harder to use outside of test generation and bug finding. It is also less efficient, as the same prefix of a path gets re-executed many times and only explores a single path at a time.

Finally KLEE [16] follows the EGT [17] line of symbolic executors. It is an actively maintained and open source symbolic executor that operates on LLVM bitcode, an intermediate representation that still captures most of the high level semantics and thus does not suffer from the same problems as binary analysis and can therefore scale to larger programs. The popularity of KLEE can also be seen from its many extensions [68, 97, 23, 92, 91, 73, 86, 15, 115, 66, 64, 94, 121].[5] KLEE is what we will be using to implement the approaches presented in this work and thus evaluate their effectiveness.

1.3 SMT Solvers

SMT solvers are programs that determine the feasibility of a logical formula. A logical formula is satisfiable if there exists an assignment to its variables that renders it true. If no such assignment exists, the formula is unsatisfiable. In general, this is a very hard problem, however advances in SMT solvers in the past two decades have made them fast enough on typical formulas generated by program analyzers. STP [37] and Z3 [32] are the two solvers that we will use. The inner workings of SMT solvers are well beyond the scope of this thesis and we will use them in a black-box manner. What is important for the context of this discourse is how a program or a path condition is presented to the solver.

SMT solvers work on a certain set of *theories*, which can be thought of as a "language" to communicate with the solver. There are four theories relevant to symbolic execution used in this thesis: theory of bitvectors, theory of arrays, theory of strings and theory of integers.

1.3.1 Theory of Bitvectors and Arrays

The theory of bitvectors lets us encode machine integers, that is integers that are represented by a finite number of bits and therefore have a bounded range. As a taster, consider the logical formula $(x > 0 \wedge x + 1 < 0)$ in the theory of bitvectors in SMT-LIB2 format [9]:

```
1 (declare-const x (_ BitVec 8))
2 (assert (bvsgt x #x00))
```

[5] A more complete list can be found at `http://klee.github.io/publications` currently listing 167 papers using or extending KLEE

```
3 (assert (bvslt (bvadd x #x01) #x00))
```

Variable x is first declared as an 8-bit vector on line 1. This is how a symbolic byte in symbolic execution is presented to a constraint solver. Each conjunct is then encoded as a separate `assert` statement, $x > 0$ on line 2 and $x + 1 < 0$ on line 3. In plain terms, what this formula is asking is: can incrementing a positive integer result in a negative integer? This is not satisfiable if x is a mathematical integer, but it is satisfiable if x is an 8 bit bitvector due to overflow. Namely, setting x to 127 ($x = 0x7f$), will lead to that overflow. This is exactly how addition works on modern computers, so the theory of bitvectors lets us easily encode the exact semantics of arithmetic instructions on modern architectures.

In addition to modeling arithmetic instructions as described above, we also need to model memory access instructions such as `load` and `store` in order to get an instruction set that can run real programs. In other words we need to model computer memory.

One way of modeling memory is to keep using the theory of bitvectors. It supports slicing, that is extracting smaller bitvectors from a larger one. Therefore the whole memory could be modeled as one giant bitvector and `load`/`store` instructions just become equality constraints on slices of the giant bitvector representing the whole memory. However, the theory of bitvectors requires slices to have concrete offsets. That means the theory of bitvectors cannot be used to model `load`/`store` instructions with a symbolic address. This can be resolved by enumerating all possible addresses the symbolic address can point to, concretizing for each case and creating a disjunction over them. This can quickly get expensive, but it is a technique used by some binary symbolic executors [102, 80, 11].

A more concise way of modeling computer memory is to use theory of arrays and bitvectors. The theory of arrays is simple and has two operations: $read(array, index)$ and $write(array, index, value)$, with two read-over-write axioms. First $\forall a, i, j, v.\ i = j \implies read(write(a, i, v), j) = v$, which simply says that reading a value from the same location the value was written to yields that value. And second $\forall a, i, j, v.\ i \neq j \implies read(write(a, i, v), j) = read(a, j)$, which simply says to ignore writes at the locations not being read. The *value* in the theory of arrays can be anything, but for the purpose of modeling computer memory it is most convenient if they are 8-bit bitvectors, to represent

an array of bytes. Note that the theory of arrays does not require *index* to be concrete. Therefore it can be used to represent `load/store` instructions of symbolic addresses more concisely without large disjunctions.

1.3.1.1 Symbolic Execution and Memory

As an example, consider again our program from before in Figure 1.1. We previously explained the basics of symbolic execution in Section 1.2. We stopped at line 15 because it involves symbolic memory. Using theory of arrays we can now present how the `key[input[i] - 'a']` might be represented in the theory of arrays.

First we have two arrays, `input` and `key`. `input` is a symbolic array in our example, so `input[i]`, can be represented simply as $read(input, i)$. In the first iteration of the loop where `i` is 0, that would be $read(input, 0)$. `key` is a concrete array with some values. To encode this in the theory of arrays, we need to write those values explicitly into the array: $write(write(...write(write(key, 0,' a'), 1,' s')), 25,' p')$.

The `key[input[i] - 'a']` expression can now be encoded as

$$
\begin{aligned}
&read(\\
&\quad write(write(...write(write(key, 0,' a'), 1,' s')), 25,' p'),\\
&\quad read(input, i) -' a')
\end{aligned}
$$

Note that this is only one way of encoding `key[input[i] - 'a']` into theory of arrays and bitvectors albeit perhaps the most straightforward to present and the one used by KLEE [16]. In this exposition we combined `input` and `key` to mean: program pointer to a chunk of memory, name of the chunk of memory and a logical array representing that chunk of memory to an SMT solver. The distinction between these three concepts is irrelevant for the example from Figure 1.1, because our pointer (`key + (input[i] - 'a')` can only point to the `key` object allocated at line 1 in Figure 1.1. We will refer to the process of identifying which memory objects a pointer can point to as *memory object resolution* or

simply resolution.

In Chapter 3 we explore cases where a pointer might resolve to multiple memory objects and present the challenges this poses for a symbolic executor. In this context we then present other methods of encoding memory and illustrate their trade-offs. We then present a segmented memory model that uses pointer alias analysis (briefly introduced in Section 1.4) to pre-compute the possible memory objects a pointer can point to. Our segmented memory model allocates these memory objects together, which enables better symbolic execution of programs heavily utilising pointers that might resolve to multiple memory objects.

1.3.2 Theory of Strings

Strings are perhaps the most widely-used datatype, at the core of the interaction between humans and computers, via command-line utilities, text editors, web forms, and many more. Therefore it can be reasonable to tailor program analysis to strings. The theory of strings allows for encoding of strings in a more natural way instead of relying on the theory of arrays. It provides a different set of primitives that talk about sequences of characters. For brevity we will present a single primitive: string concatenation, which we will denote as `++`. String concatenation simply appends the right string at the end of the left one, for example `"hello world" == "hello" ++ " " ++ "world"`.

Suppose we would want to encode a SQL statement: `SELECT * FROM {table-name};`, where `table-name` is some symbolic variable. In the theory of strings this can simply be expressed as: `"SELECT * FROM " ++ tv ++ ";"` where `tv` is some string variable. Let us go back to the theory of arrays for a moment to consider how this SQL statement could be encoded there. First we need 14 *write* expressions to write the 14 characters of `SELECT * FROM` on indices 0-13 of the array. Then we can write the first character of `tv` as the 14^{th} character, second character of `tv` as the 15^{th} and so on. But when do we stop to write the final semi-colon? That depends on the length of `tv` and theory of arrays has no way of expressing variable length nicely. It would have to rely on some form of disjunction and special case each length of `tv`. That is significantly less scalable.

The rise of string solvers [69, 64, 127, 117, 126, 10] has enabled effective program analysis

techniques for string-intensive code leveraging the theory of strings, e.g. Kuduzu [95] for JavaScript or Haderach [100] and JST [39] for Java. However, these techniques operate on domains where strings are well-defined objects (i.e. the *String* class in Java).

As mentioned before, strings in our scope of C-like languages are just arbitrary portions of memory terminated by a null character. That means interacting with strings does not have to go through a well-defined API as in other programming languages. While there are string functions in the C standard library (e.g. *strlen*, *strchr*, *strspn*, etc. defined in *string.h*), programmers can write their own equivalent loops for the functionality provided by the standard library. In Chapter 2 we show this practice is not uncommon.

An example of this can be observed in our initial program from Figure 1.1, where the loop on line 7 could be summarised into the standard *strlen* function. Note how that would compress the entire exploration tree (even the `...` part in Figure 1.2) into a simple `len = strlen(input)` expression. This can significantly reduce the number of paths that need to be explored, thus speeding up symbolic execution.

There is a flip side to the theory of strings which can be explained by looking at line 15 of Figure 1.1, where we have lookups into a string based on an index. This is not a problem in itself, because the theory of strings provides the `at` operator, which returns the character at an index. There are two problems however. First, this degrades the theory of strings into the theory of arrays that is just harder to solve. Second, the theory of strings requires indices to be mathematical integers instead of bitvectors. That means we can either drop the theory of bitvectors and model them as mathematical integers, which brings its own problems [62], or do conversion between bitvectors and mathematical integers, which is very expensive and only very recently somewhat addressed [2].

That is why having string manipulation through string functions almost exclusively is vital to the success of applying the theory of strings to symbolic execution. In languages such as Java or JavaScript this is almost free due to the language design and the theory of strings is successful in that domain, while we are unaware of any similar successes for C-like languages. In Chapter 2 we make important steps towards addressing this problem. We show a technique that can automatically summarise string loops into well known C standard

library functions. This has the potential for large gains for symbolic execution of string heavy programs, as well as applications in refactoring and compiler optimisation.

1.4 Pointer Alias Analysis

Pointer alias analysis is a type of static program analysis that computes all possible memory locations a pointer can point to. To put it another way, it tells us what pointers can point to the same memory region, that is, which pointers may alias each other. While details of pointer alias analysis are beyond the scope of this work, we use it in Chapter 3 to enhance a memory model of symbolic execution. We use SVF [111], an off-the-shelf pointer alias analysis for LLVM that implements a scalable variation of Andersen-style [5] alias analysis that is field-sensitive, flow-insensitive and context-insensitive.

In a nutshell, Andersen-style analysis works by translating a program into a set of constraints. For example, consider the following code:

```
1 char* p = malloc(10);
2 char* q = p + 1;
3 q = malloc(15)
```

Line 1 would be translated into a constraint $pts(p) \supseteq \{l_1\}$, which says the set of locations that `p` can point to (denoted as $pts(p)$) includes the memory region denoted by l_1. We use l_1 to denote all possible memory regions that can be allocated on line 1. Sometimes l_1 would be called an abstract memory object. Note that this is a static analysis, which means it is performed without running the program, so it is very hard to know how many times line 1 will be executed. Therefore we abstract the problem away by treating all the memory at line 1 as one, denoted by l_1.

Similarly, line 2 would be translated into constraint $pts(q) \supseteq pts(p)$, which simply says `q` points to whatever `p` points to. Note that the `+ 1` arithmetic operation is abstracted away as it has no impact on pointer aliasing information due to the C language standard prohibiting pointer arithmetic crossing memory object boundaries. Finally, line 3 would be encoded into $pts(q) \supseteq \{l_3\}$, similarly to line 1.

These constraints can now be solved in a flow-insensitive way, that is ignoring the order of the program statements. A flow-insensitive solution at the end of propagation would be $pts(p) = \{l_1\}$ and $pts(q) = \{l_1, l_3\}$. This is because flow insensitivity ignores the fact `q` is overwritten on line 3. A flow sensitive solution would yield $pts(p) = \{l_1\}$ and $pts(q) = \{l_3\}$, which is more precise (points to sets are smaller), but harder to compute and therefore less scalable.

Andersen focuses [5] on flow-insensitive style of analysis, which they argue is the better trade-off between scalability and precision. The original Andersen analysis is also field-insensitive and context-insensitive. That is it considers fields of a struct as one and it does not consider functions being called in different contexts.

1.5 Overview

We have introduced symbolic execution, placed it in the wider context of software engineering and glanced at some of its challenges. In the remainder of this thesis we present three techniques that are steps towards addressing some of the shortcomings of symbolic execution.

In Chapter 2 we first aim to transform programs under test in a way that makes them friendlier to symbolic execution by summarising some of the loops over strings into standard library string functions. We gather a set of loops from open source programs and show our technique can summarise over two thirds of these loops in less than 5 minute of runtime per loop. We then show these loop summaries are applicable to improvements in symbolic execution, optimising compilers and refactoring.

We then look into the multiple resolution problem in Chapter 3. Multiple resolution occurs when pointers that depend on (symbolic) input can resolve to multiple memory objects. This can amplify the path explosion problem for benchmarks that make heavy use of datastructures such as hash tables. We use pointer alias analysis to partition the memory into segments, such that each pointer can only point to a single segment, thus alleviating the issue of multiple resolution.

While working with the benchmarks in Chapter 3, we observed poor performance of

KLEE unrelated to multiple resolution. Therefore, in Chapter 4, we introduce pending constraints that aim to increase the performance of symbolic execution on those benchmarks. Pending constraints aim to aggressively explore paths that are already known to be feasible, thus using the solver more efficiently and increasing coverage.

Finally we conclude and present some possible future work in Chapter 5.

Chapter 2

Loop Summarisation

In this chapter we focus on a particular set of loops common in real C programs that usually operate on strings. We call these loops *memoryless loops*, because they do not carry information from one iteration to another. To illustrate, consider the loop shown in Figure 2.1 (taken from *bash* v4.4). This loop only looks at the current pointer and skips the initial whitespace in the string *line*. The loop could have been replaced by a call to a C standard library function, by rewriting it into `line += strspn(line, "␣\t")`.[1]

In addition to enabling gathering constraints in theory of strings for symbolic execution as presented in Section 1.3.2, replacing loops such as those of Figure 2.1 with calls to standard string functions has two other advantages.

First, from a software development perspective, such code is often easier to understand, as the functionality of standard string functions is well-documented. Furthermore, such code is less error-prone, especially since loops involving pointer arithmetic are notoriously hard to get right. Our technique is thus useful for refactoring such loops into code that calls into the C standard library. As detailed in Section 2.2, we submitted several such refactorings to popular open-source codebases, with some of them now accepted by developers.

Second, translating custom loops to use string functions can also impact native execution, as such functions can be implemented more efficiently, e.g. to take advantage of architecture-

[1] We remind the reader that `strspn(char *s, char *charset)` computes the length of the prefix of `s` consisting only of characters in `charset`.

```
1 #define whitespace(c) (((c) == ' ') || ((c) == '\t'))
2 char* loopFunction(char* line) {
3     char *p;
4     for (p = line; p && *p && whitespace (*p); p++)
5       ;
6     return p;
7 }
```

Figure 2.1: String loop from the *bash v4.4* codebase, extracted into a function.

specific hardware features.

In Section 2.1 we first define a vocabulary that can express memoryless string loops similar to the one in Figure 2.1. We then present an algorithm for counterexample-guided synthesis within a single (standard) symbolic execution run, which we use to synthesise memoryless string loops in our vocabulary and show they are equivalent up to a small bound. Finally, we build a database of suitable loops in Section 2.2.1 based on popular open-source programs, which we use to comprehensively analyse the impact of time, program size and vocabulary on summarising loops in Section 2.2.2.

2.1 Technique

The goal of our technique is to translate[2] loops such as the one in Figure 2.1 into calls to standard string functions. Our technique uses a counterexample-guided inductive synthesis (CEGIS) algorithm [105] inspired by Sharma et al.'s work on synthesising adaptors between C functions with overlapping functionality but different interfaces [101]. This CEGIS approach is based on dynamic symbolic execution, which we introduced in Section 1.2. In this section, we first discuss the types of loops targeted by our approach (§2.1.1), then present the vocabulary of operations to which loops are translated (§2.1.2), and finally introduce our CEGIS algorithm (§2.1.3).

[2]In this chapter, we use the terms translate, summarise and synthesise interchangeably to refer to the process of translating the loop into an equivalent sequence of primitives and string operations.

2.1.1 Loops Targeted

Our approach targets relatively simple string loops which could be summarised by a sequence of standard string library calls (such as `strchr`) and primitive operations (such as incrementing a pointer). More specifically, our approach targets loops that take as input a pointer to a C string (so a `char *` pointer), return as output a pointer into that same string, and have no side effects (e.g., no modifications to the string contents are permitted). Such loops are frequently coded in real applications, and implement common tasks such as skipping a prefix or a suffix from a string or finding the occurrence of a character or a sequence of characters in a larger string. Such loops can be easily synthesised by a short sequence of calls to standard string functions and primitive operations (see §2.2 which shows that most loops in our benchmarks can be synthesised with sequences of at most 5 such operations). While our technique could be extended to other types of loops, we found our choice to provide a good performance/expressiveness trade-off.

More formally, we refer to the two types of loops that we can synthesise as *memoryless forward loops* and *memoryless backward loops*, as defined below.

Definition 1 (Memoryless Forward Loop)**.** Given a string of length len, and a pointer p into this buffer, a loop is called a forward memoryless loop with cursor p iff:

1. The only string location being read inside the loop body is $p_0 + i$, where p_0 is the initial value of p and i is the iteration index, with the first iteration being 0;
2. Values involved in comparisons can only be one of the following: 0, i, len for integer comparisons; p_0, $p_0 + i$, $p_0 + len$ for pointer comparisons; and $*p$ (i.e. $p_0[i]$) and constant characters for character comparisons;
3. The conceptual return value (i.e. what the loop computes) is $p_0 + c$, where c is the number of completed iterations.[3]

Definition 2 (Memoryless Backward Loop)**.** Such loops are defined similarly to forward memory loops, except that the only string location being read inside the loop body is $p_0 + (len - 1) - i$ and the return value is $p_0 + (len - 1) - c$.

[3]The number of times execution reached the end of the loop body and jumped back to the loop header [3].

Table 2.1: The vocabulary employed by our technique.

Gadget	Regexp	Effect
rawmemchr	M(.)	result = rawmemchr(result, $1)
strchr	C(.)	result = strchr(result, $1)
strrchr	R(.)	result = strrchr(result, $1)
strpbrk	B(.+)\0	result = strpbrk(result, $1)
strspn	P(.+)\0	result += strspn(result, $1)
strcspn	N(.+)\0	result += strcspn(result, $1)
is nullptr	Z	skipInstruction = z != NULL
is start	X	skipInstruction = result != s
increment	I	result++
set to end	E	result = s + strlen(s)
set to start	S	result = s
reverse	^V	reverses the string (see §2.1.2)
return	F	return result and terminate

2.1.2 Vocabulary

Given a string loop, our technique aims to translate it into an equivalent program consisting solely of primitive operations and standard string operations. For convenience, we represent the synthesised program as a sequence of characters, each character representing a primitive or string operation. To make things concrete, we show the elements (called *gadgets*) of our working vocabulary in Table 2.1.

To summarise the loop in Figure 2.1, we need to first represent the call to `strspn`. We choose the character `P` to be the opcode for calls to `strspn`. We do not have to represent the first argument in `strspn` as it is implicitly the string the loop is operating on. The second argument is a string containing all the characters `strspn` counts in its span. We represent this using the actual list of characters passed as the second argument, followed by the \0 terminator character. Thus, the call `strspn(line, "␣\t")` would be encoded by `P␣\t\0`.

Extended regular expressions provide a concise way of presenting the vocabulary. A string matching the regular expression `P(.+)\0` represents a call to `strspn`, where the second argument consists of the characters in the capture group `(.+)` of the regular expression. In Table 2.1, we refer to the first capture group as `$1`.

To formally define the meaning of a program in our language (e.g. of a sequence like `P␣\t\0`), we define an interpreter that operates on such sequences of instructions (characters)

Algorithm 1 Program interpreter for a vocabulary of three gadgets: *strspn*, *is nullptr* and *return*.

```
 1: function Interpreter(s, program)
 2:     result ← s
 3:     skipInstruction ← false
 4:     foreach instruction ∈ program do
 5:         if skipInstruction then
 6:             skipInstruction ← false
 7:             continue
 8:         end if
 9:         arguments ← InstructionArgs(instruction)
10:         switch OpCode(instruction) do
11:             case 'P' // strspn
12:                 result ← result+ strspn(result, arguments)
13:             case 'Z' // is nullptr
14:                 skipInstruction ← result != NULL
15:             case 'F' // return
16:                 return result
17:             default
18:                 return invalidPointer
19:         end switch
20:     end foreach
21:     return invalidPointer
22: end function
```

and returns an offset in the original string. The execution of the interpreter represents the synthesised program.

A partial interpreter that can handle three of our vocabulary gadgets (*strspn*, *is nullptr* and *return*) is shown in Algorithm 1. The interpreter has an immutable input pointer register (s, the original string pointer on which the loop operates), a return pointer register (*result*) and a skip instruction flag (*skipInstruction*).

The interpreter loops through all the instructions in the *program* (lines 4–20). If the skip instruction flag is set, the interpreter skips the next instruction and resets the flag (lines 5–8). Otherwise, it reads the next instruction and interprets each gadget type accordingly, by either updating the *result* register or the *skipInstruction* flag (lines 9–16). If the loop runs out of instructions or the opcode is unknown, we return an invalid pointer (lines 17–18). This ensures that malformed programs never have the same output as the original loop and are therefore not synthesised.

Consider running the interpreter on the program `P␣\t\0F`. The interpreter initialises

`result` to the string pointer `s` and `skipInstruction` to `false`. It then reads the next instruction, `P`, and thus updates `result` to `result + strspn(result, "␣\t")`. Finally, it reads the next instruction, `F`, and returns the `result`. Therefore, the synthesised program is:

```
result = s;
skipInstruction = false;
if (!skipInstruction)
  result = result + strspn(result, "␣\t");
else skipInstruction = false;
if (!skipInstruction)
  return result;
else skipInstruction = false;
```

which is equivalent to `return = s + strspn(result, "␣\t")`.

To also illustrate the use of the *is nullptr* gadget covered by Algorithm 1, consider the slightly enhanced program `ZFP␣\t\0F`. In this case, the interpreter will synthesise the following code for the added `ZF` prefix, which returns `NULL` if the string is `NULL`:

```
result = s;
if (!skipInstruction)
  skipInstruction = result != NULL;
else skipInstruction = false;
if (!skipInstruction)
   return result;
else skipInstruction = false;
```

In summary Table 2.1 shows the full set of vocabulary we devised to express the loops we wanted to target. The first column in the table gives the name of the gadget, while the second shows the extended regular expression that describes the gadget. The third column shows the effect this gadget has on the INTERPRETER.

Meta-characters. To more easily synthesise gadgets that take sets of characters as arguments, such as `strspn`, we introduce the notion of meta-characters. These are single symbols in our synthesised programs that expand into sets of characters. By reducing the program size, this helps us synthesise loops that contain calls to macros/functions such as

`isdigit` in a more scalable way. For example, instead of having to synthesise 10 characters (`"0123456789"`), we can synthesise just a single meta-character (which we chose to be `"\a"`). Similarly, we also use a whitespace meta-character which represents `"␣\t\n"`). These are the only two meta-characters that we found beneficial in our experiments, but more could be added if needed. At the same time, we note that meta-characters are not strictly necessary, i.e. the synthesis algorithm would work without them, but would take longer.

Reverse instruction. The *reverse* instruction can only occur as the first instruction of a synthesised program. Its purpose is to facilitate the summarisation of backward loops. For example, *reverse* and *strchr* should be equal to *strrchr*. However, not all functions have a reverse version, so the purpose of the *reverse* instruction is to enable similar functionality for other functions such as *strspn*. For instance, such function can be easily expressed in terms of the vocabulary offered by string constraint solvers.

The *reverse* instruction takes the string s and copies it into a new buffer in reverse order. It then sets *result* to this new buffer. It also enhances the behaviour of the *return* instruction: instead of simply returning *result*, which now points to a completely different buffer, it maps the offset in the new buffer back into the original buffer.

2.1.3 Counterexample-Guided Synthesis

Our synthesis approach is inspired by Sharma et al.'s work on synthesising adapters between different C library implementations [101]. Their vocabulary gadgets are able to translate the interface of one library into that of another. Examples of gadgets include swapping arguments or introducing a new constant argument to a function.

Their approach uses counterexample-guided inductive synthesis (CEGIS) based on symbolic execution to synthesise the adapters in that vocabulary. In brief, this means they first take a symbolic adapter, constrain it so that it correctly translates all the currently known examples, then concretize the adapter and try to prove the two implementations are the same up to a bound. If that is successful, the synthesis terminates, otherwise they obtain a counterexample, add it to the list of known examples and repeat.

Algorithm 2 Synthesis algorithm, which is run under symbolic execution.

```
1: counterexamples ← ∅
2: while TIMEOUT not exceeded do
3:     prog ←SYMBOLICMEMOBJ(MAX_PROG_SIZE)
4:     foreach cex ∈ counterexamples do
5:         ASSUME(ORIGINAL(cex) = INTERPRETER(cex, prog))
6:     end foreach
7:     KILLALLOTHERS( )                                  ▷ Only need a single path
8:     prog ← CONCRETIZE(prog)
9:
10:    example ←SYMBOLICMEMOBJ(MAX_EX_SIZE)
11:    example[MAX_EX_SIZE − 1] ← '\0'
12:    STARTMERGE( )
13:    originalOutput ← ORIGINAL(example)
14:    synthesizedOutput ← INTERPRETER(example, prog)
15:    isEq ← originalOutput = synthesizedOutput
16:    ENDMERGE( )
17:
18:    if ISALWAYSTRUE(isEq) then
19:        return prog                                   ▷ We are done
20:    end if
21:
22:    ASSUME(!isEq)
23:    cex ← CONCRETIZE(example)
24:    counterexamples ← counterexamples ∪ {cex}
25: end while
```

We took their approach further to synthesise whole functions instead of just adapters between function interfaces.

Our approach focuses on the types of loops described in Section 2.1.1, which can be extracted into functions with a `char* loopFunction(char* s)` signature. However, the technique should be easy to adapt to loops with different signatures. Our vocabulary of gadgets is the one described in Section 2.1.2.

Algorithm 2 presents our approach in detail. The algorithm is a program that when executed under symbolic execution performs CEGIS and returns the synthesised program representing the loop. This program is just a sequence of characters which can be interpreted as discussed in Section 2.1.2.

The algorithm makes use of the following symbolic execution operations:

- SYMBOLICMEMOBJ(N) creates a symbolic memory object of size N bytes.
- ASSUME(COND) adds the condition COND to the current path constraints.

- Concretize(x) makes symbolic input x concrete, by asking the constraint solver for a possible solution.
- KillAllOthers() prunes all the paths, except the path first reaching this call.
- IsAlwaysTrue(cond) returns *true* iff the condition cond can only be true.
- StartMerge() and EndMerge() merge all the paths between these two calls into a single path, by creating a big disjunction of all the path constraints of the merged paths.

The target loop for which we are trying to synthesise a summary is represented by the Original(s) function, with s the input string. Interpreter(s, program) executes the program on string s as discussed in Algorithm 1. Our aim is to synthesise a program such that: $\forall$s.Interpreter(s, program) = Original(s).

Algorithm 2 starts by initialising the set of all counterexamples encountered so far to the empty set (line 1). Counterexamples in this context are strings C on which the original loop and the synthesised program were found to behave differently, i.e.

$$\text{Interpreter}(C, \text{program}) \neq \text{Original}(C)$$

.

The algorithm is centered around a main loop (lines 2–25), which ends either when a synthesised program is found or a timeout is exceeded. On each loop iteration, we create a new symbolic sequence of characters representing our program (line 3) and constrain it such that its output on all current counterexamples matches that of the original function (lines 4–6).

When the program is run through the Interpreter function, there might be multiple paths on which the program is equivalent to the Original function for the current counterexamples. However, we are only interested in one of them. The computation spent on exploring other paths at this point is better used in the next iteration of the main loop, which will have more counterexamples to guide the synthesis. Therefore, we only keep a

single path by calling KillAllOthers (line 7). We also concretize the PROGRAM (line 8) to make the next step computationally easier.

The remainder of the loop body then focuses on finding a new counterexample, if one exists. Lines 10–15 attempt to prove the ORIGINAL function and PROGRAM have the same effect on a fresh symbolic string of up to length MAX_EX_SIZE on all possible paths. To be able to reason about all these paths at once, we merge them using StartMerge() and EndMerge() (lines 12–16).

Variable isEq on line 15 is a boolean variable that encodes whether the original loop and the synthesised program are equivalent on strings up to length MAX_EX_SIZE. Line 18 checks whether isEq can only take value *true*.

If so, we know that the synthesised PROGRAM behaves the same as the original loop on all strings of up to length MAX_EX_SIZE and based on a small world theorem from [61] [4] on strings of arbitrary length. Intuitively, the proof in [61] shows that the loops we target cannot distinguish between executing on a "long" string and executing on its suffix. In both executions, the value returned by the loop is uniquely determined by the number of iterations it completed, and every time the loop body operates on a character c, it follows the same path, except, possibly, when scanning the first or last character. Therefore, we can successfully return the PROGRAM as it is equivalent to the original loop.

Otherwise, we need to get a new counterexample and repeat the process in a next iteration of the main loop. For this, we first prune the paths where isEq is *true* by assuming it is *false* (line 22). Then, we obtain a counterexample by asking for a concrete solution for the EXAMPLE string that meets the current path constraints (where isEq is false) (line 23) and add it to the set of counterexamples (line 24).

2.2 Evaluation

We implemented our synthesis algorithm on top of KLEE commit 4432580, where KLEE already supports creating symbolic memory objects, adding path constraints, concretiz-

[4] Omitted from this thesis as it is not our contribution.

ing symbolic objects and merging of paths. Thus, we only needed to add support for KillAllOthers and IsAlwaysTrue, which required under 20 LOC.

We split our evaluation into three parts. First we describe how we gathered a large set of loops from real programs (§2.2.1). We then explore how many programs we can synthesise from this database with respect to time, vocabulary and maximum synthesised program size (§2.2.2). Finally, we evaluate the applications of loop synthesis in scaling up symbolic execution (§2.2.3), speeding up native execution (§2.2.4), and refactoring code (§2.2.5). All the experiments were run on a modern desktop PC with Intel i7-6700 CPU on Ubuntu 16.04 with 16GB of RAM.

A replication package can be found at `https://doi.org/10.1145/3325967` or `https://srg.doc.ic.ac.uk/projects/loop-summaries/artifact.html`.

2.2.1 Loop Database

We perform our evaluation on loops from 13 open-source programs: *bash*, *diff*, *awk*, *git*, *grep*, *m4*, *make*, *patch*, *sed*, *ssh*, *tar*, *libosip* and *wget*. These programs were chosen because they are widely-used and operate mostly on strings.

The process for extracting loops from these programs was semi-automatic. First, we used LLVM passes to find 7,423 loops in these programs and filter them down to 323 candidate memoryless loops. Then we manually inspected each of these 323 loops and excluded the ones still not meeting all the criteria for memoryless loops. The next sections describe in detail these two steps.

2.2.1.1 Automatic Filtering

After compiling each of the programs to LLVM IR, we apply LLVM's *mem2reg* pass. This pass removes load and store instructions operating on local variables, and is needed in our next step. LLVM's *LoopAnalysis* was then used to iterate through all the loops in the program, and filter out loops which are not memoryless. We automatically prune loops that have inner loops and then loops with calls to functions that take pointers as arguments or return a pointer.

Table 2.2: Loops remaining after each additional filter.

	Initial loops	Inner loops	Pointer calls	Array writes	Multiple ptr reads
bash	1085	944	438	264	45
diff	186	140	60	40	14
awk	608	502	210	105	17
git	2904	2598	725	495	108
grep	222	172	72	42	9
m4	328	286	126	78	12
make	334	262	129	102	13
patch	207	172	88	67	20
sed	125	104	35	19	1
ssh	604	544	227	84	12
tar	492	432	155	106	33
libosip	100	95	39	30	25
wget	228	197	115	83	14
Total	7423	6448	2419	1515	323

Then, we filter out loops containing writes into arrays. We assume that due to *mem2reg* pass, any remaining store instructions write into arrays and not into local variables. Therefore we miss loops where this assumption fails, as they get excluded based on containing a store instruction. Finally, we remove loops with reads from multiple pointer values. This ensures that we only keep loops with reads of the form $p_0 + i$ as per Definitions 1 and 2 of memoryless loops.

Table 2.2 shows, for each application considered, how many loops are initially selected (column *Initial loops*) and how many are left after each of the four filtering steps described above (e.g., column *Pointer calls* shows how many loops are left after both loops with inner loops and loops with calls taking or returning pointers are filtered out). In the end, we were left with between 9 and 108 loops per application, for a total of 323 loops.

2.2.1.2 Manual Filtering

We manually inspected the remaining 323 loops and manually excluded any loops that still did not meet the memoryless loops criteria from Section 2.1.1.

Two loops had `goto` in them, which meant they jumped to some other part of the function unrelated to the loops. Three loops had I/O side effects, such as outputting characters with

`putc` (note that the automatic *pointer calls* filter removed most of the other I/O related loops).

A total of 74 loops did not return a pointer, and an additional 70 loops had a return statement in their body. 28 loops had too many arguments. For example, incrementing a pointer while it is smaller than another pointer would belong into this category, as the other pointer is an "argument" to the loop. Finally, 31 loops had more than one output, e.g. both a pointer and a length.

Note that some of these loops could belong into multiple categories, we just record the reason for which they were excluded during our manual inspection. In total, we manually excluded 208 loops, so we were left with 323 - 208 = 115 memoryless loops on which to apply our synthesis approach.

As part of this manual step, we also extracted each loop into a function with a `char* loopFunction(char*)` signature. While this could be automated at the LLVM level, we felt it is important to be able to see the extracted loops at the source level.

2.2.2 Synthesis Evaluation

We evaluate our synthesis algorithm in several ways. First, we report its effectiveness using a generous timeout (§2.2.2.1), then we analyse how results vary with the size of the synthesised program and the timeout (§2.2.2.2), and finally how they vary with the size and shape of the vocabulary (§2.2.2.3).

2.2.2.1 Results with a Large Timeout

We first aim to understand how well our synthesis approach performs within a fixed and generous budget. We choose to run the synthesis with a 2-hour timeout, which is a reasonably long timeout for symbolic execution. We use the full vocabulary, as we want to capture as many loops as possible. Finally, we choose a maximum synthesised program size (MAX_PROG_SIZE in Alg. 2) of 9 characters based on some exploratory runs, where going above this number made synthesis very slow. The bound for checking program equivalence (MAX_EX_SIZE in Alg. 2) was set to length 3, which is sufficient for proof in [61] that shows

Table 2.3: Successfully synthesised loops in each program and the time taken by the synthesis process (with all gadgets, MAX_PROG_SIZE=9, MAX_EX_SIZE=3 and TIMEOUT=2h).

	% synthesised	Average (min)	Median (min)
bash	12/14	5.7	5.3
diff	3/5	5.1	5.3
awk	3/3	2.1	0.2
git	18/33	3.1	2.6
grep	1/3	4.4	4.4
m4	1/5	4.4	4.4
make	0/3	n/a	n/a
patch	9/13	1.8	0.2
sed	0/0	n/a	n/a
ssh	2/2	2.7	2.7
tar	10/15	10.2	4.5
libosip	12/13	12.6	5.3
wget	6/6	6.3	1.5
Total	77/115	6.1	4.5

equivalence for all string lengths.

Table 2.3 summarises the time it took to synthesise loops in each program. In total, we successfully synthesised 77 out of 115 loops, most under 10 minutes, well below our 2-hour timeout. Note that the median is sometimes significantly smaller that the average, indicating that a few loops were particularly difficult to synthesise. For example, in *libosip* 10 loops are synthesised within 10 minutes, while another 2 take well over an hour to complete. These 2 loops are summarised by a `strspn` with a four character argument, whereas most other loops have just 1 or 2 characters as arguments. *strpbrk*, *is start* and *reverse* gadgets were not synthesised during this experiment.

2.2.2.2 Influence of Program Size and Timeout

We next investigate how our synthesis approach performs with respect to the synthesised program size and the chosen timeout value. To do so, we use an iterative deepening approach where we gradually increase the program size from 1 to 9 and plot the number of programs we can synthesise. We run each experiment with timeouts of 30 seconds, 3 minutes, 10 minutes and 1 hour.

More generally, we advocate using such an iterative deepening approach when summarising

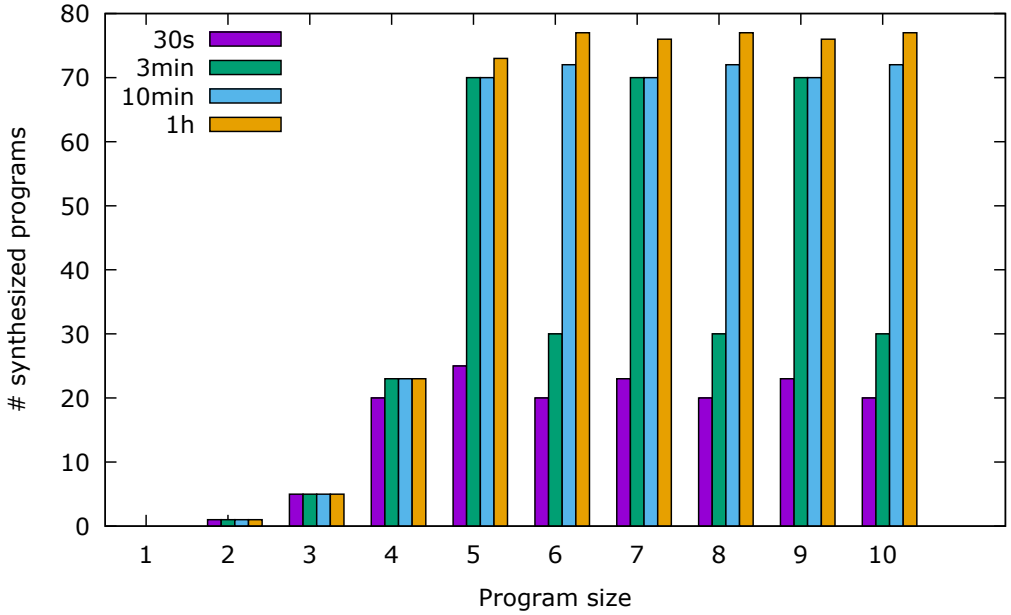


Figure 2.2: Number of programs synthesised as we increase program size, with different timeouts.

loops, as this strategy gives the smallest program we can synthesise for each loop, with reasonable additional overhead.

Figure 2.2 shows the results. Unsurprisingly, we cannot synthesise any programs with program size 1 as such a program cannot express anything but identity in our vocabulary. However, even with size 2 we can synthesise one program, namely `EF`, which iterates until the null character is encountered and then returns the pointer.

Figure 2.2 also shows that we can synthesise most loops with program size 5 and a timeout of only 3 minutes, which is encouraging for using this technique in practice. Increasing the program size or the timeout further provides only marginal benefits.

Another interesting observation is that there are some dips in the graph, e.g. when we increase the program size from 5 to 6 while keeping a 3-minute timeout. This is because increasing the program size has an impact on performance, as the search space of our synthesis algorithm increases with each additional character in the synthesised program.

2.2.2.3 Optimising the Vocabulary

We next explore how the performance of the synthesis approach depends on the gadgets included in the vocabulary, and propose a method for optimising the vocabulary used. Before we go into discussing results, let us first formalise our investigation.

Let N be the number of gadgets in our universe, and $G_1, G_2, \ldots G_N$ be the actual gadgets. In our concrete instantiation presented in Table 2.1, $N = 13$ and the gadgets are *rawmemchr*, *strchr*, etc.

We represent the vocabulary employed by the synthesis algorithm using a bitvector $v \in \{0,1\}^N$, where bit v_i of the vector is set if and only if the gadget G_i is used. For instance, a vocabulary allowing only the *rawmemchr* instruction would be represented as 1000000000000, a full vocabulary would be 1111111111111, and the vocabulary with three gadgets whose interpreter is shown in Algorithm 1 would be represented as 0000101000001.

The number of programs we can synthesise in a given time can be seen as a function of the vocabulary used, let us call it the success function $s\colon \{0,1\}^N \to \mathbb{N}$.

Understanding the influence of the vocabulary and choosing the best possible vocabulary is then equivalent to exploring the behaviour of s.

To exhaustively evaluate s we would need to perform $2^{13} = 8192$ evaluations of s. With a timeout of 5 minutes per loop and 115 loops each evaluation could take $5 * 115 = 575$ minutes. However some loops would synthesize before the timeout. In our setup described below, each evaluation of s took around 1 hour. Taking 1 hour as an optimistic estimate, we would need 8192 hours of computation time or a little over 341 days. Given enough resources this could be feasibly performed as evaluating s is trivially parallelizable, however it would be too expensive and wasteful for our purpose. Therefore, we use instead Gaussian Processes [76] to efficiently explore s.

While describing Gaussian Processes (GPs) in detail is beyond the scope of this thesis, we will give an intuition as to why they are useful in this case. GPs can be thought of as a probability distribution over functions. Initially, when we know nothing, they can represent any function (including s) albeit with small probability. The main idea is to get a real evaluation of s and refine the GP to better approximate s. Let's say we evaluate $s(v) = n$.

Table 2.4: The 7 vocabularies that perform better than the 2-hour experiment of §2.2.2.1.

Vocabularies	Synthesised programs
`MPNIFV`	81
`MPNBIFV`	80
`PNIFV`, `MPNIFVS`, `MPNBXIFV`	78

We can now refine the GP to have the value n at v with 0 variance. The variances around v also decrease, while variances far away from v do not.

We can repeat this refinement for other values of v to get an increasingly more precise model of s with the GP. The GP also tells us where to evaluate next. We can use GPs to optimise s — i.e. to optimise the vocabulary with respect to the number of programs we can synthesise — by looking at points where s is likely to have a large value [107].

We used GPyOpt [8] for an implementation of GPs. We use an expected improvement acquisition function, which is well suited for optimisation.

In our experiments, we chose to optimise the vocabulary by using a maximum program size of 7 and a timeout of 5 minutes per loop. As we will show, this is enough to beat the results of Section 2.2.2.1 that used a maximum program size of 9 and a much larger 2-hour timeout per loop. Our optimised vocabulary is obviously optimised for our benchmarks and might not generalise to a different set of benchmarks, however the optimisation technique should generalise.

The optimisation process evaluated s 40 times, and to our surprise, found 5 vocabularies that achieve better results in 5 minutes/loop than those achieved in 2 hours/loop in Section 2.2.2.1. These vocabularies are shown in Table 2.4. The largest number of loops synthesised was 81, for a vocabulary containing *rawmemchr*, *strspn*, *strcspn*, *increment*, *return* and *reverse*. The smallest vocabulary which still beats the results of Section 2.2.2.1 (in just 5 minutes) consists of only 5 gadgets: *strspn*, *strcspn*, *increment*, *return* and *reverse*. We note that the *reverse* gadget is never synthesised in the 2-hour experiment, because it is too expensive in the context of a full vocabulary. But as the GP discovered significantly smaller vocabularies, *reverse* was now used several times.

We also note that the programs synthesised by the best GP-discovered vocabulary

(`MPNIFV`) do not subsume the programs synthesised by the full vocabulary used in Section 2.2.2.1. For instance, there were 11 loops synthesised by `MPNIFV` and not the full vocabulary. These were mostly loops that look for a *strspn* from the end of the string. Conversely, there were 7 loops synthesised by the full vocabulary but not the `MPNIFV` vocabulary. Three of these were loops that required 3 arguments to *strspn*, one was a *strchr*, and 3 were a *strrchr* loop. Overall, the vocabularies in Tables 2.3 and 2.4 synthesised 88 out of the 115 loops.

2.2.3 Loop Summaries in Symbolic Execution

The next three sections discuss various scenarios that can benefit from our loop summarisation approach: symbolic execution (this section), compiler optimisations (§2.2.4) and refactoring (§2.2.5).

In Section 1.3.2 we illustrated how solving constraints in the theory of strings can benefit symbolic execution. As discussed, standard string functions can be easily mapped to the theory of strings, while custom loops cannot.

To measure the benefits of using a string solver instead of directly executing the original loops, we wrote an extension to KLEE [16] (based on KLEE revision 9723acd) that can translate our loop summaries into constraints over the theory of strings and pass them to Z3 version 4.6.1. We refer to this extension as *str.KLEE*, and to the unmodified version of KLEE as *vanilla.KLEE*.

Figure 2.3 shows the average time difference across all loops between these two versions of KLEE when we grow the symbolic string length. We use a 240-second timeout and show the average execution time across all loops. For small strings of up to length 8 the difference is negligible, but then it skyrockets until we hit a plateau where some loops start to time out with vanilla.KLEE. In contrast, str.KLEE's average execution time increases insignificantly, with an average execution time under 0.36s for all string lengths considered.

Figure 2.4 shows the speedup achieved for each loop by str.KLEE over vanilla.KLEE on a log scale. It uses symbolic input strings of length 13. For over half of the loops, str.KLEE achieves a speedup of more than two orders of magnitude, with several loops experiencing

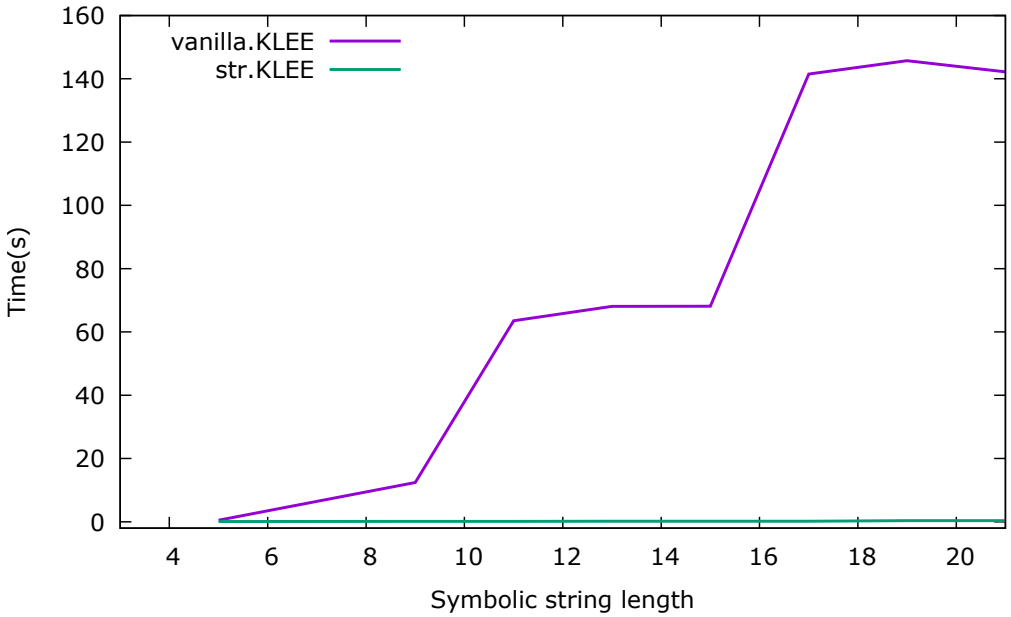


Figure 2.3: Mean time to execute all loops with str.KLEE and vanilla.KLEE, as we increase the length of input strings.

speedups of over 1000x (these include loops where str.KLEE times out, and where the actual speedup might be even higher). For others, we see smaller but still significant speedups. There is a single loop where str.KLEE does worse by a factor of 2.5x; this is a *strlen* function where it takes 1.4s, compared to 0.5s for vanilla.KLEE.

2.2.4 Loop Summaries for Optimisation

Compilers usually provide built-in functions for standard operations such as *strcat*, *strchr*, *strcmp*, etc. for optimisation purposes.[5][6] The reason is that such functions can often be implemented more efficiently, e.g. by taking advantage of certain hardware features such as SIMD instructions. In addition, compilers often try to identify loops that can be translated to such built-in functions. E.g., `LoopIdiomRecognize.cpp` in LLVM 8 "implements an idiom recogniser that transforms simple loops into a non-loop form. In cases that this kicks in, it

[5] `https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html`
[6] `https://clang.llvm.org/docs/LanguageExtensions.html`

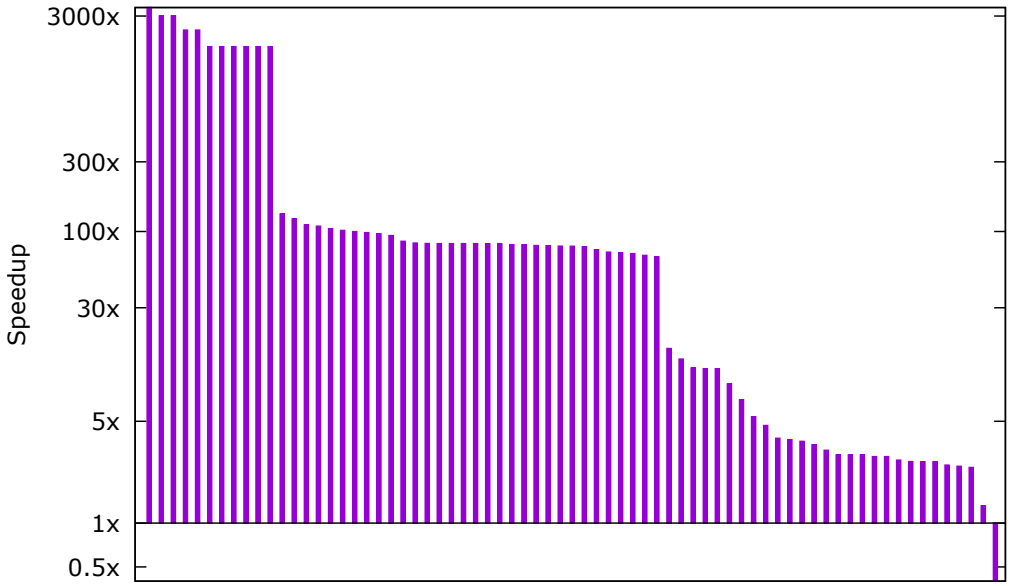


Figure 2.4: The speedup for each loop by str.KLEE over vanilla.KLEE for inputs of length 13, sorted by speedup value.

can be a significant performance win."[7]

However such loop recognisers are typically highly specialised for certain functions, so we wanted to understand if our more general technique could be helpful to compiler writers in recognising more loops. To do so, we built a simple compiler that translates our vocabulary back to C. We then benchmarked the compiled summary against the original loop function.

We ran each function for 10 million times on a workload of four strings about 20 characters in length. Picking the strings has a large impact on the execution time and it is difficult to choose strings that are representative across all loops, since they come from different programs and modules within those programs. Therefore, we make no claim that this optimisation should always be performed, rather we try to highlight that there are indeed cases where summarising loops into standard string functions can be beneficial.

The programs were compiled with GCC version 5.4.0 using -O3 -march optimisation level. Figure 2.5 shows the results of this experiment. The bars going up show cases in

[7] `http://llvm.org/doxygen/LoopIdiomRecognize_8cpp_source.html`

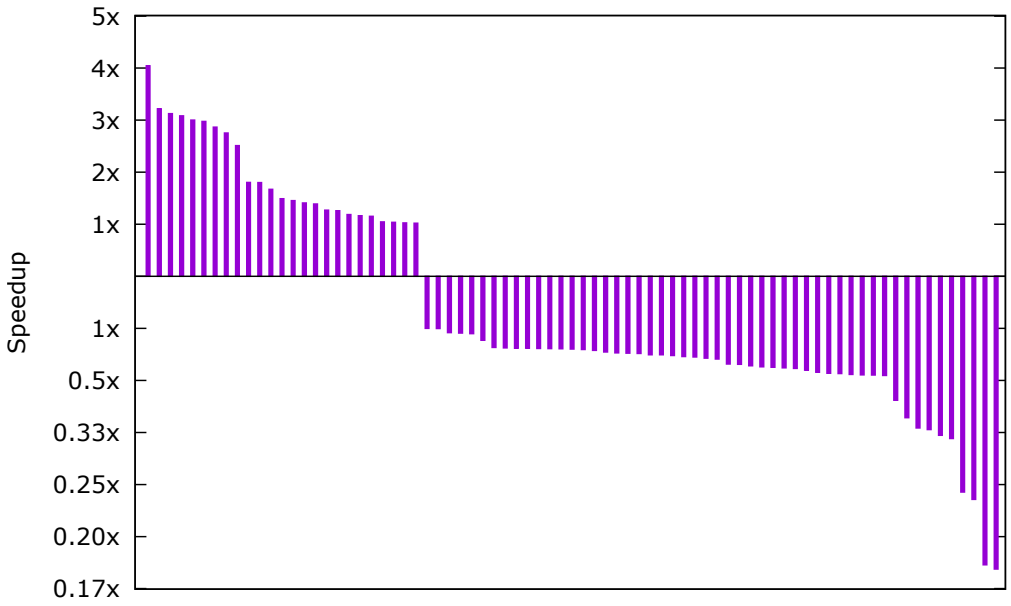


Figure 2.5: Relative time between running the original loop vs. running the synthesised program for each loop.

which the summarised version is faster, and bars going down cases in which it is slower. Whether the summary achieves a speedup or slowdown is mostly due to the function being synthesised. For instance, loops summarised by `strrchr` see important speedups because the glibc implementation of `strrchr` uses hand-crafted assembly based on fast vectorized instructions. Replacing the loop with a call to `strrchr` is beneficial here as the compiler does not optimise the loop enough to compete with handcrafted assembly.

By contrast, loops summarised by `strspn` are slower, because the glibc implementation of `strspn` builds a lookup table of the characters it needs to span over, which is expensive. This is because `strspn` is optimised for cases where one wants to span over a large set of characters and the span is large. In our example we had loops that e.g. simply skip whitespace, which does not fit this pattern.

Therefore, our data suggests that it is always beneficial to replace `strrchr`, whereas for `strspn` we should only replace when the accept string is long. A similar analysis can be performed for the other string functions in our vocabulary.

Program	Before	After
patch	`while (*s != '\n')` `  s++;`	`s = rawmemchr(s, '\n');`

libosip	`while (('␣' == *pbeg) || ('\r' == *pbeg)`
	`  || ('\n' == *pbeg) || ('\t' == *pbeg))`
	`  pbeg++;`
		`pbeg += strspn(pbeg, "␣\r\n\t");`

wget	`p = path + strlen (path);`
	`for (; *p != '/' && p != path; p--)`
	`  ;`
		`p = strrchr(path, '/');`
		`p = p == NULL ? path : p;`

Figure 2.6: Examples of loop summarisation patches accepted by developers.

2.2.5 Loop Summaries for Refactoring

Our technique could also be incorporated into a refactoring engine. We envision an IDE that would highlight certain loops and suggest to developers a change that would replace them with more readable and less error-prone calls to standard string functions.

We decided to evaluate the refactoring potential of our loop summarisation approach by manually submitting some patches that summarise loops in five open-source projects, selected from those used to build our loop database (see §2.2.1).

We submitted patches to five different applications, three of which accepted some or all of our proposed changes. Figure 2.6 shows three loop summarisation patches that were accepted in *patch*, *libosip* and *wget*.

The responses we received show that depending on the functionality involved, some developers prefer to have loops, while others prefer functions. For example, developers who rejected our `strspn` summaries cited as reasons `strspn`'s relative obscurity and the performance implications that we also discovered in Section 2.2.4. On the other hand, the developers who accepted our patches found the standard string functions more readable. In *wget*, we also noticed prior patches from developers that did similar replacements.

2.3 Limitations

Our approach, as presented, is limited by the single pointer input/output interface to which loops have to conform. This restriction could be relaxed. For example, allowing an integer

output instead of a pointer could be achieved with minor engineering effort. Allowing for loops that take two strings as input would be a larger effort. It would require both moderate engineering effort and a new small-model theorem than the one presented in [61]. The synthesis will also require new gadgets conforming to the two-pointer interface.

While the synthesis algorithm supports any code conforming to the single pointer interface as gadgets, gadgets are limited by the scalability of symbolic execution. The gadgets we chose are either constant (i.e. *isnull*), linear (i.e. *strchr*) or quadratic (i.e. *strspn*) in terms of their input size. Gadgets of higher than quadratic complexity would likely make the synthesis impractical.

We recognise that some of the loops we summarise could be recognised by more lightweight approaches such as source code pattern matching or scalar evolution approaches such as the one used by LLVM's LoopIdiomRecognize. However, it would be difficult to apply these approaches for complex loops that require additional modifications, such as conditionally incrementing a pointer after loop exit, setting it to the end of the string etc. More generally, pattern matching approaches require great manual effort in finding the patterns and then encoding them. In fact, during development we found it difficult to manually synthesise loops because equivalence checking kept finding incorrectly-handled edge cases, so we preferred to tweak the synthesis parameters rather than attempting to manually synthesise the loops.

In Section 2.2.3, we show large increases in the scalability of symbolic execution with our summaries. This does not directly imply the same speed-ups would be observed when running whole programs with loops summaries, however we believe this work is an important step towards scaling symbolic execution to large strings.

2.4 Related Work

Similar to our work, S-Looper [122] automatically summarises loops with the aim of improving program analysis. Their technique uses static analysis to enhance buffer-overflow detection. Our work is more general in that it is applicable to any analysis that operates on C directly, generating human-readable summaries that can even be used for refactoring.

Godefroid and Luchaup [49] use partial loop summarisation to enable concolic execution to reason about multiple paths through a loop at once. Their summaries consist of pre- and post-conditions, which they automatically infer during concolic execution. Similarly, *loop-extended symbolic-execution* [96] uses a combination of symbolic execution and static analysis to summarise loops in order to speed up symbolic execution. As for S-Looper, these two approaches are intertwined with their analysis, unlike our approach which can be immediately used in any technique.

STOKE [98] is an assembly level superoptimizer that speeds up loop-free code segments. With its recent extension to loops [24] their work is similar in spirit. They also use bounded verification to aid synthesis, but instead of a relying on a small-world theorem they use a sound verifier to generalise to arbitrary bounds. Their work focuses on optimising libc functions, whereas our work focuses on summarising loops in arbitrary programs, therefore we believe the work is complementary.

Srivastava et al. [108] present an approach synthesising loops from pre- and post-conditions using a verifier. While more precise, they require user-specified annotations, making it inapplicable as an automatic summarisation technique.

LLVM's LoopIdiomRecognize pass attempts to replace loops that match `memset` or `memcpy` patterns and is quite specific to these functions (other compilers, such as GCC, have similar passes that recognise patterns). It detects induction variables from which it can recognise stride load and store instructions. Their to-do includes functions like `strlen` for over six years, showing that such passes require significant expertise to implement. By contrast, our approach is more general and can easily be extended by adding a gadget.

More generally, program synthesis has seen renewed interest in recent years [104, 50, 51, 90, 52, 108, 1, 29]. Our synthesis approach is based on CEGIS [105], with the synthesizer and verifier both based on dynamic symbolic execution [101].

2.5 Conclusion

In this chapter, we presented a novel approach for summarising loops in C code. This approach uses counterexample-guided synthesis to generate the loop summaries, Gaussian processes to optimise the vocabulary used, and rely on a formal proof [61] to show that the summaries are correct for unbounded loop lengths if they are correct for the first two iterations.

We evaluated our approach on a large loop database extracted from popular open-source systems and assessed its utility in several contexts: symbolic execution, where we recorded speedups of several orders of magnitude; compiler optimisations, where several summaries resulted in significant performance improvements; and refactoring, where some of our summary patches were accepted by developers.

The work on extending KLEE to translate loop summaries into theory of string constraints required in-depth understanding of the memory models for symbolic execution of C-like programs, which we applied in the next chapter.

Chapter 3

Segmented Memory Model

One of the main strengths of symbolic execution is that it can reason precisely about *all* possible values of each control-flow execution path it explores through the code under testing. This amplifies its ability of finding bugs and other corner cases in the code being analysed. However, symbolic execution still struggles to reason in this way about complex data structures that involve pointers that can refer to multiple memory objects at once, which we briefly alluded to in Section 1.3.1.1.

To illustrate this problem in further detail, consider the C code in Figure 3.1, where we set N to 40. When `SINGLE_OBJ` is defined, the program allocates on line 3 a 2D matrix as a single object, for which all elements are initially zero. It then writes value 120 into `matrix[0][0]` on line 10. On lines 12 and 13, it declares two symbolic indexes `i,j` which are constrained to be in the range `[0,N)`. Finally, on line 15, it checks whether the element `matrix[i][j]` is positive.

The program has two paths: one on which `matrix[i][j]` is positive (which happens only when `i` and `j` are both zero), and the other on which it is not (which happens when either `i` or `j` are not zero). Unsurprisingly, a symbolic execution tool like KLEE [16] explores the two paths in a fraction of a second.

Now consider the program in the same figure, but without defining `SINGLE_OBJ`. Now the matrix is allocated as multiple objects, one for each row in the matrix. In this case, KLEE

```
 1 int main() {
 2 #ifdef SINGLE_OBJ
 3   int matrix[N][N] = { 0 };
 4 #else
 5   int **matrix = malloc(N * sizeof(int*));
 6   for (int i = 0; i < N; i++)
 7     matrix[i] = calloc(N, sizeof(int));
 8 #endif
 9
10   matrix[0][0] = 120;
11
12   int i = klee_range(0, N, "i");
13   int j = klee_range(0, N, "j");
14
15   if (matrix[i][j] > 0)
16     printf("Found␣positive␣element\n");
17 }
```

Figure 3.1: 2D matrix allocated as a single object when `SINGLE_OBJ` is defined, and as multiple objects when it is not.

will explore $N + 1$ paths, one for each row in the matrix, and an additional one for the first row in which both sides of the `if` statement are feasible. It will take KLEE significantly longer — 12s instead of 0.3s in our experiments — to explore all 41 paths.

The reason is that KLEE, as well as other symbolic executors, use the concrete addresses of memory objects to determine to which objects a symbolic pointer could refer. In our example, `matrix[i]` is a symbolic pointer formed by adding a concrete base address (the address of `matrix`) with a symbolic offset `i`. When this pointer gets dereferenced on line 15, KLEE needs to know to which memory object it points. Therefore, KLEE scans all the memory objects and asks the solver if the symbolic pointer can be within its bounds. If that is true for more than one object, we are in what is called a *multiple resolution* case. This is the case in our example, where the pointer `matrix[i]` can point to N different objects, namely those corresponding to the N rows of the matrix. (Note that when `SINGLE_OBJ` is defined this is not the case, as the compiler allocates a 2D array as a 1D array indexed by `N*i+j`.) So KLEE forks one path for each of the N objects, constraining the pointer to refer to a single memory object in each case. For the case where `matrix[i]` refers to the first row of the matrix, KLEE explores two paths, one in which `matrix[0][j]` is greater than zero, and one where it is not. For the cases in which `matrix[i]` refers to one of the other rows, only the case where `matrix[i][j]` is not greater than zero is feasible. This gives us a total

of `N+1` paths.

This *forking memory model* for symbolic pointers has several important drawbacks. First, it leads to path explosion, as each time the code dereferences a symbolic pointer, the execution is forked into as many paths as there are objects to which the symbolic pointer could refer. Second, and related to the first point, it leads to memory exhaustion, as each additional path requires a certain amount of space in memory. Third, it disables the ability of symbolic execution to reason about all possible values on a certain control-flow program path.

In this chapter, we propose a novel approach that uses pointer alias analysis to group memory objects that may alias each other into a *memory segment*. This avoids forking, because conservative alias analysis guarantees that a symbolic pointer can only point to objects within a single memory segment.

We describe our technique in detail in Section 3.1, present its implementation in Section 3.2, and evaluate it in Section 3.3. We then discuss its trade-offs in Section 3.4 and present related work in Section 3.5.

3.1 Proposed Memory Model

Consider the program in Figure 3.1 (with `SINGLE_OBJ` not defined) for $N = 3$, with a memory layout shown on the left-hand side of Figure 3.2. `matrix` is an array of pointers allocated at $0x0100$ that points to the rows of the matrix allocated at addresses $0x0200, 0x0300$ and $0x0400$. Suppose that libc also allocates two memory objects in the background, at addresses $0x0500$ and $0x0600$. There are also two symbolic pointers: `matrix[i]`, which is constrained to point to one of the three matrix rows, and `matrix[2][j]`, which is constrained to point to one of the entries of the last row.

Most symbolic executors reflect this memory layout in their internal memory model. They record the start and end addresses of each memory block and associate a corresponding array in the constraint solver. Therefore, they can handle symbolic pointers such as `matrix[2][j]` well, because their value resolves to a single memory object. The symbolic address then gets

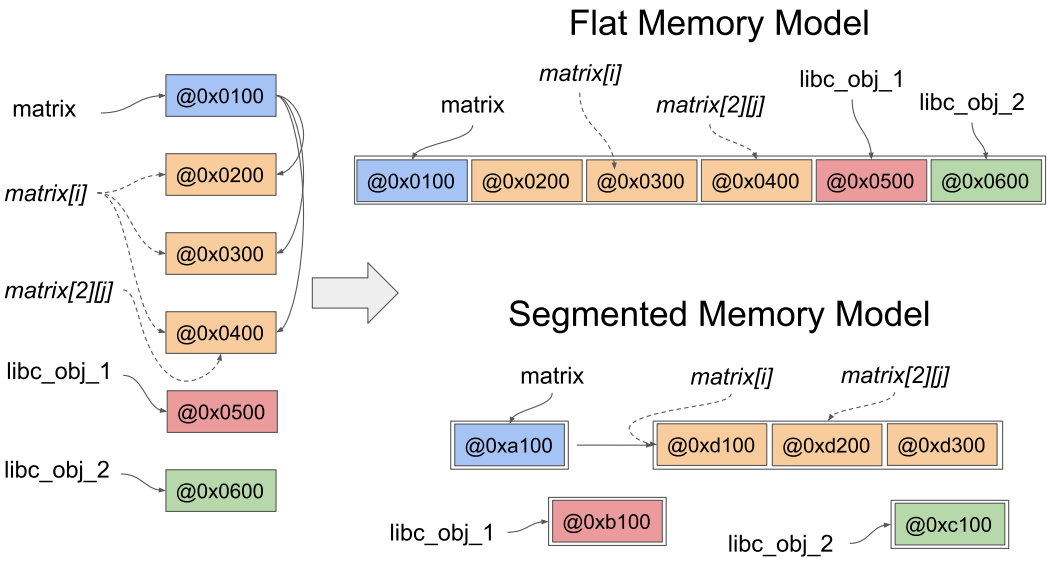


Figure 3.2: A concrete memory layout for the program in Figure 3.1 when `SINGLE_OBJ` is not defined, illustrating the flat and segmented memory models.

translated into a symbolic index into the solver array that is backing the memory object. Solvers can then solve the associated queries with the popular theory of arrays (§1.3.1).

The problem arises when a symbolic pointer can point to multiple memory objects, as is the case for pointer `matrix[i]`. This pointer resolves to three different memory objects (the matrix rows), backed by three different solver arrays. The theory of arrays cannot be used here, because it cannot express such constraints.

3.1.1 Existing Memory Models

There are several memory models for handling multiple resolution that have been considered in the context of symbolic execution: single-object, forking, flat memory and merging. Each memory model aims to resolve a dereference of a symbolic pointer to an access into a solver array.

Single-object model. In this approach, the pointer is resolved to one possible object and the other possibilities are discarded. This is the model used by most concolic executors, such as CREST [28], where the single object considered is the one given by the concrete

Algorithm 3 Forking model.

```
1: function ForkingDereference(Pointer p)
2:     foreach MemoryObject o do
3:         ForkPath( )
4:         AddConstraint(o.start ≤ p < o.end)
5:         solverArray ← GetSolverArray(o)
6:         return solverArray[p − o.start]
7:     end foreach
8: end function
```

input used in the current round of concolic execution. This is also the model used by the dynamic symbolic executor EXE [18], where the pointer is concretised to one possible value. FuzzBall [80] employs a more general version of this model, in which the pointer is concretized to multiple values that are selected randomly.

This approach scales well, but is incomplete and may miss important execution paths. For instance, both CREST and FuzzBall usually finish running the multi-object version of the code in Figure 3.1 (when slightly adapted to use the CREST/FuzzBall APIs) without exploring the feasible path where "Found positive element" is printed, incorrectly giving the impression to the user that the statement is not reachable.

Forking model. In this approach, shown in Algorithm 3, execution is forked for each memory object to which the pointer can refer (line 3), and on each forked path the pointer is constrained to refer to that object only (line 4). AddConstraint terminates the path if the added constraint is provably false (in our case, if p cannot refer to that object on that path). Finally, the corresponding access into the solver array associated with that object is returned (line 6). As a detail, ForkPath uses the initial path for the last object (instead of forking). Also, for ease of exposition, in this algorithm and the following ones, we ignore out-of-bounds checking and assume dereferences of `char*` pointers.

Consider running Algorithm 3 on our running example from Figure 3.1 and the layout from Figure 3.2, with p being `matrix[i]`. The loop will iterate six times, forking for each of the six memory objects in the program. In the first iteration, the path will be immediately killed by AddConstraint, since p cannot point to the object at $0x0100$. In the second iteration, p will be constrained to point to the first object it can point to ($0x0200$), and then the corresponding access into the solver array associated with that object is returned,

i.e. $solverArray_{0x0200}[p - 0x0200]$. The remaining iterations are similar, resulting in three paths, one for each of the three objects at $0x0200$, $0x0300$ and $0x0400$ that p can point to.

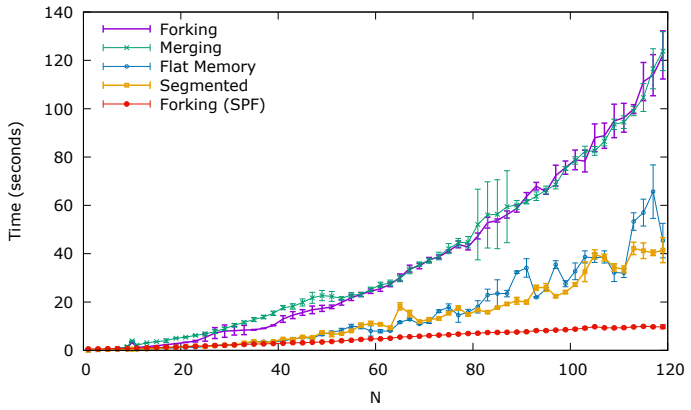
Figure 3.3 shows the behaviour of the forking model compared to other memory models. In particular, it shows the behaviour of KLEE and Symbolic PathFinder (SPF) [88], both of which implement the forking model. Figure 3.3a shows results for the code in Figure 3.1 (with `SINGLE_OBJ` not defined), while Figure 3.3b shows results for a variant of the code that performs two symbolic lookups into the matrix. These figures show that the forking model scales poorly. This is particularly clear when two symbolic accesses are performed, where the execution time increases exponentially with the dimension of the matrix. Note that SPF appears to have an efficient implementation of forking for a small number of paths, which explains its good performance on the single lookup example.

Merging model. This approach keeps a single path, and creates an *or* expression, with one disjunct for each possible object to which the pointer could refer. The write operation of the state merging model is illustrated in Algorithm 4.

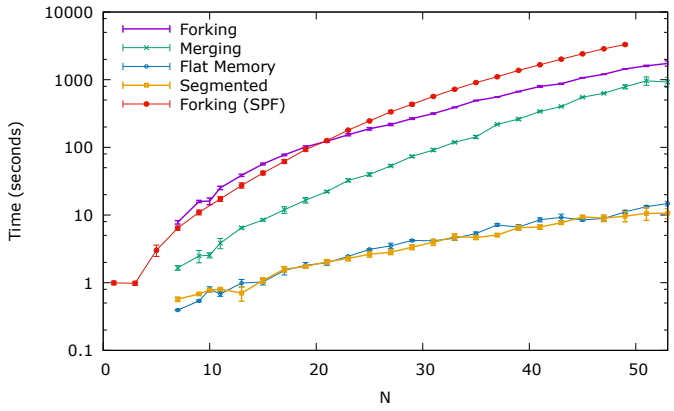
Algorithm 4 State merging.

```
1: function StateMergingWrite(Pointer p, Value v)
2:     expr ← false
3:     for MemoryObject o ∈ Objs(p) do
4:         s ← GetSolverArray(o)
5:         inBounds ← o.start ≤ p < o.end
6:         expr ← expr ∨ (inBounds ∧ (s[p − o.start] = v))
7:     end foreach
8:     return expr
9: end function
```

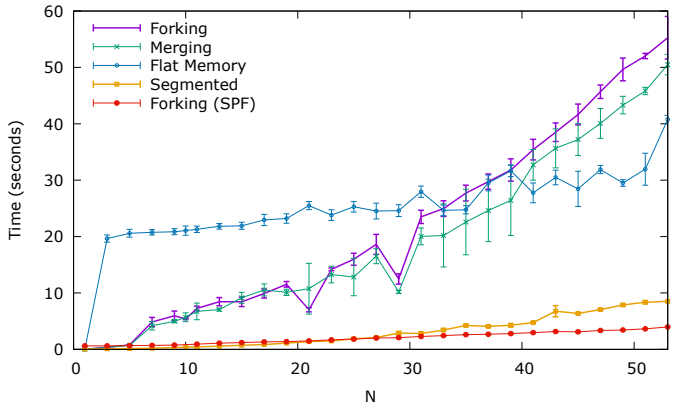
Consider running Algorithm 4 on our running example. Suppose that the symbolic executor encounters the expression $*p == 1$, with p being again $matrix[i]$. This will be translated into a disjunction, with one disjunct for each of the three memory objects to which p can refer. Each disjunct will express the fact that p points to that object and will replace the pointer with the associated solver array for that object. That is, the expression $*p == 1$ will be translated to the following disjunction, where sa stands for $solverArray$:



(a) 2D matrix with 1 symbolic lookup (linear scale)



(b) 2D matrix with 2 symbolic lookups (log scale)



(c) 2D matrix with 1 symbolic lookup and extra allocation (linear scale)

Figure 3.3: Runtime of different memory models on a family of 2D matrix benchmarks based on Figure 3.1 with `SINGLE_OBJ` undefined. All memory models are implemented in KLEE, except for the one explicitly mentioning Symbolic PathFinder.

$$(0x0200 \leq p < 0x020C \wedge sa_{0x0200}[p - 0x0200] = 1)$$
$$\vee\ (0x0300 \leq p < 0x030C \wedge sa_{0x0300}[p - 0x0300] = 1)$$
$$\vee\ (0x0400 \leq p < 0x040C \wedge sa_{0x0400}[p - 0x0400] = 1)$$

Approaches following this idea at a high level were proposed in the context of SAGE [34] and Angr [35].

As illustrated in Figures 3.3a and 3.3b, the scalability of this approach on the running example is similar or slightly better than that of forking. In practice (§3.3), its performance is benchmark dependent.

Flat memory model. This approach tackles the multiple resolution problem by treating the whole memory as a single object, backed by a single solver array. Therefore, a solver can reason about symbolic dereferences using the theory of arrays, because all queries are using a single flat array.

Algorithm 5 Flat memory model.

```
1: function NaiveFlatMemoryDereference(Pointer p)
2:     return memorySolverArray[p]
3: end function
```

The algorithm for translating a dereference is straightforward and shown in Algorithm 5. Any pointer in the program is an offset into the large solver array $memorySolverArray$. This approach is also illustrated in the top right of Figure 3.2. This is the approach taken by some binary symbolic executors [11] such as ANGR [102] and BINSEC/SE [30].

As shown in Figures 3.3a and 3.3b, this approach achieves good scalability on our running example and the variant with two symbolic lookups. However, this is because these examples allocate little memory. In an additional experiment, we modify the example to perform an additional irrelevant allocation of 30KB, which is meant to simulate a real application in which most of the memory is not involved in any single dereference. The results in Figure 3.3c show that the flat memory model also scales poorly.

3.1.2 Segmented Memory Model

The key idea behind our approach is the realisation that memory can be divided into multiple segments, such that any symbolic pointer can only refer to memory objects in a single memory segment. We can then associate one solver array per memory segment, and translate the symbolic pointer to an offset into its associated memory segment. As long as the individual memory segments are small enough, the approach can scale while still handling the problem of symbolic pointers referring to multiple objects.

On our running example, Figure 3.3 shows this approach has the advantages of the flat memory model in the first two cases, while maintaining its good performance when irrelevant memory allocations are performed.

Computing memory segments. To divide memory objects into segments, we use a conservative points-to analysis [5, 54], which we briefly introduced in Section 1.4. To recap, the result of a points-to analysis is a mapping between pointers and points-to sets. The *points-to set* of a pointer p is a set of pointers and abstract memory objects to which p may refer during execution. An *abstract memory object* is identified by the static allocation point in the program. For instance, in Figure 3.1, there are two abstract memory objects, AO_1, corresponding to the allocation at line 5, and AO_2, corresponding to the allocation at line 7.

The function ComputeMemorySegments in Algorithm 6 shows the algorithm for computing memory segments. After running the points-to analysis (line 2), the set $ptSets$ is initialised to contain all the points-to sets computed by the analysis (lines 3–6). Then, any two points-to sets s and s' that overlap are merged until no such sets exist anymore (lines 7–13). Finally, for each of the resulting points-to sets a new memory segment is created (lines 14–15) in which all the objects associated with that points-to set will be allocated. The map AssocPTSets remembers the points-to set associated with each memory segment (line 16).

In our example from Figures 3.1 and 3.2, there are two pointers we consider: `matrix` and `matrix[i]`. Pointer analysis tells us the points-to set of `matrix` is a singleton set containing the abstract object AO_1. Similarly, the points-to set of `matrix[i]` is a singleton set containing AO_2. Continuing with Algorithm 6, we therefore initialise $ptSets = \{\{AO_1\}, \{AO_2\}\}$ (lines

Algorithm 6 Computing and using memory segments.

```
 1: function ComputeMemorySegments
 2:     PerformPointsToAnalysis( )
 3:     ptSets ← ∅
 4:     foreach pointer p do
 5:         ptSets ← ptSets ∪ PointsToSet(p)
 6:     end foreach
 7:     while ptSets changes do
 8:         foreach s ∈ ptSets do
 9:             if ∃s′ ∈ ptSets . s′ ≠ s ∧ s′ ∩ s ≠ ∅ then
10:                 ptSets ← (ptSets \ {s, s′}) ∪ {s ∪ s′}
11:             end if
12:         end foreach
13:     end while
14:     foreach s ∈ ptSets do
15:         seg ← new MemorySegment( )
16:         AssocPTSet(seg) ← s
17:     end foreach
18: end function
19:
20: function GetMemorySegment(Pointer p)
21:     pts ← PointsToSet(p)
22:     foreach MemorySegment seg do
23:         if pts ⊆ AssocPTSet(seg) then
24:             return seg
25:         end if
26:     end foreach
27: end function
28:
29: function HandleAlloc(Pointer p, Size sz)
30:     seg ← GetMemorySegment(p)
31:     return AllocateIn(seg, sz)
32: end function
```

3–6). In this case there are no common elements in those sets, so lines 7–13 have no effect. Finally, we create two memory segments corresponding to the two elements in $ptSets$. AO_1 corresponds to the memory segment starting at $0xa100$ in Figure 3.2, where the array of pointers to the matrix rows will be allocated. AO_2 corresponds to the memory segment starting at $0xd100$, where the actual rows of the matrix will be allocated.

The function GetMemorySegment in Algorithm 6 returns the memory segment associated with a pointer p in the program. It does so by finding the memory segment whose associated points-to set contains the points-to set of p. (We note that an implementation could be optimised by keeping a reverse mapping from points-to sets to memory segments.)

Algorithm 7 Pure segmented memory model.

```
1: function PureSegmentedDereference(Pointer p)
2:     seg ← GetMemorySegment(p)
3:     solverArray ← GetSolverArray(seg)
4:     return solverArray[p − seg.start]
5: end function
```

Algorithm 8 Segmented memory model.

```
1: function SegmentedMemoryDereference(Pointer p)
2:     foreach MemorySegment seg do
3:         ForkPath( )
4:         AddConstraint(seg.start ≤ p < seg.end)
5:         solverArray ← GetSolverArray(seg)
6:         return solverArray[p − seg.start]
7:     end foreach
8: end function
9:
10: function AllocateIn(MemorySegment seg, Size sz)
11:     if seg.size ≤ Threshold then
12:         return Extend(seg, sz)
13:     else
14:         seg′ ← ChooseSegment( )
15:         return AllocateIn(seg′, sz)
16:     end if
17: end function
```

GetMemorySegment is then used by HandleAlloc in Algorithm 6 to handle a heap allocation of the form `p = malloc(sz)` in C. The function allocates memory for the target pointer p in its associated memory segment. In our example, it returns the memory segment starting at $0xa100$ when encountering the allocation for `matrix` on line 5. For the allocation on line 7 where row allocations are made, it returns the memory segment starting at $0xd100$.

Finally, Algorithm 7 shows how a dereference of a pointer p is handled. We first obtain the memory segment associated with p (line 2), then we get the solver array associated with that segment (line 3), and return the corresponding solver array access (line 4).

Restricting segment size. The approach of Algorithm 7 scales as long as the size of each memory segment remains small. In practice, some segments may become very large, imposing significant overheads on the solver. To ensure segment sizes remain small, we refine the technique by imposing a threshold on the maximum segment size. If the program tries to allocate memory in a certain segment and the current size of that segment already exceeds

the threshold, a new segment is used. When this happens, some pointers may now refer to several segments. In that case, dereferencing a pointer forks execution for each memory segment. Essentially, the model becomes a hybrid between the pure segmented memory model of Algorithm 7 and the forking model of Algorithm 3.

Our finalised approach is illustrated in Algorithm 8, (together with the previously discussed Algorithm 6). The dereference function, SegmentedMemoryDereference, is similar to that of the forking model, only that now we iterate over all segments, trying to find those to which p could refer. Remember that the AddConstraint function kills a path if the added constraint is *false*, that is when the pointer cannot refer to that segment.

The other change is required in the allocation function. Instead of simply allocating the required memory in the given segment, the modified $AllocateIn$ function works as follows. If the current size of the segment is under the threshold, the allocation is performed in that segment (lines 11–12). The Extend function extends the memory segment by the size and performs the allocation there. This is done to keep the segment sizes small for as long as possible. Otherwise, we choose another segment (which could either be a new segment or an existing one) and we try to allocate in there (lines 14–15).

Another consequence of this hybrid model is that it is resilient to bugs in pointer alias analysis. Should it incorrectly report two pointers not aliasing each other, putting them in different memory segments, the hybrid approach would simply fork for the two cases without losing any precision. Conversely, this kind of bug would cause the pure segmented memory model to miss paths.

In practice, the performance of the algorithm depends on the precision of the points-to analysis. If the analysis is extremely imprecise, it may put all pointers into a single points-to set, and our model would degenerate to the flat memory model. At the other extreme, if the threshold is too small (or the points-to analysis extremely buggy), our model would degenerate to the forking model.

3.2 Implementation

We implemented our approach on top of KLEE commit 0283175f, configured to use LLVM 7 and the STP constraint solver version 2.1.2. An artifact is available at `https://srg.doc.ic.ac.uk/projects/klee-segmem/artifact.html` and `https://doi.org/10.1145/3345840`. Below we provide additional details about our implementation.

Points-to analysis. We use SVF [111], a scalable LLVM-based framework for value-flow construction and pointer analysis. In particular, we use whole-program wave propagation-based Andersen analysis [89], a fast flow-insensitive, context-insensitive inclusion-based points-to analysis. We found it to be significantly faster than the basic Andersen analysis [5] while maintaining precision. It terminated in under two minutes for all our benchmarks. Compared to the high runtime of symbolic execution, we found the analysis time to be insignificant.

Memory segment implementation. For each memory segment that needs to be created in Algorithm 6 (line 15) we reserve 200KB of address space using `mmap`. At this point the segment has size 0, but can hold an object of maximum size 200KB. We use `mmap` instead of `malloc` to save memory, as in practice many memory segments will be empty and will therefore not be mapped to physical memory.

The threshold size for the hybrid approach can reasonably range between 5KB to 30KB. STP starts hitting scalability issues with arrays larger than 30KB. The lower bound of 5KB comes from KLEE's warning, where KLEE issues a warning if an object is larger than 5KB, which indicates than objects smaller than that should not be of concern. From this range we tried 10KB and it worked. That is STP was still performing well and the segments were not too fragmented.

We write the size of each allocation just before the returned pointer to enable `realloc` and `free`. The `realloc` function is handled with an LLVM pass that runs before the execution and replaces all calls to `realloc` with `malloc`, `memcpy` and `free`. We use the previously stored allocation size information to determine the size of the region to copy with `memcpy`.

The memory freed by calls to `free` is added to a list of free space in each memory segment.

That list is then scanned before a memory segment is extended for allocation. We try to find the smallest gap that fits the requested size, and if such a gap is found, we perform the allocation there. We found this to be an important optimisation to keep the sizes of memory segments small.

Constants and local variables. A common performance optimisation in pointer alias analysis is to model all constant objects (such as constant strings) as single memory objects [55]. We similarly adopt a scheme in which constant objects are disregarded by the alias analysis and allocated in their own memory segments. We make the same choice for local memory objects, namely objects allocated on the stack. We found only one program where constant objects are involved in multiple resolution, but this was a case where the forking model would work well too, as the pointer could only refer to two different objects. We found no program where local objects were involved in multiple resolution. The approach presented here is easily adapted to support these kind of objects, should a need for it arise.

Flat memory model. We found it easy to implement the flat memory model in KLEE, by making all allocations into a single memory segment backed by a single solver array.

Merging model. The merging model is implemented by leveraging KLEE's ability to merge states. Upon hitting a multiple resolution, we let KLEE fork as usual and then immediately merge all the states thus creating the *or* expression of their path conditions. We note that this approach might not be optimal — however, KLEE spent most of the time in constraint solving activities in the merging runs in our evaluation. Therefore, we believe the threat to validity against this implementation choice is minimal.

3.3 Evaluation

Symbolic pointer dereferences that trigger multiple resolutions often make existing symbolic execution tools unusable. However, only some types of programs trigger such dereferences, and we discovered that benchmark suites used in the past to evaluate symbolic execution, such as *GNU Coreutils* [40], do not trigger many multiple resolutions.

In general, benchmarks which are prone to trigger multiple resolutions are those where a

symbolic input (e.g. coming from files, command-line arguments or environment variables) flows into a data structure indexed by that input. Hash tables are the prime example of such benchmarks, widely used in a variety of programs.

Therefore, we selected several large programs that use hash tables — *m4*, *make*, *APR* and *SQLite3*— and used test harnesses that drive execution toward the parts of the code employing these hash tables. Essentially, our evaluation is meant to show that our model can make a big difference in such cases, while acknowledging that it is not relevant all the time. We further discuss this aspect in Section 3.4.

Ideally, we would like to compare the fraction of the search space explored in these programs under each memory model. Unfortunately, this is challenging to do, especially since the number of explored paths changes across models due to forking and merging. Therefore, we decided to build our test harnesses in such a way that programs terminate. Then, we can simply compare the time needed to finish the exploration under different memory models. For one program where we did not manage to write such a driver, *make*, we measure how quickly each memory model reaches coverage saturation. As a sanity check, we made sure that terminating benchmarks reach the same coverage at the end under all memory models, and that the flat memory model and the two variants of the segmented memory model explore the same number of paths (recall that we did not see any forking in these benchmarks with the hybrid segmented memory model).

We also measure the memory consumption of KLEE under each memory model as a proxy for how many states KLEE has to keep in memory at each point and to illustrate a further tangible benefit of choosing the right memory model.

We observed that execution time and memory consumption can vary significantly with different search strategies. Therefore, we conducted our experiments using three different strategies: DFS and BFS representing extreme behaviors in terms of memory consumption, and KLEE's default search heuristic[1] representing a good general strategy.

The experiments were run on an Intel(R) Core(TM) i7-6700 at 3.40GHz with 16GB of memory. We use a timeout of two hours in our experiments and a memory limit of 4GB.

[1]Default heuristic interleaves random path selection with a heuristic that aims to maximize coverage.

Table 3.1: Impact of points-to analysis on our benchmarks and its runtime.

Benchmark	Total mem. (bytes)	Max. segment size (bytes)		Analysis runtime (s)
		ideal	SVF	
APR	24,904	120	16,588	1.9
GNU m4	3,392	1,051	2,753	4.3
GNU make	17,663	664	7,936	4.7
SQLite3	45,461	506	17,604	70.3

3.3.1 Impact of Points-to Analysis

The performance of the segmented memory model is highly dependent on the size of the segments, which is directly related to the precision of the points-to analysis. In order to understand by how much our results would be improved by a more precise points-to analysis, we decided to approximate a best-case points-to analysis.

To do so, we run the forking approach on our benchmarks until it hits the first multiple resolution, where we record the allocation sites with allocation context of all the objects involved in the multiple resolution and terminate the execution. We then restart the execution, placing all objects allocated at those allocation sites in the appropriate contexts into a segment. Should a new multiple resolution be encountered, we again record the allocation site and its context, create a new segment and merge any segments if needed, and repeat this process until no more multiple resolutions are encountered (within a certain timeout). In practice, we only needed fewer than five iterations for our benchmarks. Essentially, we group into segments only the objects actually involved in multiple resolutions in a given execution. This gives us results which are at least as good as the most precise points-to analysis possible.

Table 3.1 shows the impact of the pointer alias analysis on the benchmarks we use in our evaluation (and which will be described in detail in Sections 3.3.2 to §3.3.5). The first column shows the total memory used by the dynamically-allocated memory objects in each program. This is the size of the flat memory segment.

The next two columns show the maximum segment sizes for both the SVF pointer analysis and our idealized analysis. For all benchmarks, the points-to analysis significantly reduces

```
1 define(‘A’, ‘l’)
2 define(‘P’, 2)
3 ?
4 ?
5 ifelse(?, ‘l’, ifelse(?, P, eval(1 + n)) ,‘failed’)
```

Figure 3.4: Symbolic input to *m4*, where ? denotes a symbolic character.

the largest segment size. The maximum segment size of the ideal analysis is significantly smaller than for the SVF analysis, suggesting there are important opportunities for improving the precision of SVF.

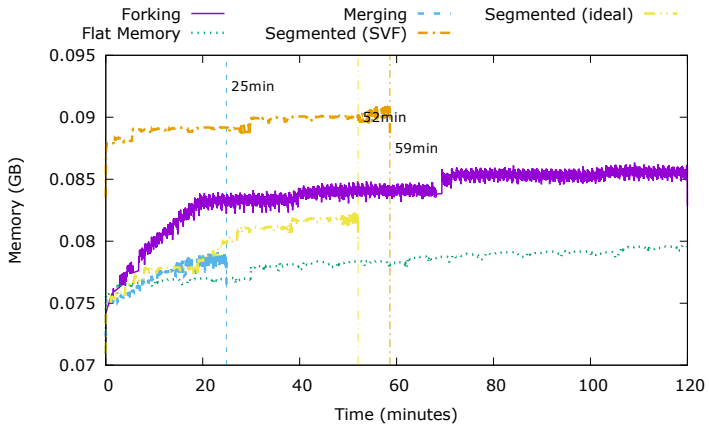
Using the SVF analysis, the *APR* and *SQLite3* benchmarks reach the maximum threshold of 10KB for our segments and get split. Despite this split, we did not observe any forking in the segmented memory model for *APR*. For *SQLite3*, we observed two forks into two paths each.

The last column of Table 3.1 shows the runtime of the SVF points-to analysis. For all benchmarks but *SQLite3*, the analysis took less than five seconds, which is insignificant compared to the cost of symbolic execution. Even for complex programs like *SQLite3*, the analysis took 70s, which is noticeable, but still little in a symbolic execution context.

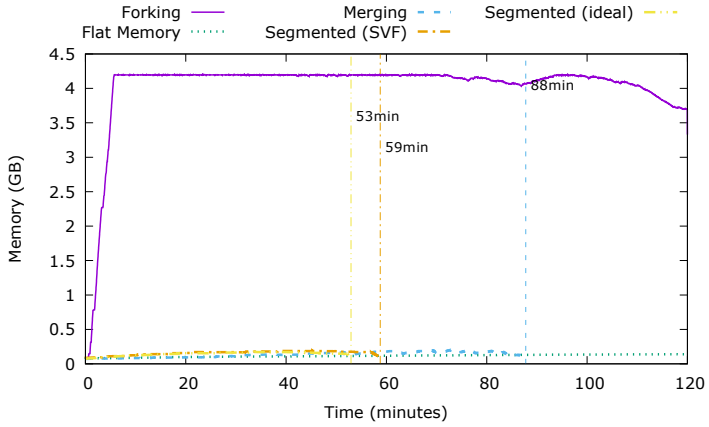
3.3.2 *GNU m4*

GNU *m4* [75] is a popular implementation of the *m4* macro processor included in most UNIX-based operating systems. It is a relatively small program consisting of about 8000 lines of code, which makes extensive use of hash tables to look up strings. In a nutshell, *m4* operates by reading in the input text as a sequence of tokens. It then looks up each token in a hash table. If the token is found, it replaces it with the value in the hash table. Otherwise, it outputs the token and continues with the next one. That makes it a good candidate to benefit from the proposed memory model, as a significant part of *m4*'s computation consists of hash table lookups which trigger forking due to multiple resolution.

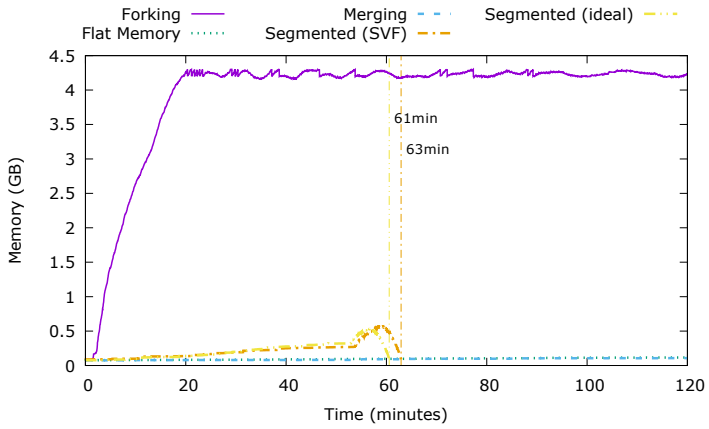
To run *m4* using symbolic execution, we need to make its input symbolic. To reach deeper parts of the code, where *m4* operates as described in the paragraph above, we run *m4* on a partially symbolic input. The outline of the input we used is shown in Figure 3.4. We



(a) DFS



(b) BFS



(c) Default

Figure 3.5: Runtime and memory consumption of the different memory models for *GNU m4* across different search strategies.

define several concrete one-character macros, using the `define` keyword. Then we expand two macros, with the name of the macro being symbolic, which results in a symbolic lookup in the hash table. Finally, we expand two more macros inside `ifelse` constructs. The example is set in a way that illustrates the multiple dereference problem, but we believe it is quite representative of how *m4* is used.

We ran KLEE on *m4* for two hours under DFS, BFS and KLEE's default strategy, with forking, merging, flat and segmented memory models. The segmented memory model is further broken down into two runs. The first one is using points-to information computed from SVF. The second one uses ideal points-to information, which was obtained with a pre-run as described in Section 3.3.1.

Figure 3.5 shows, for each search strategy, KLEE's termination time and memory usage for each of the five memory models. The forking and flat memory models do not terminate before the 2-hour timeout. The segmented memory model using SVF terminates in around one hour, with the ideal version a few minutes earlier. In contrast to the segmented memory model, which is only mildly influenced by the search strategy, the performance of the merging model heavily depends on the search strategy: for DFS, it terminates in only 25 minutes, for BFS in 88 minutes, while with KLEE's default strategy it times out after 2 hours.

Under DFS, the memory consumption is unsurprisingly low, as only a small number of states are kept in memory at one time (note that the y-axis has a different scale for DFS compared to BFS and the default KLEE strategy). The SVF segmented memory has slightly higher overall memory usage due to the fixed memory cost of keeping all points-to information. For BFS and the default search strategy, all memory models have low memory usage, except for the forking model, which quickly reaches the 4GB memory limit.

3.3.3 *GNU make*

GNU make [77] is a popular build system, used extensively in the open source community. It is a larger program, consisting of about 35,000 lines of code. It uses significantly more memory during runtime than *m4*. To make execution easier for KLEE, we reduced the sizes of several caches in *make*.

```
1 a := word1
2 b := word2
3 d := $?
4 e := $?
```

Figure 3.6: Symbolic input to *GNU make*, where ? denotes a symbolic character.

Our evaluation focused on the variable expansion capabilities of *make*. Similarly to *m4*, we made the makefiles only partially symbolic. Figure 3.6 illustrates the makefile we used as symbolic input in our experiments.

Figure 3.7 shows the memory consumption of the different memory models. This benchmark was the only one where none of the runs terminated before reaching the timeout. Unsurprisingly, the forking model's memory consumption grows the quickest. The two segmented memory models behave similarly, with the SVF version consuming a constant amount of additional memory due to holding the whole points-to set.

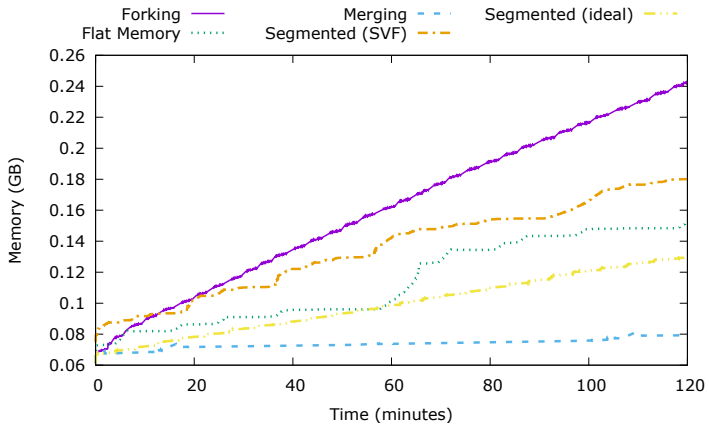
The merging model appears to have the best memory consumption, but this is only because it executes two times fewer instructions than the next slowest approach (SVF segmented memory). When compared to *m4*, the *make* runs are slower (execute fewer instructions per second), therefore the differences between the memory models are also smaller.

Since this benchmark is not terminating, we report the coverage. For DFS and BFS, the coverage was the same across all memory memory models at 21%. For the default heuristic, the forking and ideal segmented memory models had the highest coverage at 21.49%, followed by SVF segmented memory model at 18.75% and the flat memory model at 18.27%. Merging only achieved 15.44% coverage.

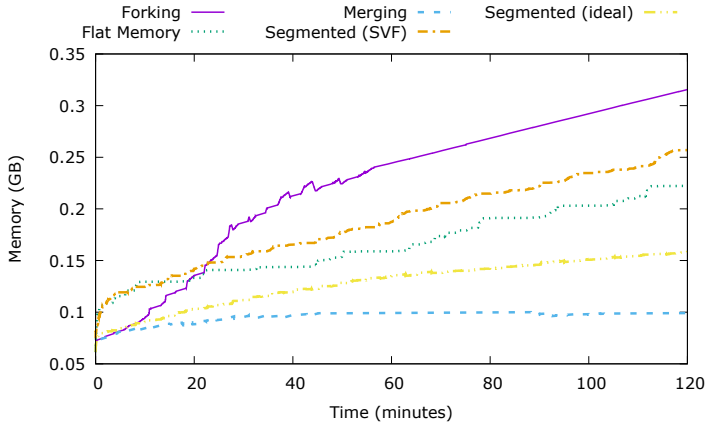
3.3.4 SQLite

SQLite3 [106] is a SQL database engine library written in C, which claims to be the most used database in the world. It has a big codebase with over 200,000 lines of code. We focused our evaluation on a part of *SQLite3* that triggers multiple resolutions in symbolic execution.

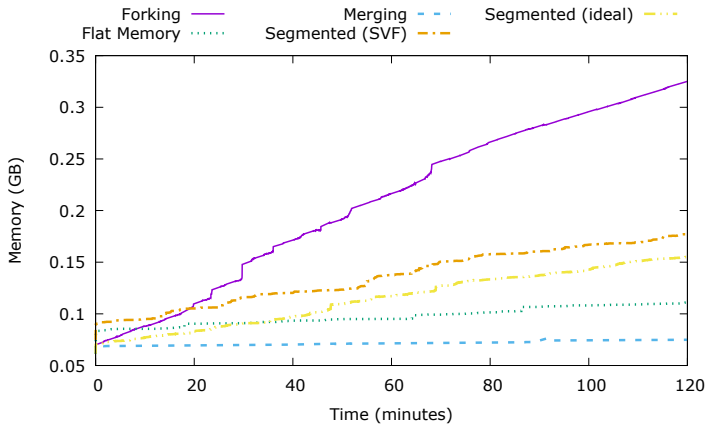
In particular, *SQLite3* uses a hash table to keep track of all triggers associated with a



(a) DFS

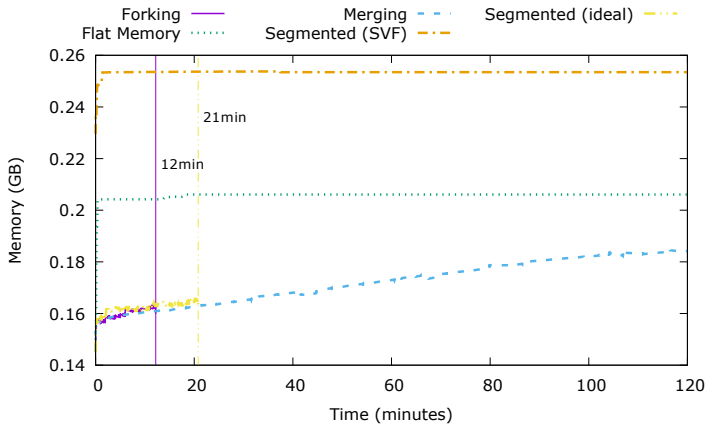


(b) BFS

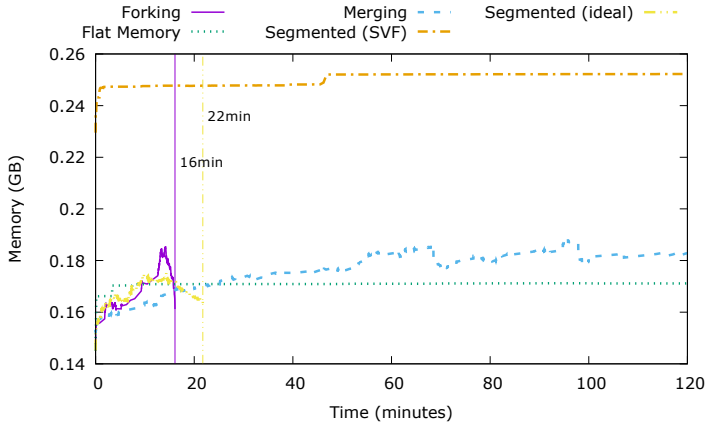


(c) Default

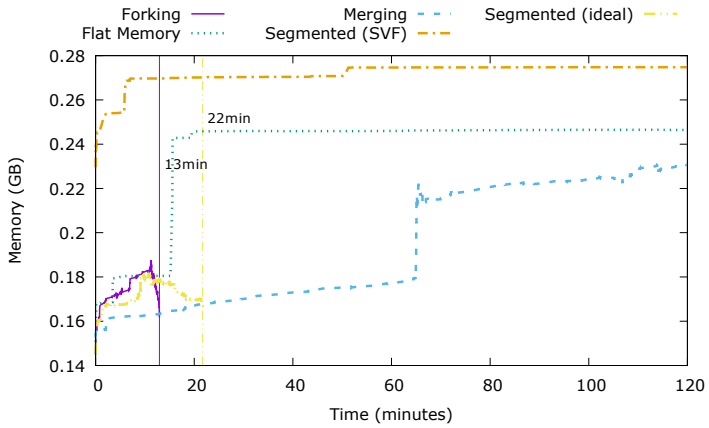
Figure 3.7: Memory consumption of the different memory models for *GNU make* across different search strategies.



(a) DFS



(b) BFS



(c) Default

Figure 3.8: Runtime and memory consumption of the different memory models for *SQLite3* across different search strategies.

table. We create several triggers with concrete names and then try to create a trigger with symbolic name, which should involve a lookup of a symbolic key in the hash table of triggers.

We use an in-memory database, create a table with two `int` columns and add 15 triggers to this table. We need to create more than 10 triggers as the *SQLite3* hash table is optimized to be just a linked-list when there are fewer than 10 elements. Choosing a higher amount of triggers would tilt the results against the forking model, but we believe 15 triggers is a good trade-off between showcasing the multiple dereference issue and a realistic use of the library. Finally, we create two more triggers whose names are symbolic.

Figure 3.8 shows the results. The forking approach terminates in 12-16 minutes, the ideal segmented memory model terminates in 21-22 minutes, while the SVF segmented memory, flat memory and merging models do not terminate within the 2-hour time budget.

There are no significant differences in memory consumption across search strategies. SVF segmented memory uses a constant additional amount of memory since it needs to hold the points-to sets in memory. The flat memory model uses more memory due to holding expressions over large arrays in memory.

This case study raises two interesting points. First, the precision of the points-to analysis significantly impacts the performance of the segmented memory model. As can be seen in Table 3.1, the difference in the size of the computed segments between ideal points-to analysis and SVF is huge. This is has an immediate impact on the performance of the segmented memory model, shown in Figure 3.8. Therefore, improving the precision of the points-to analysis is a promising future avenue for improving the segmented memory model.

Second, the segmented memory model performs worse than forking when there is little or no forking in the program. This is because grouping memory objects together increases the size of solver arrays, which makes constraints harder. This is a price the segmented memory approach has to pay regardless of whether the program causes multiple resolutions. In this case, the constraints in the forking model can be solved relatively fast, so forking can brute force its way through multiple resolutions. We note that increasing the number of triggers in the benchmarks would likely tilt results against the forking model, but our choice of 15 triggers was so that at least some runs finish within the 2-hour time budget.

3.3.5 Apache Portable Runtime

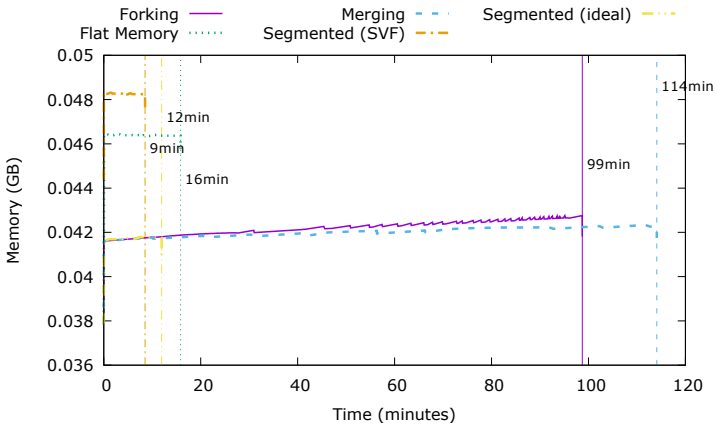
Apache Portable Runtime (*APR*) [6] is a C library that provides cross-platform functionality for memory allocation, file operations, containers, networking and threads, among others. It was initially used by Apache HTTP Server, but its use now extends to projects such as LibreOffice and Apache Subversion.

We focused our evaluation on *APR*'s hash table API. At a high level, we add 15 elements to a hash table and then do two symbolic lookups in this table. The number of keys and lookups was chosen to be the same as for *SQLite3*. The skeleton code is shown in Figure 3.10. We used the default hashing algorithm for the *int* keys, which is the popular "times 33" algorithm, used by Perl for example.

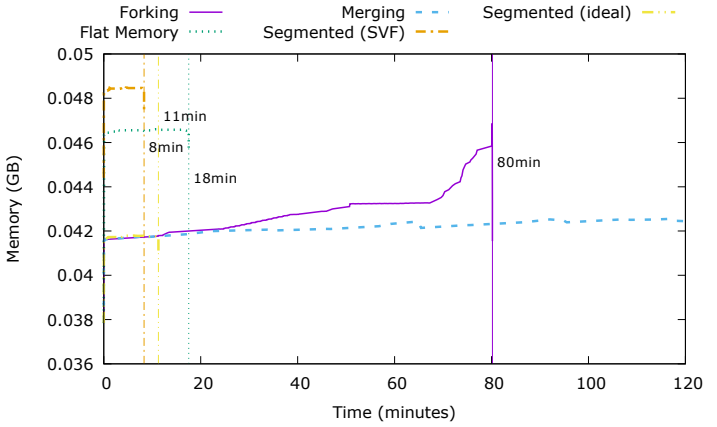
Figure 3.9 shows the results. All runs terminate, except for merging under BFS and the default search strategy. Merging is also the slowest under DFS, taking 114 minutes to terminate. Forking terminates in 80-99 minutes, depending on the strategy. Flat memory performs well on this benchmark, terminating in 16-21 minutes. The segmented memory models (SVF) perform the best, terminating in 8-13 minutes. Memory consumption is small overall.

The most interesting observation about this benchmark is that the segmented memory model with ideal points-to analysis performs slightly worse than the one with SVF analysis. This is due to an interesting interaction with the solver. To understand why, we need to describe the relevant memory objects involved. These are a large 8.2KB memory object and several small objects, totalling about 120 bytes in size. The ideal points-to analysis groups the small objects into a single segment and puts the large 8.2KB object into a separate segment. The SVF analysis bundles both the 8.2KB object and the small objects into a single segment. During execution, there are queries involving arrays associated with both the 8.2KB object and the small objects. The ideal segmented memory model generates constraints with two arrays, one with 120 bytes and one with 8.2KB. The SVF segmented memory model generates constraints with only a single array. This gives better solver query caching for SVF segmented memory model, which results in a quicker runtime.

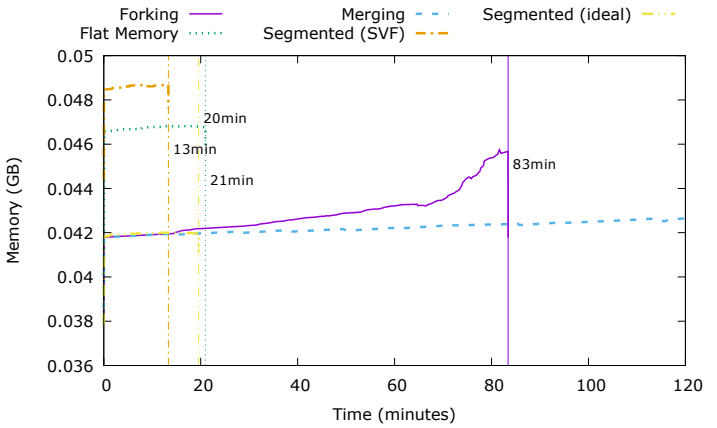
However, we note that this was the only benchmark where such a difference is observed,



(a) DFS



(b) BFS



(c) Default

Figure 3.9: Runtime and memory consumption of the different memory models for *APR* across different search strategies.

```
 1 apr_hash_t *ht = apr_hash_make();
 2 for (int i = 0; i < 15; i++) {
 3     int *key = malloc(sizeof int);
 4     *key = i;
 5     apr_hash_set(ht, key, sizeof(int), "A␣value");
 6 }
 7
 8 int i, j = symbolic();
 9 apr_hash_get(ht, &i, sizeof(int));
10 apr_hash_get(ht, &j, sizeof(int));
```

Figure 3.10: Apache Portable Runtime baseline benchmark.

and a more precise points-to analysis is otherwise associated with better runtime performance. However, this case study suggest that there is future work in improving caching in symbolic execution, which would have implications beyond this memory model.

3.4 Discussion

Our experience shows that our segmented memory model can effectively deal with multiple resolutions that occur in the context of complex data structures such as hash tables. However, the model is only useful to the extent that the code triggers such symbolic dereferences.

To get an idea of how our model performs when there are no multiple resolutions, we performed an experiment on *GNU Coreutils* version 8.29 [40]. We first ran all 105 utilities for 60 minutes with KLEE under the (default) forking model using depth-first search strategy and recorded the number of instructions KLEE executed for each utility. Then we ran each utility again up to the number of recorded instructions, with both the forking model (as a sanity check) and our segmented memory model, using a larger timeout of 80 minutes. A total of 18 utilities timed out after 80 minutes with the segmented memory model. For the remaining utilities, the segmented memory model was between 70% slower and 20% faster, but on average 4% slower. *GNU Coreutils* are an unfavorable case for our memory model, because the lack of multiple resolutions means that the model incurs a cost without any benefits.

While for most utilities, the impact is not significant, for 18 utilities it is. The cost is due to the grouping of arrays into memory segments, which can significantly increase the

difficulty of the constraints sent to the solver. Therefore, we believe that our approach should be enabled on demand: one would first run the forking approach, and only if that run reports several multiple resolution cases with large branching factors, switch to the segmented memory approach. In our experience, anything with more than five occurrences of multiple resolution with a branching factor of more than 10 should be sufficient for our proposed model to become beneficial.

Furthermore, the approach is particularly useful when one is interested in reasoning about properties requiring all-value analysis for the paths explored. When forking is used, such reasoning is not possible, as individual control-flow paths are split. If only high coverage is of interest, the single object or forking models might sometimes be more appropriate. Our approach also provides a convenient way to trade off forking and all-value reasoning, by adjusting the threshold for the size of memory segments. We also note that the flat memory and merging approaches can also perform all-value analysis.

We found the merging approach to be a competitive alternative to segmented memory. However, in our experiments the merging approach performed worse overall (e.g. unlike the segmented memory approach, merging timed out for all three *SQLite3* experiments and on two out of the three *APR* experiments) and was more sensitive to the search heuristic used. However, in some cases it performed better than the segmented approach (e.g. *m4* with DFS). As a threat to validity, as mentioned before, we note that our implementation of merging might not be optimal, but we believe this threat to be small (see §3.2).

Our comparison between the two versions of the segmented memory model — one using SVF and the other an approximation of the ideal points-to sets — shows that the precision of the points-to analysis directly influences the performance of our approach. However, we found an interesting case where better precision slightly decreased performance, suggesting that in some cases merging arrays could be beneficial due to caching.

3.5 Related work

CUTE [99] introduced a simple memory model, which only handles equalities and inequalities for symbolic pointers. As discussed in Section 3.1.1, EXE [18] and CREST [28] implement the single-memory model, FuzzBALL [80] a generalisation of it, KLEE [16] the forking model, and an extension of SAGE [34] and Angr [102] the merging model.

A recent idea paper [35] proposes a model that associates symbolic addresses with values, along with a time stamp and a condition. The symbolic memory is then represented as a list of these associations. When a read occurs, the most recent value matching the address and the condition is returned. ESBMC [27] uses a similar technique of chaining if-then-else expressions to model pointers that can point to multiple objects, in the context of model checking.

MAYHEM [21] creates a new memory object on each read operation, which is a subset of the whole memory that the read operation can alias. This approach is at a high-level similar to ours, but there are important differences. In particular, the approach does not handle writes via symbolic pointers that may refer to multiple objects — instead, these pointers are concretised as in the single-object model. Furthermore, the approach still involves solver queries, as in the forking model, to determine the objects to which the pointer may refer. Unfortunately, a direct comparison with Mayhem is not possible, as the code is closed-source, and the memory model is complex enough to make a reimplementation difficult.

David et al. [31] summarise the concretisation approaches, such as the ones employed by MAYHEM and EXE, by proposing a framework for specifying concretization-symbolisation policies. While our approach strives to avoid the need for concretisation, it still uses concrete addresses to identify memory objects.

Trtík and Strejček [118] present a fully symbolic *segment-offset-plane* memory model. They split memory operations involving different types (such as *int*s or *float*s) into different planes, which resemble memory segments, but their solution may group together memory objects which would have been separated in our model.

Research on lazy initialisation for symbolic execution of Java code explores different

ways of initialising symbolic memory object references and thus exploring different memory layouts [63, 93, 13]. By contrast, our work improves symbolic execution given a particular memory layout, notably it does not create new objects. We believe our work is complementary, and necessary to efficiently implement this work in symbolic executors for lower-level languages such as KLEE.

The idea of partitioning memory into segments is not new. Lattner and Adve [65] used points-to analysis to partition the heap as in our work. However, they only considered uses in compiler optimisation, whereas the novelty of our work stems from employing it to improve symbolic execution.

Bouillaguet et al. [12] apply a similar idea of partitioning memory based on points-to analysis to deductive verification. They require the points-to analysis to be sound and context-sensitive, whereas our approach is more tolerant to errors and imprecision in the analysis. Their evaluation does not consider real programs and only focuses on verifying a small sort function.

3.6 Conclusion

We presented a novel segmented memory model for symbolic execution, which uses pointer alias analysis to group objects into memory segments. Our segmented memory model can significantly reduce forking due to symbolic dereferences, sometimes even completely eliminating the need for forking. We evaluated our approach on *GNU m4*, *GNU make*, *SQLite3* and Apache Portable Runtime, highlighting its advantages in terms of performance and memory consumption, as well as its inherent limitations.

While working on this project, we realised that KLEE does not achieve good coverage on these harder benchmarks. Furthermore, it seemed very hard to guide the symbolic exploration to interesting parts of the program, so we had to resort to introducing partially symbolic input. Unsatisfied with the situation, we introduce pending constraints in the next chapter, which enables KLEE to better explore benchmarks at least as hard as the ones used in this chapter.

Chapter 4

Pending Constraints

In this chapter, we propose a novel mechanism for symbolic execution that aggressively explores paths whose feasibility is known via caching or seeding. Our approach tackles both scalability challenges of symbolic execution presented in Section 1.2.1. On the one hand, it enables more efficient use of solved constraints, thus reducing the burden on the solver. And on the other hand, it provides a meta-search heuristic that gives a way to guide the exploration towards interesting parts of the program.

The core of our idea revolves around *inverting* the forking process as presented in Section 1.2. Instead of doing an expensive feasibility check first and then forking the execution, we fork the execution first. The branch condition is then added as a *pending constraint*, which means its feasibility has not been checked yet. We refer to states (or paths) with pending path constraints as *pending states*.

The responsibility for feasibility checking of pending path constraints is passed to the search heuristic. This gives the search heuristic the capability to decide when and for which states it wants to pay the price of constraint solving. For example, it could solve pending states immediately, thus restoring the original algorithm from Section 1.2, or could take into account the (estimated) cost of constraint solving in its decisions.

In our approach, we take advantage of an important characteristic of symbolic execution runs: the feasibility of some paths/states can be quickly determined without using a constraint

solver. There are two common cases. First, modern symbolic execution systems like KLEE make intensive use of caching and many queries can be solved without involving the constraints solver [16, 119, 7]. Second, symbolic execution is often bootstrapped with a set of seeds from which to start exploration: these can come from regression test suites [78, 79] or greybox fuzzers in hybrid greybox/whitebox fuzzing systems [110, 23, 85]. By aggressively following paths for which feasibility can be quickly determined without using a constraint solver, our approach can minimise the constraint solving costs as well as provide an effective exploration of the program search space.

Note that a seed can be thought of as a solver solution to constraints of a path in the program. We use them similarly to cached solver solutions by pre-populating the cache with the seeds. We discuss this in further detail in section 4.1.2.2.

The rest of the chapter is organised as follows. In Section 4.1 we present in detail the design of our technique. Then in Section 4.2 we give further implementation-specific details in the context of building our technique in KLEE. We finally evaluate our approach in Section 4.3, discuss related work in Section 4.4 and conclude in Section 4.5.

4.1 Approach

For clarity and contrast, Algorithm 9 presents, in simplified form, the forking and search heuristics components of the standard symbolic execution algorithm from Section 1.2. The symbolic executor is exploring program paths encoded as a set of *states* (line 1). As discussed before, a state keeps track of all the information necessary to resume execution of the associated path (program counter, stack contents, etc.) and particularly its path condition *pc*. When a symbolic state s encounters a symbolic branch *condition*, the Fork function is called (line 3).

Fork first checks if the *condition* can be both *true* and *false* under the current path condition (line 4). If so, the state is duplicated into a *falseState* (line 5), which will be the state representing the execution of the false branch. The path conditions of the two states are then updated accordingly (lines 6 and 7) and the new state is added to the set of states

Algorithm 9 Standard symbolic execution.

```
 1: Set states
 2:
 3: function Fork(State s, SymExpr condition)
 4:     if IsSat(s.pc ∧ condition) ∧ IsSat(s.pc ∧ ¬condition) then
 5:         falseState ← s
 6:         falseState.pc ← s.pc ∧ ¬condition
 7:         s.pc ← s.pc ∧ condition
 8:         SearcherAdd(s, falseState)
 9:     end if
10: end function
11:
12: function SearcherAdd(State s1, State s2)
13:     states ← states ∪ {s1, s2}
14: end function
15:
16: function SearcherSelect()
17:     return SearchHeuristic(states)
18: end function
```

of the search heuristic by calling SearcherAdd (line 8). If the *condition* cannot be both true and false, the path condition is not updated and s continues to execute the only feasible side of the branch (the updates to the program counter are omitted for ease of exposition).

After each instruction, the symbolic executor calls SearcherSelect (lines 16–18) to select the state to be executed next. In the case of standard symbolic execution, SearcherSelect simply forwards calls to the SearchHeuristic. We mentioned some examples of search heurstics in Section 1.2.2. Below, we remind the reader of the three search heuristics which we use in this chapter:

Depth-first search is a standard graph traversal algorithm that explores states as deep as possible before backtracking.

Random path search, introduced in the original KLEE paper [16], works by randomly selecting a path through the execution tree of explored states. The algorithm starts at the root of the tree and with 50% probability follows the left-hand subtree and 50% probability the right-hand one. The process repeats until a leaf state is reached, which is selected for further exploration. By design, this search heuristic favours states closer to the root of the execution tree.

Depth-biased search is a form of non-uniform random search provided by KLEE. It works

by randomly selecting states weighted by their depth — the higher the depth of a state, the more likely for it to be selected. By design, this search heuristic favours states deeper in the execution tree.

Algorithm 10 Symbolic execution with pending constraints.

```
1: Set feasibleStates, pendingStates
2:
3: function Fork(State s, SymExpr condition)
4:     feasibleStates ← feasibleStates \ {s}
5:     s.pendingCondition ← condition
6:     falseState ← s
7:     falseState.pendingCondition ← ¬condition
8:     SearcherAdd(s, falseState)
9: end function
10:
11: function SearcherAdd(State s1, State s2)
12:     foreach State s ∈ {s1, s2} do
13:         if FastIsSat(s, s.pc ∧ s.pendingCondition) then
14:             s.pc ← s.pc ∧ s.pendingCondition
15:             s.pendingCondition ← ∅
16:             feasibleStates ← feasibleStates ∪ {s}
17:         else
18:             pendingStates ← pendingStates ∪ {s}
19:         end if
20:     end foreach
21: end function
22:
23: function SearcherSelect()
24:     while feasibleStates = ∅ do
25:         s ← SearchHeuristic(pendingStates)
26:         if IsSat(s.pc ∧ s.pendingCondition) then
27:             s.pc ← s.pc ∧ s.pendingCondition
28:             s.pendingCondition ← ∅
29:             feasibleStates ← feasibleStates ∪ {s}
30:         end if
31:         pendingStates ← pendingStates \ {s}
32:     end while
33:     return SearchHeuristic(feasibleStates)
34: end function
```

4.1.1 Pending Constraints Algorithm

Note that the calls to IsSat in standard symbolic execution are potentially very expensive and thus are the optimisation target of our proposed approach.

Algorithm 10 shows the same three functions in the pending constraints version of

symbolic execution. In this version, we maintain two disjoint sets of states: *feasibleStates*, which stores regular states which we know are feasible; and *pendingStates*, which stores pending states for which feasibility is still unknown (line 1).

In our version of FORK, the ISSAT feasibility checks are skipped and execution is forked unconditionally. First, we remove the current state s from the list of *feasibleStates* (line 4), and assign the *condition* to a special pending condition field associated with s (line 5). Second, we duplicate s into a new state *falseState* (line 6) and assign the negation of *condition* to its pending condition field (line 7). When a state becomes pending, as s and *falseState* here, it means that it should not continue execution until its pending condition is checked for feasibility. We call the process of checking the pending condition and adding it to the path condition as *reviving* the state.

Finally, we call SEARCHERADD (line 8) to let the searcher decide what to do with the newly created pending states. For each of the two states, SEARCHERADD checks whether the pending condition is feasible using a fast satisfiability checker (line 13). If so, the pending condition is added to the path condition (line 14), the *pendingCondition* field is reset (line 15) and the state is added to the set of *feasibleStates* (line 16). If the fast satisfiability check is unsuccessful, the state is added to the set of *pendingStates* (line 18). The fast satisfiability solver is discussed in Section 4.1.2.

The SEARCHERSELECT function operates as in standard symbolic execution as long as there are feasible states available (line 33). However, when all the feasible states have been exhausted, it keeps picking a pending state (line 25) until it finds one that can be revived successfully. This is done by asking the solver whether the pending condition is feasible (line 26) and if so, by adding the pending condition to the state's path condition (line 27), clearing the *pendingCondition* field (line 28), and adding the state to the set of *feasibleStates* (line 29).

4.1.2 Fast Satisfiability Checking

As discussed in the introduction, the fast satisfiability checker relies on the fact that we often have existing assignments (solutions) to symbolic constraints. There are two common

scenarios that occur in practice: first, modern symbolic execution engines use a smart form of caching to speed up constraint solving (§4.1.2.1) and second, symbolic exploration often starts with a set of seeds (§4.1.2.2).

Algorithm 11 Fast satisfiability checking.

```
1: function FastIsSat(State s, SymExpr condition)
2:     assignments ← GetAssignmentsForState(s)
3:     foreach Assignment a ∈ assignments do
4:         if Substitute(a, condition) = true then
5:             return true
6:         end if
7:     end foreach
8:     return false
9: end function
```

Algorithm 11 shows the fast satisfiability checking algorithm. More formally, an *assignment* is a mapping from symbolic variables to concrete values, e.g. $\{x \leftarrow 2, y \leftarrow 3\}$. The Substitute function takes a symbolic condition and an assignment and evaluates the condition under the assignment (line 4). That is, it substitutes all the symbolic variables in the condition with the values specified in the assignment. If all the symbolic variables in the expression are mapped by the assignment, Substitute will return a concrete value (either *true* or *false*). Otherwise it will return *false*.

FastIsSat first gets all the assignments associated with the given state using the GetAssignmentsForState function (line 2); we will discuss how this function works below. The condition is then evaluated on every assignment returned by GetAssignmentsForState and if the evaluation results in *true*, FastIsSat returns successfully (line 5). If none of the returned assignments satisfy the condition, then FastIsSat returns unsuccessfully (line 8).

4.1.2.1 Caching

When issuing satisfiability queries to the solver, modern symbolic execution engines typically also ask for a satisfying assignment when the query is satisfiable. This is because caching these assignments can be highly effective [16, 119, 7]. In this work, we similarly use cached assignments to implement our fast satisfiability checks. Substituting all assignments that were returned by the core solver at any previous point in the execution can be expensive

and here we want to ensure the check is fast. So instead, we use the same mechanism used by assignment caching in KLEE, and have GetAssignmentsForState (line 2) return only the assignments associated with a subset of the path condition of state s.

The idea behind the pending constraints approach is to use existing assignments as long as it is possible, and only ask for expensive new solutions when absolutely necessary. As a result, the search goes deep into the execution tree, because it explores the solutions it has to completion. Therefore, combining this strategy with a search heuristic that goes wide when picking a pending state can be really effective. Note that the search heuristics that often perform best, such as random path search, tend to behave like this [16]. Another way of understanding our approach is as a meta-searcher that enhances a given search heuristic, enabling it to explore deeper parts of the program. Intuitively, this is achieved by picking a path and sticking to it, while normally the same searcher would keep changing between paths. Importantly, the path it sticks to issues no additional queries to the SMT solver and therefore completes quickly.

Note that this technique only re-distributes the cost of solving constraints, so it is unlikely to help if the interesting parts of the program are locked behind hard to solve constraints. In that case it needs some help to guide pending constraints towards them. In section 4.1.2.2 we propose using seeding for this purpose.

To make things more concrete, consider the program in Figure 4.1 with `DFS_FRIENDLY` undefined. It has two symbolic variables: a boolean `isSpace` and a string `str`. It first branches on `isSpace` on line 14 and writes space as the first character of `str` on the `then` branch or a zero on the `else` branch. The two loops between lines 21–23 then introduce a large number of symbolic branches. After these loops, it computes the fifteenth Fibonacci number using an inefficient recursion. This represents some large concrete workload. Finally, we assert `isSpace` is `false`. Note that reaching this assertion failure only depends on the first branch and is completely independent of the large number of paths spawned in the loops.

Vanilla KLEE executes around 4 million instructions, taking around 10 seconds to reach the assertion failure using the random path search strategy. By contrast, the pending

```
 1 unsigned fib(unsigned n) {
 2   if (n == 0) return 0;
 3   if (n == 1) return 1;
 4   return fib(n - 1) + fib(n - 2);
 5 }
 6
 7 int main() {
 8   bool isSpace;
 9   klee_make_symbolic(&isSpace, sizeof(isSpace), "isSpace");
10   char str[6];
11   klee_make_symbolic(str, sizeof(str), "str");
12
13 #ifndef DFS_FRIENDLY
14   if (isSpace) str[0] = ' ';
15   else str[0] = '\0';
16 #else
17   if (isSpace == 0) str[0] = '\0';
18   else str[0] = ' ';
19 #endif
20
21   for (int i = 1; i < 6; i++)
22     for (char j = 0; j < str[i]; j++)
23         str[i-1]++;
24
25   fib(15);
26   assert(!isSpace);
27   return 0;
28 }
```

Figure 4.1: An example program where pending constraints find the failing assertion faster.

constraints approach executes only around 67k instructions, taking less than a second. The reason is that vanilla KLEE needs to compute a large number of solutions to the symbolic branches in the loops, which are not needed to find the assertion failure. Whereas the pending constraints approach only needs one solver solution for `str`. Because random path search is the underlying search strategy, it is also more likely to pick the pending states produced by the branch on line 14 as it is closer to the root of the process tree.

Note that DFS reaches our one-hour timeout in this case. However, it performs similarly to the pending states if the order of branches on line 14 is swapped (by defining `DFS_FRIENDLY`). With `DFS_FRIENDLY` defined, both pending states and upstream KLEE find the assertion in about 33k instructions. This shows that while DFS can be as efficient, it is highly dependent on the order of branches, unlike the pending constraints approach.

4.1.2.2 Seeding

A seed is a concrete input to a program which is given to symbolic execution to bootstrap exploration. A seed can make symbolic execution of the program easier for two reasons. First, it is used to guide symbolic execution toward parts of the program that are executed by real inputs. Second, it provides a solution for complex path constraints, therefore removing the need for expensive constraint solving. Seeding has been shown to be effective when symbolic execution is combined with regression test suites [78, 79, 87] as well as when it is integrated with a greybox fuzzer [110, 23, 85].

Our pending constraints approach can be easily extended to support seeding. In FastIsSat, we modify the GetAssignmentsForState function to return the seeds as assignments (line 2). As a result, only the states that satisfy the seeds are followed first, and there are no calls to the SMT solver until all the paths followed by the seeds are explored. In this use case, pending states represent the possible divergences from the seeds, from where we can pick up exploration once the seeding is completed.

When seeds are available, random path search is unlikely to be the most effective strategy. Once the paths followed by the seeds are completed, random path search would pick a state very close to the start of the program, as the probability of picking a state halves at every symbolic branch. This means that the exploration will behave similarly as to when no seeds are available.

On the other hand, depth-biased search would most likely pick a state toward the end of the path executed by the seeds, meaning it will start to explore paths that are very hard to reach from the start of the application. In other words, it benefits from the seed and can start exploring code out of reach of normal symbolic execution.

However, there are two limitations to this search heuristic. First, by design, this strategy misses code that is easy to reach from the start of the program. Therefore, it is best combined with a more wide-reaching strategy such as random path search. Second, this heuristic is more likely to select pending states that are infeasible, as the path conditions for those states have more constraints and thus the pending condition is more likely to be infeasible in that state.

4.2 Implementation

We implemented our approach in KLEE [16], our prototype is based on KLEE commit `0fd707b` and is configured to use LLVM 7 and STP 2.3.3. We discuss below some of the most important implementation aspects.

We make our prototype and associated artefact available at `https://srg.doc.ic.ac.uk/projects/pending-constraints` and `https://doi.org/10.6084/m9.figshare.12865973`.

4.2.1 Fast Satisfiability Solver

KLEE's constraint solving chain consists of a series of caches and other partial solvers, finally delegating the query to an external SMT solver. For our fast solver we simply used this solver chain without the final, potentially expensive, call to STP.

Seeding is easily implemented with KLEE's counterexample (assignment) cache, as we simply add the seed to the cache as an additional assignment at the start of execution. Note that this is different from how vanilla KLEE implements seeding. However, to make the comparison between vanilla KLEE and pending constraints fair, we also added the seed to the cache in vanilla KLEE.

4.2.2 Error Checks

During execution, KLEE performs various checks to find errors. Examples include checks for division by zero and out-of-bounds memory accesses. One option would be to treat these checks as we treat regular branches and create pending states. However, that would mean that errors would not be caught as soon as that code is covered, as in vanilla KLEE (and in fact they might never be caught if those pending states are killed later on due to memory pressure). Therefore, we treat these checks specially by always performing them instead of creating pending states.

However, if higher coverage is more important than bug finding, deferring these checks to pending states might make more sense. Therefore, we also run our benchmarks with these checks deferred in a version that we call pending constraints with *relaxed checks*. We explore

this version in Section 4.3.5 and observe significant coverage gains.

4.2.3 Branching without a Branch Instruction

In KLEE there are several cases where forking happens without a branch instruction. For instance, in the default configuration, a switch instruction is handled as a branch with multiple targets. This fits less neatly in our model therefore we simply configure KLEE to lower all switch instructions to a series of `if-else` statements.

As discussed in Chapter 3, when a symbolic pointer is encountered, KLEE scans the address space to find all memory objects to which it can point. If multiple objects are found, it forks execution for each object, adding appropriate constraints. In the default version of our approach, we do not create pending states in this case; instead we eagerly fork as necessary, as in vanilla KLEE. However, with relaxed checks, for each memory object to which the pointer can refer, the state gets forked into two pending states: one in-bounds of the object and the other one out-of-bounds. This second pending state encompasses all other resolutions of the symbolic pointer.

4.2.4 Releasing Memory Pressure

KLEE often hits the memory limit for large programs due to the vast size of the search space and generally broad search heuristics [15]. When exceeding the memory limit, KLEE terminates a number of randomly-chosen states to get back below the memory limit. We could follow the same approach with pending constraints, but this could delete both feasible and infeasible pending states. Feasible pending states potentially represent large parts of the search space and we should avoid their termination. By reviving pending states, we can select and terminate infeasible pending states. But this comes at a price, as reviving states requires expensive constraint solving. In our implementation, we decided to stop reviving states when the remaining number of pending states equals the number of states we still need to terminate to fall below the memory limit. At that point, we start to randomly choose states to terminate as in vanilla KLEE.

4.3 Evaluation

Our evaluation is structured as follows. We present our benchmarks in Section 4.3.1 and explain why we evaluated them on internal coverage in Section 4.3.2. We evaluate pending constraints on their ability to enhance symbolic execution in a non-seeding context in Section 4.3.3 and then in a seeding context in Section 4.3.4. We also evaluate the version with relaxed checks in Section 4.3.5. Finally, we discuss our approach in further detail in Section 4.3.6, via a case study on *SQLite3*.

We run all experiments in a Docker container on a cluster of 18 identical machines with Intel i7-4790 @ 3.60GHz CPU, 16GiB of RAM and Ubuntu 18.04 as operating system. We repeat our experiments three times and where possible plot the average and the minimum and maximum as an interval. In cases where we combine experiments, there are no intervals shown as we merge all the repetitions.

4.3.1 Benchmarks

Our aim is to demonstrate the capability of pending constraints to increase the exploration potential of symbolic execution. Therefore, we chose benchmarks that are challenging for symbolic execution.

SQLite3 [106] already introduced in Section 3.3.4. We used version 3.30.1 via its default shell interface with symbolic input and without custom drivers. However, we adjusted some compile-time flags, such as simplifying the allocation model, to make it easier for symbolic execution.

magick [57] is a command-line utility for performing various operations on images. We used version 7.0.8-68 in two configurations, converting (symbolic) jpeg images to png images and back, which makes use of *libjpeg* [71] and *libpng* [72].

tcpdump [112] is used for capturing and printing network packets. We used version 4.10.0 to print information from a symbolic capture file.

GNU m4 [44] is a macro processor introduced in Section 3.3.2. We used version 1.4.18 to process a symbolic file.

GNU make [45] is a widely-used build system we already discussed in Section 3.3.3. We used version 4.2 on a symbolic makefile.

oggenc [120] is a tool from the Ogg Vorbis audio compression format reference implementation. We used version 1.4.0 to convert a symbolic `.wav` file to the `.ogg` format.

GNU bc [41] is an arbitrary precision calculator. We used version 1.07 on symbolic standard input.

GNU datamash [43] is a tool for performing command-line calculations on tabular data. We used version 1.5 both with symbolic input and symbolic arguments.

Table 4.1 gives a more detailed overview of the benchmarks, showing their size, arguments we used and a brief description of the seeds. The program size is measured in terms of LLVM instructions as reported by KLEE. The third column shows the arguments we used to run the benchmarks. The arguments prefixed with `-sym` are replaced with symbolic data. For example, `-sym-arg 3` is replaced with a three-character symbolic argument, `-sym-stdin 20` is a 20-byte symbolic standard input stream, and `-sym-file 100` creates a file named `A` with 100 bytes of symbolic content.

The last two columns of Table 4.1 give an overview of the seeds we used. Where possible, we give the exact data used as in the case of *bc*. However, in the case of *magick* and similar utilities that would not be too informative, so we give instead a brief description of the seed. We took the smallest valid files from `https://github.com/mathiasbynens/small`. Some seeds were padded with zeroes as necessary to make the seed size match the number of symbolic bytes as specified by the arguments.

Figure 4.2 shows the percentage of time spent in the solver by vanilla KLEE on our benchmarks, on 2-hour runs, using different search strategies. As can be seen, most of our benchmarks are solver-bound, meaning they spent a very high proportion of time in the solver, for example *bc* and *make*. *datamash* is the only benchmark that is not solver-bound in any search strategy considered, meaning KLEE spends very little time constraint solving. *SQLite3* is only solver-bound with the DFS search strategy.

Table 4.1: Overview of (symbolic) arguments and used seeds for our benchmarks. The benchmark size is given as number of LLVM instructions in the bitcode file.

Benchmark	Size	Arguments	Seed set 1 ($s1$)	Seed set 2 ($s2$)
bc	34,311	`-sym-stdin 20`	`435 / 4 + 6 - 3421`	`x=6 sqrt(2 * x + 5)`
datamash	63,405	`-sym-arg 3 -sym-arg 1 -sym-arg 4 -sym-arg 1 -sym-stdin 20`	`sum 1 mean 3` on a 3×3 matrix	`md5 1 sha1 2` on 2×3 matrix
m4	93,169	`-G -H37 -sym-arg 2 -sym-arg 5 -sym-stdin 20`	`-G -DM=6` with `M is M also M is B`	`-G -DM=6` with `ifdef('M', ha, nah)`
$magick_{jpg}$	1,368,912	`jpeg:fd:0 png:fd:1 -sym-stdin 285`	smallest valid JPEG file	1×1 JPEG image created with *GIMP*
$magick_{png}$	1,368,912	`png:fd:0 jpg:fd:1 -sym-stdin 70`	smallest valid PNG file	1×1 PNG image created with *GIMP*
make	80,790	`-n -f A -sym-arg 2 -sym-arg 5 -sym-file 20`	`-n -Bfds` with `$(info $$helo ther)`	`-n -Bfds` with `a:=4 $(info $a)`
oggenc	87,142	`A -sym-file 100`	smallest valid WAV file	sine wave WAV file created by *SciPy*
SQLite3	283,020	`-sym-stdin 20`	`SELECT * FROM t;`	`CREATE TABLE t (i);`
tcpdump	353,196	`-r A -K -n -sym-file 100`	100 byte captured packet	another 100 byte captured packet

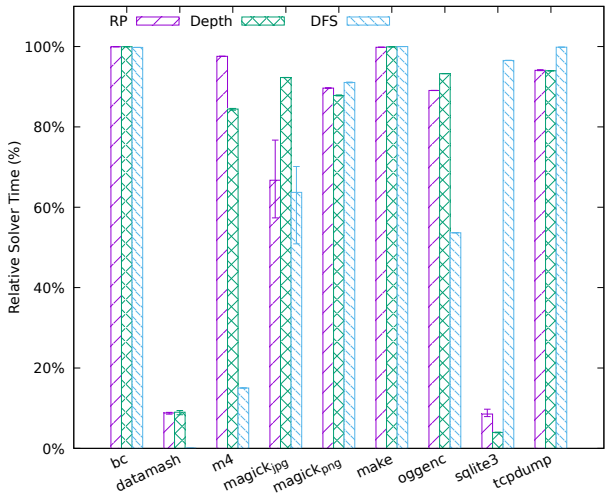


Figure 4.2: Relative time spent in the SMT solver (STP) by vanilla KLEE in our non-seeded experiments with the random path (RP), depth-biased (Depth) and DFS strategies.

4.3.2 Internal Coverage vs. *GCov* Coverage

When measuring the coverage achieved by vanilla KLEE and our pending constraints extension, we can use either the internal coverage reported by KLEE or the coverage reported by *GCov* [38] when running the test inputs generated by KLEE.

The internal coverage reported by KLEE includes the lines of code that KLEE executed symbolically, at the LLVM bitcode level. By contrast, the *GCov* coverage is the coverage reported by *GCov* at the source code level, when rerunning the test inputs generated by KLEE on a version of the program compiled with *GCov* instrumentation. Both approaches have advantages and disadvantages. On the one hand, *GCov* coverage has the advantage of being disconnected from KLEE and LLVM; often developers just want a tool like KLEE to generate a high-coverage test suite and assess this coverage independently, at the source code level. On the other hand, *GCov* coverage includes parts of the program that KLEE did not execute symbolically; these parts of code were not error-checked by KLEE, and thus bugs could be missed. It is important to note here that when KLEE performs an error check, it finds a bug if there are *any* values that can trigger it on that path. By contrast, a test case that covers the buggy path might not necessarily reveal the bug. Therefore, if KLEE is

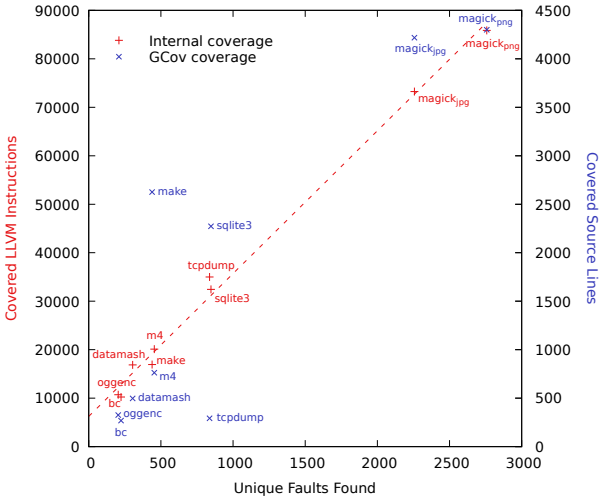
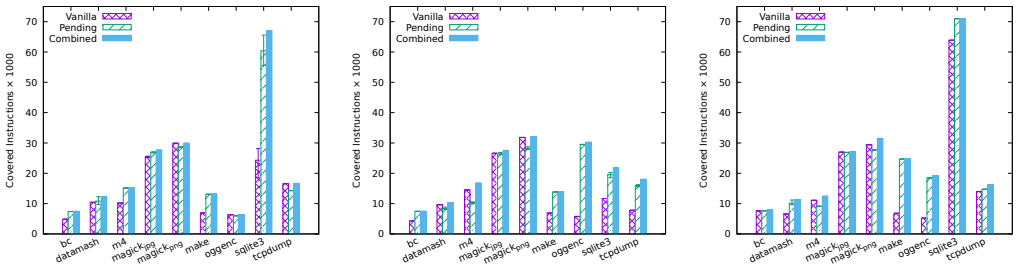


Figure 4.3: Dual-axis scatter plot of internal coverage (left y-axis) and *GCov* coverage (right y-axis) against the number of injected faults found.



(a) Random Path (b) Depth-Biased (c) DFS

Figure 4.4: Instructions covered by vanilla KLEE and pending constraints alongside their combination in two hours non-seeded runs.

used primarily to reveal bugs, internal coverage is more appropriate.

To illustrate this issue, we injected a division-by-zero error in every basic block of some of our benchmarks. These division-by-zero errors are triggered by a different value in each basic block. We then run vanilla KLEE for two hours on these fault-injected programs and measure both internal and *GCov* coverage.

Figure 4.3 shows a dual-axis scatter plot of both internal coverage on the left y-axis and *GCov* coverage on the right y-axis against the number of unique injected faults found.

As can be seen, internal coverage seems to highly correlate with unique faults found,

whereas *GCov* coverage shows no such correlation. We discuss this in more detail in a blog post [19].

In our work, we observed that pending constraints do not necessarily improve *GCov* coverage, but significantly improve internal coverage. Therefore, while accepting the former, we show in our experiments that it can improve the latter, and thus report internal coverage in our experiments.

4.3.3 Non-seeded Experiments

To evaluate pending constraints in a setting without seeds, we ran each of our benchmarks for two hours with the random path, DFS and depth-biased strategies. The results are shown in Figure 4.4. For some benchmarks, such as *bc* and *make*, pending constraints consistently cover more instructions for all search strategies. For others, such as *m4* and *tcpdump*, the relative performance is dependent on the search strategy.

If we look at the combined coverage of vanilla KLEE and pending constraints, we see that there is some complementarity to the approaches. For instance, while vanilla KLEE obtains better coverage for *m4* with the depth-biased search, the pending constraints reach code that is not covered by vanilla KLEE. Overall, pending constraints reached 35%, 24% and 34% more instructions across our benchmarks with random path, DFS and depth-biased search respectively. These results show that pending constraints can significantly increase the power of symbolic execution on some benchmarks by themselves and/or cover different parts of the code when compared to vanilla KLEE. Therefore they seem to be an effective tool for non-seeded exploration in symbolic execution.

Comparing the search strategies for pending constraints, we observe that for these experiments DFS performs best overall, covering 11% more instructions than random path, which in turns covers 14% more instructions than depth-biased search.

One advantage of the pending constraints approach is that it usually spends less time solving queries that turn out to be infeasible. Figure 4.5 shows the time spent constraint solving queries that turn out to be infeasible relative to total constraint solving time. We averaged across all search strategies for brevity and clarity. For most benchmarks, pending

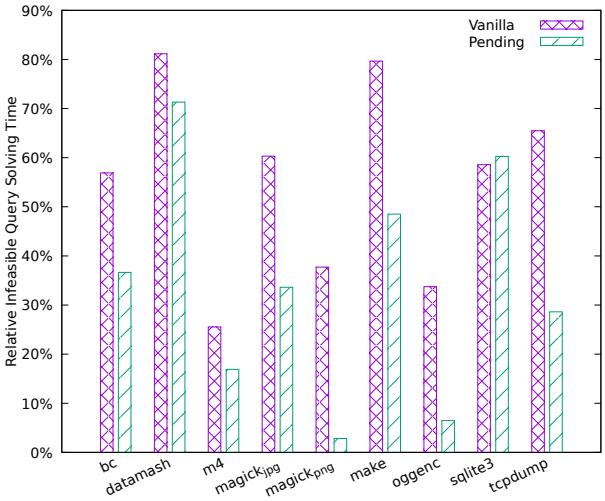


Figure 4.5: Relative time spent solving queries that were infeasible, averaged across all three search strategies.

constraints spend significantly less time constraint solving queries that are infeasible. The only exception is *SQLite3*, where the absolute time spent solving infeasible queries is still lower for pending constraints across all three search strategies.

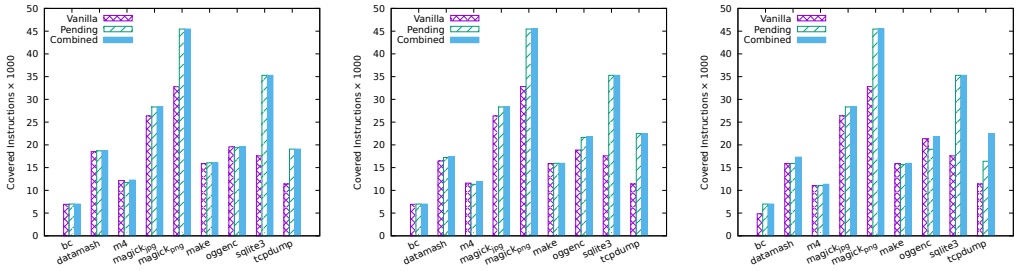
4.3.4 Seeded Experiments

To evaluate our approach in the context of seeding, we ran both vanilla KLEE and pending constraints for two hours with each seed from Table 4.1. Both versions are given the seeds — vanilla KLEE as assignment in the (counter-example) cache and additionally as concrete input.

Figure 4.6 shows the coverage achieved by vanilla KLEE and pending constraints, for each search strategy. Coverage results for the two seeds are merged in each configuration.

For some benchmarks, such as *magick*$_{png}$, *SQLite3* and *tcpdump*, pending constraints significantly outperform vanilla KLEE for all search strategies. For others, such as *datamash* and *make*, there does not seem to be a large difference. Finally, for *m4*, pending constraints perform slightly worse than vanilla KLEE. This is due to dereferences of symbolic pointers being very common in *m4*, which do not benefit from pending constraints.

From the combined coverage bars, we can observe there is very little complementarity



(a) Random Path (b) Depth-Biased (c) DFS

Figure 4.6: Covered instructions by vanilla KLEE and pending constraints alongside their combination in seeded runs on our benchmarks. Each seed set is run for two hours in each configuration and the coverage results for the two seeds are then merged.

between the two approaches, with the exception of *tcpdump* under DFS, where the combined coverage is significantly higher. In this case, both runs explore the seed, but pending constraints go deeper in the exploration. We discuss in more detail the advantages provided by pending constraints with seeding in the *SQLite3* case study of Section 4.3.6.

Overall, pending constraints cover 25% more instructions with random path search, 30% more with depth-biased search, and 23% more with DFS. In the pending constraints experiments, depth-biased search covered 2% more instructions overall when compared to random path search, which in turn covered 3.5% more than DFS.

Figure 4.7 shows the coverage achieved with only seed set 1 between our pending constraints approach and vanilla KLEE with the random path search strategy. For most utilities, we can draw similar conclusions as for the experiments of Figure 4.6 where the coverage for the two seed sets is merged. *oggenc* is an exception as it shows no coverage improvement on only seed set 1.

4.3.5 Pending Constraints with Relaxed Checks

Figure 4.7 also shows the coverage achieved with relaxed checks for critical instructions (§4.2.2). As can be seen for some tools like *bc* or *SQLite3*, relaxing these checks can lead to large increases in number of covered lines. In the case of *bc* the instruction coverage more than doubles. For most other utilities the increase is smaller, but not insignificant, with *oggenc*, $magick_{png}$ and *tcpdump* being the notable exceptions showing no or small decreases

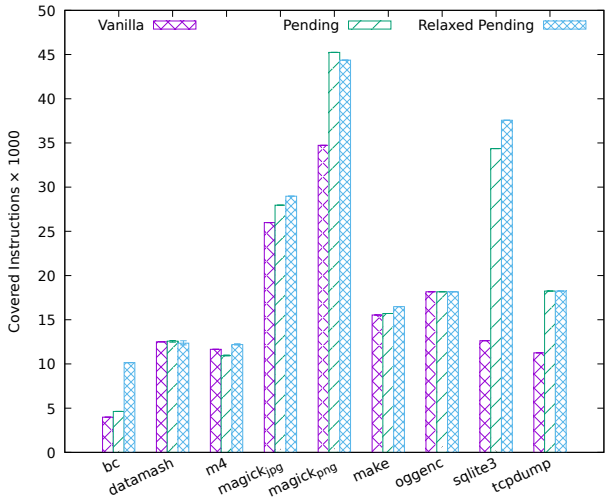


Figure 4.7: Covered instructions on seed set 1 of vanilla KLEE against pending constraints with both strong and relaxed checks with the Random Path search strategy.

in coverage.

These results show that pending constraints with relaxed checks can be used to effectively reach deeper into the program, but with the downside of errors being missed during exploration.

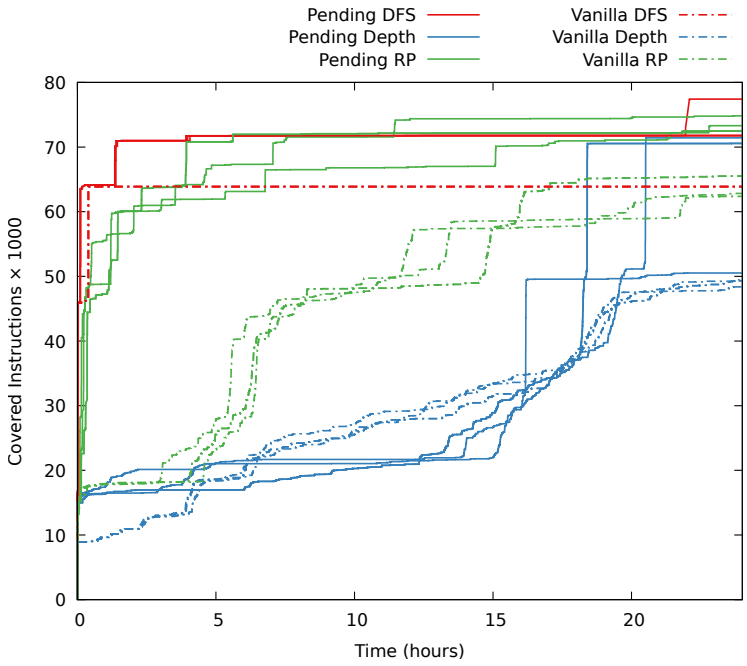
4.3.6 Case Study: *SQLite3*

To provide more insight into our approach, we now discuss one of our benchmarks, *SQLite3*, in more detail. In particular, we look at the evolution of coverage on long-running experiments and examine various pairwise combinations of configurations to uncover how different seeding approaches and search strategies complement each other.

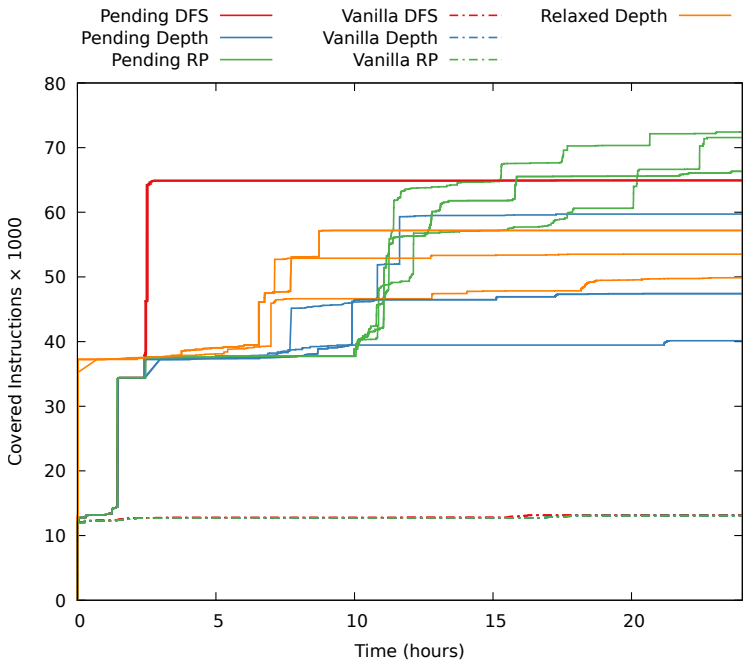
4.3.6.1 24-hour Non-seeded Runs

We first performed 24-hour experiments with *SQLite3* without seeding, with each of the three considered search strategies. Each experiment was repeated three times to account for the randomness of the depth-biased and random path strategies. The results are shown in Figure 4.8a.

DFS performs surprisingly well for this benchmark. However, with vanilla KLEE, it



(a) Non-Seeded



(b) Seeded

Figure 4.8: Coverage of *SQLite3* over a 24-hour run. The experiments were repeated three times, represented as lines of the same type. Both non-seeded and seeded runs are shown.

achieves no additional coverage after the first hour. By contrast, pending constraints with DFS continue to make progress and end up achieving the highest coverage overall.

We inspected the test inputs generated using DFS to understand why it performs so well. Vanilla KLEE with DFS appears to be lucky, as it manages to find two short keywords, such as `AS` and `BY`, which seem to be located close to the edge of the search space. These two keywords are not found by vanilla KLEE using the other two search strategies. Pending constraints with DFS does not find these short keywords, but instead finds longer ones such as `INSERT`, `FILTER`, `VACUUM` and `WINDOW`. This is not surprising, as the search space of pending constraints with DFS is often quite different from that of regular DFS.

Pending constraints with random path gain coverage quickly due to its broader exploration, but makes limited progress in the last 10 hours. Pending constraints with the depth-biased strategy make little progress for a long time, but gain a lot of coverage towards the end of the run, achieving coverage similar to that of random path. Vanilla KLEE makes steady progress with both search strategies, but does not overtake pending constraints.

4.3.6.2 24-hour Seeded Runs

We also performed 24-hour experiments with *SQLite3* with seed set 1, with each of the three considered search strategies. In addition, we also run in this setting pending constraints with relaxed checks, using the depth biased strategy. As in the non-seeded runs, each experiment was repeated 3 times.

Figure 4.8b shows the results. The seed covers 37.2k instructions. Vanilla KLEE performs similarly across the three search strategies (all the lines for the vanilla KLEE runs are at the bottom of Figure 4.8b) and never manages to complete the seeded path, stalling around 13k covered instructions. It makes very slow progress, covering less than 500 new instructions in the final 20 hours of its run. This is due to KLEE's eager feasibility checks, with large constraints that take a long time to solve.

Our pending constraints approach manages to cover all instructions on the seeded path. With error checks in place, it takes up to 3 hours to complete the seeded path, whereas with relaxed checks it only takes 4 minutes. There are about 303 memory bounds checks

on the seeded path and solving the associated constraints accounts for the majority of the time difference between the configuration without and with relaxed checks. Note that 4 minutes is still significantly slower than the pure interpretation time of KLEE on this input. Our approach still performs some solving activities such as computing independent sets and substituting the seed in the constraints.

There is no difference between search strategies in the initial seeding phase as the exploration is guided by the seed. After pending constraints finish with the seeding, they make little progress for up to several hours. During this time they are attempting to revive pending states, most of which are infeasible and therefore killed, thus giving no additional coverage. Finally, depending on the search strategy, a feasible state is revived, leading to the execution of a different path, potentially giving large coverage gains.

The depth-biased search, which picks good states to revive in 2 out of 3 of our runs, achieves significant new coverage. However, random path search outperforms the depth-biased search in all of the 3 runs and seems to generally achieve higher coverage. Due to determinism of DFS, there is no difference between its runs. They all achieve coverage in-between the random path and depth biased search strategies.

We inspected the generated test inputs for the depth-biased, DFS and random path runs. The test inputs generated by the depth-biased runs were mostly very similar to the seed, with only a small number of characters changed (e.g. `SELECT * TEMp ;.  .` instead of the seed `SELECT * FROM t;`).[1] Even when it manages to find new keywords, such as `FILTER`, `HAVING`, `UPDATE`, `VACUUM`, and `WINDOW`, the test input is close to the initial seed (e.g. `uPdATE * FROM t;....`). DFS-generated test inputs are almost identical to the seed, with only the last 2 characters changing in a depth-first fashion. The random path search strategy, on the other hand, also finds several more new keywords, such as `BEFORE`, `HAVING`, `INSERT`, `FILTER`, and `VACCUM`, but diverges early from the seed and finds additional keywords by exercising different code in the parser (e.g. `./.c....;expLain....`). This difference between the generated test inputs across the three search strategies is expected, as depth-biased search and DFS pick states towards the end of the path, where the seed has a greater impact.

[1] Inputs shorter than the specified symbolic input size are padded with \0 and newlines (\n, \r) are shown as "."

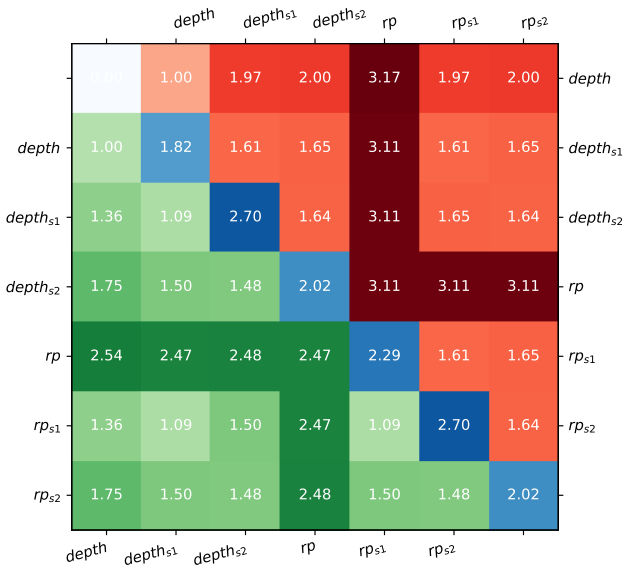


Figure 4.9: Heatmaps of coverage combination pairs for vanilla KLEE (below the diagonal) and pending constraints (above the diagonal) for *SQLite3*. Section 4.3.6.3 describes how to read them.

4.3.6.3 Pairwise Coverage Combinations

Finally, we wanted to explore how the different combinations (non-seeded, seeded with seed sets 1 and 2, and different search strategies) complement each other. Therefore, we looked at the pairwise combinations of coverage for 2-hour runs of each configuration. We decided to omit showing the DFS runs in this case to illustrate the complementary aspects of seeding versus non-seeding runs more clearly. As shown in Figure 4.4c, DFS is really effective in *SQLite3* without seeds. The union of non-seeded DFS and other approaches cover only 10-100 more instructions than DFS by itself. However, we note that DFS is not always the best-performing strategy, as shown by other benchmarks.

Analysing this amount of data is hard, so we devised the visualisation in Figure 4.9, which shows pairwise comparisons of combined coverage across the 6 different configurations for *SQLite3*. The labels indicate the search strategy used and the seed set is indicated in the subscript. Un-subscripted labels indicate non-seeding runs.

The figure consists of three different heatmaps. The red heatmap above the diagonal shows the coverage achieved by pending constraints. The green heatmap below the diagonal shows the coverage achieved by vanilla KLEE. The blue diagonal shows the ratio between pending states and vanilla for non-combined runs. The data in red and green heatmaps is normalised with respect to their upper-left corner, that is the two hour depth-biased run without seeds. The diagonal of the red and green heatmaps therefore show the non-combined coverage.

As this graphic is dense with information, we will walk through a couple of examples. Looking at the top-left corner of both red and green heatmap, we see that both pending and vanilla KLEE with depth-biased strategy have the value of 1.00 — this is because everything is normalised to this value. However, the blue 1.82 value tells us that our pending constraints approach covered 82% more instructions than vanilla KLEE with depth-biased search strategy.

Now focusing on two configurations from green heatmap below the diagonal: depth-biased without seeds and depth-biased with seeds for vanilla KLEE. The coverage of the run with a seed from set 1 is 9% (1.09) higher than that of the non-seeded run, while the combined coverage of both achieve 36% (1.36) more coverage. This indicates that there is complementarity between the coverage achieved with and without a seed.

Looking at the same two combinations in the red heatmap above the diagonal, we can see that with pending constraints, seeding with set 1 achieves 61% (1.61) more coverage than the non-seeding run. Furthermore, their combination achieves 97% (1.97) more coverage than solely the non-seeding run. That indicates again that there is complementarity between seeding and non-seeding runs as with vanilla KLEE.

The random path search strategy without seeding achieves more than twice the coverage of the depth-biased search strategy in both vanilla KLEE (2.47) and with pending constraints (3.11). However, pending constraints cover 129% (2.29) more lines than vanilla KLEE. There is also some complementarity of coverage between depth-biased search and random path with pending constraints. Their union covers 217% (3.17) more instructions than just depth-biased as opposed to 211% (3.11) achieved by random path. Vanilla KLEE behaves similarly.

Finally, looking at the union of coverage between non-seeded and seeded runs, we can see that for pending constraints seeding complements well with depth-biased exploration, achieving over 30% points (1.61 to 1.97 and 1.65 to 2.00) more coverage when combined. With random path, we do not see any such complementarity (3.11). Vanilla KLEE follows a similar pattern.

4.3.7 ZESTI

ZESTI [78] is a promising extension of KLEE that combines EGT-style symbolic execution with seeding via regression test suites. Unfortunately, its original implementation was never upstreamed, partly because it is quite large and intrusive. In this section we show that pending constraints can be used to build a lightweight and effective version of ZESTI.

ZESTI consists of several parts. The two most important ones are the ability to use a variety of inputs as seeds and the so called ZESTI searcher, whose purpose is to explore paths around sensitive instructions. The idea of the ZESTI searcher is to take a single seed and execute it to completion, while recording sensitive instructions and divergence points. A divergence point is a symbolic branch where the path not taken by the seed is feasible. ZESTI then starts bounded symbolic execution from the divergence point closest to a sensitive instruction. It then moves to the next closest divergence point and so on.

Re-implementing ZESTI on top of pending constraints is straightforward. Consider a symbolic execution run with pending constraints after a single seed has been executed to completion. There are no normal states and many pending states representing the divergence points described above. The ZESTI searcher is implemented by considering pending and normal states separately. We prioritise normal states, but only if their depth does not exceed the depth of the last revived pending state plus some bound. Pending states are revived in the order of distance to sensitive instructions. This is equivalent to the ZESTI searcher and is easy to implement. For simplicity, our implementation currently considers only memory accesses as sensitive instructions.

We found the benchmarks used for evaluation to be too hard for ZESTI. For example, as seen in Section 4.3.6.2 it takes three hours to execute a simple seed with *SQLite3*. Thus

running the whole test suite of *SQLite3* and exploring a significant amount of paths around a seed is impractical. Therefore, we chose three tools that KLEE can execute more comfortably: *dwarfdump* [70], *readelf* [42] and *tar* [46]. These are inspired by the original ZESTI paper [78], where we replaced *GNU Coreutils* with *tar* as recent modifications in the *Coreutils* build system make it harder to use with ZESTI.

To capture the seeds, we replaced each binary with a script and ran the application test suite. We then removed large seeds of over 8.1MiB to keep the execution time associated with an individual seed short. This resulted in 1273, 313 and 5 seeds for *dwarfdump*, *tar* and *readelf* respectively. Since we wanted to run each seed for 30 minutes as per the original ZESTI paper and keep the overall time under 12 hours, we ran at most 200 seeds per benchmark.

These experiments found one bug in *dwarfdump*[2] and one in *tar*[3] which have already been confirmed and fixed by the developers. Both of these bugs were found by the ZESTI searcher and were not triggered by the original seed. Vanilla KLEE seeded with the seed ZESTI mutated was not able to find these bugs with a 2-hour timeout.

4.4 Related work

Concolic executors such as CREST [14], DART [47] or SAGE [48] briefly described in Section 1.2.3 also drive each test input to completion, which is similar to the behaviour of pending constraints. However, these tools suffer from the disadvantages of concolic executors such as re-executing path prefixes and exploring a single path at a time. Our approach brings some of the strengths of concolic execution to EGT-style tools like KLEE while maintaining their advantages.

KLEE [16] has an existing seeding mechanism, which is also used by ZESTI [78] and KATCH [79]. However, when following a seed, KLEE eagerly performs feasibility checks at every symbolic branch, unlike pending constraints which defer these checks for when a pending state is revived. This in turn can have a big impact on coverage, as we have shown in Section 4.3.4.

[2] `https://www.prevanders.net/dwarf.html` (28 June 2020 update)
[3] `http://git.savannah.gnu.org/cgit/tar.git/commit/?id=dd1a6bd37a0d57eb4f002f01f49c51fa5c6bb104`

KLUZZER [66], a whitebox fuzzer based on KLEE, implements a similar idea of delaying the satisfiability checks by only following the currently satisfied branch given by a seed. However, their approach goes further by trading off the benefits of EGT-style symbolic execution completely and reverting to concolic execution.

UC-KLEE [91] introduces *lazy constraints*. Here, the executor continues exploring paths although the underlying constraint solver can't determine the feasibility in a given amount of time. The corresponding expensive constraint is added as lazy constraint to the respective path condition and only evaluated when some goal is satisfied, e.g. a bug is found, to suppress false positives. This leads to more states in memory and thus to more solver queries but can also reduce the overall solver time as the additional constraints along such paths narrow down the solution space for the constraint solver. Our approach explores a different design point, which always favours states that are known to be feasible, either via caching or seeding, and when just pending states are left, they are only explored further if they are determined to be feasible.

Similarly, speculative symbolic execution [124] delays checking the feasibility of constraints until a specified number of path conditions has been accumulated. While targeting similar problem space to pending constraints their approach can spend time exploring infeasible paths, while pending constraints only delay feasibility checking. They also propose *absurdity based optimization*, which skips the feasibility check for the else branch if the then branch was shown to be infeasible. This is similar, but complementary to our use of caches or seeds for fast feasibility checks.

Hybrid fuzzing approaches such as Driller [110], QSYM [125] or SAVIOR [23] pass concrete inputs between a greybox fuzzer and a symbolic executor. These approaches could directly benefit from pending constraints to achieve a tighter integration between fuzzing and symbolic execution.

4.5 Conclusion

We have presented pending constraints, a strategy that achieves a more efficient use of constraint solving and a deeper exploration of programs with symbolic execution. The key idea is to aggressively follow paths that are known to be feasible either via caching or seeding, while deferring all other paths by storing states with *pending constraints*. We implemented this approach in KLEE and evaluated it on eight hard benchmarks, including *make*, *SQLite3* and *tcpdump*. Our evaluation shows that pending constraints can significantly increase the coverage achieved by symbolic execution in both seeded and non-seeded exploration.

Chapter 5

Conclusion

We have presented three ways of enhancing symbolic execution.

In Chapter 2, we presented a novel approach for summarising loops in C code. This approach uses counterexample-guided synthesis to generate loop summaries. We evaluated our approach on a large loop database extracted from popular open-source systems and assessed its utility in several contexts: symbolic execution, where we recorded speedups of several orders of magnitude; compiler optimisations, where several summaries resulted in significant performance improvements; and refactoring, where some of our summary patches were accepted by developers.

Then we focused on programs that use complex heap data structures, in which a pointer is allowed to refer to more than one memory object, such as hash tables. These often cause path explosion and memory exhaustion in symbolic execution. In Chapter 3, we presented a novel segmented memory model for symbolic execution, which uses pointer alias analysis to group objects into memory segments. Our segmented memory model can significantly reduce forking due to symbolic dereferences, sometimes even completely eliminating the need for forking.

Finally, we presented pending constraints in Chapter 4. Pending constraints are a strategy that achieves a more efficient use of constraint solving and a deeper exploration of programs with symbolic execution. The key idea is to aggressively follow paths that are known to be

feasible either via caching or seeding, while deferring all other paths by storing states with *pending constraints*. We implemented this approach in KLEE and evaluated it on eight hard benchmarks, including *make*, *SQLite3* and *tcpdump*. Our evaluation shows that pending constraints can significantly increase the coverage achieved by symbolic execution in both seeded and non-seeded exploration.

We can see several avenues for possible future work on each of our chapters.

The path forward for the loop summarisation approach of Chapter 2 is clear. The process of creating a loop summary could become fully automated, with significant engineering work. Additional gadgets and loop types could also be supported. Both of these changes would significantly improve the usability and adaptability of the approach, however it remains unclear how useful it would be in practice. There are some loops that cannot reasonably be summarised into standard functions which would still cripple symbolic execution of C-like programs using the theory of strings. The usefulness of loop summaries as a refactoring tool is intriguing, but still limited by its speed, which is not appropriate for interactive use.

Trabish and Rinetzky [116] explore an interesting direction of future work for the segmented memory model of Chapter 3. Instead of using alias analysis and grouping memory objects into segments ahead of time, they do it dynamically on the fly. This results in a significant performance boost as the segments are no larger than needed. A different direction for improving the segmented memory model could be to use the layered memory model proposed by Nowack [84], which could also alleviate the problem of segments becoming unnecessarily large.

Finally, the pending constraints presented in Chapter 4 show most potential for impactful future work. On one hand, different methods of fast feasibility checking could be explored in addition to caching and seeding. For example, a regular SMT solver with a low timeout in the order of tens of milliseconds could be used to quickly try to determine feasibility. Another option would be to revive a pending state and use an SMT-sampler [33] to quickly get a large number of solutions for it, which can then be explored aggressively to completion again. Pending constraints could also be used to implement the parallelization idea from Siddiqui and Khurshid [103], where a seed would be used to split the symbolic execution search space

into two. Each of the two workers would execute the seed using pending constraints. Then one worker would prune all the pending states on the *then* branches, while the other worker would prune all the *else* branches. That would effectively partition and distribute the search space between them. The two workers would then proceed normally.

Bibliography

[1] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Counterexample guided inductive synthesis modulo theories. In *Proc. of the 30th International Conference on Computer-Aided Verification (CAV'18)*, July 2018.

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Julian Dolby, Petr Janků, Hsin-Hung Lin, Lukáš Holík, and Wei-Cheng Wu. Efficient handling of string-number conversion. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'20)*, June 2020.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, 2006.

[4] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. JPF-SE: A Symbolic Execution Extension to Java PathFinder. In *Proc. of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, March-April 2007.

[5] Lars Ole Andersen. Program analysis and specialization for the C programming language. Technical report, 1994.

[6] APR. Apache Portable Runtime. https://apr.apache.org/, 2019.

[7] Andrea Aquino, Giovanni Denaro, and Mauro Pezzè. Heuristically matching solution spaces of arithmetic formulas to efficiently reuse solutions. In *Proc. of the 39th International Conference on Software Engineering (ICSE'17)*, May 2017.

[8] The GPyOpt authors. GPyOpt: A bayesian optimization framework in python. `http://github.com/SheffieldML/GPyOpt`, 2016.

[9] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at `www.SMT-LIB.org`.

[10] M. Berzish, V. Ganesh, and Y. Zheng. Z3str3: A string solver with theory-aware heuristics. In *Proc. of the 17th Formal Methods in Computer-Aided Design (FMCAD'17)*, October 2017.

[11] Luca Borzacchiello, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. Memory models in symbolic execution: key ideas and new thoughts. *Software Testing, Verification and Reliability*, 29(8), 2019.

[12] Quentin Bouillaguet, François Bobot, Mihaela Sighireanu, and Boris Yakobowski. Exploiting pointer analysis in memory models for deductive verification. In *Proc. of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'19)*, January 2019.

[13] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. Combining symbolic execution and search-based testing for programs with complex heap inputs. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'17)*, July 2017.

[14] Jacob Burnim. CREST: A Concolic Test Generation Tool for C, 2020.

[15] Frank Busse, Martin Nowack, and Cristian Cadar. Running symbolic execution forever. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'20)*, July 2020.

[16] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. of the 8th*

USENIX Symposium on Operating Systems Design and Implementation (OSDI'08), December 2008.

[17] Cristian Cadar and Dawson Engler. Execution generated test cases: How to make systems code crash itself. In *Proc. of the 12th International SPIN Workshop on Model Checking of Software (SPIN'05)*, August 2005.

[18] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. EXE: Automatically Generating Inputs of Death. In *Proc. of the 13th ACM Conference on Computer and Communications Security (CCS'06)*, October-November 2006.

[19] Cristian Cadar and Timotej Kapus. Measuring the coverage achieved by symbolic execution. `https://ccadar.blogspot.com/2020/07/measuring-coverage-achieved-by-symbolic.html`, July 2020.

[20] Cristian Cadar and Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. *Communications of the Association for Computing Machinery (CACM)*, 56(2):82–90, 2013.

[21] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing Mayhem on binary code. In *Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'12)*, May 2012.

[22] Avik Chaudhuri and Jeffrey S. Foster. Symbolic security analysis of ruby-on-rails web applications. In *Proc. of the 17th ACM Conference on Computer and Communications Security (CCS'06)*, October-November 2010.

[23] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Taowei, and Long Lu. Savior: Towards bug-driven hybrid testing, 2019.

[24] Berkeley Churchill, Rahul Sharma, JF Bastien, and Alex Aiken. Sound loop superoptimization for google native client. In *Proc. of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, April 2017.

[25] Peter Collingbourne, Cristian Cadar, and Paul H.J. Kelly. Symbolic crosschecking of floating-point and SIMD code. In *Proc. of the 6th European Conference on Computer Systems (EuroSys'11)*, April 2011.

[26] Peter Collingbourne, Cristian Cadar, and Paul H.J. Kelly. Symbolic testing of OpenCL code. In *Proc. of the Haifa Verification Conference (HVC'11)*, December 2011.

[27] L. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Transactions on Software Engineering (TSE)*, 38(4):957–974, July 2012.

[28] CREST: Automatic Test Generation Tool for C. `https://github.com/jburnim/crest`.

[29] Cristina David, Pascal Kesseli, Daniel Kroening, and Matt Lewis. Program synthesis for program analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 40(2):5:1–5:45, May 2018.

[30] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M. L. Potet, and J. Y. Marion. Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis. In *Proc. of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, March 2016.

[31] Robin David, Sébastien Bardin, Josselin Feist, Laurent Mounier, Marie-Laure Potet, Thanh Dinh Ta, and Jean-Yves Marion. Specification of concretization and symbolization policies in symbolic execution. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'16)*, July 2016.

[32] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, March-April 2008.

[33] R. Dutra, J. Bachrach, and K. Sen. Smtsampler: Efficient stimulus generation from complex smt constraints. In *Proc. of the 37th International Conference on Computer-Aided Design (ICCAD'19)*, November 2018.

[34] Bassem Elkarablieh, Patrice Godefroid, and Michael Y. Levin. Precise pointer reasoning for dynamic test generation. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'09)*, July 2009.

[35] Camil Demetrescu Emilio Coppa, Daniele Cono D'Elia. Rethinking pointer reasoning in symbolic execution. In *Proc. of the 32nd IEEE International Conference on Automated Software Engineering (ASE'17)*, October 2017.

[36] Pete Evans. Heartbleed bug: RCMP asked Revenue Canada to delay news of SIN thefts. `https://www.cbc.ca/news/business/heartbleed-bug-900-sins-stolen-from-revenue-canada-1.2609192`, 2014.

[37] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proc. of the 19th International Conference on Computer-Aided Verification (CAV'07)*, July 2007.

[38] gcov – A Test Coverage Program. `gcc.gnu.org/onlinedocs/gcc/Gcov.html`.

[39] Indradeep Ghosh, Nastaran Shafiei, Guodong Li, and Wei-Fan Chiang. Jst: An automatic test generation tool for industrial java applications with strings. In *Proc. of the 35th International Conference on Software Engineering (ICSE'13)*, May 2013.

[40] GNU. GNU Coreutils. `https://www.gnu.org/software/coreutils/`, 2019.

[41] GNU bc. `https://www.gnu.org/software/bc/`, 2020.

[42] GNU Binutils. `https://www.gnu.org/software/binutils/`, 2020.

[43] GNU datamash. `https://www.gnu.org/software/datamash/`, 2020.

[44] GNU M4. `https://www.gnu.org/software/m4/`, 2020.

[45] GNU Make. `https://www.gnu.org/software/make/`, 2020.

[46] GNU tar. `https://www.gnu.org/software/tar/`, 2020.

[47] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'05)*, June 2005.

[48] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Proc. of the 15th Network and Distributed System Security Symposium (NDSS'08)*, February 2008.

[49] Patrice Godefroid and Daniel Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'11)*, July 2011.

[50] Sumit Gulwani. Dimensions in program synthesis. In *Proc. of the 12th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'10)*, July 2010.

[51] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proc. of the 38th ACM Symposium on the Principles of Programming Languages (POPL'11)*, January 2011.

[52] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'11)*, June 2011.

[53] The Heartbleed bug. `http://heartbleed.com/`, April 2014.

[54] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proc. of the 2nd ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, June 2001.

[55] Michael Hind and Anthony Pioli. Evaluating the effectiveness of pointer alias analyses. *Science of Computer Programming*, 39(1):31 – 55, 2001. Static Program Analysis (SAS'98).

[56] Carly Hysell. Garmin Outage Press Release. https://newsroom.garmin.com/newsroom/press-release-details/2020/Garmin-issues-statement-on-recent-outage/default.aspx, 2020.

[57] ImageMagick. https://imagemagick.org, 2020.

[58] Wei Jin and Alessandro Orso. Bugredux: Reproducing field failures for in-house debugging. In *Proc. of the 34th International Conference on Software Engineering (ICSE'12)*, June 2012.

[59] Timotej Kapus, Frank Busse, and Cristian Cadar. Pending constraints in symbolic execution for better exploration and seeding. In *Proc. of the 35th IEEE International Conference on Automated Software Engineering (ASE'20)*, September 2020.

[60] Timotej Kapus and Cristian Cadar. A segmented memory model for symbolic execution. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*, August 2019.

[61] Timotej Kapus, Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Cristian Cadar. Computing summaries of string loops in C for better testing and refactoring. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'19)*, June 2019.

[62] Timotej Kapus, Martin Nowack, and Cristian Cadar. Constraints in dynamic symbolic execution: Bitvectors or integers? In *Proc. of the 13th International Conference on Tests and Proofs (TAP'19)*, October 2019.

[63] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proc. of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, April 2003.

[64] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: A solver for string constraints. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'09)*, July 2009.

[65] Chris Lattner and Vikram Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'05)*, June 2005.

[66] Hoang M. Le. Kluzzer: Whitebox fuzzing on top of llvm. In *Automated Technology for Verification and Analysis (ATVA)*, October 2019.

[67] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):1841, July 1993.

[68] Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan. KLOVER: a symbolic execution and automatic test generation tool for C++ programs. In *Proc. of the 23rd International Conference on Computer-Aided Verification (CAV'11)*, July 2011.

[69] Tianyi Liang, Andrew Reynolds, Nestan Tsiskaridze, Cesare Tinelli, Clark Barrett, and Morgan Deters. An efficient SMT solver for string constraints. *Formal Methods in System Design (FMSD)*, 48(3):206–234, June 2016.

[70] libdwarf. `"https://www.prevanders.net/dwarf.html`, 2020.

[71] libjpeg. `http://libjpeg.sourceforge.net/`, 2020.

[72] libpng. `http://www.libpng.org/`, 2020.

[73] Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair Donaldson, Rafael Zhl, and Klaus Wehrle. Floating-point symbolic execution: A case study in n-version programming. In *Proc. of the 32nd IEEE International Conference on Automated Software Engineering (ASE'17)*, October 2017.

[74] Jacques-Louis Lions, Lennart Luebeck, Jean-Luc Fauquembergue, Gilles Kahn, Wolfgang Kubbat, Stefan Levedag, Leonardo Mazzini, Didier Merle, and Colin OHalloran.

Ariane 5 flight 501 failure report by the inquiry board. `http://zoo.cs.yale.edu/classes/cs422/2010/bib/lions96ariane5.pdf`, 1996.

[75] M4. GNU M4. https://www.gnu.org/software/m4/, 2019.

[76] David J. C. MacKay. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, 2002.

[77] Make. GNU Make. https://www.gnu.org/software/make/, 2019.

[78] Paul Dan Marinescu and Cristian Cadar. make test-zesti: A symbolic execution solution for improving regression testing. In *Proc. of the 34th International Conference on Software Engineering (ICSE'12)*, June 2012.

[79] Paul Dan Marinescu and Cristian Cadar. KATCH: High-coverage testing of software patches. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, August 2013.

[80] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. Path-exploration lifting: Hi-fi tests for lo-fi emulators. In *Proc. of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*, March 2012.

[81] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the Association for Computing Machinery (CACM)*, 33(12):32–44, 1990.

[82] Ettay Nevo. What happened to Baresheet. `https://davidson.weizmann.ac.il/en/online/sciencepanorama/what-happened-beresheet`, 2019.

[83] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program repair via semantic analysis. In *Proc. of the 35th International Conference on Software Engineering (ICSE'13)*, May 2013.

[84] Martin Nowack. Fine-grain memory object representation in symbolic execution. In *Proc. of the 34th IEEE International Conference on Automated Software Engineering (ASE'19)*, November 2019.

[85] Saahil Ognawala, Thomas Hutzelmann, Eirini Psallida, and Alexander Pretschner. Improving function coverage with munch: A hybrid fuzzing and directed symbolic execution approach. April 2018.

[86] Hristina Palikareva and Cristian Cadar. Multi-solver support in symbolic execution. In *Proc. of the 25th International Conference on Computer-Aided Verification (CAV'13)*, July 2013.

[87] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. Shadow of a doubt: Testing for divergences between software versions. In *Proc. of the 38th International Conference on Software Engineering (ICSE'16)*, May 2016.

[88] Corina S. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. Symbolic PathFinder: Integrating symbolic execution with model checking for Java bytecode analysis. In *Proc. of the 28th IEEE International Conference on Automated Software Engineering (ASE'13)*, September 2013.

[89] Fernando Magno Quintao Pereira and Daniel Berlin. Wave propagation and deep propagation for pointer analysis. In *Proc. of the 7th International Symposium on Code Generation and Optimization (CGO'09)*, March 2009.

[90] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'14)*, June 2014.

[91] David A. Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *Proc. of the 24th USENIX Security Symposium (USENIX Security'15)*, August 2015.

[92] Anthony Romano. *Methods for Binary Symbolic Execution*. PhD thesis, Stanford University, 2014.

[93] N. Rosner, J. Geldenhuys, N. M. Aguirre, W. Visser, and M. F. Frias. Bliss: Improved symbolic execution by bounded lazy initialization with sat support. *IEEE Transactions on Software Engineering (TSE)*, 41(7):639–660, July 2015.

[94] Richard Rutledge, Sunjae Park, Haider Khan, Alessandro Orso, Milos Prvulovic, and Alenka Zajic. Zero-overhead path prediction with progressive symbolic execution. In *Proc. of the 41st International Conference on Software Engineering (ICSE'19)*, May 2019.

[95] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'10)*, May 2010.

[96] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'09)*, July 2009.

[97] Daniel Schemmel, Julian Bning, Csar Rodrguez, David Laprell, and Klaus Wehrle. Symbolic partial-order execution for testing multi-threaded programs. In *Proc. of the 32nd International Conference on Computer-Aided Verification (CAV'20)*, July 2020.

[98] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Proc. of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*, March 2013.

[99] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*, September 2005.

[100] Daryl Shannon, Sukant Hajra, Alison Lee, Daiqian Zhan, and Sarfraz Khurshid. Abstracting symbolic execution with string analysis. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, September 2007.

[101] V. Sharma, K. Hietala, and S. McCamant. Finding substitutable binary code for reverse engineering by synthesizing adapters. In *Proc. of the IEEE International Conference on Software Testing, Verification, and Validation (ICST'18)*, April 2018.

[102] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'16)*, May 2016.

[103] Junaid Haroon Siddiqui and Sarfraz Khurshid. Scaling symbolic execution using ranged analysis. In *Proc. of the 27th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'12)*, October 2012.

[104] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. Programming by sketching for bit-streaming programs. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'05)*, June 2005.

[105] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proc. of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, October 2006.

[106] SQLite. SQLite Database Engine. `https://www.sqlite.org/`, 2019.

[107] Niranjan Srinivas, Andreas Krause, Sham Kakade, and Matthias Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. In *Proc. of the 27th International Conference on International Conference on Machine Learning (ICML'10)*, June 2010.

[108] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *Proc. of the 37th ACM Symposium on the Principles of Programming Languages (POPL'10)*, January 2010.

[109] Appleinsider Staff. Bug in iOS Unicode handling crashes iPhones with a simple text. https://appleinsider.com/articles/15/05/26/bug-in-ios-notifications-handling-crashes-iphones-with-a-simple-text, 2015.

[110] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proc. of the 23rd Network and Distributed System Security Symposium (NDSS'16)*, February 2016.

[111] Yulei Sui and Jingling Xue. SVF: Interprocedural static value-flow analysis in LLVM. In *Proc. of the 25th International Conference on Compiler Construction (CC'16)*, March 2016.

[112] tcpdump. https://www.tcpdump.org/, 2020.

[113] Nikolai Tillmann and Jonathan De Halleux. Pex: white box test generation for .NET. In *Proc. of the 2nd International Conference on Tests and Proofs (TAP'08)*, April 2008.

[114] David Trabish, Timotej Kapus, Noam Rinetzky, and Cristian Cadar. Past-sensitive pointer analysis for symbolic execution. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*, November 2020.

[115] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. Chopped symbolic execution. In *Proc. of the 40th International Conference on Software Engineering (ICSE'18)*, May 2018.

[116] David Trabish and Noam Rinetzky. Relocatable addressing model for symbolic execution. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'20)*, July 2020.

[117] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *Proc. of the 21st ACM Conference on Computer and Communications Security (CCS'14)*, November 2014.

[118] Marek Trtík and Jan Strejček. Symbolic memory with pointers. In *Automated Technology for Verification and Analysis (ATVA)*, November 2014.

[119] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. Green: reducing, reusing and recycling constraints in program analysis. In *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'12)*, November 2012.

[120] Vorbis. `https://xiph.org/vorbis/`, 2020.

[121] Qianqian Wang and Alessandro Orso. Mimicking user behavior to improve in-house test suites. In *Proc. of the 41th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion'19)*, May 2019.

[122] Xiaofei Xie, Yang Liu, Wei Le, Xiaohong Li, and Hongxu Chen. S-looper: Automatic summarization for multipath string loops. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'15)*, July 2015.

[123] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. Automatically generating malicious disks using symbolic execution. In *Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'06)*, May 2006.

[124] Zhang Yufeng, Chen Zhenbang, and Wang Ji. Speculative symbolic execution. In *Proc. of the 23rd International Symposium on Software Reliability Engineering (ISSRE'12)*, November 2012.

[125] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In *Proc. of the 27th USENIX Security Symposium (USENIX Security'18)*, August 2018.

[126] Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Murphy Berzish, Julian Dolby, and Xiangyu Zhang. Z3str2: An efficient solver for strings, regular expressions,

and length constraints. *Formal Methods in System Design (FMSD)*, 50:249–288, June 2017.

[127] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: A z3-based string solver for web application analysis. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, August 2013.