The Raymond and Beverly Sackler Faculty of Exact Sciences

The Blavatnik School of Computer Science

Novel Memory Models for Symbolic Execution

Thesis submitted for the degree of Doctor of Philosophy

by

David Trabish

This work was carried out under the supervision of

Prof. Noam Rinetzky

Submitted to the Senate of Tel Aviv University

March 2024

Acknowledgements

First, I would like to thank my supervisor Prof. Noam Rinetzky who guided me throughout this work. Then, I would like to thank the other co-authors in this work: Cristian Cadar, Shachar Itzhaky, Sharon Shoham, Timotej Kapus, and Vaibhav Sharma. Finally, I would like to thank my family for the support all along.

Abstract

Symbolic execution (SE) is a program analysis technique that systematically explores paths in the program while checking their feasibility using an SMT solver. SE has been used successfully in many applications, both academic and industrial, but still suffers from three main challenges: *path explosion*, expensive *constraint solving*, and *incompleteness*. The *memory model* is a key component in SE, as it determines the representation of the address space and the memory objects. This, in turn, affects path exploration and constraint encoding, which are directly related to the aforementioned challenges. In this thesis, we propose several novel memory models, and show how they help to scale SE.

First, we propose a memory model which uses symbolic base addresses instead of the standardly-used concrete base addresses. This memory model allows us to dynamically modify the layout of address spaces and the representation of memory objects. We use this capability to achieve faster solving of array-theory constraints and to reduce forking, and thus mitigate path explosion, when handling symbolic pointers.

Another advantage of that memory model is the ability to distinguish between address expressions and non-address expressions. We exploit this to perform efficient query caching of queries that depend on address expressions.

Then, we propose a memory model where the size of a memory object can be symbolic, and not only concrete. To avoid arbitrarily large memory objects, which require unbounded memory, the size of memory objects is limited by a user-specified capacity. This memory model allows us to perform a more *complete* analysis, but it also increases the number of forks. To address this, we apply an optimized form of state merging when those additional forks are introduced.

The effectiveness of the state merging approach mentioned above might be limited in the presence of complex disjunctions and *if-then-else* expressions, especially when many symbolic states are being merged. To address this, we propose a novel state merging technique which uses quantified encoding instead of the standard quantifier-

free encoding. To efficiently handle the new encoding, we propose a specialized solving procedure which takes advantage of the specific structure of our quantified constraints.

We address the limitations of SE using novel memory models, and make it thus more scalable and complete. We implement our ideas, evaluate them on real-world benchmarks, and find bugs. Our replication packages are freely available and contain all the resources needed to run our experiments.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Symbolic execution (SE) [76] is a program analysis technique that has many applications in both academic and industrial areas, including: high-coverage test generation [38, 39, 89], bug finding [39, 66], patch testing [83, 93], automatic program repair [84, 86], debugging [72], cross checking [42, 74], reverse engineering [40], and side-channel analysis [34, 35, 90]. In symbolic execution, the program is executed with an unconstrained (or a partially constrained) symbolic input, rather than with a concrete one. Whenever the execution reaches a branch that depends on one of the symbolic inputs, an SMT solver [49] is used to determine the feasibility of each branch side, and the relevant paths are further explored while their paths constraints are updated with the corresponding constraints. Once the execution of a given path completes, the SMT solver generates a satisfying assignment using the corresponding path constraints, i.e., a concrete input (test case) that can be used to replay that path.

The main challenges in symbolic execution are *path explosion*, *constraint solving*, and *incompleteness* [25, 37]. *Path explosion* relates to the challenge of navigating the huge number of paths in real-world programs, which is usually at least exponential to the number of static branches in the code. *Constraint solving* relates to the challenge of solving the generated SMT queries, which are numerous, complex, and whose solving usually dominates the analysis time, especially when analyzing real-world programs. *Incompleteness* relates to partial exploration of the state space, which can be caused by concretizations, i.e., when a symbolic value is constrained to a specific concrete value. Such concretizations are performed, for example, when there are *symbolic-size allocations*, i.e., when the number of allocated bytes is symbolic.

A key component in symbolic execution is the *memory model* as it determines

the representation of the address space and the memory objects (i.e., variables and heap allocations). The representation of the address space determines the way in which memory objects are allocated, referenced, and deallocated. The representation of a memory object determines the encoding of its contents, its size, and the way in which read and write operations are performed. These representation choices affect, as explained in Section 1.1, the path exploration and the encoding of constraints, two central aspects in symbolic execution which are directly related to the aforementioned challenges, i.e., path explosion, constraint solving, and incompleteness.

In this thesis, we propose several approaches for modeling the memory, and show how these approaches help to mitigate the challenges hindering symbolic execution.

In the rest of this chapter, we discuss in more detail the various aspects of the memory model (Section 1.1) and provide a high-level description of our main contributions (Section 1.2).

1.1 Background

In this section, we provide sufficient technical details about symbolic execution to understand our main contributions.

1.1.1 Symbolic Execution

In symbolic execution, we run the program with a symbolic (unconstrained or partially constrained) input, rather than a concrete input. Similarly to concrete (or native) execution, the symbolic state records the instruction counter and the state of the memory (i.e., stack, heap, and global variables). However, the memory may contain expressions over symbolic inputs instead of concrete values. In addition, the symbolic state maintains the so-called *path constraints*, a set of logical formulas over the symbolic input variables, which describe the branches that were taken during the execution. We start the execution of the program with an initial symbolic state whose path constraints are empty. When the execution reaches a non-branching statement (e.g., an assignment), we update the symbolic state following the concrete semantics of that statement, but while operating on symbolic values rather than concrete values. When the execution reaches a branch statement (e.g., *if* and *switch*), we examine the symbolic expression that corresponds to the branch condition, and using an SMT solver we check the feasibility of both branch sides w.r.t. the path constraints. If only one of the branch sides is feasible, then we simply follow the relevant branch side. Otherwise, we *fork*, i.e., create a copy of the current symbolic state, and add the relevant condition (the branch condition or its negation) to the path constraints of the two symbolic states. Once a given symbolic state finishes executing the program, we use an SMT solver to obtain a satisfying model for its path constraints. This model gives us a concrete test case (input), which can be used to replay the execution path traversed by that symbolic state.

For example, consider the function `get_sign` in Figure 1.1. Assume that the parameter `x` is a 32-bit integer. In concrete execution, if we want to exhaustively test the program, i.e., cover all of its possible execution paths, then, conceptually, we need to consider all 2^{32} possible inputs. In contrast, in symbolic execution, we start the execution with an initial symbolic state in which the value of `x` is symbolic. When we reach the first branch at line 2, we use the SMT solver to check the feasibility of the

```
 1 int get_sign(int x) {
 2     if (x == 0) {
 3         return 0;
 4     }
 5
 6     if (x < 0) {
 7         return -1;
 8     } else {
 9         return 1;
10     }
11 }
```

Figure 1.1: A program that returns the sign of an integer input.

two branch sides: $x = 0$ and $x \neq 0$. In this case, both branch sides are feasible, so we fork the current symbolic state and update the path constraints in each of the symbolic states. In the symbolic state which takes the branch at line 2, the path constraints are $x = 0$, and in the other one, the path constraints are $x \neq 0$. The first symbolic state finishes the execution at line 2, and the resulting test case maps `x` to 0. The other symbolic state continues the execution and reaches the second branch at line 6. Here, the SMT solver determines the feasibility of both branch sides, and we fork again. In the symbolic state which takes the branch at line 6, the path constraints are $x \neq 0 \wedge x < 0$, and in the other one, the path constraints are $x \neq 0 \wedge x \geq 0$. The two symbolic states finish the execution at lines 7 and 9, respectively. A possible test case for the symbolic state that takes the branch at line 6 maps `x` to -7, and a possible test case for the other one maps `x` to 3. As can be seen, when we execute this program symbolically, we produce three test cases that cover all the possible execution paths of the program.

1.1.2 Memory Modeling

In SE engines [39, 85, 103], the address space is modeled as a linear space of memory objects,[1] where each memory object has a concrete base address, a concrete size, and a (possibly symbolic) content. The address space must preserve the *non-overlapping* property, i.e., the address interval associated with a given memory object must not overlap with the address intervals of other memory objects.

We now discuss several aspects of the memory model, focusing on the challenges they impose. At the end of this section, we provide a simple example illustrating these aspects.

[1] The size of the address space is determined by the pointer size, which is typically 32 or 64 bit. In our examples, we assume 32-bit pointers.

1.1.2.1 Memory Objects

Variables and heap-allocated objects are represented using *memory objects*. A memory object has a concrete base address and a concrete size, and its contents are represented as a sequence of bytes, which is typically encoded as an SMT array [39] using array theory [59]. We assume SMT arrays whose indices and values are 32-width and 8-width bit-vectors, respectively, and use the standard operators *select* and *store*: $select(a, i)$ returns the i-th cell of a, and $store(a, i, v)$ returns a new array obtained from a by replacing the i-th cell with v. When we attempt to access the i-th byte of a memory object of size s, i.e., read or write, we first need to make sure that i is a valid offset. To do so, we generate an SMT query that encodes the fact that $i \geq s$ (w.r.t. the path constraints), and check its satisfiability using the SMT solver. If the query is satisfiable, then we found a test case leading to an out-of-bounds memory access, and report an error. Otherwise, we can safely access the memory object with the offset i. Now, assume that a is the SMT array that represents the contents of the memory object. When we read the i-th byte, the read value is encoded using a *select* expression: $select(a, i)$. When we write an expression e to the i-th byte, the SMT array a is replaced with a *store* expression: $store(a, i, e)$.[2] For clarity of explanations, we use the $select_k(a, i)$ and $store_k(a, i, v)$ operators, instead of employing the standard bit-vector concatenation operations: $select_k(a, i)$ reads an $(8 \cdot k)$-width value from offsets $i, \ldots, i + k - 1$ in a and $store_k(a, i, v)$ writes an $(8 \cdot k)$-width value v to these locations.

Imposed Challenges When we perform a read or write operation, we check the memory safety of that operation by solving SMT queries, which increases the cost of constraint solving.

1.1.2.2 Allocation

When an allocation of an object of size s occurs during symbolic execution, we add to the address space a new memory object with a base address a and a size s. If the allocation size s is symbolic, the common practice is to *concretize* it, i.e., fix it to one of the possible concrete values that satisfy the path constraints. To preserve the non-overlapping property of the address space, we need to make sure that the address

[2] SE engines that do not use array theory handle the access operations using *ite* (if-then-else) expressions. Note that *select* and *store* expressions can be eliminated by using *ite*'s.

interval associated with the newly allocated memory object, i.e., $[a, a + s)$, does not overlap with the address intervals of the existing memory objects.

Imposed Challenges The concretizations caused by allocations of symbolic size may lead to missed execution paths, i.e., incomplete exploration. Furthermore, if the sizes of memory objects are allowed to be an arbitrary symbolic values, then the preservation of the non-overlapping property can be costly.

1.1.2.3 Pointer Resolution

When we need to dereference a pointer p, be it concrete or symbolic, we scan the address space in order to find all the memory objects that might be pointed to by that pointer. To determine if a given memory object may be pointed to by p, we generate a so-called *resolution* query, an SMT query that encodes the fact that the value of p belongs to the address interval of that memory object, and check its satisfiability. If we find a memory object pointed to by p, then assuming that the base address of the resolved memory object is a, we can access the desired memory location at offset $p - a$ of the SMT array of the memory object. Note that if p is not a concrete pointer, then it may point to multiple memory objects. If that happens, then the common practice is to fork the execution so that p points to only one memory object in each forked symbolic state.

Imposed Challenges Pointer resolution is performed by solving SMT queries, which increases the cost of constraint solving. Moreover, when symbolic pointers are resolved to multiple memory objects, the execution is forked, which amplifies path explosion.

1.1.2.4 Deallocation

When a pointer p is deallocated, for example, when `free(p)` is executed, we first resolve p, and then remove all the resolved memory objects from the address space.

Imposed Challenges Deallocation requires resolving pointers, therefore, it imposes the challenges related to pointer resolution.

1.1.2.5 Example

Consider the code in Figure 1.2. At line 1 we define the symbolic variable `n`, and at line 2 we reach the allocation of the array of pointers `ptrs`. Since the allocation size, i.e., the

```
1 size_t n; // symbolic
2 char **ptrs = malloc(n * sizeof(char *));
3 for (int i = 0; i < n; i++) {
4     ptrs[i] = malloc(10);
5 }
6
7 int k; // symbolic
8 if (0 <= k && k < 2) {
9     // symbolic pointer dereference
10    char *s = ptrs[k];
11    // symbolic pointer dereference with multiple resolutions
12    s[0] = 'a';
13 }
14
15 for (int i = 0; i < n; i++) {
16    free(ptrs[i]);
17 }
18 free(ptrs);
```

Figure 1.2: A simple example with memory operations.

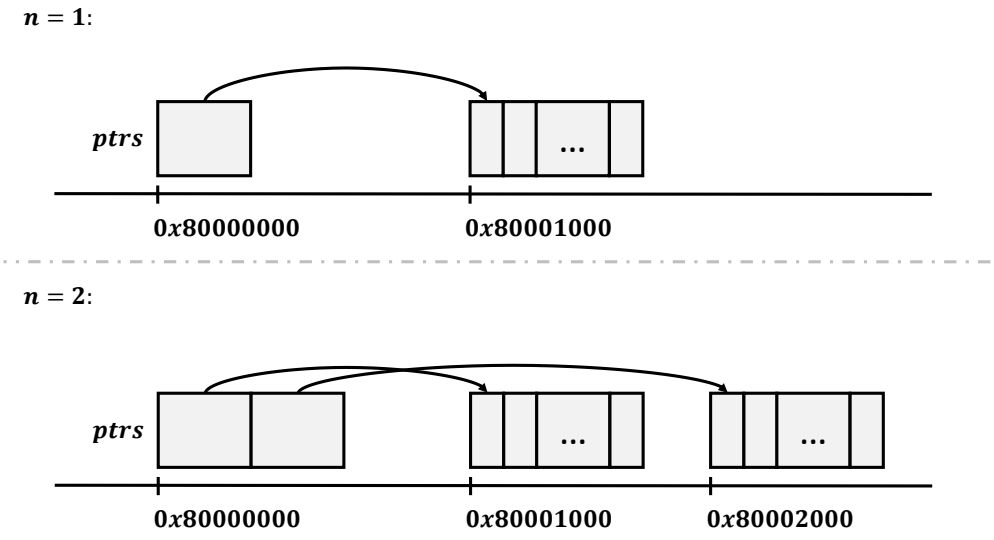


Figure 1.3: Memory diagrams of the program from Figure 1.2 with different concretization values of `n`.

value of `n`, is symbolic, we are forced to perform a concretization before the allocation itself. Assuming that the value of `n` is concretized to 1, we then allocate the array `ptrs`, and initialize its only cell at line 4. Assume that the base addresses of the memory objects allocated at lines 2 and 4 are $0x80000000$ and $0x80001000$, respectively, and that the SMT array of the memory object associated with `ptrs` is a_{ptrs}. The upper part of Figure 1.3 depicts the state of the memory after line 5 when `n` is concretized to 1. At line 7, we define the symbolic variable `k` whose value, denoted by k, is unconstrained. Then, we execute the branch at line 8, and both of the branch sides are feasible here, but for the sake of the example, assume that we take the branch. In that case, we add the condition $0 \leq k < 2$ to the path constraints and execute lines 10-12.

Recall that the value of `k` is symbolic, so the pointer used to access `ptrs` at line 10 is symbolic as well. More specifically, the value of this pointer is $0x80000000 + 4 \cdot k$. In order to dereference this pointer, we first need to resolve it. To do so, we scan the address space, which contains two memory objects, and the resulting resolution queries are:

$$0 \leq k < 2 \ \wedge \ 0 \leq ((0x80000000 + 4 \cdot k) - 0x80000000) < 4$$

$$0 \leq k < 2 \ \wedge \ 0 \leq ((0x80000000 + 4 \cdot k) - 0x80001000) < 10$$

In this case, only the first query is satisfiable, so we conclude that this pointer is resolved to the memory object associated with `ptrs`. Before dereferencing this pointer, we need to check the validity of the accessed offset, which is given by:

$$(0x80000000 + 4 \cdot k) - 0x80000000$$

and can be simplified to $4 \cdot k$. To do so, we check the satisfiability of the following query:

$$0 \leq k < 2 \ \wedge \ 4 \cdot k \geq 4$$

In this case, the query is satisfiable when $k = 1$, so we detect an out-of-bounds memory access.

Now, let us examine a different path under the assumption that the value of `n` at line 2 is concretized to 2 instead of 1. Similarly to the first path, we allocate the array `ptrs` at line 2, and initialize its cells in the two iterations at line 4. As before, assume that the memory object associated with `ptrs` has the base address $0x80000000$

and its SMT array is a_{ptrs}. In addition, assume that the two memory objects allocated at line 4 have the base addresses $0x80001000$ and $0x80002000$, respectively, and that their SMT arrays are a_1 and a_2, respectively. The lower part of Figure 1.3 depicts the state of the memory after line 5 when `n` is concretized to 2.

As in the first path, when we reach the dereference of the symbolic pointer $0x80000000 + 4 \cdot k$ at line 10, we have to resolve it. To do so, we scan the address space, which this time contains three memory objects, and the resulting resolution queries are:

$$0 \leq k < 2 \ \wedge \ 0 \leq ((0x80000000 + 4 \cdot k) - 0x80000000) < 8$$

$$0 \leq k < 2 \ \wedge \ 0 \leq ((0x80000000 + 4 \cdot k) - 0x80001000) < 10$$

$$0 \leq k < 2 \ \wedge \ 0 \leq ((0x80000000 + 4 \cdot k) - 0x80002000) < 10$$

Here, only the first query is satisfiable, so we conclude that this pointer is resolved to the memory object associated with `ptrs`. As before, the accessed offset here is given by:

$$(0x80000000 + 4 \cdot k) - 0x80000000$$

which can be simplified to $4 \cdot k$. To check its validity, we check the satisfiability of the following query:

$$0 \leq k < 2 \ \wedge \ 4 \cdot k \geq 8$$

In this case, the query is unsatisfiable, so we can safely access the resolved memory object.

Since the value of `k` is symbolic, the pointer `s` obtained at line 10 is symbolic as well. More specifically, the value of this pointer is:

$$select_4(a, (0x80000000 + 4 \cdot k) - 0x80000000)$$

where a denotes the SMT array of `ptrs` at this point:

$$a \triangleq store_4(store_4(a_{ptrs}, 0, 0x80001000), 4, 0x80002000)$$

In order to dereference the pointer `s` at line 12, we scan the address space again, and

the resulting resolution queries are:

$$0 \leq k < 2 \ \wedge \ 0 \leq (select_4(a, 4 \cdot k) - 0x80000000) < 8$$

$$0 \leq k < 2 \ \wedge \ 0 \leq (select_4(a, 4 \cdot k) - 0x80001000) < 10$$

$$0 \leq k < 2 \ \wedge \ 0 \leq (select_4(a, 4 \cdot k) - 0x80002000) < 10$$

Here, the two last queries are satisfiable, so we conclude that `s` may point to the two memory objects allocated at line 4. Since `s` may point to multiple memory objects, we fork, and obtain two symbolic states: In one case, `s` points to the memory object allocated in the first iteration at line 4, and in the other case, `s` points to the memory object allocated in the second iteration. The path constraints corresponding to these symbolic states are identical to the second and third resolution queries above. Consider, for example, the first case, where `s` is resolved to the memory object whose base address is $0x80001000$. When `s` is accessed at line 12, the offset is given by:

$$select_4(a, 4 \cdot k) - 0x80001000$$

and after the update of `s[0]`, the SMT array of the updated memory object is set to:[3]

$$store(a_1, select_4(a, 4 \cdot k) - 0x80001000, 97)$$

After that, we deallocate at lines 15-18 the memory objects that were allocated at the beginning of the program. Let us see, for example, how the deallocation is performed in the first iteration of the loop at lines lines 15-17. Here, the value of the pointer stored in the first cell is:

$$select_4(a, 0)$$

To deallocate this pointer, we first need to resolve it, and to do so, we scan the address space and generate the following resolution queries:

$$0 \leq k < 2 \ \wedge \ select_4(a, 0) = 0x80000000$$

$$0 \leq k < 2 \ \wedge \ select_4(a, 0) = 0x80001000$$

$$0 \leq k < 2 \ \wedge \ select_4(a, 0) = 0x80002000$$

[3] Recall that the ASCII code of `a` is 97.

In this case, only the second query is satisfiable, so we conclude that this pointer is resolved to the memory object allocated in the first iteration at line 4. The deallocation of the remaining memory objects is performed in a similar manner.

```
1 char **arrays[2];
2 for (int i = 0; i < 2; i++) {
3     arrays[i] = malloc(2 * sizeof(char *));
4     for (int j = 0; j < 2; j++) {
5         arrays[i][j] = malloc(10);
6     }
7 }
8
9 int i, j; // symbolic
10 char **array = arrays[0];
11 if (0 <= i && i < 2 && 0 <= j && j < 10) {
12     char *buffer = array[i];
13     buffer[j] = 'a';
14 }
```

Figure 1.4: A symbolic pointer with multiple resolution.

1.2 Main Results

In this section, we present the main results of this thesis using a series of simple examples. We do not use a single example, since that will require from us to present a rather complicated program, which might be hard to follow.

1.2.1 Relocatable Memory Model

Handling symbolic pointers is challenging for existing SE engines [38, 39, 89]. First, to resolve a symbolic pointer we need to solve resolution queries, which increases the cost of constraint solving. Second, if a symbolic pointer is resolved to multiple memory objects, then the execution is forked according to the number of resolved memory objects, which increases path explosion.

Consider, for example, the program from Figure 1.4. We start with the execution of the initialization loop at lines 2-7, where at each iteration we allocate an array of pointers at line 3, and initialize the pointer values in the inner loop at line 5. Figure 1.5 depicts the state of the memory after executing lines 2-7. The two arrays allocated at line 3 correspond to $array_1$ and $array_2$. The two buffers allocated in the first iteration of the outer loop at line 5 correspond to $buffer_1$ and $buffer_2$. Then, we access the first array ($array_1$) at line 10, and perform a memory safety check at line 11 in order to access the `i`-th buffer at the `j`-th offset. Since `i` is symbolic, then the pointer accessed at line 12 is a symbolic pointer, and it is resolved to the memory object ($array_1$) allocated in the first iteration of the outer loop at line 3. Since `j` is symbolic, then the pointer accessed at line 13 is symbolic as well. This symbolic pointer is resolved to the two memory

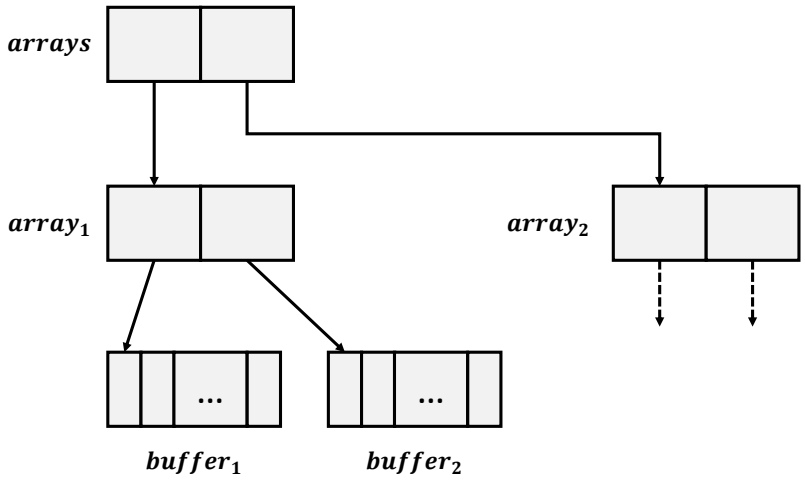


Figure 1.5: A memory diagram of the program from Figure 1.4.

objects (*buffer*$_1$ and *buffer*$_2$) allocated in the first iteration of the outer loop at line 5, so standard symbolic execution must fork.

One way to mitigate this problem, i.e., avoid forking, is to use the *segmented memory model* [73]. This approach uses static pointer analysis to partition the *abstract objects* in the program, i.e., the static allocation sites,[4] into disjoint groups, such that abstract objects that may be pointed to by the same pointer are mapped to the same group. During the symbolic execution, each group is associated with a segment, i.e., a memory block. When there is an allocation of a memory object whose static allocation site belongs to a given group, then the memory object is allocated in the segment associated with that group. This way, every symbolic pointer is guaranteed to be resolved to at most one segment, which helps to reduce the resolution queries and the forking. The main disadvantage of this memory model is the dependence on static pointer analysis. In large programs, static pointer analysis is usually imprecise, so the partitioning may result in large groups that contain many abstract objects. During the symbolic execution, such large groups will lead to the allocation of large segments, which in turn will result in large SMT arrays that slow down the SMT solver, thus overshadowing the benefit of reduced forking.

To illustrate this, consider the program from Figure 1.4 again. Typical pointer analyses are index-insensitive and thus cannot distinguish between the different buffers

[4] In static pointer analysis, a static allocation site is the program location where the corresponding allocation occurs.

that are allocated at line 5, as they all have the same static allocation site, so these buffers will be allocated in the same segment. Note, however, that this segment clearly contains redundant memory objects, since the symbolic pointer at line 13 is resolved to the buffers ($buffer_1$ and $buffer_2$) associated with the first array only ($array_1$).

To mitigate this problem, we propose a *relocatable* memory model. In this memory model, the base addresses observable by the symbolic state are symbolic rather than concrete. The non-overlapping property is preserved by maintaining additional address constraints, which constrain each symbolic base address to a constant value. This gives us the ability to relocate memory objects, i.e., modify their underlying address constraints in a way that is transparent to the symbolic state. Now, instead of partitioning the memory objects into segments ahead of time, we can use the relocatable memory model to create the segments on demand during the symbolic execution. If a symbolic pointer is resolved to multiple memory objects, we create a new segment and relocate the resolved memory objects, which were allocated before, to that segment. Note that the newly created segment contains only memory objects that can be pointed to by that symbolic pointer, without any spurious ones. This way, we can have segments containing fewer memory objects while still handling symbolic pointers without forking. Smaller segments result in smaller SMT arrays which, in turn, help to reduce the cost of constraint solving.

In our example, the symbolic pointer at line 13 is resolved to the two buffers associated with the first array ($array_1$). Therefore, we will create a new segment, and relocate the two resolved memory objects into that segment. This way, the created segment does not contain spurious memory objects, i.e., the buffers associated with the second array.

In our experiments on real-world benchmarks, we show that our approach accelerates constraint solving, while still preserving the benefits of the forking approach for handling symbolic pointers.

The results were published in [112] and are discussed in Chapter 3.

```
1 int z; // symbolic
2 if (z == 0) {
3     allocate_objects();
4 }
5
6 char **array = calloc(2, sizeof(char *));
7 for (int i = 0; i < 2; i++) {
8     array[i] = calloc(10, 1);
9 }
10
11 int i, j; // symbolic
12 if (0 <= i && i < 2 && 0 <= j && j < 10) {
13     char *buffer = array[i];
14     if (buffer[j] == 'a') {
15         // do something...
16     }
17 }
```

Figure 1.6: A simple example with address-dependent queries.

1.2.2 Address-Aware Query Caching

Another problem imposed by the standard memory model relates to *query caching* [39, 103, 119]. Query caching is an optimization used in symbolic execution to reduce the cost of constraint solving, which is based on a cache that memoizes the satisfiability of solved queries. If a given query does not exist in the cache, then its satisfiability is determined using the SMT solver and the cache is updated accordingly. Otherwise, the satisfiability of that query is determined by the cached result without using the SMT solver.

In the standard memory model, where concrete base addresses are used, existing query caching techniques cannot efficiently handle *address-dependent* queries, i.e., queries that involve address values, which are generated in SE engines such as KLEE [39], ANGR [103], Manticore [85], and SAGE [56].

Consider, for example, the program from Figure 1.6. We start with the execution of the branch at line 2, which leads to a fork because the variable `z` is symbolic.

Let us first follow the path that does not take the branch at line 2. In that case, we then allocate an array of pointers at line 6, and initialize the pointers at line 8. At line 12, we perform a memory safety check in order to access the `i`-th buffer at the `j`-th offset. Recall that the values written to `array` are the base addresses of the buffers allocated at line 8, and since `i` is symbolic, the value of `buffer` at line 13 is symbolic and depends on the values of the assigned base addresses. More specifically, assuming that the base addresses of the two memory objects allocated at line 8 are $0x80001000$

and $0x80002000$, and that the SMT array of the memory object associated with `array` is a_{array}, the value of `buffer` at line 13 is:

$$select_4(store_4(store_4(a_{array}, 0, 0x80001000), 4, 0x80002000), i * 4)$$

where i is the value of the symbolic variable `i`. Therefore, the query generated at line 14 is address-dependent.

Now, let us follow the other path which takes the branch at line 2. Here, we allocate some memory objects at line 3, and then execute the same flow as before. Note, however, that this time the base addresses of the memory objects allocated at line 8 might differ from those in the first path. When that happens, for example, if the base addresses of these two memory objects are $0x80003000$ and $0x80004000$, then the value of `buffer` at line 13 will be:

$$select_4(store_4(store_4(a_{array}, 0, 0x80003000), 4, 0x80004000), i * 4)$$

As a result, the query generated at line 14 will differ from the analogous query generated in the first path.

The only difference between the queries generated in the two paths comes from the address values,[5] due to a different sequence of allocations in each path. Intuitively, since the behavior of the program in Figure 1.6 is agnostic to the allocated base addresses, the queries generated in the two paths are equisatisfiable. However, in the standard memory model, we cannot distinguish address values from non-address values, which makes it hard to detect the relation between the queries generated in the two paths. Therefore, in the second path, we will not be able to reuse the results of the queries solved in the first path, which will hurt the effectiveness of query caching.

To mitigate this problem, we use the relocatable memory model in which concrete base addresses are replaced with symbolic ones (see Section 1.2.1). This way, the symbolic base addresses propagate to the queries, which allows us to distinguish address values from non-address values. This, in turn, helps detecting queries that are identical up to renaming of symbolic addresses. When such queries are detected, and the address spaces of the corresponding symbolic states are identical up to reordering of address intervals, then we can conclude that the queries are equisatisfiable.

[5] Technically, the queries contain different conditions on `z`, but these can be sliced away.

In our experiments on real-world benchmarks, we show that our query caching approach improves the cache utilization and reduces the cost of constraint solving compared to standard query caching.

The results were published in [114] and are discussed in Chapter 4.

```
1 size_t n; // symbolic
2 int z; // symbolic
3
4 char *p = malloc(n);
5 for (int i = 0; i < n; i++) {
6     if (z == 0) {
7         break;
8     }
9     if (i > 0) {
10        p[i] = i;
11    }
12 }
```

Figure 1.7: A simple example with a symbolic-size allocation.

1.2.3 Bounded Symbolic-Size Model

Another problem with the standard memory model relates to the allocation of memory objects whose size is symbolic. In existing SE engines [39, 85, 103], every memory object has a concrete size. Therefore, if the user wants to analyze a program that has variable-size inputs, e.g., arrays or strings, then the size of these inputs must be fixed before the analysis starts. Moreover, if during the symbolic execution there is an allocation of a symbolic size, then the allocation size must be concretized to a specific concrete value. All this usually leads to an incomplete analysis, i.e., partial path exploration, which then has a negative impact on code coverage and bug finding.

Consider, for example, the code in Figure 1.7. At lines 1 and 2, we define the symbolic variables `n` and `z`, and at line 4, we allocate a buffer of a symbolic size `n`. In the standard memory model, the size of the allocated memory object must be concrete, so the symbolic size expression `n` is concretized. For the sake of the example, suppose that `n` is concretized to 1. After the allocation, we execute the first iteration of the loop at line 5, and fork at line 6. One of the forked symbolic states exits the loop at line 7, and the other one executes the branch at line 5 and exits the loop as well, due to the constraint on `n`. In summary, in the standard memory model, we are able to explore only two paths, while missing the path that reaches line 10. Note that this problem would not have occurred if we chose, for example, $n = 3$, but then, in another situation, we might not cover code that is reachable only when the buffer is too short.

To mitigate this problem, we propose a memory model that supports symbolic-size allocations, and thus enable the analysis of programs with inputs whose size belongs to a range of values. In this memory model, the size of a memory object can be concrete

or symbolic. When the size of a memory object is symbolic, it is bounded by a user-specified *capacity*. As we bound the sizes of memory objects that have a symbolic size, we can still use the linear address space as is done in the standard memory model, without making any other changes. Therefore, our memory model can be easily integrated with existing SE engines.

Now, let us see how the new memory model works with our example. When we reach the allocation at line 4, we allocate a memory object with a symbolic size `n`. Assuming that the capacity is 3, for example, we add to our path constraints a condition that limits the value of the symbolic size `n` by 3. The underlying memory object is comprised of 3 bytes but this is unobservable to the user for whom the size is n. In the new memory model, we are able to explore five paths: Four paths which execute k iterations of the loop ($k = 0, 1, 2, 3$) and exit the loop at line 5, and another path which executes one iteration and exits the loop at line 7.

As can be seen from the example, our memory model makes the analysis more complete, but it also increases the number of forks due to the additional symbolic expressions, i.e., the symbolic sizes. This is particularly noticeable in *size-dependent* loops, i.e., loops where the number of iterations depends on a symbolic size expression. To cope with the amplified path explosion, we propose to use *state merging*. State merging [69, 77] is a technique which allows merging multiple symbolic states together. The merged symbolic state, obtained from merging multiple symbolic states, is encoded using disjunctions and *ite*'s. More specifically, the path constraints of the merged symbolic state are encoded as a *disjunction* over the path constraints of the original symbolic states, and the value of each memory location is encoded as an *ite* expression over the values of that memory location in the original symbolic states. We propose to apply state merging in size-dependent loops, i.e., locations where our memory model introduces additional forking. On one hand, applying state merging this way leads to more complex constraints, but on the other hand, it reduces forking and makes the exploration more similar to the exploration obtained with the standard memory model, as if no additional forking was ever introduced.

In our example, we detect the loop at lines 5-12 as size-dependent, and apply state merging on the resulting symbolic states. When merging symbolic states, they must have the same instruction counter, so we create two merging groups: One group that contains the symbolic states that exit the loop at line 5, and another group that contains

the symbolic state that exits the loop at line 7. This eventually results in two paths instead of the five paths that were obtained with the forking approach when state merging was not applied.

In our experiments on real-world benchmarks, we show that the symbolic-size model indeed helps increase the coverage compared to the concrete-size model (Section 1.1.2). In addition, when the symbolic-size model is used, we show that our state merging approach often outperforms the forking approach in terms of analysis time and code coverage. We also show real-world examples where concretizations caused by symbolic-size allocations lead to missed bugs.

The results were published in [113] and are discussed in Chapter 5.

```
 1 char *strchr(const char *s, int c) {
 2     int i = 0;
 3     while (s[i] != '\0') {
 4         if (s[i] == c) {
 5             return s + i;
 6         }
 7         i++;
 8     }
 9     return NULL;
10 }
11
12 int main() {
13     size_t n; // symbolic
14     char *s = malloc(n + 1); // symbolic
15     s[n] = 0;
16
17     char *p = s;
18     for (int j = 0; j < 2; j++) {
19         p = strchr(p, 'a');
20         if (p != NULL) {
21             p++;
22         } else {
23             break;
24         }
25     }
26
27     return 0;
28 }
```

Figure 1.8: Encoding explosion with a sequence of invocations to `strchr`.

1.2.4 State Merging with Quantifiers

As was mentioned above, state merging [69, 77] is a technique which allows merging multiple symbolic states together. On one hand, state merging helps reducing the forks, but on the other hand, it introduces disjunctive constraints and *ite*'s. The latter usually results in complex constraints which are hard to solve, especially when the number of symbolic states being merged is high.

This happens, for example, in the approach presented in Section 1.2.3, which applies state merging in loops where the symbolic-size model introduces additional forking. That approach indeed helps reducing the forks, however, we observed that in cases where the merged values propagate to the path constraints, the solving time may be high which, in turn, may eliminate the benefit of reducing the number of forks. This is especially noticeable when the number of symbolic states being merged is high, as typically happens when the input capacity is high.

To illustrate this, consider the symbolic execution of the program in Figure 1.8 using the symbolic-size model described in Section 1.2.3. At lines 13-15, we allocate

the symbolic (null-terminated) buffer `s` whose symbolic size is `n + 1`. Suppose that the capacity of the allocated buffer is 4, i.e., $n+1 \leq 4$. In the first iteration of the *for* loop, we call `strchr` at line 19 with the allocated buffer, and suppose that we apply state merging in the loop in `strchr` (lines 3-8). For the sake of the example, consider the three symbolic states that reach line 5, whose path constraints are:

(1) $n \leq 3 \wedge select(s,0) \neq 0 \wedge select(s,0) = 97$

(2) $n \leq 3 \wedge select(s,0) \neq 0 \wedge select(s,0) \neq 97 \wedge select(s,1) \neq 0 \wedge select(s,1) = 97$

(3) $n \leq 3 \wedge select(s,0) \neq 0 \wedge select(s,0) \neq 97 \wedge select(s,1) \neq 0 \wedge select(s,1) \neq 97 \wedge select(s,2) \neq 0 \wedge select(s,2) = 97$

When we merge those three symbolic states, then in the resulting merged symbolic state, the path constraints φ are:

$$\begin{aligned}\varphi \triangleq\ & (n \leq 3 \wedge select(s,0) \neq 0 \wedge select(s,0) = 97) \vee \\ & (n \leq 3 \wedge select(s,0) \neq 0 \wedge select(s,0) \neq 97 \wedge select(s,1) \neq 0 \wedge select(s,1) = 97) \vee \\ & (n \leq 3 \wedge select(s,0) \neq 0 \wedge select(s,0) \neq 97 \wedge select(s,1) \neq 0 \wedge select(s,1) \neq 97 \wedge select(s,2) \neq 0 \wedge select(s,2) = 97)\end{aligned}$$

and the merged value v of the variable `i` is:

$$\begin{aligned}v \triangleq\ & ite(n \leq 3 \wedge select(s,0) \neq 0 \wedge select(s,0) = 97, \\ & \quad 0, \\ & \quad ite(n \leq 3 \wedge select(s,0) \neq 0 \wedge select(s,0) \neq 97 \wedge select(s,1) \neq 0 \wedge select(s,1) = 97, \\ & \qquad 1, \\ & \qquad 2))\end{aligned}$$

Now, let us see what happens when we continue the execution with the merged symbolic state from above. Recall that we merged the symbolic states in which the character `'a'` is found, so the return value of `strchr` in the first iteration at line 19 is not *null*, so we execute another iteration and call `strchr` again at line 19.

Suppose that we perform another state merging operation, where similarly to the first one, we merge the symbolic states that reach line 5. In the first iteration of the *for* loop, the pointer returned by `strchr` at line 19 was incremented by one at line 21. Therefore, this time, the path constraints in the resulting merged symbolic state are:

$$\begin{aligned}& (\varphi \wedge select(s, v+1) \neq 0 \wedge select(s, v+1) = 97) \vee \\ & (\varphi \wedge select(s, v+1) \neq 0 \wedge select(s, v+1) \neq 97 \wedge select(s, v+2) \neq 0 \wedge select(s, v+2) = 97)\end{aligned}$$

and the merged value of the variable `i` is:

$$\begin{aligned}&ite(\varphi \wedge select(s, v+1) \neq 0 \wedge select(s, v+1) = 97,\\ &\quad 1,\\ &\quad 2)\end{aligned}$$

where φ and v are the merged path constraints and merged value of `i` obtained in the first merging operation, respectively.

One can see that if we increase the number of symbolic states being merged, for example, by setting the capacity of the buffer allocated at line 14 to 100 instead of 4, or increase the number of subsequent invocations to `strchr`, then the encoding of the resulting merged symbolic states will become even more complex. When state merging is applied in real-world programs, the merged values are usually complex and often propagate to the path constraints, which eventually leads to *encoding explosion*.

To mitigate this problem, we propose using a different formula encoding: When the path constraints of the symbolic states being merged share some form of uniformity, we can encode the merged symbolic states using quantifiers, while using less disjunctions and *ite*'s.

For example, in the first merging operation from above, the path constraints in the merged symbolic state can be encoded as:

$$\begin{aligned}&n \leq 3 \wedge\\ &0 \leq k_1 \leq 2 \wedge\\ &\forall i.\ 1 \leq i \leq k_1 \rightarrow (select(s, i-1) \neq 0 \wedge select(s, i-1) \neq 97) \wedge\\ &(select(s, k_1) \neq 0 \wedge select(s, k_1) = 97)\end{aligned}$$

where k_1 is a fresh auxiliary variable. Moreover, when the path constraints are encoded this way, the merged value of `i` can be simplified to k_1. Then, in the second merging operation, the path constraints can be encoded as:

$$\begin{aligned}&n \leq 3 \wedge\\ &0 \leq k_1 \leq 2 \wedge\\ &\forall i.\ 1 \leq i \leq k_1 \rightarrow (select(s, i-1) \neq 0 \wedge select(s, i-1) \neq 97) \wedge\\ &(select(s, k_1) \neq 0 \wedge select(s, k_1) = 97) \wedge\\ &0 \leq k_2 \leq 1 \wedge\\ &\forall i.\ 1 \leq i \leq k_2 \rightarrow (select(s, k_1+i) \neq 0 \wedge select(s, k_1+i) \neq 97) \wedge\\ &(select(s, k_1+k_2+1) \neq 0 \wedge select(s, k_1+k_2+1) = 97)\end{aligned}$$

where k_2 is another fresh auxiliary variable, and the merged value of `i` can be simplified to $k_1 + k_2 + 1$.

By using this encoding, we are able to reduce the encoding explosion. In practice, however, solving the generated quantified queries is often more expensive than solving the analogous quantifier-free queries. To overcome this, we propose a specialized solving procedure that leverages the particular structure of the generated quantified queries. This solving procedure helps to reduce the solving time in many cases.

In our experiments on real-world benchmarks, we show that our approach often leads to significant performance gains compared to standard state merging and standard symbolic execution.

The results were published in [115] and are discussed in Chapter 6.

1.2.5 Artifacts Availability

To facilitate the spread and application of the ideas presented here, we provide publicly available tools and replication packages.

Outline. The rest of the thesis is organized as follows: Chapters 3 to 6 describe in detail the results presented in Sections 1.2.1 to 1.2.4, respectively. Chapter 7 discusses other approaches related to this thesis. Chapter 8 summarizes the contributions of this thesis. Chapter 9 presents future research directions.

Chapter 2

Preliminaries

2.1 Logical Notations

We encode symbolic path constraints and memory contents in first-order logic modulo theories using *formulas* and *terms*, respectively. A term is either a constant, a variable, or an application of a function to terms.[1] A formula is either an application of a predicate symbol to terms or obtained by applying boolean connectives ($\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$) or quantifiers ($\forall$, $\exists$) to formulas.

Let φ and φ' be formulas and m a model. We write $m \models \varphi$ to denote that m is a model of φ. We write $\varphi \models \varphi'$ to denote that every model of φ is a model of φ'. We write $\varphi \equiv \varphi'$ to denote that $m \models \varphi$ if and only if $m \models \varphi'$ for any model m. We write $\varphi \doteq \varphi'$ to denote that φ and φ' are syntactically equal.

Let t and t' be terms and m a model. We denote by $m(t)$ the value assigned by m to t. We write $t \equiv t'$ to denote that $m(t) = m(t')$ for any model m.

We use the following functions from the standard theory of arrays [59]: $K(c)$ returns an array whose cells are initialized to c, $\mathit{select}(a, i)$ returns the value of the i-th cell of a, and $\mathit{store}(a, i, v)$ returns a new array obtained from a by replacing the contents of the i-th cell with v. In some cases, we write $a[e]$ as a shorthand for $\mathit{select}(a, e)$.

We use $\triangleq$ to denote definitions.

[1] In our examples, we use the functions from the bit-vector and array theories [59].

2.2 Standard Memory Model

In modern SE engines, e.g., KLEE [39] and Manticore [85], a memory object is represented as a tuple:

$$(b, s, a) \in N^+ \times N^+ \times A$$

where b is a concrete base address, s is a concrete size, and a is an SMT array that tracks the values written to the memory object.[2] Given a memory object mo, we denote its base address, size, and SMT array by $mo.addr$, $mo.size$, and $mo.array$, respectively.

The address space is represented as a set of *non-overlapping* memory objects, i.e., every memory object has its own unique address interval which does not intersect with address intervals of other memory objects: For every two distinct memory objects (b_1, s_1, a_1) and (b_2, s_2, a_2), it holds that:

$$b_1 > b_2 + s_2 \ \vee \ b_2 > b_1 + s_1$$

This *non-overlapping* property reflects the fact that different memory objects are located at different parts of the memory, and enables identifying memory objects by addresses, i.e., a concrete address can belong to at most one memory object. Thus, when the program accesses a concrete address, the SE engine can determine which SMT array represents the content at that address and act accordingly.

When a pointer p is accessed, the SE engine needs first to resolve it, i.e., find the memory objects that p may point to. Given a memory object $mo \triangleq (b, s, a)$, the SE engine determines if p may point to mo by checking if the following *resolution* query is satisfiable (w.r.t. the path constraints):

$$b \leq p < b + s$$

When p may point to mo, the offset i with which the memory object is accessed is given by: $p - b$. The access operations, i.e., *read* and *write*, are encoded using the *select* and *store* operators from array theory [59], respectively. When the byte at offset i of mo is read, its value is expressed by $select(a, i)$. When a value v is written at offset i of mo, the SMT array a is replaced by a new SMT array expressed by $store(a, i, v)$.[3]

[2] SMT arrays are in fact unbounded, but the SE engine records the allocated size and never accesses cells beyond it.

[3] SE engines use an optimized representation when a memory object is accessed only with concrete

To detect out-of-bounds memory accesses, i.e., buffer overflows, the SE engine allocates after each memory object in the address space a so-called *red zone*, an unmapped memory region residing between each two consecutive memory objects. We note, however, that the SE engine is not guaranteed to detect all the possible bugs, since the size of the red zone is bounded.

2.3 Symbolic State

A *symbolic state* s consists of (1) an instruction counter $s.ic$, (2) a path constraint $s.pc$, (3) a symbolic store $s.vars$ that associates variables V with symbolic expressions,[4] (4) and a heap $s.heap$ which is a set of memory objects.

In addition, we define the auxiliary function *get-memory-object-by-address*(s, b), which receives a symbolic state s and a base address b, and returns the memory object from $s.heap$ whose base address is b, if such exists, and *null* otherwise.

2.4 Execution Tree

An *execution tree* [76] is a tree where every node n is associated with a symbolic state $n.s$ and a symbolic condition $n.c$ corresponding to the taken branch such that the conditions associated with any two sibling nodes are mutually inconsistent and the condition of the root node is *true*. The execution tree characterizes the analysis of an arbitrary code fragment, which is not necessarily the whole program. The root node corresponds to the symbolic state that reached the entry point of the code fragment, and the leaf nodes correspond to the symbolic states that completed the analysis of the code fragment.

Definition 2.4.1. Let t be an execution tree with a root r. If there is a path from n_1 to n_k in t, then we denote the sequence of nodes on that path by $\pi_t(n_1, n_k)$, and write $\pi_t(n_k)$ when n_1 is the root r. Given a path $\pi_t(n_1, n_k) \triangleq n_1; n_2; ...; n_k$ in t, we define its *tree path condition* (tpc) and *tree path condition tail* ($\overline{tpc}$):

$$tpc_t(n_1, n_k) \triangleq n_1.c \wedge \overline{tpc}_t(n_1, n_k) \qquad \overline{tpc}_t(n_1, n_k) \triangleq \bigwedge_{1<i\leq k} n_i.c$$

(non-symbolic) offsets. For simplicity, we avoid describing this optimization.

[4] Strictly speaking, V contains only the global variables of the program while the memory state also maintains a stack of frames (symbolic stores) that maintain the values of local variables. For simplicity, we elide this detail and assume that $s.vars$ tracks the values of all the variables in the program.

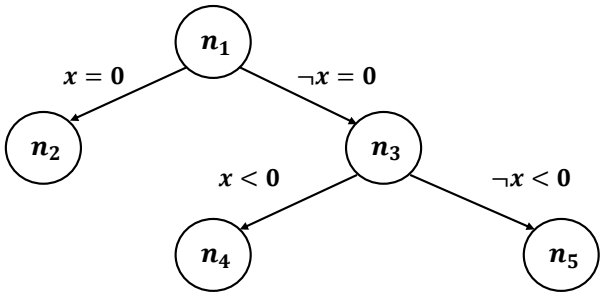


Figure 2.1: The execution tree of the program from Figure 1.1.

If there is no path from n_1 to n_k in t, then we define:

$$tpc_t(n_1, n_k) \triangleq false, \quad \overline{tpc}_t(n_1, n_k) \triangleq false$$

We write $tpc_t(n) \triangleq tpc_t(r, n)$ and $\overline{tpc}_t(n) \triangleq \overline{tpc}_t(r, n)$ as shorthands. We omit the tree subscript when it is clear from the context.

Definition 2.4.2. Given an execution tree t with root r, we say that t is *valid* if the following holds for every node:

$$n.s.pc \equiv r.s.pc \wedge tpc_t(n)$$

Note that $r.s$ is not necessarily the initial symbolic state of the whole program, so $tpc_t(n)$ is a suffix of the path constraints. From now on, we assume that all execution trees are valid.

For example, consider the symbolic execution of the function `get_sign` from Figure 1.1 when the variable `x` has the symbolic value x. The resulting execution tree is depicted in Figure 2.1, where the symbolic condition associated with each node is depicted on the incoming edge of the node. Here, for example, the node n_5 corresponds to the symbolic state that reaches line 9, and:

$$n_5.c \triangleq \neg x < 0, \quad n_5.s.pc \triangleq \neg x = 0 \wedge \neg x < 0$$

Consider the path between n_3 and n_5, then:

$$\begin{aligned} \pi(n_3, n_5) &\triangleq n_3; n_5 \\ tpc(n_3, n_5) &\triangleq \neg x = 0 \wedge \neg x < 0 \\ \overline{tpc}(n_3, n_5) &\triangleq \neg x < 0 \end{aligned}$$

2.5 State Merging

As was mentioned above, a heap is a set of memory objects. Two heaps h_1 and h_2 are *merge-compatible* if for every memory object (b_1, s_1, a_1) in h_1, there exists is a memory object (b_2, s_2, a_2) in h_2 such that $b_1 = b_2$ and $s_1 = s_2$, and vice versa. Symbolic states are *merge-compatible* if: (1) they have the same instruction counter, (2) they contain the same variables in their stores, (3) and their heaps are *merge-compatible*.

Definition 2.5.1. The merged symbolic state resulting from the merging of the merge-compatible symbolic states $\{s_i\}_{i=1}^n$ is the symbolic state s defined as follows:

$$\begin{aligned} &s.ic \triangleq s_1.ic, \\ &s.pc \triangleq \textit{merge-pc}(\{s_i\}_{i=1}^n) \\ &s.vars \triangleq \lambda v \in V.\ \textit{merge-values}(\{s_i\}_{i=1}^n, v) \\ &s.heap \triangleq \{\textit{merge-object}(\{s_i\}_{i=1}^n, b) \mid (b, s, a) \in s_1.heap\} \end{aligned}$$

where *merge-pc*, *merge-values*, and *merge-object* are defined at Algorithm 1.

The *merge-pc* function creates a disjunction over the path constraints of the input symbolic states. The *merge-values* function takes the values of the variable v in each of the input symbolic states, and creates an *ite* expression over those values. Since we assume that the path constraints of the merged symbolic state hold, the right value of the innermost *ite* expression is not guarded by its corresponding condition. The *merge-object* function takes from each of the input symbolic states the memory object whose base address is b, and then merges the arrays of those memory objects in a bytewise fashion. Recall that the heaps are assumed to be merge-compatible, so if one of the symbolic states contains a memory object whose base address is b, then each of the other symbolic states is guaranteed to contain a memory object with that base

Algorithm 1 State merging algorithm.

1: **function** *merge-conditions*$(\{\varphi_i\}_{i=1}^n)$
2: **return** $\bigvee_{i=1}^{n} \varphi_i$
3: **function** *merge-pc*$(\{s_i\}_{i=1}^n)$
4: **return** *merge-conditions*$(\{s_i.pc\}_{i=1}^n)$
5: **function** *merge-values*$(\{\varphi_i\}_{i=1}^n, \{v_i\}_{i=1}^n)$
6: **if** $n = 1$ **then**
7: **return** v_1
8: **else**
9: **return** $ite(\varphi_1, v_1, \textit{merge-values}(\{\varphi_i\}_{i=2}^n, \{v_i\}_{i=2}^n))$
10: **function** *merge-var*$(\{s_i\}_{i=1}^n, v)$
11: **return** *merge-values*$(\{s_i.pc\}_{i=1}^n, \{s_i.vars[v]\}_{i=1}^n)$
12: **function** *merge-object*$(\{s_i\}_{i=1}^n, b)$
13: $\{mo_i\}_{i=1}^n \leftarrow \{\textit{get-memory-object-by-address}(s_i, b)\}_{i=1}^n$
14: $s \leftarrow mo_1.size$
15: $a \leftarrow \textit{new-smt-array}()$
16: **for** $0 \leq j < s$ **do**
17: $e \leftarrow \textit{merge-values}(\{s_i.pc\}_{i=1}^n, \{\textit{select}(mo_i.array, j)\}_{i=1}^n)$
18: $a \leftarrow \textit{store}(a, j, e)$
19: **return** (b, s, a)

address.

State merging is usually applied on a given code fragment, typically a loop or a function. Once the symbolic exploration of the code fragment is complete, the resulting symbolic states are partitioned into (merge-compatible) merging groups. Then, each merging group is transformed into a single merged symbolic state. Finally, the resulting merged symbolic states are added to the state scheduler [39] of the SE engine to continue the exploration.

Chapter 3

Relocatable Addressing Model for Symbolic Execution

This chapter is based on the results published in [112].

3.1 Introduction

In SE engines such as KLEE [39] and ANGR [103], the address space of a symbolic state is modeled using a set of memory objects, where each memory object has a fixed concrete address and its own unique and non-overlapping address interval (Section 2.2). This model is a reasonable implementation choice, but identifying memory objects using concrete addresses is not essential: As long as the non-overlapping property of the memory objects holds, the values of the assigned addresses should not affect the execution.

We propose a new addressing model where the base addresses observable by the symbolic state are symbolic values rather than concrete ones. We preserve the non-overlapping property by maintaining additional *address constraints*, which constrain each symbolic base address to some constant value. This addressing model gives us the ability to *relocate* a given memory object, i.e., modify its underlying address constraint in a way that is *transparent* to the symbolic state. Note that relocating a memory object is not possible under the existing addressing model, since a memory object is allocated at a concrete base address that cannot be modified.

To illustrate the benefits of such addressing model, consider the program from Figure 3.1, inspired by code found in our benchmarks. First, the program executes

an initialization loop, where at each iteration it creates a hash table (line 43) and initializes it with some values (line 45). Then, at line 49 it performs a lookup on one of the tables with a symbolic key k. The lookup function `table_lookup` computes the hash of the input key, and iterates over the nodes of the relevant bucket to find the matching element. When the function `table_lookup` is called at line 49, the value of the pointer `node` at line 18 is symbolic, since it depends on the symbolic hash value which is derived from the symbolic value `k`. Therefore, at line 20, `node->key` dereferences a symbolic pointer.

Symbolic pointers impose a challenge for SE [38, 73]. Each memory object is associated with a unique SMT array, and then queries involving memory objects are translated to constraints over the corresponding SMT arrays. When a symbolic pointer is dereferenced, the SE engine needs to *resolve* that symbolic pointer, i.e., determine the memory objects it can refer to. If the symbolic pointer is resolved to more than one memory object, then we are in the case of *multiple resolution*. Several memory models for handling multiple resolutions have been considered in the past: The *forking model* [39, 89] forks the current symbolic state for each of the resolved memory objects, and in each forked state, it constrains the symbolic pointer to the address interval of the resolved memory object. This approach is relatively efficient in terms of constraint solving, but it may contribute to path explosion due to the forking. The *merging model* [66, 103] creates a disjunction with one disjunct for each of the resolved memory objects. This way, forks are avoided but the path constraints become more complex due to the introduction of disjunctions.

Similarly to the merging model, the *segmented memory model* [73] proposes an approach which avoids forking, but it achieves that by using *array theory* [59] rather than disjunctions. In this model, the memory is split into segments using static pointer analysis [70, 97, 107, 109], such that each pointer (concrete or symbolic) refers to memory objects in a single segment. The segments are computed as follows: First, static pointer analysis is invoked to compute the points-to set of each pointer, i.e., a set of abstract memory objects which are identified by static allocation sites. Then, every two intersecting points-to sets are merged into one points-to set, until a fixpoint is reached, i.e., all the points-to sets are disjoint. A segment is created for each of these disjoint points-to sets, such that all the memory objects associated with that points-to set will be allocated in that segment.

```
 1 typedef struct node_s {
 2     unsigned long data;
 3     unsigned key;
 4     struct node_s *next;
 5 } node_t;
 6 typedef struct {
 7     node_t **buckets;
 8     size_t size;
 9 } table_t;
10
11 void table_init(table_t *t, size_t n) {
12     t->buckets = calloc(n, sizeof(node_t *));
13     t->size = n;
14 }
15
16 node_t *table_lookup(table_t *t, unsigned k) {
17     unsigned long h = hash(&k, sizeof(k)) % t->size;
18     node_t *node = t->buckets[h];
19     while (node != NULL) {
20         if (memcmp(&node->key, &k, sizeof(unsigned)) == 0) {
21             return node;
22         }
23         node = node->next;
24     }
25     return NULL;
26 }
27
28 void table_insert(table_t *t, unsigned k, int data) {
29     if (table_lookup(t, k)) {
30         return;
31     }
32     unsigned long h = hash(&k, sizeof(k)) % t->size;
33     node_t *node = calloc(1, sizeof(node_t));
34     node->key = k;
35     node->data = data;
36     node->next = t->buckets[h];
37     t->buckets[h] = node;
38 }
39
40 int main(int argc, char *argv[]) {
41     table_t tables[3];
42     for (unsigned i = 0; i < 3; i++) {
43         table_init(&tables[i], 300);
44         for (unsigned j = 0; j < 5; j++) {
45             table_insert(&tables[i], j, 7);
46         }
47     }
48     unsigned k; // symbolic
49     table_lookup(&tables[0], k);
50     return 0;
51 }
```

Figure 3.1: Motivating example.

With this approach, forks are avoided when symbolic pointers are encountered, as each pointer is guaranteed to point to exactly one segment. However, this approach is limited by the precision of pointer analysis, and the created segments might contain too many memory objects. A large segment corresponds to a large SMT array which results in more complex constraints. To illustrate the limitations of pointer analysis, consider again the program at Figure 3.1. Pointer analysis cannot distinguish between the different memory objects that are allocated at line 12, as they all have the same static allocation site. As a result, the bucket arrays of all 3 hash tables are allocated in one segment. Similarly, all the nodes allocated at line 33 are allocated in one segment as well. The SMT arrays of both segments are involved in the constraints, as both are accessed with a symbolic offset: The segment of buckets at line 18, and the segment of nodes at line 20. The sizes of these SMT arrays are at least three times bigger than those created by the forking model, due to merging of spurious memory objects. Therefore, despite of the reduction in the number of paths, the forking model still outperforms the segmented memory model in this case.

With our addressing model, we can dynamically relocate a memory object, in a way that is transparent to the symbolic state. Instead of determining the segments ahead of time, we create them on demand: If a symbolic pointer is resolved to multiple memory objects, we create a new segment and relocate the resolved memory objects to that segment. Now, a segment will contain only memory objects that can be pointed to by a given symbolic pointer, without any spurious ones. At line 49, the function `table_lookup` receives the first hash table as an argument, so the symbolic pointer at line 18 is resolved to a single memory object, the bucket array of the first hash table, which does not require creating a segment. The symbolic pointer at line 20 is resolved to nodes from the first hash table only, so the created segment does not contain nodes from the other two hash tables. This way, we are able to avoid forking while creating smaller SMT arrays, which allows us to outperform both the segmented and the forking memory models.

Another challenge arising from the Program in Figure 3.1 relates to solving *array-theory* [59] constraints. An array access with a concrete offset can be handled similarly to scalar variables, but accessing an array with a symbolic offset is more challenging, as the symbolic offset can refer to multiple locations in the array. In that case, the accessed value is expressed using an SMT formula over arrays, which creates a variable for each

offset of the array. Such formulas are hard to solve, especially with large arrays, thus hindering symbolic execution. The hash tables created at line 43 are initialized with a bucket array of 300 entries, therefore the size of its corresponding SMT array is 2400 bytes. When calling `table_lookup` at line 49, the bucket is accessed with a symbolic offset at line 18, which triggers the usage of array theory. This constraint propagates into the path constraints that are created later during execution, thus slowing down the exploration.

With our addressing model, when a big enough memory object is accessed with a symbolic offset, we can relocate that memory object and split it into several smaller adjacent memory objects. For example, the symbolic pointer at line 18 was resolved to exactly one memory object, the bucket array of the hash table. Now, when this memory object is relocated and split, that symbolic pointer is resolved to multiple memory objects with smaller SMT arrays. In this case, despite of having more explored paths due to additional multiple resolutions, the reduced complexity of the constraints eventually results in faster exploration.

Main contributions:

1. We propose a new addressing model, that allows *seamless* and *dynamic* relocation of memory objects during symbolic execution.

2. We provide an implementation based on KLEE [39], a state-of-the-art symbolic executor, which we make available as open source.[1]

3. We show the benefits of our addressing model in two scenarios: improving the segmented memory model, and reducing the cost of solving array-theory constraints with large arrays.

Outline. In Section 3.2, we present our relocatable addressing model and its applications. In Sections 3.3 and 3.4, we discuss our implementation and evaluation, respectively.

3.2 Proposed Addressing Model

In this section, we present our relocatable addressing model and its applications.

[1] https://davidtr1037.github.io/ram/

3.2.1 Relocatable Addressing Model

We propose a new addressing model, where the base address of a memory object is a symbolic value, rather than a concrete one. As before, the program's address space is represented as a set of *memory objects*:

$$mo \triangleq (\beta, s, a) \in 2^{E \times N^+ \times A}$$

However, the base address of a memory object is now a symbolic value $\beta \in E$ and not a concrete value.[2] We enforce the non-overlapping property using *hidden* concrete base addresses: Whenever the program allocates a memory object, we create an *address pair* which consists of two values: a symbolic one β and a concrete value c. The concrete value c is used to ensure that the allocated memory objects do not overlap in the same way that is done in the existing model. The symbolic value β is the value that propagates to the symbolic state.

We maintain the correlation between the symbolic base addresses and the concrete ones using *address constraints* (AC). These constraints record equalities of the form:

$$\beta = e$$

where e is an expression over the hidden concrete base addresses and the symbolic ones. The address constraints are separate from the path constraints, and they are only used when we pass a query to the SMT solver. In this model, the base addresses are symbolic, and any other expression might depend on these symbolic values, which are not constrained by the path constraints. Therefore, we substitute the address constraints before passing an expression e to the SMT solver, i.e., obtain $e[e_i/\beta_i]$ (for each address constraint $\beta_i = e_i$).

Remark Another way to preserve the non-overlapping property is to extend the path constraints of the symbolic state with appropriate constraints over the values of the symbolic base addresses. If we have memory objects $(\beta_1, s_1, a_1), ..., (\beta_n, s_n, a_n)$, then we could have used the following constraints:

$$\forall i, j. \quad i \neq j \rightarrow [\beta_i, \beta_i + s_i) \cap [\beta_j, \beta_j + s_j) = \emptyset.$$

[2] E is the domain of symbolic variables.

```
 1 #define N (2)
 2
 3 char **array = calloc(N, sizeof(char *));
 4 for (unsigned int i = 0; i < N; i++) {
 5     array[i] = calloc(256, 1);
 6 }
 7
 8 unsigned int i; // symbolic, i < 2
 9 unsigned int j; // symbolic, j < 100
10 if (array[i][j] == 1) {
11     // do something...
12 }
```

Figure 3.2: A simple program allocating a two dimensional matrix using an array of pointers and multiple buffers.

However, we found this approach not scalable, as it requires additional constraints (quadratic w.r.t. the number of memory objects in the address space), thus making constraint solving harder.

To illustrate our new addressing model, consider the program from Figure 3.2. When the array of pointers is allocated at line 3, we do the following: Assuming that a pointer size is 4 bytes, we first create a new memory object $mo_1 \triangleq (\beta_1, 8, a_1)$ with an address pair (β_1, c_1), and add mo_1 to the address space. Then, we add a new address constraint $\beta_1 = c_1$, and the symbolic value β_1 is assigned to be the value of the local variable `array`. We note that c_1 is chosen such that the address interval $[c_1, c_1 + 8)$ does not intersect with address intervals of existing memory objects. If the memory objects allocated at line 5 are mo_2 and mo_3 with the corresponding address pairs (β_2, c_2) and (β_3, c_3), then before executing line 10 the address space consists of:

$$\{mo_1, mo_2, mo_3\}$$

and the address constraints are:

$$\{\beta_1 = c_1, \beta_2 = c_2, \beta_3 = c_3\}$$

At line 10, where `array` is accessed with the symbolic offset i, the value of `a[i]` is a *select* expression:

$$select_4(store_4(store_4(a_1, 0, \beta_2), 4, \beta_3), i * 4)$$

This symbolic pointer has to be resolved using the SMT solver, and to do so we first

substitute our address constraints, and then the actual expression passed to the SMT solver is:

$$select_4(store_4(store_4(a_1, 0, c_2), 4, c_3), i * 4)$$

With this model, memory objects can be now seamlessly relocated. Suppose that after the loop at line 6 we want to relocate the memory object mo_2 to a new address. This is achieved by the following steps: First, we allocate a new memory object mo_4 with an address pair (β_4, c_4) of the same size as mo_2, and copy the contents of mo_2 to mo_4. Second, we update the address space by removing mo_2 and adding mo_4. Finally, we modify the address constraint $\beta_2 = c_2$ to be $\beta_2 = c_4$. After the relocation, the expression obtained by the symbolic read `a[i]` at line 10 is the same as before:

$$select_4(store_4(store_4(a_1, 0, \beta_2), 4, \beta_3), i * 4)$$

But now, the substituted expression that will be passed to the SMT solver is different:

$$select_4(store_4(store_4(a_1, 0, c_4), 4, c_3), i * 4)$$

In this addressing model the address values observable by the symbolic state are always symbolic, but those passed to the SMT solver are concrete, which allows efficient constraint solving. Using this addressing model we can perform *dynamically* two operations which are not possible with the existing model: merging multiple memory objects into one segment (Section 3.2.3), and splitting a memory object to multiple smaller memory objects (Section 3.2.4).

3.2.2 Limitations

Our relocatable addressing model is applicable for *well-behaved* programs where the actual address of a memory object should not affect the behavior of the program. In particular, the program should only compare pointers of different memory objects for equality, as required by the C standard, and avoid using fixed addresses besides *null*.

3.2.3 Application: Inter-object Partitioning

In this section, we show how the relocatable addressing model can be used to dynamically merge the representations of several memory objects into a single memory

object, thus reducing the cost of forking when symbolic pointers are dereferenced.

3.2.3.1 Segmented Memory Model

Symbolic pointers impose a challenge for symbolic execution [38, 73]. Since a symbolic pointer can potentially refer to multiple memory objects, the SE engine first needs to *resolve* the pointer, i.e., find all the memory objects to which the pointer could refer to, such that the right SMT arrays can be referenced. The case of multiple resolution, where we have more than one resolved memory object, is especially challenging, and several approaches have been proposed in the past: *forking model*, *merging model*, and *segmented memory model* (Section 3.1).

In the *segmented memory model*, memory objects pointed to by any pointer (symbolic or concrete) are guaranteed to reside in exactly one segment, which makes the process of symbolic pointer resolution much more efficient. However, this approach is limited by the precision of pointer analysis, and the computed segments might be too large in complex programs. A larger segment results in a larger SMT array, thus affecting the performance of the SMT solver.

3.2.3.2 Dynamically Segmented Memory Model

Instead of conservatively computing the segments ahead of time using pointer analysis, we propose a dynamic memory partitioning strategy that creates the segments on the fly using our relocatable addressing model. When we encounter a symbolic pointer that refers to multiple memory objects $mo_1, ..., mo_n$ where:

$$mo_i = (\beta_i, s_i, a_i)$$

we create a new segment (a memory object of an appropriate size), and relocate all these memory objects to this segment. First, we allocate a new segment (β_s, s_s, a_s) with an address pair (β_s, c_s), such that $s_s = \sum_i s_i$. Then we copy the contents of the memory objects $mo_1, ..., mo_n$ into that segment, such that after the copy it holds that:

$$\forall j. \quad 0 \leq j < s_i \rightarrow select(a_s, o_i + j) = select(a_i, j)$$

$$\text{where} \quad o_i = \sum_{k<i} s_k$$

Finally, we remove from the address space the memory objects $mo_1, ..., mo_n$, and the address constraints on β_i are updated to:

$$\beta_i = \beta_s + o_i$$

In the example from Figure 3.2, the symbolic pointer obtained by reading `a[i]` (at line 10) is resolved to two memory objects:

$$mo_2 = (\beta_2, 256, a_2), \;\; mo_3 = (\beta_3, 256, a_3)$$

We then create a new segment $mo_4 = (\beta_4, 512, a_4)$ with the address pair (β_4, c_4), add mo_4 to the address space, and add the address constraint $\beta_4 = c_4$. Then we remove from the address space mo_2 and mo_3, and update the address constraints of β_2 and β_3 to:

$$\beta_2 = \beta_4, \quad \beta_3 = \beta_4 + 256$$

Finally, we copy the contents of mo_2 and mo_3 to the appropriate locations in mo_4. After this transformation, our symbolic pointer is resolved to only one memory object (mo_4), thus reducing the number of forks. Note that with this approach, the segments that we dynamically create do not contain redundant memory objects, those that are not pointed to by the symbolic pointer. We merge only the memory objects that were resolved using the SMT solver, thus reducing the size of the created segments.

The above transformation is graphically depicted in Figure 3.3. The state of the memory is shown before and after the transformation. Note that the contents of the memory object associated with `array` are not affected by the transformation. In addition, note that the transformation does not create a red zone (Section 2.2) between mo_2 and mo_3, which may result in undetected out-of-bounds memory accesses. This, however, can be addressed by creating an internal red zone within the segment itself.

3.2.3.3 Optimizations

The segments created using our approach are guaranteed not to be larger than the segments created by the segmented memory model [73]. However, the resolution process with our approach is more expensive, as a symbolic pointer still may point to multiple memory objects, before those are merged into a single segment. The array-theory

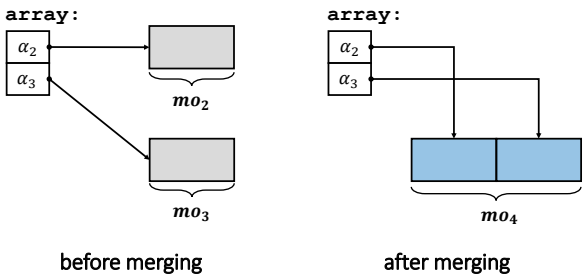


Figure 3.3: Merging multiple memory objects into a single segment.

constraints which are added due to the merging of memory objects are harder to solve, which makes constraint solving and symbolic pointer resolution more expensive. To address these issues, we propose several optimizations.

Context Based Resolution Resolving symbolic pointers is a challenging task, as a symbolic pointer may refer to multiple memory objects. To determine these memory objects, SE engines (such as KLEE) scan the entire memory, and generate resolution queries for the scanned memory objects. The dependence on the SMT solver makes this process expensive, especially when the number of memory objects is high.

We observed that the resolution process can be optimized when using a context abstraction of the allocated memory objects. When a memory object is allocated, its k-context abstraction is obtained by the calling instructions of the last k stack frames, including the current instruction. Once we learn the contexts of the resolved memory objects at a given location (instruction), we can use that information to speed up the resolution process at the next time we have a resolution at that location. When memory objects are scanned during the resolution process, a memory object whose context is not one of the recorded contexts will be skipped, that is, a query will not be sent to the SMT solver. Once the resolved memory objects are merged into a new segment (as described in Section 3.2.3.2), we can check the completeness of the resolution process with a single query that checks if the symbolic pointer *must* point to the newly created segment. If that's not the case, we fallback to the default resolution mechanism. Note that applying context-based resolution in the forking model is not beneficial, as checking completeness requires scanning the entire memory.

Reusing Segments SE engines use various heuristics for optimizing constraint solving. One of the key heuristics used in KLEE is query caching [38, 39], which

associates query expressions to their satisfiability results. Consider again the program from Figure 3.2, and suppose that two symbolic states execute the branch instruction at line 10. With the dynamically segmented memory model, when the first state executes the branch, the resolved memory objects pointed to by `a[i][j]` are merged to a new segment. Suppose that the memory object allocated at line 3 is (β_1, s_1, a_1), and the created segment is $(\beta_2, 512, a_2)$ with the address pair (β_2, c_2). In that case, the expression corresponding to the branch condition `a[i][j] == 1` after substituting the address constraints will be:

$$select(a_2, (select_4(a_1, i * 4) + j) - c_2) = 1$$

Similarly, if in the second symbolic state the created segment is $(\beta_3, 512, a_3)$, then the expression for the same condition will be:

$$select(a_3, (select_4(a_1, i * 4) + j) - c_3) = 1$$

The query caching is perform *syntactically* on the expression level, therefore the second symbolic state will have a cache miss for this query and will invoke the SMT solver.

To handle this issue we attempt to reuse previously allocated segments. If at a program location L, a symbolic pointer was resolved to memory objects $\{mo_i\}_{i=1}^n$ that were merged to a segment (β_s, s_s, a_s) whose address pair is (β_s, c_s), then we record the mapping between the tuples $(L, \{mo_i\}_{i=1}^n)$ and (c_s, a_s). If later another symbolic state resolves a symbolic pointer at program location L to the same set of memory objects $\{mo_i\}_{i=1}^n$, the created segment will be (β'_s, s_s, a_s) with the address pair (β'_s, c_s).

This way, when the second symbolic state performs the merge at line 10, its address pair will be (β_3, c_2), and its SMT array will be a_2. Therefore, the expression of the branch condition will be equal to that of the first symbolic state, which will result in a cache hit. In Chapter 4, we propose a more general approach for increasing cache hits.

3.2.4 Application: Intra-object Partitioning

In this section, we show how the relocatable addressing model can dynamically transform the memory state such that a single memory object allocated by the program can be represented by several adjacent smaller memory objects, thus reducing the size of the SMT arrays and consequently the cost of constraint solving.

When a memory object is accessed with a symbolic offset, the resulting values are translated using array theory to *select* and *store* expressions. Solving array-theory constraints is usually much harder than regular bit-vector constraints, especially when the arrays are large. During symbolic execution, we might have many such queries, which then results in a significant slowdown.

To make constraint solving more efficient, we attempt to reduce the size of large SMT arrays by dynamically splitting their corresponding memory objects into smaller ones. We split only memory objects which were accessed with a symbolic offset, as array theory will not be used for memory objects that are accessed only with concrete offsets.

The split transformation of a memory object $mo \triangleq (\beta, s, a)$ with an address pair (β, c) works as follows: We allocate n new memory objects $\{mo_i\}_{i=1}^{n}$ with address pairs $\{(\beta_i, c_i)\}_{i=1}^{n}$, such that their addresses are consecutive:

$$\forall i. \quad 1 \leq i < n \rightarrow c_{i+1} = c_i + s_i$$

and add the address constraints $\{\beta_i = c_i\}_{i=1}^{n}$. We initialize the contents of each memory object $mo_i \triangleq (\beta_i, s_i, a_i)$ using mo such that:

$$\forall j. \quad 0 \leq j < s_i \rightarrow select(a_i, j) = select(a, o_i + j)$$

$$\text{where} \quad o_i = \sum_{k<i} s_k$$

Then we remove from the address space the memory object mo, and update the address constraint on β to $\beta = \beta_1$, the symbolic base address of the first split memory object. The sizes $\{s_i\}_{i=1}^{n}$ of the memory objects $\{mo_i\}_{i=1}^{n}$ are determined according to a given partitioning strategy.

Consider the symbolic execution of the program from Figure 3.2 under the forking model. Assuming that the memory object allocated at line 3 is $(\beta_1, 8, a_1)$, the expression of the pointer from which the value `a[i][j]` is read is:

$$select_4(a_1, i * 4) + j$$

This symbolic pointer is resolved to the two memory objects allocated at line 5, namely mo_2 and mo_3, and the symbolic state is forked. If we continue the execution with the

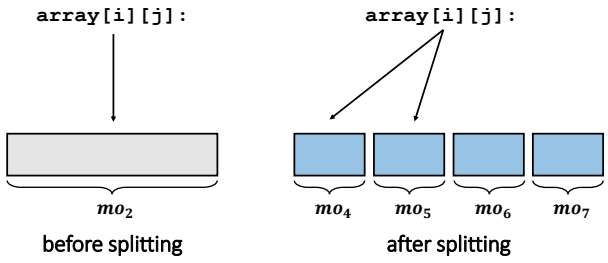


Figure 3.4: Splitting a memory object into adjacent smaller memory objects.

symbolic state which constrains the symbolic pointer to the memory object:

$$mo_2 = (\beta_2, 256, a_2)$$

as depicted at the left part of Figure 3.4, then the value of `a[i][j]` would be:

$$select(a_2, select_4(a_1, i * 4) + j - \beta_2)$$

Since this value is read from mo_2 with a symbolic offset, we would like to split mo_2. Suppose that we choose a partitioning strategy that splits a given memory object into n memory objects of equal size. Then for $n = 4$ we split mo_2 into four memory objects: mo_4, mo_5, mo_6, mo_7, as depicted in the right part of Figure 3.4. After the split, we re-execute the load instruction that references the previous symbolic pointer, but now it is resolved to two memory objects, mo_4 and mo_5, due to the constraint $j < 100$.

Assuming that $mo_4 \triangleq (\beta_4, s_4, a_4)$, in the first newly forked state the expression of the value `a[i][j]` is now:

$$select(a_4, select_4(a_1, i * 4) + j - \beta_4)$$

Due to the split we explore an additional path, the one that constrains the symbolic pointer to mo_5, but we get smaller SMT arrays: a_4 and a_5. The size of a_4 is 64, which is four times smaller than the size of a_2, which makes the new expression much easier to solve.

The effect of the split transformation depends on the partitioning strategy: If a memory object is split to smaller memory objects then the SMT arrays are smaller, but the number of resolved memory objects is higher, and therefore the number of

forks is higher as well. If a memory object is split to larger memory objects then the number of resolved memory objects and forks is lower, but the resulting SMT arrays are consequently larger. The partitioning strategy also affects the resolution process which works by scanning the entire memory. We investigate this trade-off in Section 3.4.3.

3.3 Implementation

We implemented our addressing model on top of KLEE [39], configured with LLVM 7.0.0 and STP 2.3.3. We modified KLEE's allocation API to return symbolic base addresses instead of concrete ones, and extended the symbolic state with the address constraints. Our addressing model is actually implemented as a mixed *concrete-symbolic* one, i.e., we allow allocation of memory objects with concrete addresses as well. Obviously, the applications described in Sections 3.2.3 and 3.2.4 cannot be applied to such memory objects.

In our implementation, symbolic base addresses can be assigned to both stack and heap memory objects. By default, we do not assign symbolic base addresses for stack variables, as they are rarely involved in multiple resolutions or array-theory constraints. For technical reasons, we currently do not support global variables with symbolic base addresses, but this can be solved by automatically rewriting the program such that global variables would be allocated on the heap upon the program's startup.

When a memory object is split, we need to ensure that reads and writes to primitive fields are performed within the bounds of a single memory object. We assume that struct fields are aligned to 8 bytes, so the size of each split memory object must be aligned to 8 bytes as well.

3.4 Evaluation

We perform several experiments in our evaluation: In Section 3.4.1, we empirically validate the correctness of our implementation. In Sections 3.4.2 and 3.4.3 respectively, we show the benefits of our addressing model when applied in the context of inter-object partitioning (dynamic merging) and intra-object partitioning (dynamic splitting).

Experimental Setup. We performed our experiments on an a machine running Ubuntu 16.04, equipped with an Intel i7-6700 processor and 32GB of RAM.

Table 3.1: The benchmarks used throughout the evaluation, with their versions and number of source code lines (SLOC).

Benchmark	Version	SLOC
m4	1.4.18	80K
make	4.2	28K
SQLite	3.21	127K
apr	1.6.3	60K
gas	2.31.1	266K
libxml2	2.9.8	197K
coreutils	8.31	188K

Benchmarks. The benchmarks used in our evaluation are listed in Table 3.1. *GNU m4* [81] is a macro processor included in most Unix-based systems. *GNU Make* [82] is a tool which controls the generation of executables and other non-source files, also widely used in Unix-based systems. *SQLite* [108] is one of the most popular SQL database libraries in the world. *Apache Portable Runtime* [23] (APR) is a library used by the Apache HTTP server that provides cross-platform functionality for memory allocation, file operations, containers, and networking. *GNU Assembler* [60], commonly known as *gas*, is the assembler used by the GNU project, and is the default back-end of GCC. The *libxml2* [6] library is a XML parser and toolkit developed for the Gnome project. *GNU Coreutils* [63] is a collection of utilities for file, text, and shell manipulation.

3.4.1 Empirical Validation

In this experiment, we empirically validate the correctness of the implementation of our addressing model. To do so, we validate that the existing addressing model (vanilla KLEE) and our addressing model are consistent in terms of path exploration. Here we use our addressing model without applying the merging (Section 3.2.3) and splitting (Section 3.2.4) operations, therefore the number of explored paths in both models is expected to be identical. For this experiment, we used the programs listed in Table 3.1, where in *coreutils* we selected 15 programs which behave deterministically across multiple runs.

For each program, we proceed with the following evaluation process: First, we run KLEE with its default addressing model for roughly one hour, and record the number of executed instructions. Then, we run KLEE with our addressing model up to the number of recorded instructions. Finally, we validate that the number of paths explored by both

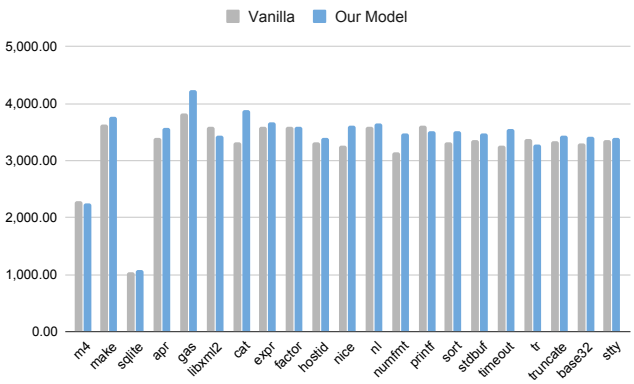


Figure 3.5: The termination times in *seconds* for each program with the existing addressing model (**Vanilla**) and our addressing model (**Our Model**).

addressing models is identical. To enforce determinism, we ran each program with the DFS search heuristic and the deterministic memory allocator.

Our experiments confirmed that the number of explored paths with both addressing models is indeed the same. We also measured the runtime overhead induced by our addressing model, which comes from substituting expressions and maintaining additional symbolic values for address expressions, as described in Section 3.2.1. Figure 3.5 shows the termination time (in seconds) for each program with the two addressing models. For the programs we tested, the maximum runtime overhead was 16% (in *cat* from *coreutils*), and the average runtime overhead was 4%.

3.4.2 Inter-object Partitioning

In this experiment we compare the performance of our dynamically segmented memory model (*DSMM*), the segmented memory model (*SMM*) proposed in [73], and the forking model (*FMM*) used in vanilla KLEE. We perform the same experiment as in [73] (Figures 6,7,8,9 from [73]). The benchmarks of this experiment are: *m4*, *make*, *SQLite*, and *apr*[3]. These programs use hash tables with symbolic keys, which eventually results in symbolic pointers with multiple resolutions. Programs in which symbolic pointers with multiple resolutions are few (*gas*) or absent (*libxml2*, and *coreutils*) are not used in this experiment. The impact of our addressing model on the runtime overhead for these programs is discussed in Section 3.4.1.

We run each program with a timeout of 24 hours using three search heuristics (DFS,

[3]In *SQLite* we disabled the *counter-example caching* query optimization, as it lead to inconsistent termination times. For *SMM*, the result was timeout with and without this optimization. Note that this optimization is different from the *query caching* discussed in Section 3.2.3.3.

Table 3.2: Maximum segment size in bytes.

Program	**Max. Size**	
	SMM	*DSMM*
m4	2753	1008
make	7574	1776
SQLite	17604	528
apr	8316	240

Table 3.3: Termination time in *hh:mm* or *TO* (timeout) and memory usage in *GB* or *OOM* (out-of-memory) with different memory models: *FMM*, *SMM* and *DSMM*.

		Time			**Memory**		
	Search	*FMM*	*SMM*	*DSMM*	*FMM*	*SMM*	*DSMM*
m4	DFS	21:03	01:04	00:26	0.1	0.2	0.3
	BFS	09:32	01:04	00:30	*OOM*	0.3	0.4
	Default	09:41	01:10	00:35	*OOM*	0.5	0.5
make	DFS	06:35	22:49	03:35	1.6	0.8	0.5
	BFS	06:33	23:03	03:34	1.6	0.8	0.6
	Default	07:27	23:04	03:38	1.5	0.8	0.4
SQLite	DFS	00:18	T.O.	01:36	0.2	0.8	0.4
	BFS	00:18	T.O.	01:34	0.3	0.7	0.4
	Default	00:18	T.O.	01:36	0.2	0.7	0.5
apr	DFS	01:01	00:07	00:19	0.1	0.1	0.1
	BFS	00:59	00:07	00:19	0.1	0.1	0.1
	Default	00:57	00:07	00:19	0.1	0.1	0.1

BFS and KLEE's default search heuristic) with the deterministic memory allocator, and measure the termination time and the memory consumption with each memory model. *DSMM* is run with the optimizations described in Section 3.2.3.3.

Table 3.2 shows the maximum segment sizes for both *SMM* and *DSMM*. The maximum segment size created using our approach is reduced on average by 83%, where the reduction is 63% in *m4*, 77% in *make*, 97% in *SQLite*, and 97% in *apr*. The sizes of the created segments are crucial for the performance, as the sizes of their corresponding SMT arrays affect the complexity of constraint solving.

Table 3.3 shows for each benchmark and search heuristic the termination time and the memory consumption obtained with each memory model. We first discuss the performance comparison between *SMM* and *DSMM*, and then discuss the performance of *FMM* compared to the segmentation-based memory models (*SMM* and *DSMM*).

In *m4*, *DSMM* achieves an average speedup of 2.2× compared to *SMM*, with a

slightly higher memory usage. In *make*, *DSMM* achieves an average speedup of 6.3× compared to *SMM*, and the memory usage is roughly the same. In *SQLite*, *SMM* does not terminate before the 24 hours time limit (with all search heuristics), and *DSMM* achieves an average speedup of at least 14.2×. The memory usage is roughly the same with both approaches in this case. In *apr*, the memory usage is equally low in both approaches, but in terms of termination time, this is the only case where *DSMM* performs worse. As was mentioned above, the maximum segment size with *DSMM* is significantly smaller, but *SMM* is still 2.3× faster on overage. The program allocates several small memory objects using *libc*'s standard allocation API, and some other memory objects using a custom pool allocator, that internally uses an array of 8192 bytes. During the symbolic execution of the program, the SMT array associated with the large array (of the pool allocator) is involved in the queries, which slows down the exploration. In the case of *SMM*, the large array and the other small memory objects are merged into one segment. With *DSMM*, some of the small memory objects are dynamically merged into one segment, but the large array remains untouched. In both approaches, we have a large array of roughly the same size which is involved in the constraints, thus slowing down the SMT solver. The advantage of *DSMM* is having smaller segments, but symbolic pointers are still needed to be resolved (possibly to multiple memory objects), which leads to higher resolution time. In *SMM*, a symbolic pointer is guaranteed to point to a single segment, so the resolution process is less costly. In this case, the cost of resolution with *DSMM* is indeed higher, as each resolution query involves the large array that was mentioned before. In fact, the resolution process takes roughly 50% of the total execution time, which explains the extra time required for *DSMM* to terminate.

When comparing the performance of *FMM* with the segmentation-based memory models (*SMM* and *DSMM*), the results are mixed. In *m4* and *apr*, *FMM* performs significantly slower (with all search heuristics) than other memory models. The memory usage in *apr* is basically identical across all memory models, but in *m4* the memory usage with *FMM* reaches the limit of 8GB (with BFS and Default), which leads to an incomplete exploration and early termination. In *make*, *FMM* performs faster than *SMM* but slower than *DSMM*, and its memory usage is higher compared to other memory models. In *SQLite*, *FMM* outperforms *SMM* and *DSMM* in terms of termination time and memory usage.

Table 3.4: The number of resolution queries with different context abstractions.

Program	**Default**	**K-Context**				
		0	1	2	3	4
m4	12050	11434	6920	6920	6808	6808
make	8104	2632	2365	2365	2365	2365
SQLite	9134	6816	6816	3320	3320	1668
apr	96	94	94	94	94	94

> *DSMM* achieves a significant speedup compared to *SMM* and *FMM* in most of the cases.

3.4.2.1 Context-Based Resolution

Here we further investigate the impact of context-based resolution (Section 3.2.3.3). We use the same benchmarks as in Section 3.4.2, and run each of them with the DFS search heuristic and the deterministic memory allocator.

To understand how a given context abstraction affects the process of symbolic pointer resolution, we examine the number of resolution queries. We evaluate two resolution mechanisms: The default resolution mechanism which scans the entire memory, and our context-based resolution with the k-context abstraction, which takes into account the last k calling instructions from the stack trace of a given allocation site (including the current instruction).

Table 3.4 shows the number of resolution queries with different resolution mechanisms: The default resolution mechanism and the context-based resolution with $0 \leq k \leq 4$. The largest reduction occurs when $k = 4$, where the number of queries is decreased by 44% in *m4*, 71% in *make*, 82% in *SQLite*, and 2% in *apr*. We can also see that as k increases, the number of resolution queries (non-strictly) decreases. The impact of k on the reduction rate varies across benchmarks: In *make* and *m4* we have a significant reduction with $k = 0$ and $k = 1$, but increasing k further does not result in a significant improvement. In *SQLite* a reduction of 25% is obtained already with $k = 0$, and increasing k further to 2 and 4 results in a reduction of 64% and 82%, respectively. Compared to other benchmarks, the number of created memory objects in *apr* is relatively small, so the resolution process with the default mechanism is almost optimal. The number of resolution queries is reduced by only two queries for $k = 0$,

and using higher values does not give any better results.

Columns *None* and *K-Context* of Table 3.5 show the termination time with the default resolution mechanism and the context-based resolution (with $k = 4$). In *m4*, *make*, and *SQLite*, the termination time was reduced by 26%, 13%, and 62% respectively. In *apr*, the number of reduced resolution queries was minor, therefore the termination time was not affected. Note that the reduction in termination time depends not only on the number of reduced queries, but also on the relative proportion of resolution time: If the resolution time is already low, then reducing the number of resolution queries is likely to result in a minor improvement. When the resolution time is high, a significant speedup can be achieved even with a minor reduction in the number of resolution queries.

In general, using the highest value for k (or a full-context abstraction with $k = \infty$) is not guaranteed to be beneficial. If the value of k is too high, our context-based resolution might skip relevant memory objects, which will result in an *incomplete* resolution. In that case, it will resort to the default resolution mechanism.

> Context-based resolution reduces the number of resolution queries and speeds up the analysis in most of the cases.

3.4.2.2 Reusing Segments

Here we further investigate the impact of reusing segments (Section 3.2.3.3). We use the same benchmarks as in Section 3.4.2, and run each of them with the DFS search heuristic and the deterministic memory allocator.

This optimization attempts to achieve speedup by improving the query caching, i.e., reducing the number of queries which are actually passed to the SMT solver. The impact of reusing segments can be seen in columns *None* and *RS* of Table 3.5. In *m4* and *apr*, the termination time was reduced by 85% and 17% respectively, while in *make* and *SQLite* the reduction was relatively small with 3% and 8% respectively. As was mentioned before, the benchmarks in this experiment use hash tables with buckets, which are typically implemented using arrays of pointers. In the case of *m4* and *apr*, the hash tables are not modified after their initialization, so the corresponding SMT arrays are identical for all the symbolic states, which allows efficient caching when segments are reused. In the case of *make* and *SQLite*, the hash tables are modified after the

Table 3.5: Termination time in *hh:mm* in different modes: *None*: without any optimizations, *K-Context*: with context-based resolution (for $k = 4$), *RS*: with reusing segments, and *All*: with both optimizations.

Program	**Termination Time**			
	None	*K-Context*	*RS*	*All*
m4	03:34	02:37	00:33	00:26
make	04:14	03:42	04:06	03:35
SQLite	04:16	01:37	03:58	01:36
apr	00:23	00:23	00:19	00:19

initialization, so when different symbolic states update a hash table by adding a new element, the corresponding SMT arrays are different as well, which makes the reuse mechanism less efficient.

> Reusing segments can significantly speedup the analysis.

3.4.3 Intra-object Partitioning

In this experiment, we investigate the impact of the splitting approach (Section 3.2.4) on the termination time and the number of explored paths. We use programs that generate array-theory constraints with large arrays. For each program we compare the results obtained by vanilla KLEE and the splitting approach with different partitioning strategies. When the splitting approach is used with a partitioning strategy P_n, a memory object is split into smaller memory objects of size n. We use a *split threshold* of 300 bytes, i.e., we split only memory objects whose size exceeds that given threshold. We use the DFS search heuristic and the deterministic memory allocator.

The benchmarks in this experiment are: *m4*, *make*, *SQLite*, *apr*, *gas*, and *libxml2*. Similarly to the experiments in Section 3.4.2, we achieve termination by running these programs with a partially symbolic input, except for *libxml2* which is run with a fully symbolic input. In Section 3.4.2, the programs *m4* and *make* (which were taken from [73]) were run with decreased sizes for some of the arrays: In *m4*, the hash table size was decreased using one of the program's command line flags (`-H`), and in *make* some of the arrays were manually patched to have smaller sizes. In this experiment, we restore the default array sizes in order to test the splitting approach with larger arrays.

Table 3.6 shows the termination time and the number of explored paths with both

vanilla KLEE and the splitting approach (with different partitioning strategies).[4] In terms of termination time, the speedup of the splitting approach compared to vanilla KLEE varies between 6.0×-13.4× in *m4*, 1.2×-17.9× in *make*, 0.9×-4.0× in *SQLite*, 13.3×-132.1× in *apr*, 33.6×-43.8× in *gas*, and 1.0×-2.5× in *libxml2*. Nevertheless, there were two cases where the splitting approach performed worse: In *SQLite*, the size of the split memory object is 328 bytes, therefore using P_{512} affects neither the memory objects nor the number of explored paths, with the termination time being higher by 4% due to the overhead incurred by our addressing model. Running *libxml2* with P_{32} resulted in a slightly higher termination time, mainly due to the increased number of explored paths.

Table 3.7 shows the maximum size of a split memory object for each benchmark. The sizes vary between roughly 500KB in *gas* and only 328 bytes in *SQLite*, which shows that the splitting approach can be successfully applied with both large arrays and relatively small ones.

The partitioning strategy used in the splitting approach directly affects the termination time and the number of explored paths. When a memory object is split according to some partitioning strategy, a more refined partitioning will (non-strictly) increase the number of memory objects that a symbolic pointer can point to. Since we are in the forking model, when we decrease n, i.e., refine the partitioning, the number of resolved memory objects with P_n increases together with the number of explored paths. In addition, when the partitioning is more refined, the number of memory objects grows, which may slowdown symbolic pointer resolution. Nevertheless, using a more refined partitioning creates smaller SMT arrays, which makes constraint solving easier. This tradeoff between the complexity of the constraints on one side, and the number of explored paths and the resolution time on the other side, eventually determines the termination time with a given partitioning strategy.

When trying to understand the impact of a given partitioning strategy, we observed two main patterns. When n is decreased in *SQLite* and *apr*, the overhead of forks and resolution remains relatively low and SMT arrays also become smaller, which results in better overall performance. In other benchmarks (*m4*, *make*, *gas*, and *libxml2*), decreasing n toward small values (32) results in a slowdown due to an increased number

[4] In one of the cases, when *make* was analzed with the partitioning strategy P_{32}, we had to use the standard (libc) memory allocator and not the deterministic one used in all other experiments, since the latter does not reuse addresses and ran out of memory.

Table 3.6: Termination time in *hh:mm:ss* and number of explored path with vanilla KLEE and different splitting strategies.

Program	**Mode**	**Time**	**Paths**
m4	Vanilla	00:39:46	82
	P_{512}	00:06:40	82
	P_{256}	00:02:58	101
	P_{128}	00:03:16	145
	P_{64}	00:03:50	257
	P_{32}	00:05:08	485
make	Vanilla	09:04:13	3386
	P_{512}	01:13:50	4488
	P_{256}	00:30:26	6088
	P_{128}	00:39:47	10136
	P_{64}	01:53:15	21304
	P_{32}	07:44:19	55928
SQLite	Vanilla	00:18:38	147
	P_{512}	00:19:26	147
	P_{256}	00:15:21	213
	P_{128}	00:09:46	259
	P_{64}	00:07:26	355
	P_{32}	00:04:38	465
apr	Vanilla	01:01:38	961
	P_{512}	00:04:39	1024
	P_{256}	00:02:21	1225
	P_{128}	00:01:16	1444
	P_{64}	00:00:47	1849
	P_{32}	00:00:28	2025
gas	Vanilla	05:18:26	5
	P_{512}	00:07:16	5
	P_{256}	00:07:25	5
	P_{128}	00:07:43	5
	P_{64}	00:08:23	5
	P_{32}	00:09:29	5
libxml2	Vanilla	01:22:27	4413
	P_{512}	00:33:26	5003
	P_{256}	00:33:38	5230
	P_{128}	00:39:13	5821
	P_{64}	00:59:06	6885
	P_{32}	01:23:00	8718

Table 3.7: Maximum size of split memory objects.

Program	**Size**
m4	4072
make	8192
SQLite	328
apr	8192
gas	524411
libxml2	4096

of explored paths. In *make* and *m4*, increasing n too much toward high values (512) results in a slowdown as well, due to the growing complexity of constraints over larger SMT arrays. The sweet spot value for n lies somewhere between 64 and 256.

> The splitting approach achieves a significant speedup in programs that generate complex array-theory constraints.

Chapter 4

Address-Aware Query Caching for Symbolic Execution

This chapter is based on the results published in [114].

4.1 Introduction

Symbolic execution often spends most of its time on solving queries [87], therefore the effectiveness of the SE engine depends on the effectiveness of its underlying SMT solver. To reduce the cost of constraint solving, SE engines perform their own optimizations before invoking the SMT solver. Among these optimizations are: caching [39, 103, 119], slicing [39, 103, 119], expression rewriting [39, 91], and logical implications [71].

Symbolic executors generate a large number of queries, so one of the most vital optimizations is *query caching*. In this optimization, queries are transformed to a normal form, which is used as a key for maintaining a cache. Various normalization strategies [39, 103, 119] have been proposed, including: canonization, renaming variables, rewriting equalities, and arithmetic simplifications.

Unfortunately, there are still some types of queries that are inefficiently handled by existing query caching approaches. An example for that is *address-dependent* queries, i.e., queries that involve address expressions. Such queries are generated in various SE engines such as KLEE [39], ANGR [103], Manticore [85], and SAGE [56].

To illustrate when such queries are encountered, consider the program in Figure 4.1. When the value of `array[i][j]` is read at line 17, the corresponding expression depends on the content of `array`, as it is accessed with the symbolic offset `i`. The contents of

```
 1 #define N (2)
 2 #define MAGIC (7)
 3
 4 int z; // symbolic
 5 if (z > 0) {
 6     /* allocate objects... */
 7 }
 8
 9 char **array = calloc(N, sizeof(char *));
10 for (unsigned int i = 0; i < N; i++) {
11     array[i] = calloc(N, 1);
12 }
13 array[0][1] = MAGIC;
14
15 unsigned int i; // symbolic, i < N
16 unsigned int j; // symbolic, j < N
17 if (array[i][j] == MAGIC) {
18     /* do something... */
19 }
```

Figure 4.1: Motivating example.

`array` are the address values assigned at line 11, so the query generated for the branch at line 17 is address-dependent. Due to the branch at line 5, the same flow described above is executed again while exploring a different execution path. Depending on the utilized allocation scheme, the addresses assigned at line 11 by the other symbolic state may be different from those assigned by the previously discussed symbolic state. Therefore, the queries generated at line 17 by these two symbolic states would be syntactically different, although they are clearly *equisatisfiable*. Since existing query caching techniques rely on some form of syntactic normalization, an opportunity to reuse the result of the first query for the second one would be missed. Notice that even if each symbolic state had its own local memory allocator, synchronizing the assigned addresses would be difficult: For example, if additional memory objects are allocated at line 6, then the two symbolic states forked at line 5 are likely to produce different allocation sequences.

We introduce a novel query caching technique that can efficiently handle address-dependent queries. Such queries are prevalent in programs that dereference *symbolic pointers*, i.e., pointers whose values depend on the symbolic input. A prolific source for such symbolic pointers are programs where the symbolic input propagates into a data structure indexed by that input, as happens, for example, with hash tables.

At a high level, we utilize the *relocatable addressing model* proposed in Section 3.2.1 to modify the representation of the expressions generated by the SE engine, such that the base addresses returned by the memory allocator are symbolic values. This allows

us to track the propagation of address values to the symbolic states and the resulting queries, and distinguish address expressions from non-address expressions. Using this model, we are able to detect address-dependent queries which are not syntactically equivalent but are nonetheless equisatisfiable, thus improving the cache utilization.

Main contributions:

1. We propose a novel query caching technique, that allows efficient handling of address-dependent queries.
2. We provide a formal proof for the correctness guarantees of our technique.
3. We provide a KLEE-based implementation, which we make available as open-source.[1]
4. In our evaluation, we empiricallly validate the correctness of our technique, and show that it can achieve significant performance gains.

Outline. In Section 4.2, we provide background on standard query caching. In Section 4.3, we present our query caching approach for address-dependent queries. In Sections 4.5 and 4.6, we discuss our implementation and evaluation, respectively.

4.2 Standard Query Caching

To understand why efficiently caching address-dependent constraints is challenging, we first give some background on the existing query caching approaches in modern analysis tools [33, 38, 39, 103, 119].

SE engines [38, 39, 103] and other analysis frameworks [33, 119] use some form of *syntactic* query caching to improve the performance of constraint solving. Each query is transformed to an equivalent normal form according to some syntactic rules, and this normal form is used as a key for maintaining the cache: In a case of a miss, the query is solved using the SMT solver and its result is memoized. Otherwise, the result of a previously solved query is reused without invoking the SMT solver. The normalization is typically achieved via variable renaming, canonization, arithmetic simplification, and equality rewriting.

[1] https://github.com/davidtr1037/klee-aaqc

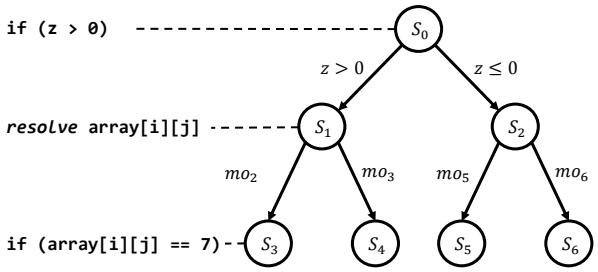


Figure 4.2: The execution tree of the program from Figure 4.1.

Table 4.1: The queries generated at line 17 in different states with the two addressing models. The upper section corresponds to the standard addressing model, and the lower section corresponds to the symbolic addressing model.

State	Symbolic Pointer	Query	AC
s_3	$p_1 \triangleq select_4(store_4(store_4(a_1, 0, 200), 4, 300), i * 4) + j$	$q_1 : i < 2 \wedge j < 2 \wedge 200 \leq p_1 < 202 \wedge select(a_2, p_1 - 200) = 7$	-
s_4		$q_2 : i < 2 \wedge j < 2 \wedge 300 \leq p_1 < 302 \wedge select(a_3, p_1 - 300) = 7$	-
s_5	$p_2 \triangleq select_4(store_4(store_4(a_1, 0, 500), 4, 600), i * 4) + j$	$q_3 : i < 2 \wedge j < 2 \wedge 500 \leq p_2 < 502 \wedge select(a_5, p_2 - 500) = 7$	-
s_6		$q_4 : i < 2 \wedge j < 2 \wedge 600 \leq p_2 < 602 \wedge select(a_6, p_2 - 600) = 7$	-
s_3	$p_3 \triangleq select_4(store_4(store_4(a_4, 0, \beta_2), 4, \beta_3), i * 4) + j$	$q_5 : i < 2 \wedge j < 2 \wedge \beta_2 \leq p_3 < \beta_2 + 2 \wedge select(a_2, p_3 - \beta_2) = 7$	$\beta_2 = 200$
s_4		$q_6 : i < 2 \wedge j < 2 \wedge \beta_3 \leq p_3 < \beta_3 + 2 \wedge select(a_3, p_3 - \beta_3) = 7$	$\beta_3 = 300$
s_5	$p_4 \triangleq select_4(store_4(store_4(a_4, 0, \beta_5), 4, \beta_6), i * 4) + j$	$q_7 : i < 2 \wedge j < 2 \wedge \beta_5 \leq p_4 < \beta_5 + 2 \wedge select(a_5, p_4 - \beta_5) = 7$	$\beta_5 = 500$
s_6		$q_8 : i < 2 \wedge j < 2 \wedge \beta_6 \leq p_4 < \beta_6 + 2 \wedge select(a_6, p_4 - \beta_6) = 7$	$\beta_6 = 600$

For instance, consider the following two queries:

$$x < 1 \ \wedge \ x + y + 4 < 7 \qquad x + z + 1 < 4 \ \wedge \ x < 1$$

These queries are syntactically different, but they can be reduced to the same normal form:

$$v_0 < 1 \wedge v_0 + v_1 < 3$$

Therefore, if one of these queries was already solved, the result of the other one could be deduced later if needed. This query caching mechanism is effective in practice, but does not provide a *complete method* for determining if two formulas are equisatisfiable: There are queries that are equisatisfiable, but which cannot be reduced to the same normal form, as we exemplify in Section 4.3.

4.3 Address-Aware Query Caching

Our technique enables a more efficient caching of address-dependent queries. We achieve that by using the *relocatable addressing model* proposed in Section 3.2.1, which

modifies the representation of expressions in the symbolic state in a way that enables distinguishing address expressions from non-address expressions. In the following section, we show how this model helps to efficiently handle address-dependent queries.

4.3.1 Motivation

To illustrate the need for the relocatable addressing model, consider again the program in Figure 4.1 whose execution tree is shown in Figure 4.2. The analysis starts with the initial state s_0, which executes the symbolic branch at line 5 and forks. Then each of the forked states, s_1 and s_2, allocates a two-dimensional matrix using an array of pointers (line 9), and initializes one of the cells to some constant value `MAGIC` (line 13). Suppose that the memory objects allocated in state s_1 at lines 9 and 11 are:

$$mo_1 \triangleq (100, 8, a_1),\ \ mo_2 \triangleq (200, 2, a_2),\ \ mo_3 \triangleq (300, 2, a_3)$$

where mo_1 corresponds to the array of pointers `array` and the two others correspond to the buffers allocated inside the loop. At line 17, where the value of `array[i][j]` is read in s_1, the value of the accessed pointer is:

$$p_1 \triangleq select_4(store_4(store_4(a_1, 0, 200), 4, 300), i * 4) + j$$

which is also shown under the column *Symbolic Pointer* of Table 4.1. This is a symbolic pointer that can refer to two memory objects: mo_2 and mo_3. As a result, the execution is forked again, resulting in two new symbolic states: s_3 and s_4. When we follow s_3, the symbolic state that resolves p_1 to mo_2, the query generated for the branch condition at line 17 is q_1, which is shown in column *Query* of Table 4.1. Note that this query does not contain the constraint $z > 0$, since the expression of the branch condition at line 17 does not depend on the symbolic value z. This optimization is known as *slicing*, and it is widely used in SE engines [39, 103].

As for s_2, the other symbolic state forked at line 5, suppose that the memory objects allocated at lines 9 and 11 are:

$$mo_4 \triangleq (400, 8, a_4),\ \ mo_5 \triangleq (500, 2, a_5),\ \ mo_6 \triangleq (600, 2, a_6)$$

Similarly, s_5 executes the same flow as s_2, and the query generated for the branch

condition at line 17 is q_3. The concrete base addresses assigned in s_3 and s_5 are different, so the mentioned queries (q_1 and q_3) cannot be reduced to the same normal form. Therefore, standard query caching cannot reuse the result of the first query for the second one.

Note, however, that these two queries (q_1 and q_3) are equisatisfiable, and this is not a coincidence. The query generated at line 17 is *address-agnostic*: For any sequence of allocations occurring on a path that leads to line 17, the generated query is guaranteed to be equisatisfiable to the queries above.

In order to detect equisatisfiable address-dependent queries, we need to know which expressions in the constraints are address expressions. Since the standard addressing model encodes pointer values as integers, detecting these address expressions is hard without additional annotations. However, that can be easily achieved with the relocatable addressing model: In this model, the assigned base addresses are symbolic values, so if the normal form of one query can be obtained from the normal form of another query by renaming the symbolic base addresses, and the sizes of the memory objects corresponding to the matched symbolic base addresses are equal, then these queries are equisatisfiable.

Using the relocatable addressing model in our example, instead of the queries q_1 and q_3, we generate q_5 and q_7 in conjunction with the corresponding address constraints shown in column AC of Table 4.1. Since q_7 can be obtained from q_5 by renaming β_2 to β_5 and β_3 to β_6, and the sizes of mo_2 and mo_3 match those of mo_5 and mo_6, then these two queries can be determined as equisatisfiable.

Table 4.1 shows the query generated at line 17 by each of the four symbolic states, with both the standard addressing model and the symbolic one. A similar equisatisfiability observation can be made regarding the queries q_6 and q_8 that are generated by s_4 and s_6, respectively.

4.3.2 Algorithm

Our algorithm for determining the equisatisfiability of two queries is given in Algorithm 2. We assume that the expressions e_1 and e_2 passed to the function *equi-sat* are represented using the relocatable addressing model, i.e., base addresses are symbolic values. We also assume that these expressions are already in a canonical form, and that they are represented using an abstract syntax tree (AST) with support for: integers,

Algorithm 2 Equisatisfiability Algorithm.

```
1:  function equi-sat(e1, e2, m)
2:      if e1 and e2 are unary expressions
3:          op1(e1') ← e1,  op2(e2') ← e2
4:          return op1 = op2 and equi-sat(e1', e2', m)
5:      ...
6:      if e1 and e2 are atomic symbolic base addresses
7:          s1 ← get-size(e1), s2 ← get-size(e2)
8:          return add-pair(m, e1, e2) and s1 = s2
9:      if e1 and e2 are atomic arrays
10:         K(c1) ← e1,  K(c2) ← e2
11:         return c1 = c2
12:     if e1 and e2 are store expressions
13:         store(a1, i1, v1) ← e1,  store(a2, i2, v2) ← e2
14:         return equi-sat(a1, a2, m) and equi-sat(i1, i2, m) and equi-sat(v1, v2, m)
15:     return false
```

symbolic values, unary and binary operations, *select* and *store* operations, etc.

The algorithm is almost identical to a standard recursive equality checking routine, except for two main cases:

Symbolic Base Addresses. We use the bidirectional map m, to compute a bijection between the symbolic base addresses in e_1 and e_2, if such bijection exists. First, we update at line 8 the map m with the new pair (e_1, e_2) using the function *add-pair*, which returns *true* if the bijection property is preserved, and *false* otherwise. Then, if *add-pair* succeeds, we take the sizes of the memory objects corresponding to e_1 and e_2 (fetched at lines 7), and check their equality.

Arrays. We assume that an atomic array, i.e., an array without *store*'s, is an array whose cells are initialized to some constant (Section 2.1). If e_1 and e_2 are atomic arrays (line 9), then we check if they are initialized with the same constant. In the case of *store* expressions (line 12), we perform a recursive check on the corresponding arrays, indices, and values.

In the context of a given symbolic state, every expression e induces an address space, which is defined by the memory objects whose symbolic base addresses appear in e. If two address spaces contain the same number of memory objects, and for every memory object in one address space, there exists a memory object with the same size in the other address space (and vice versa), then these two address spaces are considered to be *isomorphic*. If *equi-sat* succeeds, then in particular, we establish that the address spaces induced by e_1 and e_2 are isomorphic.

```
1 char *p = malloc(10), *q = malloc(50);
2 if (p > q)
3     ...
4 if (*(p + 100) == *q)
5     ...
```

Figure 4.3: An ill-behaved program due to unsafe pointer arithmetic.

4.3.3 Limitations

The approach described in Section 4.3.1 cannot be applied to queries which are not address-agnostic, i.e., queries where the ordering of the memory objects in the address space affects the satisfiability. Such queries may be generated internally by the SE engine, or when analyzing programs that incur undefined behavior.

Undefined Behavior. Indeed, there are programs whose execution depends on the relationships between the numerical address values. For example, the result of the branch statement at line 2 from Figure 4.3 clearly depends on the allocation scheme implemented by the C standard library. For this reason, the behavior of such statements is commonly considered to be undefined. Note that not all pointer comparisons are necessarily address-dependent. Comparisons such as `p + i < p + j` have well-defined semantics, and comparisons between pointers within the same memory object are commonly used and introduced as part of standard compiler optimizations. However, it is known that checking the presence or absence of undefined behavior in a given program is hard [70].

Similarly, pointer arithmetic can also expose address dependency, as demonstrated in the program from Figure 4.3: The branch condition at line 4 holds, for example, when $p = 100$ and $q = 200$, but it may be false under other address assignments. Again, verifying the absence of out-of-bounds pointer arithmetic is too hard in general [62, 118]. Symbolic executors can detect such bugs under some address assignments, but not all, and definitely cannot prove their absence.

In the presence of such undefined behavior, our query caching approach presented in Section 4.3.1 may exhibit unsoundness or incompleteness. However, we would like to point out that in these cases symbolic executors cannot provide such guarantees anyway, although our approach can exacerbate the problem.

Engine-Internal Queries. The SE engine itself may internally generate queries which are not address-agnostic. For example, when KLEE resolves a symbolic pointer

p, it generates the query $\beta \leq p < \beta + s$ in order to check if p may refer to the memory object (β, s, a). To optimize the search procedure, it generates an additional validity query of the form:

$$p < \beta + s$$

to determine if the search can be completed without scanning additional memory objects.

Clearly, the latter query is not address-agnostic. Let p be the symbolic pointer encountered at line 17 from Figure 4.1:

$$p \triangleq select_4(store_4(store_4(a_1, 0, \beta_2), 4, \beta_3), i * 4) + j$$

If the address constraints are:

$$\beta_2 = 200,\ \beta_3 = 300,\ \beta = 700$$

then the query is valid, which is not the case if $\beta = 100$.

In the case of *engine-internal* queries, the SE engine itself generates the queries, so it is easy to tag those and locally disable the query caching optimization in such cases.

In Section 4.4, we formulate the sufficient conditions under which address-dependent queries generated by the SE engine are guaranteed to be address-agnostic, and prove the correctness of our query caching approach for such queries.

4.4 Correctness

In this section, we justify the query caching approach described in Section 4.3 by arguing that from the satisfiability or unsatifiability of a given query follows the same result under any isomorphic address spaces.

We consider formulas in *array theory* [32, 58], with one-dimensional arrays whose index sort is *Int*, as these are the ones that occur in satisfiability queries during symbolic execution of low-level program representations (e.g., LLVM IR). The theory includes the interpreted function symbols: *select*, *store*, and K (Section 2.1).

Definition 4.4.1. Let L_1 be the language of unquantified formulas in the array theory with two sorts for scalars: *Int* and *Ptr*. The *Int* sort admits all the linear integer

arithmetic operations. The *Ptr* sort admits equality, a special constant *null*, and a pointer arithmetic operator:

$$+ : \mathit{Ptr} \times \mathit{Int} \to \mathit{Ptr}$$

The intended interpretation for *Ptr* will therefore be numeric, and we assume it to be isomorphic to $\mathbb{N}$. In particular, *null* is interpreted as 0, + is interpreted as addition, and we assume that:

$$(p + n) + m \equiv p + (n + m)$$

Ptr terms can be tested only for equality (or inequality), and not for order. In addition, *Ptr* and *Int* terms *cannot* be mixed freely in L_1 formulas, for example, comparing pointers with integers is prohibited.

Definition 4.4.2. An *address space* is a set of disjoint integer intervals, canonically written as $S = \{[c_i, e_i]\}_{1 \le i \le r}$ where $0 < c_i \le e_i$. The size of an interval $[c, e]$ is defined as follows:

$$\big|[c, e]\big| \triangleq e - c + 1$$

We are interested only in models where the interpretation of every uninterpreted constant $p : \mathit{Ptr}$ is either *null* or a base address, i.e., a beginning of an interval. This is expressed in the following definition:

Definition 4.4.3. Let m be a model and S be an address space. We say that m *respects* S if the following holds:

- For every uninterpreted constants $p_1, p_2 : \mathit{Ptr}$, if $p_1 \neq p_2$ then:

  $$m(p_1) = m(p_2) = 0 \text{ or } m(p_1) \neq m(p_2)$$

- For every uninterpreted constant $p : \mathit{Ptr}$, if $m(p) \neq 0$ then there exists an interval $[c, e] \in S$ such that $m(p) = c$.

Definition 4.4.4. We say that a term $p : \mathit{Ptr}$ *respects* an address space S in a model m if the following holds:

- If p is an uninterpreted constant and $m(p) \neq 0$, then there exists an interval $[c, e] \in S$ such that $m(p) = c$.

- If $p \triangleq a + n$, where $a : \mathit{Ptr}$ and $n : \mathit{Int}$, then there exists an interval $[c, e] \in S$ such that $m(a) \in [c, e]$ and $m(a + n) \in [c, e]$.

A formula φ in L_1 *respects* an address space S in a model m, when all its sub-terms of sort Ptr respect it in m.

Definition 4.4.4 is crucial to our treatment of formulas that contain pointer arithmetic operations: It means that whenever such operation occurs in φ, it may not take an address that resides inside one interval and create an address that resides in a different interval.

Definition 4.4.5. By writing $m_1 =_{Int} m_2$ we denote that for every uninterpreted constant $t : Int$,[2] it holds that:

$$m_1(t) = m_2(t)$$

Definition 4.4.6. Let m_1 and m_2 be models and S_1 and S_2 be address spaces. We say that m_1 and m_2 are *consistent* w.r.t. S_1 and S_2 if the following holds:

- For every $p : Ptr$, $m_1(p) = 0 \iff m_2(p) = 0$
- For every $p : Ptr$, if $m_1(p) \in [c_1, e_1]$ and $m_2(p) \in [c_2, e_2]$, where $[c_1, e_1] \in S_1$ and $[c_2, e_2] \in S_2$, then $\big|[c_1, e_1]\big| = \big|[c_2, e_2]\big|$.

Definition 4.4.7. A formula φ in $L \supseteq L_1$ is *address-agnostic* if for every two address spaces S_1 and S_2, and for every two models m_1 and m_2, the following holds:

If:

- $m_1 =_{Int} m_2$
- m_1 and m_2 respect S_1 and S_2, respectively
- m_1 and m_2 are consistent w.r.t. S_1 and S_2
- φ respects S_1 in m_1

then:

- $m_1 \models \varphi \iff m_2 \models \varphi$

Lemma 4.4.8. *If φ_1 and φ_2 are formulas in $L \supseteq L_1$ which are address-agnostic, then $\neg\varphi_1, \varphi_1 \wedge \varphi_2, \varphi_1 \vee \varphi_2$ are address-agnostic as well.*

The proof is in Appendix A.1.1.

[2] We write $t : T$ to denote that the sort of the term t is T.

Lemma 4.4.9. *Every formula φ in L_1 is address-agnostic.*

The proof is in Appendix A.1.2.

Lemma 4.4.9 states that in the context of L_1, the concrete values of interval boundaries in the address space has no effect on the truth value of the formula, and they can be freely rearranged into any other locations. However, L_1 is not expressive enough for our purposes, as it allows construction of pointer values from integers but not vice versa, thus preventing the use of terms such as $select(a, p_1 - p_2)$, which are routinely generated by the SE engine for the representation of pointer dereference operations.

Definition 4.4.10. Let L_2 be the extension of L_1 with a second pointer arithmetic operation:

$$- : Ptr \times Ptr \rightarrow Int$$

Ptr still corresponds to a numeric domain so that the subtraction operation is meaningful. In particular, we assume that:

$$(p + n) - (p + m) \equiv n - m$$

We will use $p_1 < p_2$, $p_1 \leq p_2$ as abbreviations for $p_1 - p_2 < 0$, $p_1 - p_2 \leq 0$, respectively.

In order for our formulas to still be address-agnostic, we would have to avoid terms such as $p_1 - p_2$ (where p_1 and p_2 are *Ptr* terms), since these may take different values depending on the relative positioning of the intervals containing p_1 and p_2, if they happen to reside in different address intervals.

Definition 4.4.11. Let $p : Ptr$ be a term in L_1, $a : Ptr$ be an uninterpreted constant, and $n : Int$ be a term in L_1. The *guard* constraint of p over a and n, denoted by $\gamma(p, a, n)$, is given by the formula:

$$p \geq a \wedge p < a + n$$

Note that Definition 4.4.7 assumes models and address spaces that are *respected* by the formula, so under such assumptions the term $a + n$ cannot cross the boundaries between intervals.

Lemma 4.4.12. *A guard constraint is address-agnostic.*

The proof is in Appendix A.1.3.

Lemma 4.4.13. *Let ψ be a formula in L_2, $t : Int$ be a term in L_2, and $n : Int$ be a term in L_1. If $t \equiv n$ and $\psi[n/t]$ is address-agnostic, then ψ is address-agnostic as well.*

The proof is in Appendix A.1.4.

Lemma 4.4.14. *Let $\gamma \wedge \psi$ be a formula in L_2, where γ is a guard constraint, i.e., $\gamma(p, a, n)$, and ψ is in L_2. If $p, p' : Ptr$, and there exists $k : Int$ such that:*

- $\gamma \models p - p' = k$
- $\psi[k/(p - p')]$ *is address-agnostic*

then $\gamma \wedge \psi$ is address-agnostic.

The proof is in Appendix A.1.5.

As an example for the application of Lemma 4.4.14 consider again the program from Figure 4.1. The access of `array[i][j]` triggers a dereference of the symbolic pointer p:

$$p \triangleq select_4(store_4(store_4(K(0), 0, \beta_2), 4, \beta_3), i * 4) + j$$

This pointer is resolved to $mo_2 \triangleq (\beta_2, 2, a_2)$ using the following query:

$$0 \leq i < 2 \ \wedge \ 0 \leq j < 2 \ \wedge \ p \geq \beta_2 \ \wedge \ p < \beta_2 + 2$$

After the resolution, the query generated for the branch at line 17 is given by:

$$\varphi \triangleq 0 \leq i < 2 \ \wedge \ 0 \leq j < 2 \ \wedge \ p \geq \beta_2 \ \wedge \ p < \beta_2 + 2 \ \wedge \ select(a_2, p - \beta_2) = 7$$

Here:

$$\gamma \triangleq p \geq \beta_2 \ \wedge \ p < \beta_2 + 2, \quad \psi \triangleq 0 \leq i < 2 \ \wedge \ 0 \leq j < 2 \ \wedge \ select(a_2, p - \beta_2) = 7$$

It clearly holds that:

$$\gamma \models p - \beta_2 = j$$

and the substitution $\psi[j/(p - \beta_2)]$ results in:

$$0 \leq i < 2 \ \wedge \ 0 \leq j < 2 \ \wedge \ select(a_2, j) = 7$$

which is a formula in L_1 (and therefore address-agnostic), so we can apply Lemma 4.4.14 to conclude that φ is address-agnostic.

The only sources of pointer subtraction terms are program statements and representations of pointer dereferences. We assume that the program is well-behaved, i.e., subtraction cannot be applied between pointers that correspond to different memory objects, and boundaries between memory objects cannot be crossed using pointer arithmetic. Under these assumptions we can formulate the following theorem:

Theorem 4.4.15. *Let pc be some path constraints generated by the SE engine. Then pc is address-agnostic.*

The proof is in Appendix A.1.6.

Using Theorem 4.4.15, we can show that our query caching algorithm (Section 4.3) is correct. Let $q_1 \triangleq \varphi_1 \wedge ac_1$ be a cached query, and let $q_2 \triangleq \varphi_2 \wedge ac_2$ be a new query whose satisfiability we want to determine, where ac_1 and ac_2 are some address constraints. Suppose that our algorithm determined that q_1 and q_2 are equisatisifiable: φ_1 and φ_2 are identical up to renaming of symbolic base addresses (uninterpreted constants of sort *Ptr*), and the address spaces induced by ac_1 and ac_2 are isomorphic. Now, we need to show that q_1 and q_2 are indeed equisatisifiable.

Since φ_1 and φ_2 are identical up to renaming, we can assume that they are actually identical, while the only difference comes from the address constraints:

$$q_1 \triangleq \varphi \wedge ac_1, \quad q_2 \triangleq \varphi \wedge ac_2$$

If q_1 is satisifiable, then φ is satisfiable, so there exists a model m_1 such that $m_1 \models \varphi_1$. Now, we construct a model m_2 that is identical to m_1 except for the interpretation of symbolic base addresses, which is set according to the address constraints ac_2. We assume that the program is well-behaved, so in particular, the boundaries between intervals cannot be crossed using pointer arithmetic. Therefore, φ respects the address space induced by ac_1 in m_1. In addition, m_1 and m_2 satisfy the preconditions of Definition 4.4.7. Since we assume that φ is address-agnostic, then we conclude that $m_2 \models \varphi$. The interpretation of the symbolic base addresses in m_2 are consistent with the address constraints ac_2, so we conclude that $\varphi \wedge ac_2$, i.e., q_2, is satisfiable as well. Similarly, we can show that if q_2 is satisfiable then so is q_1.

Remark. Our correctness arguments hold also for formulas where a pointer is

allowed to refer to multiple memory objects, as happens in tools that apply state merging [66, 103], since the address-agnostic property is preserved under disjunction. The notion of the *red zone* (Section 2.2) can be easily incorporated into our correctness arguments and requires no modifications to the proof: It is simply sufficient to consider each interval to consist of the padding block and the address block allocated for the respective memory object. Since the size of the padding is constant for all allocations, the resulting address spaces will still be isomorphic.

4.5 Implementation

We implemented our query caching approach on top of KLEE [39], configured with the STP [58] solver. In order to use the relocatable addresses model, we relied on the implementation from Section 3.3.

In KLEE, the existing query cache is implemented using a hash table of queries. To make the lookup and insert operations efficient, a hash value is maintained for each query (and each expression), which is then used as a key for that hash table. Then, the bucket retrieved with that key is scanned to find the matching query based on *syntactic equality*. To enable efficient caching for our query caching approach as well, we maintain an additional hash value for each expression, which captures its structure regardless of the symbolic base address expressions. More technically, when we compute this hash value, we set the hash value of symbolic base addresses to a pre-defined constant. For example, the following queries will have the same hash value:

$$select(store_4(a, 1, \beta_2)) = 0 \qquad select(store_4(a, 1, \beta_3)) = 0$$

As discussed in Section 4.3, our approach does not apply to queries whose satisfiability depends on the ordering of the memory objects in the address space. When such queries are not *engine-internal*, i.e., generated as a result of a branch in the program, our approach may lead to incorrect results. We note, however, that such incorrect results can be pruned later by running dynamically the generated test cases.

4.6 Evaluation

In our experiments, we evaluate our address-aware query caching approach (*AA*) against the standard approach used in vanilla KLEE (*Base*). The challenge of caching address-dependent queries was partially addressed using the *segment reuse* heuristic (Section 3.2.3.3), which reuses previously allocated base addresses when a new segment is allocated, and this way increasing the chance for cache hits. Therefore, we evaluate our approach under two memory models: The *forking* memory model (*FMM*), i.e., vanilla KLEE, and the *dynamically segmented* memory model (*DSMM*) described in Section 3.2.3.2. When applying *AA* under *DSMM*, we disable the *segment reuse* heuristic, and when applying *Base* under *DSMM*, we enable that heuristic.

Our evaluation is structured as follows: In section 4.6.1 we present our benchmarks. In section 4.6.2 we provide an empirical validation for our approach. In Section 4.6.3 we show the effectiveness of our approach on benchmarks that generate address-dependent queries. In Section 4.6.4 we measure the overhead of our approach on benchmarks that do not generate address-dependent queries, where our approach is not expected to produce speedups. Our replication package is available at `https://doi.org/10.6084/m9.figshare.13042277`.

Experimental Setup. We performed our experiments on a machine running Ubuntu 16.04, equipped with an Intel i7-6700 processor and 32GB of RAM.

4.6.1 Benchmarks

In our experiments, we used the following benchmarks: *GNU m4* [81] (80K SLOC) is a macro processor included in many Unix-based systems. *GNU make* [82] (28K SLOC) is a tool which controls the generation of executables and other non-source files, also widely used in Unix-based systems. *SQLite* [108] (127K SLOC) is one of the most popular SQL database libraries. *Apache Portable Runtime* [23] (60K SLOC) is a library used by the Apache HTTP server that provides cross-platform functionality for memory allocation, file operations, containers, and networking. The *libxml2* [6] (197K SLOC) library is an XML parser and toolkit developed for the Gnome project. The *expat* [4] (23K SLOC) library is a stream-oriented XML parser, used in many open-source projects including Mozilla, Perl, Python and PHP. *GNU bash* [2] (106K SLOC) is the well-known Unix shell written for the GNU project. The *json-c* [5] (7K SLOC) library is used for encoding and

decoding JSON objects. *GNU Coreutils* [63] (188K SLOC) is a collection of utilities for file, text, and shell manipulation. The *libosip* [7] (11K SLOC) library is used for parsing SIP messages. The *libyaml* [1] (9K SLOC) library is used for parsing and emitting data in the YAML format.

In Section 4.6.3 we evaluate our approach on a set of terminating programs that generate address-dependent queries. Such queries are typically generated in the presence of symbolic pointers, which are created, for example, when data structures such as hash tables are indexed using a symbolic value as key. Our benchmarks consist of both whole-program utilities (*m4*, *make*, *bash*) and libraries (*SQLite*, *apr*, *libxml2*, *expat* and *json-c*). Four of our benchmarks (*m4*, *make*, *SQLite*, and *apr*) were used in previous work related to symbolic pointers [73]: In *m4* and *make*, which are language-processing utilities, hash tables are used to store the values of variables, functions, and strings. To avoid the analysis of these programs from getting stuck in the early stages and to achieve its termination, these programs are run with a partially symbolic input. The test driver in *apr* focuses on the runtime's hash table API, and the test driver in *SQLite* creates database triggers using concrete and symbolic SQL queries. Our four additional benchmarks are *libxml2*, *expat*, *bash*, and *json-c*: As was done in the cases of *m4* and *make*, we ran *bash* with a partially symbolic input in order to reach the deeper parts of the code that operate on the various tables that store variables, strings, and functions. In *libxml2* and *expat* we built test drivers that parse symbolic HTML and XML inputs, respectively. In *json-c* we built a test driver that constructs a JSON object which internally uses hash tables.

In Section 4.6.4 we evaluate our approach on a set of programs that do not generate address-dependent queries: We chose 10 utilities from *coreutils* that behave deterministically across multiple runs, and built test drivers for the main parsing APIs in both *libosip* and *libyaml*.

4.6.2 Empirical Validation

In this experiment, we provide an empirical validation for the correctness of our approach using the following methodology: We ran KLEE on each of the benchmarks with *Base* and *AA*, and as our approach (*AA*) must not affect the exploration, we validated that both the number of explored paths and the achieved coverage are indeed identical in both runs. When running with *AA*, we additionally checked for each cached

Table 4.2: Classification of queries and their amounts.

Program	Total	C_1	C_2	C_3
m4	14,022	9,589	9,127	6,394
make	2,565,399	2,477,145	90,027	69,535
SQLite	26,990	18,407	15,589	13,783
apr	15,013	8,960	8,960	8,448
libxml2	708,101	410,789	347,420	347,420
expat	1,797,033	945,192	102,903	102,903
bash	54,078	19,051	10,840	7,860
json-c	28,476	17,484	17,263	14,311

query that its cached result is correct by simply comparing it with the result reported by the SMT solver. We performed this experiment using both memory models (*FMM* and *DSMM*).

4.6.3 Performance

In this experiment, we compare the performance of two query caching approaches: *Base* and *AA*. In the default configuration of KLEE, two constraint solving heuristics are used: standard query caching (*Base*) and counter-example (CEX) caching. Therefore, in order to have a complete comparison against vanilla KLEE, we enable the CEX caching in our experiments as well. For each approach, we run KLEE with the DFS search heuristic and the deterministic memory allocator, until all the paths are explored. In each run we record the following parameters: the number of queries reaching the SMT solver, the termination time, the size of the query cache, and the memory usage.

Table 4.2 gives an insight into the type of queries generated by vanilla KLEE in our benchmarks. In KLEE, the query caching heuristic handles only *satisfiability* and *validity* queries, and does not handle *model* (assignment) queries. As discussed in Section 4.3, not all the queries passed to the query caching heuristic can be handled with our approach. Column *Total* shows the total number of queries generated during the analysis. Column C_1 shows the number of queries passed to the query caching heuristic. Column C_2 shows the number of queries passed to the query caching heuristic which can be handled by our approach. Column C_3 shows the number of queries passed to the query caching heuristic which can be handled by our approach and are address-dependent. Note that the number of queries that pass through the cache but cannot be handled by our approach, i.e., $C_2 - C_3$, is relatively low.

Table 4.3: Number of queries with both approaches.

Program	***FMM***		***DSMM***	
	Base	*AA*	*Base*	*AA*
m4	10,792	4,265	1,600	1,289
make	347,324	45,471	50,558	9,753
SQLite	5,622	4,681	14,563	12,993
apr	445	300	126	86
libxml2	124,782	6,118	124,782	6,118
expat	89,740	31,747	89,736	31,761
bash	8,538	4,479	7,542	4,098
json-c	15,364	5,246	2,757	1,523

Table 4.4: Termination time in *hh:mm:ss*.

Program	***FMM***		***DSMM***	
	Base	*AA*	*Base*	*AA*
m4	00:13:16	00:04:59	00:19:17	00:14:55
make	06:46:44	02:30:51	03:56:42	01:47:23
SQLite	00:17:20	00:14:24	04:00:17	03:12:22
apr	00:57:33	00:39:05	00:20:20	00:13:39
libxml2	02:33:33	00:17:09	02:27:35	00:17:12
expat	00:26:02	00:23:19	00:25:13	00:23:06
bash	02:37:48	01:23:30	02:39:04	01:14:18
json-c	00:31:36	00:13:20	00:08:05	00:04:19

Table 4.3 shows the number of queries for each benchmark with the two approaches and the different memory models. Here, we report the number of queries that reached the SMT solver itself, i.e., those that were not handled by any of the constraint solving heuristics (query caching or CEX caching). In *FMM*, the reduction in the number of queries with *AA* varies between 1.20× (in *SQLite*) and 20.40× (in *libxml2*), and its average is 5.11×. In *DSMM*, the reduction varies between 1.12× (in *SQLite*) and 20.40× (in *libxml2*), and its average is 4.48×.

Table 4.4 shows the termination time for each benchmark with the two approaches and the different memory models. In *FMM*, the speedup of *AA* compared to *Base* varies between 1.11× (in *expat*) and 8.96× (in *libxml2*), and its average is 2.80×. Note that the speedup depends not only on the reduction in the number of queries, but also on the complexity of the queries. For example, the reduction in the number of queries in *make* is roughly 4 times higher than in *bash* (7.63× *vs.* 1.90×), but the speedup in *make* is

only 1.43× higher than in *bash* (2.70× vs 1.90×) as the queries in *bash* are more complex due to larger SMT arrays, i.e., array constraints with more *store* expressions. In *DSMM*, the speedup varies between 1.09× (in *expat*) and 8.59× (in *libxml2*), and its average is 2.73×. The queries in *expat* are relatively simple compared to other benchmarks, therefore the performance gains are less significant there.

Table 4.5 shows the size of the query cache for each benchmark with the two approaches and the different memory models. In *FMM*, the reduction in the cache size varies between 1.29× (in *SQLite*) and 24.68× (in *libxml2*), and its average is 6.06×. In *DSMM*, the reduction varies between 1.18× (in *SQLite*) and 24.68× (in *libxml2*), and its average is 4.93×.

In general, the number of explored paths in *DSMM* is guaranteed to be at most as high as in *FMM*, and lower in programs whose analysis triggers symbolic pointers with multiple resolutions. Therefore, in such programs there is also a reduction in the number of queries and the cache size. However, in *DSMM* the queries are potentially more complex, so the termination time is not necessarily lower compared to *FMM*, as was shown in Section 3.4 and as can be seen in Table 4.4. The number of explored paths in *libxml2* and *expat* is identical with *FMM* and *DSMM*, but in *expat* there is a slight difference in the number of queries (and cache size) due to non-determinism that was introduced by the SMT solver in model (assignment) queries.

Figure 4.4 shows the memory usage for each of the benchmarks with the two approaches under the different memory models. Clearly, the size of the query cache affects the memory usage, i.e., a smaller cache should result in lower memory usage. In benchmarks where the cache size is relatively low (*m4*, *SQLite*, *apr*, *bash* and *json-c*), the reduction in the cache size has little effect on the memory usage, which is roughly the same with both approaches. However, in other benchmarks where the cache is larger, the difference in the cache size results in larger difference in memory usage: For example, when *FMM* is used, the memory usage in *make* and *libxml2* is reduced by roughly 800MB and 100MB, respectively.

The overall performance improvement with our approach suggests that there are queries that are handled by our approach and are not handled by the CEX caching [3]. Therefore, the CEX caching should be seen as complementary to our approach.

[3] We internally experimented also with an optimized version of the CEX caching heuristic (using the *cex-cache-try-all* option), which resulted in even better improvement for our approach.

Table 4.5: The size of the query cache with both approaches.

Program	***FMM***		***DSMM***	
	Base	*AA*	*Base*	*AA*
m4	11,780	3,493	1,631	1,341
make	348,210	41,927	51,064	12,404
SQLite	6,898	5,354	10,680	9,071
apr	496	279	130	81
libxml2	136,165	5,517	136,165	5,517
expat	92,383	34,226	92,382	34,226
bash	8,774	4,453	7,712	4,308
json-c	15,998	3,634	2,906	1,336

> Our query caching approach (*AA*) increases the number of cache hits, speeds up the analysis, and reduces the memory usage.

4.6.4 Overhead

In Section 4.6.3, we showed that our approach can improve the performance in programs that generate address-dependent queries. However, our approach imposes additional computational overhead due to two main reasons: maintaining additional symbolic values for address expressions and substituting expressions (Section 3.2.1).

In this experiment, we show the runtime overhead of our approach in programs that do not generate address-dependent queries: *coreutils*, *libosip* and *libyaml*. For each program, we proceed with the following methodology under each of the memory models: First, we ran KLEE for roughly one hour and recorded the number of executed instructions, and then we re-ran the analysis up to the recorded number of instructions with both of the approaches (*Base* and *AA*).

In *FMM*, with regards to termination time, our approach had a maximum overhead of 17% (in *libosip*) and an average overhead of 6%. There was no significant difference in memory usage between the two approaches, while our approach had an overhead of 3%. Similarly to our query caching approach, *DSMM* relies on the relocatable addressing model, which is the main source of overhead compared to vanilla KLEE (*FMM*). Therefore, in *DSMM*, where the relocatable addressing model is used in both of the approaches, the performance is almost identical in terms of time and space.

Similarly to Section 4.6.3, we validated that the number of explored paths is identical

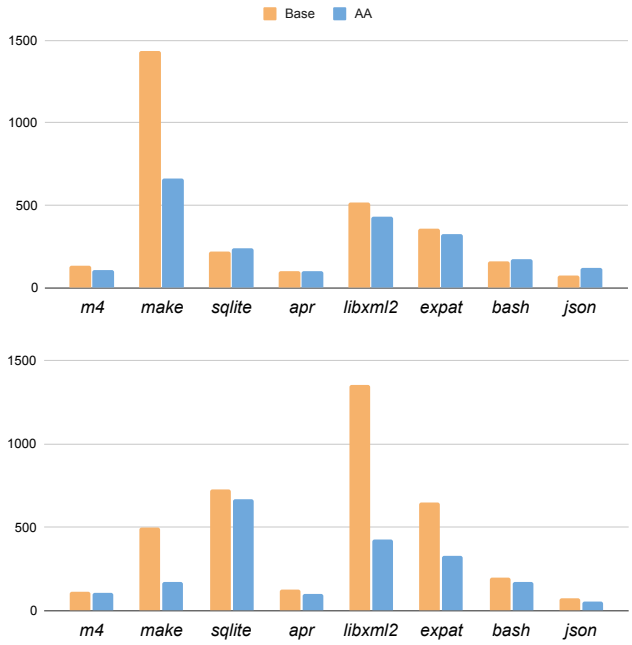


Figure 4.4: Memory usage (in *MB*) with both approaches under *FMM* (top) and (*DSMM*) (bottom).

with both approaches, and performed an additional run for each program to validate the correctness of our query caching approach w.r.t. the SMT solver.

> In programs that do no generate address-dependent queries, our query caching approach (*AA*) incurrs a reasonable overhead.

Chapter 5

A Bounded Symbolic-Size Model for Symbolic Execution

This chapter is based on the results published in [113].

5.1 Introduction

In SE engines [39, 85, 103], the memory is modeled using a linear address space where each memory object has a fixed *concrete* size. Such model imposes two main limitations: When the inputs of the program under test are of variable size, e.g., array and strings, the size of these inputs must be concretely determined before the analysis takes off. Moreover, if an allocation of symbolic size is encountered during the analysis of the program, then the size of that allocation has to be concretized. Therefore, while being a natural design choice, the existing model may lose coverage and miss bugs.

As a motivating example, consider the code fragments taken from *libosip* [12] shown in Figure 5.1, that depict two bugs found during our experiments. The `osip_via_parse` function is responsible for parsing the *VIA* header of a request message in the *Session Initiation Protocol* (SIP). It first looks for the leading occurrence of `'/'` (line 4), and if found, it looks for the second occurrence of `'/'` (line 7). If the distance between the two occurrences of the `'/'` characters is too small (line 10), then the parsing is stopped. Otherwise, it validates that there is at least one space after the second occurrence of `'/'`, and if that is the case, it skips additional spaces in the input (the loop in line 17). The `host` pointer is incremented at the beginning of each iteration (line 18), so `host` may point to the null-terminator of the string after the execution of the loop. When

```
1 int osip_via_parse(osip_via_t *via, const char *hvalue) {
2     if (hvalue == NULL)
3         return OSIP_BADPARAMETER;
4     const char *version = strchr(hvalue, '/');
5     if (version == NULL)
6         return OSIP_SYNTAXERROR;
7     const char *protocol = strchr(version + 1, '/');
8     if (protocol == NULL)
9         return OSIP_SYNTAXERROR;
10    if (protocol - version < 2)
11        return OSIP_SYNTAXERROR;
12    ...
13    const char *host = strchr(protocol + 1, ' ');
14    if (host == NULL)
15        return OSIP_SYNTAXERROR;
16    if (host == protocol + 1) {
17        while (0 == strncmp(host, " ", 1)) {
18            host++;
19            if (strlen(host) == 1)
20                return OSIP_SYNTAXERROR;
21        }
22        // out-of-bounds read
23        host = strchr(host + 1, ' ');
24        ...
25    }
26    ...
27 }
28 int osip_uri_parse_headers(osip_uri_t *url,
29                            const char *headers) {
30   const char *equal = strchr(headers, '=');
31   // out-of-bounds read
32   const char *_and = strchr(headers + 1, '&');
33   ...
34 }
```

Figure 5.1: Bugs found in *libosip* 5.2.0.

that happens, the call to `strchr` at line 22 results in an *out-of-bounds* read, because `host + 1` points to an invalid memory. You may notice that this bug is reachable only if the length of the input string is at least 5 (including the null-terminator). Therefore, if the user decides to analyze this function with a shorter input, then the bug would remain undetected. Picking a longer input string would help in the last case, but may similarly lead to missed bugs in other cases. To see why, consider the second function `osip_uri_parse_headers`. It searches the input string `headers` for the first occurrence of `'='` (line 30), and then independently looks for the first occurrence of `'&'` starting from the second character of `headers` (line 32). Clearly, this results in an *out-of-bounds* read if the input string is empty. Therefore, if the user decides to analyze this function with a longer input, then the bug would remain undetected.

We propose a model that supports symbolic-size allocations, and thus enable the

```
1 size_t n; // symbolic
2 char *p = calloc(n, 1);
3 char *q = calloc(10, 1);
```

Figure 5.2: Unbounded symbolic size.

analysis of programs with inputs whose size belongs to a range of values. Designing an *unbounded* model, i.e., a model where the symbolic size of a memory object is unconstrained, imposes several difficulties. SE engines typically use a linear address space where the base addresses of memory objects are concrete, so address intervals of distinct memory objects may overlap in the presence of unbounded memory objects. To overcome this, one would have to adopt a two-dimensional address space where each memory object has its own address space, or use symbolic base addresses with some additional constraints that will ensure that address intervals do not overlap (Section 2.2). Besides, SE engines use the QF_ABV logic fragment [29, 31, 59] to track the contents of memory objects, meaning that the value at each offset within a memory object is maintained explicitly. As a result, the analysis with big enough memory objects will not be possible, as it will require an amount of memory unavailable on modern machines. Therefore, we design a *bounded* model, where the symbolic size of memory objects is bounded by a user-specified *capacity*. This model does not require changing the modeling of the address space, thus can be easily integrated with existing SE engines.

Our model, however, increases the number of forks due to the introduction of additional symbolic expressions, i.e., the symbolic sizes. This is particularly noticeable in loops where the number of iterations depends on a symbolic size expression, leading to a number of forks which is typically at least linear in the size. To cope with the amplified path explosion, we propose a state merging approach which is applied in *symbolic-size dependent* loops, a common scenario in which our model introduces additional forking.

Main contributions:

1. We present a bounded symbolic-size model that enables analysis with variable-size inputs.

2. We propose a state merging approach to mitigate the path explosion introduced by our model.

3. We implement a KLEE-based prototype, which we make available as open-source.

4. We evaluate our model in the context of API and whole-program testing, and find previously unknown bugs.

Outline. In Section 5.2, we present our bounded symbolic-size model and our state merging approach for reducing the additional forks caused by this model. In Sections 5.3 and 5.4, we discuss our implementation and evaluation, respectively.

5.2 Technique

In Section 5.2.1, we present our bounded symbolic-size model. In Section 5.2.2, we propose a state merging approach to cope with the additional forks introduced by that model. In Section 5.2.3, we present optimizations that allow us to reduce the size of constraints in merged symbolic states. In Section 5.2.4, we discuss the limitations of our model.

5.2.1 Bounded Symbolic-Size Model

Ideally, we would like to have a model where the symbolic size of a memory object can be arbitrarily large, i.e., *unbounded*. However, such model imposes several challenges.

In the concrete-size model (Section 2.2), every memory object has a fixed address interval, so when a new memory object has to be allocated, the memory allocator can easily pick a new address interval which does not intersect with the existing ones. To illustrate why this is no longer true with symbolic-size allocations, consider the example from Figure 5.2. Let us assume that the first memory object (line 2) is allocated at address 0x80000000 and has an unbounded symbolic size n. The SE engine cannot allocate the second memory object (line 3) at a concrete base address after the first memory object as the resulting address interval might overlap with the address interval of the first memory object, thus violating the non-overlapping property. To overcome this, we will have to allocate the second memory object at some symbolic base address β, and encode the non-overlapping property directly in the path constraints. In our example, the non-overlapping of the two memory objects can be encoded using the following constraints:

$$0x80000000 > \beta + 10 \ \vee \ \beta > 0x80000000 + n$$

As the number of such constraints is expected to grow with the size of the address space, i.e., the number of memory objects, this will eventually become a burden on the SMT solver.

Another problem is that SE engines typically use the QF_ABV logic fragment to encode *read* and *write* operations. As this logic fragment is quantifier-free, when we read or write to some offset within a memory object, this operation results in an explicit encoding. Note that in the example from Figure 5.2, the first memory object is allocated with `calloc`, which initializes the memory with zeros. If we do not have a concrete bound for the size of this memory object, i.e., a maximum value for n, then we would be forced to use some form of universal quantifiers to express the side effects of `calloc`. Even if we have such concrete bound, the value stored at each offset is encoded separately, so the size of the encoding for the whole memory object would be at least *linear*. This means that for large enough memory objects, the analysis will be impossible due to an extremely high memory usage, suggesting that the size of a given memory object should be limited anyway.

We thus propose a *bounded* symbolic-size model in which a memory object is represented as a tuple:

$$(b, \sigma, c, a) \in N^+ \times E \times N^+ \times A$$

where b is a concrete base address, σ is a symbolic expression describing the size of the memory object, c is the maximum concrete value for σ, i.e., the *capacity* of the memory object, and a is an SMT array that tracks the values written to the memory object. Similarly to Section 2.2, given a memory object mo, we denote its base address, size, capacity, and SMT array by $mo.addr$, $mo.size$, $mo.capacity$, and $mo.array$, respectively.

When a memory object (b, σ, c, a) is allocated, we add to the path constraints the *capacity constraint* $\sigma \leq c$, where c is user-specified. If that condition is unsatisfiable, then we increase the capacity c until that capacity constraint becomes satisfiable. The pointer resolution process together with the read and write operations, which are described in Section 2.2, remain almost unmodified with the small change of replacing the concrete size s with the symbolic one σ.

```
1 size_t n; // symbolic
2 size_t z; // symbolic
3 char *p = malloc(n); // capacity is 3
4 for (unsigned i = 0; i < n; i++) {
5     if (z == 0) {
6         break;
7     }
8     p[i] = i;
9 }
```

Figure 5.3: Symbolic-size dependent loop.

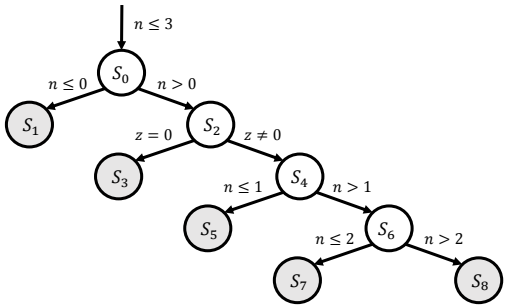


Figure 5.4: The execution tree of the program from Figure 5.3.

5.2.2 Mitigating Path Explosion By State Merging

Our model enables a more *complete* analysis since it supports the symbolic execution of programs with memory objects whose size can have a range of values rather than only a fixed one. Nevertheless, this does not come without a cost: Our model introduces additional symbolic values that describe sizes of memory objects, which in turn may lead to additional forks and more complex constraints, thus amplifying the known problems of path explosion and constraint solving.

To illustrate this, consider the program from Figure 5.3. In the concrete-size model, the value of n is concretized to some concrete value, and the symbolic execution of the program may fork only at line 5, thus exploring at most two paths. Note that with our symbolic-size model, the branch condition `i < n` at line 4 is now a symbolic expression. Therefore, each iteration of the loop will potentially produce a new fork. For example, assuming that the capacity for the allocation at line 3 is three, the memory object allocated at this line would be $(b, n, 3, a)$, and the constraint $n \leq 3$ will be added to the path constraints. Then, five paths will be explored: One path which does not enter the loop when $n = 0$, another path that executes the first iteration and breaks from the loop at line 6, and three paths that execute k full iterations for $1 \leq k \leq 3$.

To cope with the problem of path explosion, we employ state merging [69, 77], a technique that enables the merging of multiple execution paths. We apply state merging specifically in locations where our model introduces additional forking, as typically happens in symbolic-size dependent loops, rather than applying it opportunistically in every possible location.

5.2.2.1 Merging Symbolic-Size Memory Objects

In our model, the merging of symbolic states is defined similarly to previous works [69, 77], except for the case of symbolic-size memory objects which requires a special treatment. Let s_1 and s_2 be two symbolic states, and let $mo_1 \triangleq (b, \sigma, c, a_1)$ and $mo_2 \triangleq (b, \sigma, c, a_2)$ be two memory objects in s_1 and s_2, respectively. The merged memory object (b, σ, c, a) is constructed such that the following holds:

$$\forall i.\ 0 \leq i < c \rightarrow\ \ select(a, i) = ite(s_1.pc, select(a_1, i), select(a_2, i))$$

Note that the actual size bound of a given memory object (b, σ, c, a), i.e., the maximum value of σ, can be less than its capacity c. For example, this would happen in the program from Figure 5.2, if at the moment of allocation at line 2 we had the constraint $n < 5$ and the user-specified capacity was 10. Before accessing the i-th cell of a, the SE engine always validates that i is a valid offset, i.e., $i < \sigma$. Therefore, even if i is an out-of-bounds offset in mo_1 (or mo_2), our representation of the merged memory object is still valid, since the i-th offset will never be accessed anyway.

5.2.2.2 Loops

A function's *control flow graph* (CFG) is a directed graph whose nodes are basic blocks (or instructions), where the edges represent possible transitions of the control flow between nodes. A *loop* is a strongly connected component in the CFG. A *loop exit* of a loop L is a node in the CFG which does not belong to L, but has a predecessor in L. We assume that for any given basic block (and instruction), we can tell if it belongs to a loop L or not.

5.2.2.3 Detecting Symbolic-Size Dependent Loops

We apply state merging *selectively* only in loops whose execution depends on a symbolic size expression. We detect such loops dynamically: When the SE engine allocates a memory object (b, σ, c, a) with a symbolic size, we mark the (atomic) symbolic variables in σ as *tainted*. If later, during the execution of a loop L, we encounter a branch instruction that results in a fork while the corresponding branch condition is tainted, then L is considered to be *symbolic-size dependent*.

5.2.2.4 Loop Merging

Algorithm 3 depicts the application of state merging for symbolic-size dependent loops[1]. Before explaining the algorithm, we extend the definition given in Section 2.3: A symbolic state s additionally consists of a *merging context* $s.ctx$, which consists of (1) a loop *loop*, (2) a set of states *states*, (3) and a liveness counter *counter*. In addition, we extend the definition of merge-compatibility for heaps (Section 2.5): Two heaps h_1 and h_2 are *merge-compatible* if for every memory object $(b_1, \sigma_1, c_1, a_1)$ in h_1, there is a memory object $(b_2, \sigma_2, c_2, a_2)$ in h_2 such that $b_1 = b_2$, $\sigma_1 = \sigma_2$, and $c_1 = c_2$, and vice versa.

When a symbolic state s executing a loop l reaches a forking branch whose condition c is tainted, we associate s with a new *merging context* (line 5), if not already set. We then associate the states s_t and s_f (forked at line 7) with the merging context ctx, add them to ctx, update the liveness counter, and update the worklist of the state scheduler (lines 8-13). When the execution of s reaches a loop exit of a loop l that matches the loop of the associated merging context (line 15), we decrement the liveness counter (line 16). If after that, the liveness counter is zero, i.e., all the states of the merging context ctx finished executing the loop, then we finally perform the merging. As the merging of two symbolic states is allowed only when their *instruction counter* points to the same location, we first split the symbolic states to groups based on the loop exit that was taken (line 18). Then we merge each group of symbolic states separately (line 20) using the merging procedure explained next, and update the worklist of the state scheduler accordingly (line 21).

Algorithm 4 depicts the state merging procedure which receives a merging context and a set of (merge-compatible) symbolic states. First, we compute the common

[1]The lines marked in grey will be discussed later and should be ignored for now.

Algorithm 3 Loop merging algorithm. (The merging procedure *merge* in line 20 is defined in Algorithm 4.)

```
1:  function on-forking-branch(s, c, l)
2:    if is-tainted(c) then
3:      ctx ← s.ctx
4:      if ctx = null then
5:        ctx ← {.loop : l, .states : {s}, .counter : 1}
6:        ctx.root ← s.n ← {.s : s, .c : true, .l = null, .r = null}
7:      s_t, s_f ← fork(s, c)
8:      s_t.ctx ← ctx, s_f.ctx ← ctx
9:      ctx.states ← (ctx.states \ {s}) ∪ {s_t, s_f}
10:     ctx.counter ← ctx.counter + 1
11:     s.n.l ← s_t.n ← {.s : s_t, .c : c, .l = null, .r = null}
12:     s.n.r ← s_f.n ← {.s : s_f, .c : ¬c, .l = null, .r = null}
13:     worklist ← (worklist \ {s}) ∪ {s_t, s_f}

14: function on-loop-exit(s, l)
15:   if s.ctx.loop = l then
16:     s.ctx.counter ← s.ctx.counter − 1
17:     if s.ctx.counter = 0 then
18:       groups ← group-by-loop-exit(s.ctx.states)
19:       for states ∈ groups do
20:         m ← merge(s.ctx, states)
21:         worklist ← (worklist \ states) ∪ {m}
```

constraints of the input symbolic states (line 12), which can be also given by the path constraints of the symbolic state that initialized the merging context. Then, we extract for each state s_i the suffix constraints ψ_i (line 13), i.e., the constraints of $s_i.pc$ that do not appear in φ, which are the constraints that were added by s_i during the execution of the loop till its exit. The path constraints of the merged symbolic state are set to the conjunction of φ and the disjunction of all the suffix constraints (line 14). For each variable v, its value in the merged symbolic state is set to an *ite* expression (line 16), which results in the value $s_i.vars[v]$ if $s_i.pc$ holds. In the *ite* expression, we can actually use ψ_i instead of $s_i.pc$, since φ is implied by the path constraints of the merged symbolic state. The merging of the heap, which is described at lines 19-21, is similar to the standard definition in Section 2.5. The only difference is that the number of array cells that are merged is limited by the capacity of the memory object, rather than by its concrete size.

As an example, consider again the program from Figure 5.3 whose execution tree is given in Figure 5.4. Once the allocation at line 3 occurs, the symbolic expression n is marked as tainted. Later when the first iteration of the loop is executed (line 4), the

Algorithm 4 Merging algorithm. (The functions *merge-values* and *merge-pc* are defined in Algorithm 1.)

1: **function** *merge-symbolic-size-object*$(ctx, \{s_i\}_{i=1}^n, b)$
2: $\{mo_i\}_{i=1}^n \leftarrow \{get\text{-}memory\text{-}object\text{-}by\text{-}address(s_i, b)\}_{i=1}^n$
3: $\sigma \leftarrow mo_1.size$
4: $c \leftarrow mo_1.capacity$
5: $a \leftarrow new\text{-}smt\text{-}array()$
6: **for** $0 \leq j < c$ **do**
7: $e \leftarrow merge\text{-}values(\{s_i.pc\}_{i=1}^n, \{select(mo_i.array, j)\}_{i=1}^n)$
8: $e \leftarrow merge\text{-}values\text{-}opt(\{s_i\}_{i=1}^n, ctx.root, \{s_i \mapsto select(a_i, j)\})$
9: $a \leftarrow store(a, j, e)$
10: **return** (b, σ, c, a)
11: **function** *merge*$(ctx, \{s_i\}_{i=1}^n)$
12: $\varphi \leftarrow common\text{-}constraints(\{s_i.pc\}_{i=1}^n)$
13: $\{\psi_i\}_{i=1}^n \leftarrow \{suffix\text{-}constraints(s_i.pc, \varphi)\}$
14: $pc \leftarrow \varphi \wedge merge\text{-}conditions(\{\psi_i\}_{i=1}^n)$
15: $pc \leftarrow \varphi \wedge merge\text{-}conditions\text{-}opt(\{s_i\}_{i=1}^n, ctx.root)$
16: $vars \leftarrow \lambda v \in V.\ merge\text{-}values(\{\psi_i\}_{i=1}^n, \{s_i.vars[v]\}_{i=1}^n)$
17: $vars \leftarrow \lambda v \in V.\ merge\text{-}values\text{-}opt(\{s_i\}_{i=1}^n, ctx.root, \{s_i \mapsto s_i.m[v]\})$
18: $heap \leftarrow \emptyset$
19: **for** $mo \in s_1.heap$ **do**
20: $mo \leftarrow merge\text{-}symbolic\text{-}size\text{-}object(ctx, \{s_i\}_{i=1}^n, mo.addr)$
21: $heap \leftarrow heap \cup \{mo\}$
22: **return** $\{.pc : pc,\ .vars : vars,\ .heap : heap,\ .ctx = null\}$

loop is detected as symbolic-size dependent, since the branch condition $n > 0$ is tainted. The loop at line 4 has two loop exits (lines 6 and 9), so once the exploration of the loop is completed, the symbolic states are split into two merging groups. The first group contains only the symbolic state s_3 whose path constraints are $n > 0 \wedge z = 0$, and the second group contains the rest of the symbolic states, i.e., $\{s_1, s_5, s_7, s_8\}$. In the merged symbolic state of the second group, for example, the paths constraints will be:

$$(n \leq 3) \wedge (\varphi_1 \vee \varphi_5 \vee \varphi_7 \vee \varphi_8)$$

where:

$$\varphi_1 \triangleq (n \leq 0)$$

$$\varphi_5 \triangleq (n > 0 \wedge z \neq 0 \wedge n \leq 1)$$

$$\varphi_7 \triangleq (n > 0 \wedge z \neq 0 \wedge n > 1 \wedge n \leq 2)$$

$$\varphi_8 \triangleq (n > 0 \wedge z \neq 0 \wedge n > 1 \wedge n > 2)$$

and assuming that the memory object pointed to by `p` is $(b, n, 3, a)$, the value of `p[2]`, for instance, will be:

$$
\begin{aligned}
&ite(\varphi_1,\\
&\quad select(a, 2),\\
&\quad ite(\varphi_5,\\
&\qquad select(store(a, 0, 0), 2),\\
&\qquad ite(\varphi_7,\\
&\qquad\quad select(store(store(a, 0, 0), 1, 1), 2),\\
&\qquad\quad select(store(store(store(a, 0, 0), 1, 1), 2, 2), 2))))
\end{aligned}
$$

which can be simplified using store eliminations to:

$$ite(\varphi_1, select(a, 2), ite(\varphi_5, select(a, 2), ite(\varphi_7, select(a, 2), 2)))$$

Once state merging has been applied, we were able to reduce the number of explored paths. However, that resulted in a more complex representation of the merged symbolic states, due to the introduction of *ite* and *disjunctive* expressions. From our experience, the representation complexity of queries has a direct impact on SMT solving times, so the merged symbolic states should be represented as compactly as possible.

5.2.3 Optimizations

When we merge symbolic states, we generate *ite* and *disjunctive* expressions that may contain duplicate or redundant expressions. This happens, for example, in the program from Figure 5.3, when we merge the symbolic states from the merging group that corresponds to the loop exit at line 9. Let φ_i be again the constraints that were added to the state s_i during the execution of the loop till its exit. Then, the path constraints of the resulting merged symbolic state is given by (as mentioned in Section 5.2.2):

$$(n \leq 3) \wedge (\varphi_1 \vee \varphi_5 \vee \varphi_7 \vee \varphi_8)$$

Note that there are conditions that unnecessarily repeat across the different constraints: For example, the condition $n > 0 \wedge z \neq 0$ appears in the last three disjuncts (φ_5, φ_7,

and φ_8), and the condition $n > 1$ appears in both φ_7 and φ_8. Moreover, the disjunction of the last three constraints $(\varphi_5, \varphi_7, \varphi_8)$ is actually equivalent to:

$$n > 0 \wedge z \neq 0$$

since:

$$(n \leq 1) \vee (n > 1 \wedge n \leq 2) \vee (n > 1 \wedge n > 2) \equiv true$$

The redundant conditions mentioned above occur also when we merge the contents of the memory object allocated at line 3.

Instead of optimizing the expressions resulting from the original merging algorithm, we generate them in an *equivalent* and *reduced* form beforehand. To do so, we use the execution tree (Section 2.4) constructed during the symbolic execution of the loop in a given merging context. Recall that each node in the execution tree is either a leaf node that corresponds to a final symbolic state reaching a loop exit, or an intermediate node that has exactly two successors corresponding to the *true* and *false* sides of a branch. In addition, each node is associated with the corresponding symbolic state and the condition because of which it was forked, where the condition of the root node is initialized to *true*. In the execution tree from Figure 5.4, for example, the leaf and intermediate nodes are marked in grey and white, respectively.

To support the construction of the execution tree in a given merging context, we extend Algorithm 3 with the lines marked in grey. In line 6, we initialize the node of the initial state, which is set to the root of the execution tree. In lines 11 and 12, we extend the execution tree by setting the children of the node associated with s to the nodes of s_t and s_f, which are associated with the conditions c and $\neg c$, respectively.

Our optimized constraint merging procedure is given in Algorithm 5. The procedure *merge-conditions-opt* receives a set of symbolic states *states* and a node n from the execution tree, and returns a pair of a flag and a condition. The flag indicates whether the subtree originating from n is *complete*, i.e., all the symbolic states associated with its nodes belong to *states*, and the condition is the resulting merged constraint. If n is a leaf node, then we return its associated condition if its symbolic state belongs to *states* (lines 2-6), and *false* otherwise. If n is an intermediate node, then we recursively generate the constraints φ_l and φ_r for the children nodes (lines 7-8). If f does not hold, i.e., n's sub-tree contains at least one symbolic state that does not belong to *states*,

Algorithm 5 Optimized constraint merging

```
1:  function merge-conditions-internal(states, n)
2:    if is-leaf(n) then
3:      if n.s ∈ states then
4:        return true, n.c
5:      else
6:        return false, false
7:    f_l, φ_l = merge-conditions-internal(states, n.l)
8:    f_r, φ_r = merge-conditions-internal(states, n.r)
9:    f ← f_l ∧ f_r
10:   if f then
11:     φ ← n.c
12:   else
13:     φ ← n.c ∧ (φ_l ∨ φ_r)
14:   return f, φ
15: function merge-conditions-opt(states, n)
16:   f, φ = merge-conditions-internal(states, n)
17:   return φ
```

then we return the conjunction of the current condition with the disjunction of the children's constraints (line 13). The disjunction of the constraints corresponding to any *complete* sub-tree, i.e., a sub-tree in which all the symbolic states belong to *states*, is guaranteed to be equivalent to *true*. Therefore, if f holds, then we are in the case where n's sub-tree is complete, and the term $\varphi_l \vee \varphi_r$ can be further simplified to *true* (line 11). Note that the current condition $n.c$ is always added only once, thus avoiding duplicate occurrences. In addition, if φ_l (or φ_r) is *false*, i.e., the sub-tree originating from $n.l$ (or $n.r$) does not contain states from *states*, then we can propagate only φ_r (or φ_l).

As an example, consider the application of Algorithm 5 with *states* as the merging group $\{s_1, s_5, s_7, s_8\}$, and n as the root of the execution tree from Figure 5.4. Without the simplification at line 11, the resulting constraint will be:

$$(n \leq 0) \vee (n > 0 \wedge z \neq 0 \wedge (n \leq 1 \vee (n > 1 \wedge (n \leq 2 \vee n > 2))))$$

which already reduces the size of the constraint from $O(n^2)$ to $O(n)$. When applying the simplification at line 11, the constraint is further reduced to:

$$(n \leq 0) \vee (n > 0 \wedge z \neq 0)$$

Our optimized value merging procedure *merge-values-opt* is given in Algorithm 6. It

Algorithm 6 Optimized value merging

```
1:  function merge-values-opt(states, n, m)
2:    if is-leaf(n) then
3:      if n.s ∈ states then
4:        return m[n.s]
5:      else
6:        return null
7:    v_l = merge-values-opt(states, n.l, m)
8:    v_r = merge-values-opt(states, n.r, m)
9:    if v_l = null ∧ v_r = null then
10:     v ← null
11:   else if v_l = null ∧ v_r ≠ null then
12:     v ← v_r
13:   else if v_l ≠ null ∧ v_r = null then
14:     v ← v_l
15:   else
16:     v ← ite(n.l.c, v_l, v_r)
17:   return v
```

receives a set of symbolic states *states*, a node n from the execution tree, and a mapping m which associates a symbolic state with the value of the target memory location (e.g., a variable). For a leaf node n, we return the value associated with $n.s$ if the latter exists in the mapping, and *null* otherwise (lines 2-6). For intermediate nodes, if a merged value is available only in one of the children (lines 12 and 14), then we pass on that value. Otherwise, we handle the case in which both v_l and v_r are available, where we return an *ite* expression choosing between those values (line 16).

For example, when we use the original procedure to merge the value of `p[2]` for the merging group $\{s_1, s_5, s_7, s_8\}$, then assuming that the memory object pointed to by p is $(b, n, 3, a)$, we get:

$$ite(\varphi_1, select(a, 2), ite(\varphi_5, select(a, 2), ite(\varphi_7, select(a, 2), 2)))$$

When applying the optimized procedure, we get:

$$ite(n \leq 0, select(a, 2), ite(n \leq 1, select(a, 2), ite(n \leq 2, select(a, 2), 2)))$$

which reduces the size of the expression from $O(n^2)$ to $O(n)$.

To incorporate Algorithms 5 and 6 in Algorithm 4, we replace the original invocations of *merge-conditions* and *merge-values* at lines 14, 16, and 7 with the

invocations of *merge-conditions-opt* and *merge-values-opt* at lines 15, 17, and 8, respectively.

Algorithms 5 and 6 are independent of the search heuristic used by the SE engine, since the structure of the execution tree is derived only from the program. The time (and space) complexity of these algorithms is linear in the number of nodes, i.e., linear in the number of symbolic states to be merged. This is an improvement over Algorithm 4, where the worst case complexity is quadratic.

In Appendix A.2, we prove that Algorithms 5 and 6 are sound and complete w.r.t. standard state merging.

5.2.4 Limitations

The size bound of a symbolic-size memory object, i.e., the maximum concrete value of its symbolic size, can be less than its specified capacity. In such cases, if we read a value at a symbolic offset from this memory object, this value will contain array updates (*store* expressions) over offsets that will be never accessed in the future. This does not affect the read value, but leads to a more complex expression which might have a negative effect on the SMT solver. For example, consider the following code snippet:

```
1 size_t n; // symbolic
2 char *p = calloc(n, 1); // capacity is 3
3 if (n > 0 && n < 3) {
4     p[n - 1] = 17;
5     if (p[0] == 0) {
6         ...
7     }
8 }
```

Assuming that the allocated memory object at line 2 is $(b, n, 3, a)$, the value of `p[0]` at line 5 will be:

$$select(store(store(store(store(a, 0, 0), 1, 0), 2, 0), n - 1, 17), 0)$$

The branch at line 3 forces the constraint $1 \leq n \leq 2$, so the array update that writes 0 at offset 2 becomes irrelevant as the actual size bound is 2. To overcome this, one needs to know the actual bound of a given symbolic size, which is not straightforward as it requires generating additional expensive queries.

The approaches presented in Section 5.2.2 and Section 5.2.3 have the known limitations of state merging: In practice, the number of symbolic states that can be merged while keeping the analysis efficient is limited, as complex representations affect both memory usage and constraint solving.

5.3 Implementation

We implemented our symbolic-size model on top of KLEE [39], a *state-of-the-art* symbolic executor that operates on LLVM bitcode [78]. Our extension of KLEE is configured with LLVM 7.0.0 and STP 2.3.3 [59]. Originally, when a memory object is allocated with a symbolic size, KLEE concretizes the size expression and performs the allocation with the resulting concrete size. We modified that part such that instead of concretizing, we allocate a symbolic-size memory object (as described in Section 5.2.1), while its capacity is given by the user via a *command-line* option. If the user-specified capacity is too low, i.e., the symbolic size is always greater than the capacity under the path constraints of the corresponding symbolic state, then the capacity is gradually increased until that constraint becomes feasible. In addition, we modified the relevant parts which handle memory access operations and pointer resolution, in order to handle appropriately symbolic-size memory objects. We avoid state merging when the number of states exceeds a user-specified threshold,[2] as applying it in such cases leads to high memory usage and poor performance of the SMT solver. We disable state merging in loops which contain function calls and nested loops as the number of explored states in such cases is typically too high.

5.4 Evaluation

In our experiments, we evaluate the *concrete-size* and the *symbolic-size* models in the context of *API* testing and *whole-program* testing.

5.4.1 Experimental Setup

Vanilla KLEE concretizes symbolic size expressions using the SMT solver, which means that the user has little control over the resulting concretized size values. Throughout experimentation, we observed that concretizations often result in small size values, e.g.,

[2] In the evaluation, this threshold is set to 10,000.

0 or 1, which leads to fast analysis times with a rather low code coverage. Therefore, we chose a more competitive baseline mode (*Base*) which concretizes the symbolic size to its maximum feasible value w.r.t. the specified capacity.

The other modes we use in the evaluation are denoted as *range* modes, where all the possible sizes of a given symbolic-size memory object are considered, resulting in a complete exploration w.r.t. the specified capacity. We compare between several range modes: Under the concrete-size model, we use an eager forking mode (*ForkEager*) which forks at allocation time for each feasible value of the symbolic size expression. Under the symbolic-size model described in Section 5.2.1, we use a lazy forking mode (*ForkLazy*) which forks on-demand, and the two merging modes (*SM* and *SMOpt*) described in Section 5.2.2 and Section 5.2.3, respectively.

We performed our experiments on several machines with Intel i7-6700, 32 GB of RAM, and Ubuntu 16.04 as the operating system. We make our implementation[3] and the associated replication package[4] available as open-source.

5.4.2 API Testing

The benchmarks used in this experiment are: *libtasn1* [14] v4.16.0 (15K SLOC), *libpng* [13], v1.6.37 (56K SLOC), and *libosip* [12] v5.2.0 (19K SLOC). The *libtasn1* library is used for processing data in the Abstract Syntax Notation One (ASN.1) format, the *libpng* library is the official PNG image file format reference, and the *libosip* library is used for parsing and building messages for the SIP protocol. We chose these libraries as they are challenging for symbolic execution and contain many APIs that depend on variable-size memory objects.

In each benchmark, we focused on APIs whose inputs can be modeled using symbolic-size memory objects, i.e., arrays and strings. We manually constructed a test driver for each such API, based on the available documentation and the various usage examples found in the corresponding library.

We analyzed a total number of 65 APIs across the different benchmarks: 17 APIs from the decoding and encoding modules in *libtasn1*, 13 APIs from the *pngread* and *pngwrite* modules in *libpng*, and 35 APIs from the *osipparser2* module in *libosip*.

For each API, we run KLEE in the five modes (*Base*, *ForkEager*, *ForkLazy*, *SM*,

[3] https://github.com/davidtr1037/klee-symsize
[4] https://doi.org/10.6084/m9.figshare.14724453

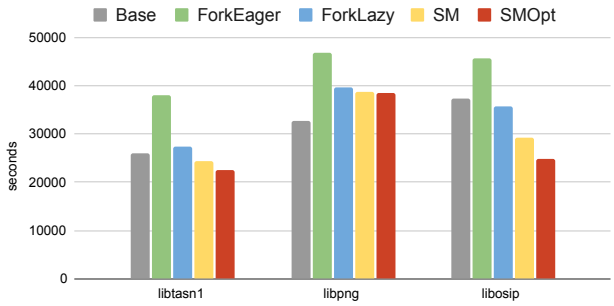


Figure 5.5: Total analysis time.

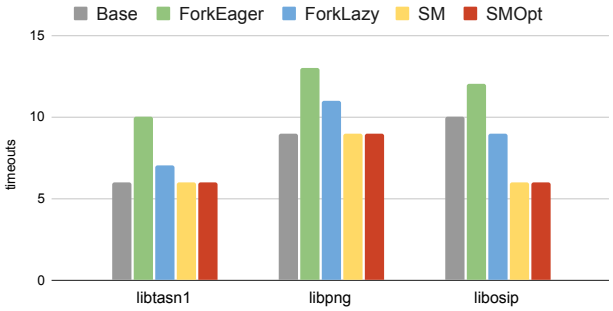


Figure 5.6: Number of timeouts.

and *SMOpt*) with the deterministic DFS search heuristic, a one hour time limit, and a 4GB memory limit. In each run we check the following metrics: analysis time, number of queries, line coverage computed with GCov [11], and number of explored paths.

5.4.2.1 Empirical Validation

In APIs where all the range modes achieved full exploration, i.e., completed the analysis before the timeout, we validated that the achieved coverage is identical across these modes. Note that the *Base* mode is not considered here, as it is generally less complete in terms of exploration, making a coverage based comparison meaningless. As the optimizations described in Section 5.2.3 must not affect the exploration during the analysis, we additionally validated that the number of explored paths in the *SM* and *SMOpt* modes is indeed identical.

5.4.2.2 Analysis Time

The *Base* mode uses concretization to handle symbolic-size allocations, therefore it cannot explore more paths than *ForkEager* and *ForkLazy*, which are also forking-based approaches. As a result, the analysis time with *Base* is expected to be lower compared

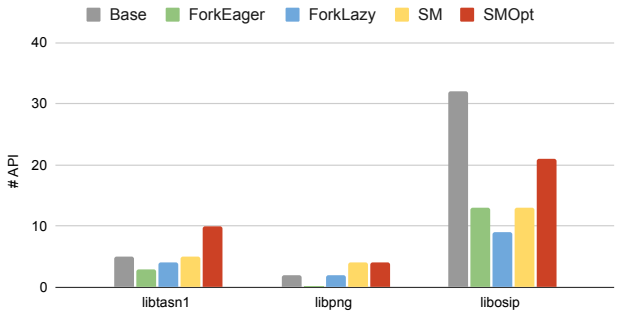


Figure 5.7: Analysis time scoreboard.

to *ForkEager* and *ForkLazy*. The *SM* and *SMOpt* modes use state merging, which might result in less paths compared to other modes and faster analysis even compared to *Base*.

Figure 5.5 shows for each benchmark and mode the total time required to analyze all the APIs. The *Base* mode was the fastest in *libpng* but still slower than *SMOpt* in *libtasn1* and *libosip*, despite its less complete handling of symbolic-size allocations. Among the range modes, *SMOpt* had the lowest analysis time across all the benchmarks. The slowest mode among all the modes was *ForkEager*, mainly due to its early forking mechanism.

As we analyze each API with a timeout of one hour, we also examine the cases which resulted in a timeout. Figure 5.6 shows for each benchmark and mode the number of APIs in which a timeout occurred. The merging modes had the lowest number of timeouts across all the benchmarks, and in *libtasn1* and *libpng* the *Base* mode had the same number of timeouts as the merging modes. In each of the benchmarks, the highest number of timeouts occurred in the *ForkEager* mode.

We now examine the results of 44 APIs in which the analysis completed before the timeout at least in one of the modes. Figure 5.7 shows for each benchmark and mode the number of APIs in which a given mode had the fastest analysis time. The highest score was achieved by *SMOpt* in *libtasn1* and *libpng*, and by *Base* in *libosip*. Note that in *libosip*, *SMOpt* had relatively high scores as well.

A more detailed comparison between the two merging modes is given in the scatter plot from Figure 5.8, where the x-axis and y-axis represent the analysis times for *SM* and *SMOpt*, respectively. Across all the APIs, the speedup of *SMOpt* relatively to *SM* varies between 0.9×-4.5×, and its average is 1.4×. The *SM* mode was faster than the *SMOpt* mode only in one case, where the difference was less than 5 seconds.

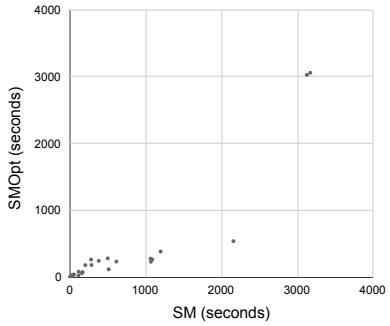


Figure 5.8: Analysis times of *SM* vs *SMOpt* (in *seconds*).

> *SMOpt* achieves the lowest total analysis time among the range modes, and in some cases, it even outperforms *Base*, whose exploration is less complete.

5.4.2.3 SMT Solver Queries

In addition to comparing the analysis times, we also compare the number of queries generated by each of the modes. Here, we report the number of queries that actually reached the SMT solver, i.e., those that were not handled by any of the constraint solving heuristics in KLEE (e.g., query caching). Note that here we consider 30 APIs in which all the modes reached full exploration, as otherwise the comparison would be meaningless.

The lowest number of queries was generated by the *SMOpt* mode, with an average of 3147 queries per API. The number of queries with *SM* was slightly higher, and as for the other modes, the relative increase in the average number of queries w.r.t. *SMOpt* was 15% in *Base*, 41% in *ForkLazy*, and 68% in *ForkEager*. When comparing between the two merging modes *SM* and *SMOpt*, the number of queries is roughly the same. The slight differences between these modes originates from the different representation of the merged symbolic states, which affects the various heuristics used in KLEE's solver chain.

> The state merging modes (*SM* and *SMOpt*) generate less queries compared to other modes.

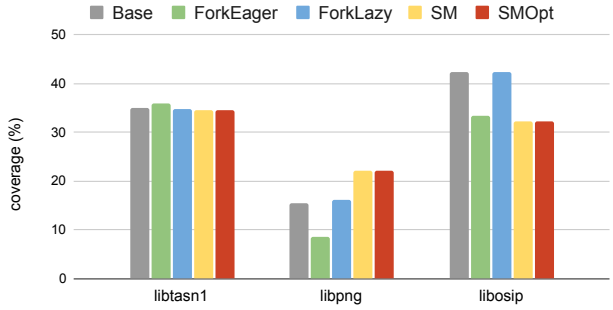


Figure 5.9: Total line coverage.

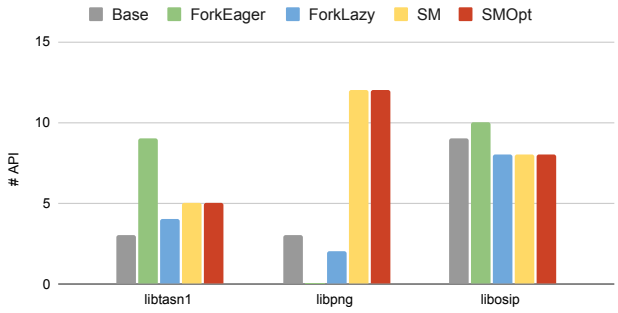


Figure 5.10: Line coverage scoreboard.

5.4.2.4 Coverage

Figure 5.9 shows for each benchmark and mode the total coverage generated by the test cases of all the APIs. In *libtasn1*, all the modes achieved similar results, with the *ForkEager* mode having a slight advantage. In *libpng*, the highest coverage was achieved by the merging modes, with an improvement of 37% compared to *ForkLazy*, 42% compared to *Base*, and 160% compared to *ForkEager*. In *libosip*, the highest coverage was achieved by *Base* and *ForkLazy*, with an improvement of 25% and 30% compared to *ForkEager* and the merging modes, respectively.

We now discuss in more detail the results of 35 APIs in which at least one of the modes had a timeout. Figure 5.10 shows for each benchmark and mode the number of APIs in which a given mode achieved the highest coverage compared to other modes. In *libtasn1* and *libpng*, the highest scores were obtained by the *ForkEager* mode and the merging modes, respectively. In *libosip*, the *ForkEager* mode had the highest score, while the other modes had slightly lower scores.

In 20 out of the 35 APIs mentioned above, the merging mode had a timeout and achieved the same coverage. To further evaluate the results in these cases, we use an additional evaluation metric: *path coverage*. Note that comparing between the

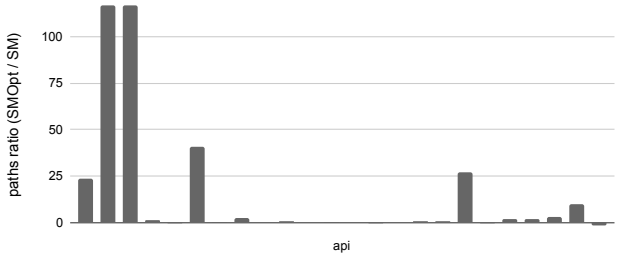


Figure 5.11: Increase in path coverage (%) of *SMOpt* vs. *SM* in cases where line coverage is identical.

merging modes using this metric makes sense: The only difference between the merging modes is the representation of merged symbolic states, therefore the exploration order of the search space remains identical. Note that this metric cannot be used to compare the other modes with the merging modes, since their exploration differs due to the introduction of state merging.

Figure 5.11 shows the increase in the number of explored paths with *SMOpt* relatively to *SM*. The increase in path coverage is 18% on average and varies between -1% and 116%. In the three cases were *SM* explored more paths than *SMOpt*, the improvement was negligible and resulted from the non-determinism of the timeout mechanism in KLEE, which may lead to slightly different running times under the same timeout configuration.

> The state merging modes (*SM* and *SMOpt*) achieve more line coverage in some cases, while in other cases, the forking modes perform better. In terms of path coverage, *SMOpt* generally performs better than *SM*.

5.4.2.5 Merging Complexity

The main problem of state merging originates from *disjunctive* constraints and *ite* expressions introduced during the merging, which propagate to the queries thus making constraint solving harder. We provide an additional comparison between the two merging modes (*SM* and *SMOpt*) based on the representation complexity of the symbolic states, i.e., the size of the constraints and the memory values resulting from the merging. In order to have a meaningful comparison, we compare only the results of 44 APIs in which both of the modes reached full exploration.

Figure 5.12 shows for each API the ratio of the total size of all the merged constraints

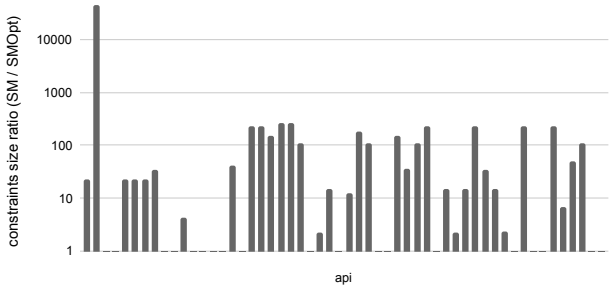


Figure 5.12: Decrease in constraints complexity.

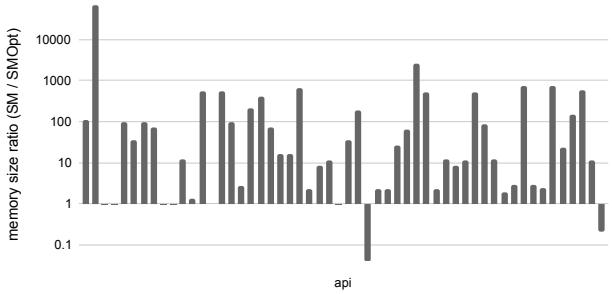


Figure 5.13: Decrease in memory complexity.

between *SM* and *SMOpt*.[5] The size of the constraints with *SMOpt* is never greater than in *SM*, while the average ratio is 12×. Figure 5.13 shows for each API the ratio of the total size of all the merged values, i.e., variables and heap memory objects, between *SM* and *SMOpt*. Here again, *SMOpt* has a clear advantage over *SM*, while the average ratio is 24×.

> *SMOpt* significantly reduces the encoding size of the merged symbolic states compared to *SM*.

5.4.2.6 Case Study: *libosip*

In *libtasn1* and *libpng*, the merging modes performed well compared to other modes, but in *libosip*, these modes were less efficient. We now characterize the cases where the merging modes performed better or worse compared to other modes. A scenario where state merging worked better, occurs when we have multiple independent operations on different inputs. On the other side, a scenario where state merging worked worse, occurs when we have multiple subsequent operations on the same input, while each operation depends on the previous ones. Two APIs that demonstrate these two scenarios

[5]The size of an expression is defined by the number of nodes in its AST representation.

Table 5.1: Crashing inputs for the *libosip* bugs

API	Input	Size Range
osip_message_set_via	'/\x01/ '	≥ 5
osip_uri_parse_headers	'='	≥ 2
osip_uri_parse_headers	''	$= 1$
osip_uri_parse_params	''	$= 1$

are *sdp_messsage_to_str* and *osip_message_parse*. In the first case, we receive a struct describing an *SDP* (session description protocol) message and translate it to a string. This struct contains several fields which are strings as well, and the translation is performed on each of them independently. In this case, *SMOpt* completes the analysis in 37 seconds, *Base* runs for 148 seconds, *ForkLazy* runs for 337 seconds, and *ForkEager* hits the timeout of one hour. In the second case, we receive a symbolic-size string, parse it, and return a struct that describes the parsed message. Here, we have a chain of *strchr* invocations on the symbolic input, where each invocation depends on the previous one. The loop inside *strchr* is merged, as it is detected as size dependent, so the next call to *strchr* operates on a more complex symbolic state than the previous call. In this case, *Base* and *ForkEager* complete the analysis in 2 seconds, *ForkLazy* in 13 seconds, while in *SM* and *SMOpt* it takes 284 and 186 seconds, respectively.

5.4.2.7 Found Bugs

Throughout our experiments, we found five bugs in two of our benchmarks: *libosip* and *libtasn1*. In *libosip*, we found three *out-of-bounds read* bugs and one *integer underflow* bug, all of which were triggered by symbolic-size memory objects, strings in this case. Table 5.1 shows the APIs in which the bugs were found, the corresponding triggering inputs, and the range of input sizes under which the bug is reachable. We reported the bugs and they were confirmed and fixed by the official maintainers of *libosip* [15, 16]. We note that *Base* misses some of these bugs when the capacity is too high or too low, due to its single-size out-of-bounds reasoning. In *libtasn1*, we found another *out-of-bounds read* bug in the `ETYPE_OK` macro, which is used in several APIs in the library. The bug happens due to an incorrect range-check of an array index, that can be triggered with a specific *element type* value. In this case, the bug was not directly triggered by a symbolic-size memory object, although changing the size of some API parameters makes this bug unreachable.

5.4.3 Whole-Program Testing

As a benchmark for whole-program testing, we chose 99 programs from *coreutils* [10]. With vanilla KLEE, these programs are analyzed with symbolic command-line arguments (*argv*) and files (*stdin*, *stdout*, etc.). Every such symbolic input is modeled as a concrete-size memory object with a user-specified size. Symbolic-size allocations are not common in *coreutils*, so in order to have a more insightful evaluation in our context, we model those inputs using symbolic-size memory objects. Note that the *Base* mode with such modeling behaves like vanilla KLEE with the original modeling.

In this experiment, we run each program in the five modes with a timeout of one hour, and measure the analysis time and the line coverage. We had only five programs in which not all the modes had a timeout: In two cases, all the modes terminated within a second except for the *ForkEager* mode. In the other three cases, the merging modes terminated faster compared to other modes with an average speedup of 3.0× compared to *Base*, 3.3× compared to *ForkLazy*, and 151.8× compared to *ForkEager*. In the other 94 programs where all the modes had a timeout, the average coverage with the different modes varies between 28.3%-31.9%, where the best and worst result was achieved by *ForkEager* and *ForkLazy*, respectively. The highest coverage was achieved in 38 cases with *ForkEager*, in 35 cases with *Base*, in 24 cases with *SMOpt*, in 24 cases with *SM*, and in 19 cases with *ForkLazy*. In some of the cases, several modes achieved the same coverage.

The two merging modes achieved identical coverage in all but 16 cases (out of 94). In three of these cases, *SMOpt* generated more test cases and achieved higher coverage. In the rest 13 cases, both modes generated the same number of test cases. However, some of the test cases were generated differently due to the difference in the representation of the constraints with the two modes, which eventually resulted in slightly different coverage. There was no significant difference between these two modes in terms of path coverage, and *SMOpt* had a slight advantage over *SM*.

> *ForkEager* performs better in most of the programs, but the state merging modes (*SM* and *SMOpt*) still perform better in a considerable number of programs.

5.4.4 Discussion

We evaluated several approaches that consider a range of size values: *ForkEager*, *ForkLazy* and the two merging modes. There is a tradeoff here between the number of explored paths and the complexity of the resulting path constraints: The eager approach has the highest number of paths and the least complex constraints, the merging approach has the lowest number of paths and the most complex constraints, and the lazy approach lies between them.

We believe that this classification allows to explain the results of our experiments: We experiment with programs that operate on both textual (*libosip* and *coreutils*) and binary (*libtasn1* and *libpng*) inputs. The *character-by-character* sequential processing of strings requires considering every size in the range, thus giving an advantage to the eager approach. In contrast, the relatively higher granularity of binary data processing, i.e., accessing larger data chunks such as integers, filters out some irrelevant size values, thus giving the advantage to the other range modes (*ForkLazy*, *SM*, and *SMOpt*). This difference becomes even more significant when the programs operate on multiple symbolic-size memory objects. Furthermore, we model strings by assuming a null-terminator at the last byte, while permitting its occurrence earlier in the buffer. This allows the baseline approach (*Base*) to effectively consider a range of *logical* string sizes, and achieve similar coverage to the range modes.

Chapter 6

State Merging with Quantifiers in Symbolic Execution

6.1 Introduction

A key remaining challenge in symbolic execution is path explosion [37]. State merging [69, 77] is a well-known technique for mitigating this problem, which trades the number of explored paths with the complexity of the generated constraints. More specifically, merging multiple symbolic states together results in a symbolic state in which the path constraint is expressed using a disjunction of constraints and the memory contents are expressed using *ite* (if-then-else) expressions.

Unfortunately, the introduction of disjunctive constraints and *ite* expressions makes constraint solving harder and slows down the exploration, especially when the number of symbolic states to be merged is high. Consider, for example, the function `memspn` from Figure 6.1 which is based on the implementation of `strspn` in *uClibc* [117].[1] The function `memspn` receives a buffer `s`, the size of the buffer `n`, and a string `chars`, and returns the size of the initial segment of `s` which consists entirely of characters in `chars`. Suppose that `memspn` is called with a symbolic buffer `s`, a symbolic size `n` bounded by some constant m, and the constant string `"a"`. The exploration of the loop at lines 3-8 results in $O(m)$ symbolic states. If we merge these symbolic states, then the encoding of the merged symbolic state, which records, among others, the path constraint and the value of the variable `count`, is of size at least linear in m. Now, suppose that the merged return value of `memspn` is used later, for example, in the parameter `s` in another call of

[1] `strspn` receives null-terminated buffers which slightly complicates the presentation.

Figure 6.1: Motivating example.

```
 1 int memspn(char *s, size_t n, char *chars) {
 2     char *p = chars; int count = 0;
 3     while (*p && count < n) {
 4         if (*p == s[count]) {
 5             count++; p = chars;
 6         } else
 7             p++;
 8     }
 9     return count;
10 }
```

`memspn`. In that case, if we perform a similar merging operation, then the encoding of the merged symbolic state will be of size at least quadratic in m, since the merged value propagates to the path constraints. Such *encoding explosion* is typically encountered during the analysis of real-world programs, thus drastically limiting the effectiveness of state merging in practice.

We propose a state merging approach that reduces the encoding complexity of the path constraints and the memory contents, while preserving soundness and completeness w.r.t. standard symbolic execution. At a high level, our approach takes as an input the execution tree [76], which characterizes the symbolic branches occurring during the symbolic execution of the analyzed code fragment, and dynamically detects regular patterns in the path constraints of the symbolic states in the tree, which allows to partition them into merging groups of symbolic states whose path constraints have a similar *uniform* structure. This enables us to encode the merged path constraints using quantified formulas, which in turn may also simplify the encoding of *ite* expressions representing the merged memory contents.

We observed that the generic method employed by the SMT solver to solve the resulting quantified queries often leads to subpar performance compared to the solving of the quantifier-free variant of the queries. To address this, we propose a specialized solving procedure which leverages the particular structure of the generated quantified queries, and resort to the generic method only if our approach fails.

Main contributions:

1. We propose a state merging approach which uses quantified constraints.
2. We propose a specialized solving procedure to handle the resulting quantified constraints.

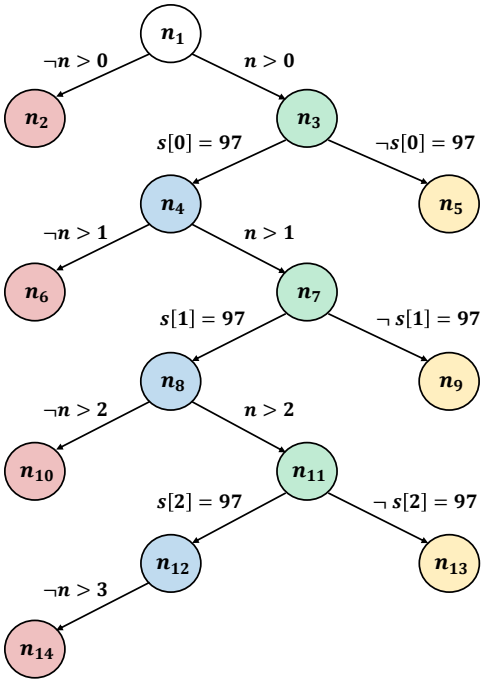


Figure 6.2: The execution tree of the loop from Figure 6.1 when `chars` is set to `"a"`. (Recall that the ASCII code of `a` is 97.)

3. We implement our approach on top of KLEE [39].

4. We evaluate our approach on real world benchmarks and find bugs.

Outline. In Section 6.2, we present our state merging approach which encodes the merged symbolic states using quantified constraints. In Section 6.3, we present an incremental state merging approach for reducing the size of complex execution trees. In Section 6.4, we present a specialized solving procedure to handle our quantified constraints. In Sections 6.5 and 6.6, we discuss our implementation and evaluation, respectively.

6.2 State Merging with Quantifiers

In this section we describe our approach for state merging with quantifiers. We start with a motivating example, and subsequently formalize our approach.

Motivating Example. Consider the symbolic execution of `memspn` (Figure 6.1) with a symbolic buffer `s`, a symbolic size `n`, and `"a"`, where `n` is bounded by 3 (i.e., $n_1.s.pc \triangleq n \leq 3$). The corresponding execution tree is depicted in Figure 6.2, where

the symbolic condition associated with each node is depicted on the incoming edge of the node.[2] Consider the symbolic states associated with the nodes n_5, n_9, and n_{13} from the execution tree in Figure 6.2, whose tree path conditions (Section 2.4) are:

$$tpc(n_5) \triangleq n > 0 \wedge \neg s[0] = 97$$
$$tpc(n_9) \triangleq n > 0 \wedge s[0] = 97 \wedge n > 1 \wedge \neg s[1] = 97$$
$$tpc(n_{13}) \triangleq n > 0 \wedge s[0] = 97 \wedge n > 1 \wedge s[1] = 97 \wedge n > 2 \wedge \neg s[2] = 97$$

The path constraint of the initial symbolic state ($n_1.s$) is $n \leq 3$, so applying standard state merging (Definition 2.5.1) on the symbolic states of the nodes above will result in a symbolic state whose path constraint is equivalent to:

$$n \leq 3 \wedge (tpc(n_5) \vee tpc(n_9) \vee tpc(n_{13}))$$

Note, however, that each of the disjuncts above has the following uniform structure: It uses k formulas (for $k = 0, 1, 2$) of the form $n > \_ \wedge s[\_] = 97$ to encode that the size of the buffer (n) is big enough to contain k consecutive occurrences of `a` characters, and another formula $n > k \wedge \neg s[k] = 97$. This uniformity is exposed when rewriting each disjunct using universal quantifiers as follows:

$$\big(\forall i.1 \leq i \leq 0 \rightarrow n > i - 1 \wedge s[i-1] = 97\big) \wedge n > 0 \wedge \neg s[0] = 97$$
$$\big(\forall i.1 \leq i \leq 1 \rightarrow n > i - 1 \wedge s[i-1] = 97\big) \wedge n > 1 \wedge \neg s[1] = 97$$
$$\big(\forall i.1 \leq i \leq 2 \rightarrow n > i - 1 \wedge s[i-1] = 97\big) \wedge n > 2 \wedge \neg s[2] = 97$$

To exploit the common structure of the rewritten disjuncts, we can introduce an auxiliary variable (k) and obtain an *equisatisfiable* merged path constraint:[3]

$$n \leq 3 \wedge (k = 0 \vee k = 1 \vee k = 2) \wedge$$
$$\big(\forall i.1 \leq i \leq k \rightarrow n > i - 1 \wedge s[i-1] = 97\big) \wedge$$
$$(n > k \wedge \neg s[k] = 97)$$

The auxiliary variable allows us to achieve similar savings in the encoding of the

[2] For now ignore the color of the nodes.
[3] Note that $(k = 0 \vee k = 1 \vee k = 2)$ can be rewritten as $0 \leq k \leq 2$.

merged memory contents. Consider, for example, the variable `count`. Its value in the symbolic states corresponding to n_5, n_9, and n_{13} is 0, 1, and 2, respectively, so its merged value with standard state merging is:

$$ite(tpc(n_5), 0, ite(tpc(n_9), 1, 2))$$

Note, however, that with the rewritten merged path constraint, the path constraints of the symbolic states corresponding to n_5, n_9, and n_{13} are now correlated with the values of k: 0, 1, and 2. As the values of `count` can be encoded as a function of those values, we can simply rewrite the complex *ite* expression above to k.

Our Approach. Our goal is to reduce the number of disjunctions and *ite* expressions introduced in standard state merging. Given a set of merge-compatible symbolic states, our state merging approach works as follows. First, we compute partitions of symbolic states based on the similarity of the path constraints (Section 6.2.1). Then, for each partition, we attempt to synthesize the merged symbolic state using universal quantifiers (Sections 6.2.2 and 6.2.3), and resort to standard state merging if that fails.

6.2.1 Identifying Merging Groups via Regular Patterns

To identify similarity between symbolic states, we use the execution tree of the analyzed code fragment. Recall that the symbolic states in each merging group are associated with leaf nodes and respective paths in the execution tree. We abstract each path to a sequence of numbers using a specialized *hash function*, which allows us to detect similarity between paths based on a shared regular pattern.

Definition 6.2.1. A *hash function* h maps constraints (formulas) to numbers ($\mathbb{N}$). We say that h is *valid* for an execution tree t if for any two sibling nodes n_1 and n_2:

$$h(n_1.c) \neq h(n_2.c)$$

In the sequel, we assume a fixed arbitrary valid execution tree t and a fixed arbitrary valid hash function h for t.[4] We now extend h to paths as follows:

[4] In practice, we use a hash function that distinguishes between a condition and its negation, which effectively ensures validity for any execution tree.

Definition 6.2.2. Given an execution tree t, the hash of a path $\pi(n_1, n_k) \triangleq n_1; ...; n_k$ in t is defined as a sequence of numbers:

$$h(\pi(n_1, n_k)) \triangleq h(n_1.c)\ h(n_2.c)\ \ldots\ h(n_k.c) \in \mathbb{N}^+$$

Note that the validity of h ensures that every path in t is identified uniquely by its hash value.

Definition 6.2.3. A *regular pattern* is a tuple $(\omega_1, \omega_2, \omega_3)$, where $\omega_1, \omega_2, \omega_3 \in \mathbb{N}^*$ are words (sequences) of numbers. Given an execution tree t, leaf nodes $\{n_j\}_{j=1}^n$ in t, and numbers $\{k_j\}_{j=1}^n \subseteq \mathbb{N}$, we say that $\{(n_j, k_j)\}_{j=1}^n$ *match* the regular pattern $(\omega_1, \omega_2, \omega_3)$ if for every $j = 1, ..., n$:

$$h(\pi(n_j)) = \omega_1 \omega_2^{k_j} \omega_3$$

where $\pi(n_j)$ denotes the sequence of nodes on the path from the root of t to the leaf node n_j (Section 2.4).

Definition 6.2.4. Given an execution tree t, a set of leaf nodes $\{n_j\}_{j=1}^n$ in t is called a *regular partition* if there exists a regular pattern $(\omega_1, \omega_2, \omega_3)$ and a set $\{k_j\}_{j=1}^n \subseteq \mathbb{N}$ such that $\{(n_j, k_j)\}_{j=1}^n$ match that pattern. A *regular partitioning* of leaf nodes in t is a partitioning into disjoint regular partitions.

Example 6.2.1. Consider a hash function h that operates on the abstract syntax tree (AST) of a formula and assigns the same pre-defined value to all the constant numerical terms. Such a hash function ensures that formulas with a similar shape will be assigned the same hash value, for example:

$$h(n > 0) = h(n > 1) = h(n > 2)$$

$$h(s[0] = 97) = h(s[1] = 97)$$

Figure 6.2 shows the resulting hash values of the nodes in the execution tree. For simplicity, we visualize every hash value as a distinct color: white (W), red (R), blue (B), green (G), and yellow (Y). Here, $\{(n_5, 0), (n_9, 1), (n_{13}, 2)\}$ match the regular pattern (W, GB, GY) since:

$$h(\pi(n_5)) = WGY,\ h(\pi(n_9)) = WGBGY,\ h(\pi(n_{13})) = WGBGBGY$$

Table 6.1: A regular partitioning of the leaf nodes of the execution tree in Figure 6.2, and the resulting merged symbolic states.

Pattern / Partition	**Pattern-Based Merged Symbolic States**
(W, GB, GY) /	*formula pattern* : $(true, n > x - 1 \wedge s[x-1] = 97, n > x \wedge \neg s[x] = 97)$
$\{n_5, n_9, n_{13}\}$	*pc* : $n \leq 3 \wedge 0 \leq k \leq 2 \wedge (\forall i.1 \leq i \leq k \rightarrow n > i - 1 \wedge s[i-1] = 97) \wedge (n > k \wedge \neg s[k] = 97)$
	vars : $[\texttt{count} \mapsto k, \quad \texttt{p} \mapsto chars + 1, \quad \texttt{s} \mapsto s, \quad \texttt{n} \mapsto n, \quad \texttt{chars} \mapsto chars\,]$
(W, GB, R) /	*formula pattern* : $(true, n > x - 1 \wedge s[x-1] = 97, n \leq x)$
$\{n_2, n_6, n_{10}, n_{14}\}$	*pc* : $n \leq 3 \wedge 0 \leq k \leq 3 \wedge (\forall i.1 \leq i \leq k \rightarrow n > i - 1 \wedge s[i-1] = 97) \wedge \neg n > k$
	vars : $[\texttt{count} \mapsto k, \quad \texttt{p} \mapsto chars, \quad \texttt{s} \mapsto s, \quad \texttt{n} \mapsto n, \quad \texttt{chars} \mapsto chars\,]$

A (possible) regular partitioning of the leaf nodes in Figure 6.2 is given in Table 6.1, which shows in the left column the regular patterns and their corresponding regular partitions.

In the next sections, we show how given a regular partition and its corresponding regular pattern, we can synthesize the resulting merged symbolic state using quantifiers.

6.2.2 Pattern-Based State Merging

A regular pattern indicates the potential existence of a uniform structure in the path conditions of the symbolic states in the associated regular partition. We formalize this intuition using *formula patterns*.

Definition 6.2.5. A *formula pattern* is a tuple $(\varphi_1, \varphi_2(x), \varphi_3(x))$, where φ_1 is a closed formula, and $\varphi_2(x)$ and $\varphi_3(x)$ are formulas with a free variable x. We say that $\{(n_j, k_j)\}_{j=1}^{n}$ *match* the formula pattern $(\varphi_1, \varphi_2(x), \varphi_3(x))$, if for every $j = 1, ..., n$:

$$tpc(n_j) \doteq \varphi_1 \wedge \Big(\bigwedge_{i=1}^{k_j} \varphi_2[i/x] \Big) \wedge \varphi_3[k_j/x]$$

The uniform structure exposed by formula patterns enables us to perform state merging with quantifiers:[5]

Definition 6.2.6. Let $\{n_j\}_{j=1}^{n}$ be a set of leaf nodes in an execution tree t such that:

- $\{n_j.s\}_{j=1}^{n}$ are merge-compatible (Section 2.5), and

[5]For simplicity of presentation, we do not describe here the handling of stack variables and heap-allocated objects. Our implementation supports both.

- $\{(n_j, k_j)\}_{j=1}^{n}$ match the formula pattern $(\varphi_1, \varphi_2(x), \varphi_3(x))$

The *pattern-based merged symbolic state* of $\{n_j.s\}_{j=1}^{n}$ is a symbolic state s whose path constraint, i.e., $s.pc$, is:

$$r.s.pc \wedge (\bigvee_{j=1}^{n} k = k_j) \wedge \varphi_1 \wedge (\forall i.\ 1 \leq i \leq k \rightarrow \varphi_2[i/x]) \wedge \varphi_3[k/x]$$

where k is a fresh constant, i is a fresh variable, and r is the root of t.

The symbolic store of s is defined as follows. For every variable v, if there exists a term $t(x)$ with a free variable x such that:

$$t[k_j/x] \doteq n_j.s.vars(v) \qquad \text{(for every } j = 1, \ldots, n)$$

then the value of v is encoded as follows:

$$s.vars(v) \triangleq t[k/x]$$

Otherwise, we use the standard merging algorithm (Definition 2.5.1):

$$s.vars(v) \triangleq \mathit{merge\text{-}var}(\{n_j.s\}_{j=1}^{n}, v)$$

Example 6.2.2. Consider the regular partition $\{n_5, n_9, n_{13}\}$ shown in the first row of Table 6.1. The formula pattern:

$$(true, n > x - 1 \wedge s[x-1] = 97, n > x \wedge \neg s[x] = 97)$$

is matched by $(\{(n_5, 0), (n_9, 1), (n_{13}, 2)\}$. The pattern-based merged symbolic state induced by that formula pattern is shown in the right column in Table 6.1 (pc and $vars$). Note that for the variable `count`, the term $t(x) \triangleq x$ satisfies:

$$t[0/x] = 0, t[1/x] = 1, t[2/x] = 2$$

so the merged value of that variable can be simplified to k. The merging of the other variables is rather trivial as the symbolic states being merged agree on their values.

Pattern-based state merging is sound and complete w.r.t. standard state merging. This is formalized in the following theorem:

Theorem 6.2.7. *Under the premises of Definition 6.2.6, let s be the pattern-based merged symbolic state of $\{n_j.s\}_{j=1}^{n}$, and let s' be their merged symbolic state obtained with standard state merging (Definition 2.5.1). The following holds for any model m:*

1. *$m \models s'.pc$ iff $m[k \mapsto \tilde{k}] \models s.pc$ for some $\tilde{k} \in \mathbb{N}$.*
2. *If $m[k \mapsto \tilde{k}] \models s.pc$ for some $\tilde{k} \in \mathbb{N}$, then $m(s'.vars(v)) = m[k \mapsto \tilde{k}](s.vars(v))$ for every variable v.*

The proof is in Appendix A.3.1. At a high-level, to prove (1), we use the fact that if $\{(n, k)\}$ matches the formula pattern $(\varphi_1, \varphi_2(x), \varphi_3(x))$, then:

$$tpc(n) \equiv \varphi_1 \wedge (\forall i.\ 1 \leq i \leq k \rightarrow \varphi_2[i/x]) \wedge \varphi_3[k/x]$$

and to prove (2), we use (1) and the fact that the formulas $\{tpc(n_j)\}_{j=1}^{n}$ are pairwise unsatisfiable.

6.2.3 Synthesizing Formula Patterns

So far, we did not discuss how formula patterns are obtained. We now describe an approach which attempts to synthesize a formula pattern given a regular pattern and its associated regular partition. As explained in Section 6.2.2, this enables us to perform state merging with quantifiers.

Our hash function h, which we assume to be valid for t (Definition 6.2.1), has the following useful property:

Lemma 6.2.8. *The following holds for any two nodes n_1, n_2 in t:*

1. *If $h(\pi(n_1)) = h(\pi(n_2))$ then $n_1 = n_2$.*
2. *If $h(\pi(n_1))$ is a prefix of $h(\pi(n_2))$, then there is a single path $\pi(n_1, n_2)$ in t.*

The proof is in Appendix A.3.2. At a high-level, the proof is based on the fact that t is a *valid* execution tree (Section 2.4).

Accordingly, we define:

Definition 6.2.9. Let $\omega_1, \omega_2 \in \mathbb{N}^+$ be two words such that:

$$h(\pi(n_1)) = \omega_1, \quad h(\pi(n_2)) = \omega_1\omega_2$$

for some nodes n_1, n_2 in t. Then we define:

$$extract(\omega_1) \triangleq tpc(n_1),\ extract(\omega_1, \omega_1\omega_2) \triangleq \overline{tpc}(n_1, n_2)$$

Note that Lemma 6.2.8 ensures that n_1 and n_2 are uniquely determined by ω_1 and ω_2.

In the following lemma, we show how a *tree path condition* can be represented using *extract*. This will be used later to prove Theorem 6.2.11.

Lemma 6.2.10. *Let n be a leaf node in an execution tree t, and suppose that:*

$$h(\pi(n)) = \omega_1\omega_2...\omega_j$$

Then:

$$\begin{aligned} tpc(n) \doteq\ & extract(\omega_1) \wedge \\ & extract(\omega_1, \omega_1\omega_2) \wedge \\ & ... \\ & extract(\omega_1...\omega_{j-1}, \omega_1...\omega_{j-1}\omega_j) \end{aligned}$$

The proof is in Appendix A.3.3. At a high-level, the proof is based on the definitions of *extract*, *tpc*, and $\overline{tpc}$ (Definition 6.2.9 and Section 2.4).

Now, we use *extract* to define the sufficient requirements to obtain a formula pattern from a given regular pattern.

Theorem 6.2.11. *Given an execution tree t and a set $\{n_j\}_{j=1}^n$ of leaf nodes in t, suppose that $\{(n_j, k_j)\}_{j=1}^n$ match the regular pattern $(\omega_1, \omega_2, \omega_3)$. If $(\varphi_1, \varphi_2(x), \varphi_3(x))$ is a formula pattern that satisfies:*

$$\begin{aligned} & \varphi_1 \doteq extract(\omega_1) \\ & \varphi_2[i/x] \doteq extract(\omega_1\omega_2^{i-1}, \omega_1\omega_2^i) \quad (i = 1, ..., max\{k_j\}_{j=1}^n) \\ & \varphi_3[k_j/x] \doteq extract(\omega_1\omega_2^{k_j}, \omega_1\omega_2^{k_j}\omega_3) \quad (j = 1, ..., n) \end{aligned}$$

then $\{(n_j, k_j)\}_{j=1}^n$ match $(\varphi_1, \varphi_2(x), \varphi_3(x))$.

The proof is in Appendix A.3.4. At a high-level, the proof applies Lemma 6.2.10 for each of the leaf nodes, and then uses the assumption about the given formula pattern.

Based on Theorem 6.2.11, we reduce the problem of finding a formula pattern to two synthesis tasks, for φ_2 and φ_3. (Note that φ_1 is trivially obtained from the first requirement of the theorem.) Each synthesis task has the form:

$$\varphi[d_\ell/x] \doteq \psi_\ell \quad (\ell = 1, ..., p)$$

where (1) $\varphi(x)$ is the formula to be synthesized (i.e., φ_2 or φ_3), (2) p is the number of equations (which is either $max\{k_j\}_{j=1}^{n}$ in the case of φ_2 or n in the case of φ_3), (3) $\{\psi_\ell\}_{\ell=1}^{p}$ are formulas (obtained from the extracted path constraints), and (4) $\{d_\ell\}_{\ell=1}^{p}$ are constant numerical terms (which are the i's in the case of φ_2 or the k_j's in the case of φ_3).

As synthesis is a hard problem in general, we focus on the case where all formulas in $\{\psi_\ell\}_{\ell=1}^{p}$ are syntactically identical up to a constant numerical term, i.e., there exists a formula $\theta(y)$ such that $\theta[\gamma_\ell/y] \doteq \psi_\ell$ for some numerical constants $\{\gamma_\ell\}_{\ell=1}^{p}$. To obtain $\varphi(x)$ from $\theta(y)$ it remains to synthesize a term that will express each γ_ℓ using the corresponding d_ℓ. Technically, if there exists a term $t(x)$ such that:

$$t[d_\ell/x] \equiv \gamma_\ell \quad (\ell = 1, ..., p)$$

then the desired formula $\varphi(x)$ will be given by $\theta[t(x)/y]$. When looking for such $t(x)$, we restrict our attention to terms of the form $a \cdot x + b$ where a and b are constant numerical terms that must satisfy:

$$\bigwedge_{\ell=1}^{p} (a \cdot d_\ell + b = \gamma_\ell)$$

The existence of such a and b can be checked using an SMT solver.

Example 6.2.3. Consider again the regular pattern (W, GB, GY) which is matched by

$\{(n_5, 0), (n_9, 1), (n_{13}, 2)\}$. We look for a formula pattern $(\varphi_1, \varphi_2(x), \varphi_3(x))$ that satisfies:

$$\begin{aligned}
\varphi_1 &\doteq true && extract(W)\\
\varphi_2[1/x] &\doteq n > 0 \wedge s[0] = 97 && extract(W, WGB)\\
\varphi_2[2/x] &\doteq n > 1 \wedge s[1] = 97 && extract(WGB, WGBGB)\\
\varphi_3[0/x] &\doteq n > 0 \wedge \neg s[0] = 97 && extract(W, WGY)\\
\varphi_3[1/x] &\doteq n > 1 \wedge \neg s[1] = 97 && extract(WGB, WGBGY)\\
\varphi_3[2/x] &\doteq n > 2 \wedge \neg s[2] = 97 && extract(WGBGB, WGBGBGY)
\end{aligned}$$

Consider, for example, the formulas associated with φ_2. First, note that they are identical up to a constant numerical term, e.g., for $\theta(y) \triangleq n > y \wedge s[y] = 97$:

$$\theta[0/y] \doteq n > 0 \wedge s[0] = 97 \qquad \theta[1/y] \doteq n > 1 \wedge s[1] = 97$$

Now we look for constant numerical terms a and b such that:

$$(0 = (a \cdot x + b)[1/x]) \wedge (1 = (a \cdot x + b)[2/x])$$

which is satisfied by $a \triangleq 1$ and $b \triangleq -1$, therefore:

$$\varphi_2(x) \triangleq \theta[(x-1)/y] \doteq n > x - 1 \wedge s[x-1] = 97$$

We similarly synthesize $\varphi_3(x) \triangleq n > x \wedge \neg s[x] = 97$.

If we succeeded to synthesize a formula pattern $(\varphi_1, \varphi_2(x), \varphi_3(x))$ matched by $\{(n_j, k_j)\}_{j=1}^{n}$, we attempt to synthesize the merged value of a variable v by synthesizing a term $t(x)$ that satisfies:

$$t[k_j/x] \doteq n_j.s.vars(v) \quad (j = 1, ..., n)$$

Such terms are synthesized similarly to formula patterns.

For each regular partition shown in Table 6.1, we automatically synthesize the formula pattern and the induced merged symbolic state using the aforementioned technique.

6.3 Incremental State Merging

When symbolically analyzing code fragments that contain disjunctive conditions, the number of generated symbolic states as well as the size of the generated execution trees might be exponential. In such cases, the exploration of the code fragment might not terminate within the allocated time budget and the analysis might not even reach the point where state merging, and pattern-based state merging in particular, can be applied.

To address this issue, we propose an *incremental* approach for state merging, in which we merge leaves in the execution tree not only with other leaves, but also with intermediate nodes, during the construction of the execution tree. This allows us to compress the execution tree as it is constructed. Once the construction of the execution tree is complete, we can apply our pattern-based state merging approach on the leaves. Technically, in addition to the *active* symbolic states, i.e., those that are stored in the current leaf nodes, we keep also the *non-active* symbolic states, i.e., those that are stored in the intermediate nodes. When a new leaf n_1 is added to the execution tree, we search for the highest node n_2, i.e., closest to the root, such that $n_1.s$ and $n_2.s$ are merge-compatible and have the same symbolic store *w.r.t. live variables* [20]. We additionally require that n_1 is unreachable from n_2 to avoid infinite sequences of merges. If such a node n_2 is found, we replace n_1 and n_2 (including the subtree of n_2) with a single merged node n_{new} that is added as a child of their lowest common ancestor, n_{lca}. We fix $n_{new}.c \triangleq \overline{tpc}(n_{lca}, n_1) \vee \overline{tpc}(n_{lca}, n_2)$ and $n_{new}.s$ is the merged symbolic state of $n_1.s$ and $n_2.s$. After the above, if a node p remains with a single child n, we remove p, redirect its incoming edge to n, and update the condition of n to $n.c \wedge p.c$. As we merge intermediate nodes, our approach does not rely on the search heuristic to synchronize between the active symbolic states to produce successful merges. To avoid nodes with more than two children, we require that n_{lca} is the parent of n_1 or n_2. (This restriction can be easily lifted.)

Example 6.3.1. Consider again the function `memspn` from Figure 6.1. When symbolically analyzing `memspn` while setting the value of the `chars` parameter to `"ab"`, instead of `"a"`, this results in an exponential execution tree. The upper part of Figure 6.3 shows the partial execution tree with some of the nodes that were added during the execution of the first iterations of the loop at line 3. Assuming that n_2 is added last, we merge it

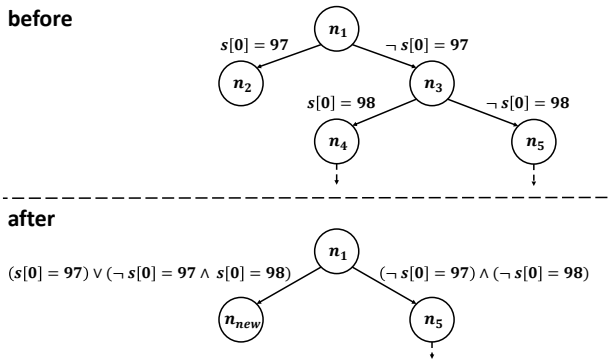


Figure 6.3: Execution tree transformation when `memspn` is called with `chars` set to `"ab"`.

with n_4 as the symbolic states associated with n_2 and n_4 are both located at line 5 and their symbolic stores w.r.t. live variables are identical, since `p` is dead at this location. We remove n_2 and n_4 together with its subtree, and add a new node n_{new} as a child of n_1, the lowest common ancestor of n_2 and n_4. Then, n_3 is left with its own child, n_5, so we remove n_3 and update the condition of n_5 appropriately. This results in the execution tree shown in the lower part of Figure 6.3. After applying similar steps in the subsequent iterations of the loop, the final execution tree is similar to the one from Figure 6.2, and can be obtained from it by replacing $s[i] = 97$ and $\neg s[i] = 97$ with $s[i] = 97 \vee (\neg s[i] = 97 \wedge s[i] = 98)$ and $\neg s[i] = 97 \wedge \neg s[i] = 98$, respectively (for $i = 0, 1, 2$). Now, the pattern-based approach can be applied similarly to the example given in Section 6.2.

The incremental state merging approach uses a standard liveness analysis [20] to find symbolic states to be merged. If the computed liveness results are imprecise, our approach will not be able to find matching symbolic states and therefore will not be able to compress the execution tree. In that case, our approach will only impose the overhead of maintaining snapshots of non-active symbolic states.

6.4 Solving Quantified Queries

In general, the quantified queries generated by our approach (Section 6.2) can be solved using an SMT solver that supports quantified formulas, e.g., Z3 [46]. In practice, however, we observed that the generic method employed by Z3[6] to solve such queries often leads to subpar performance compared to the solving of the quantifier-free variant of the queries. Hence, we devise a solving procedure which leverages the particular

[6] CVC5 [27] and Yices [53] failed to solve most of our queries.

structure of the generated quantified formulas, and resort to the generic method if our approach fails.

Before describing our solving procedure for quantified queries, we set some needed notations.

6.4.1 Notations

We assume closed formulas $\varphi = \bigwedge c$ where each clause c is either a quantifier-free formula θ or a universal formula of the form:

$$\forall i.\ 1 \leq i \leq k \rightarrow \psi$$

where ψ is a quantifier-free formula with a free variable i.

Definition 6.4.1. Given a quantified clause:

$$c \triangleq \forall i.\ 1 \leq i \leq k \rightarrow \psi$$

we denote its bound variable i by $\mathit{bound}(c)$.

Definition 6.4.2. Given a closed formula φ, we denote by $q(\varphi)$ and $qf(\varphi)$ the set of quantified and quantifier-free clauses of φ, respectively.

Definition 6.4.3. Given a formula f, we define:

$$\mathit{uconsts}(f) \triangleq \{n \mid n \textit{ is an uninterpreted constant in } f\}$$

Definition 6.4.4. An *access pair* is a pair (a, e) comprised of an SMT array a and a term e.

Definition 6.4.5. Given a formula f, we define:

$$\mathit{reads}(f) \triangleq \{(a, e) \mid a[e] \textit{ is a subterm of } f\}$$

to be the set of all access pairs coming from array access terms in f.

Definition 6.4.6. Given a formula f and a variable x, we define:

$$\mathit{vreads}(f, x) \triangleq \{(a, e) \in \mathit{reads}(f) \mid x \text{ is a subterm of } e\}$$

to be the set of all access pairs in which the index term contains the variable x.

Definition 6.4.7. Given a closed formula φ, we define:

$$qarrays(\varphi) \triangleq \bigcup_{c \in q(\varphi)} \{a \mid (a, e) \in vreads(c, bound(c))\}$$

to be the set of arrays accessed using a quantified variable.

Definition 6.4.8. Given a model m and an access pair (a, e), we define:

$$m(a, e) \triangleq (m(a), m(e))$$

and refer to it as a *semantic access pair*. We extend this notation to sets of such pairs in a point-wise manner.

6.4.2 Solving Procedure

Our solving procedure is given in Algorithm 7. Its main function is *compute-model* which works in four stages.

6.4.2.1 Quantifier stripping by formula weakening

The function *compute-model* starts by invoking $strip(\varphi)$ (line 36) which weakens φ into a quantifier-free formula φ_{QF} by replacing quantified clauses with implied quantifier-free clauses. More specifically, each quantified clause $\forall i.\ 1 \leq i \leq k \rightarrow \psi$ in φ is replaced with quantifier-free clauses stating that (a) the instantiation of ψ to $i = 1$, denoted $\psi[1/i]$, must hold if $0 < k$, and (b) if $\neg\psi[t/i]$ holds for some term t then t cannot be in the range $[1, k]$. Intuitively, the former provides a quantifier-free clause which partially preserves the properties imposed by the quantified clause, and the latter reduces the chances of getting a model of φ_{QF} that does not satisfy φ, since:

$$1 \leq t \leq k \models (\forall i.\ 1 \leq i \leq k \rightarrow \psi) \rightarrow \psi[t/i]$$

If the SMT solver fails to find a model for φ_{QF} than φ is also unsatisfiable. If a model was found, we check, optimistically, whether it is also a model of φ (line 38).

Example 6.4.1. Consider the following query, a simplification of a representative query from our experiments:

$$\varphi \triangleq (s[n] = 0) \wedge (1 \leq k \leq 10) \wedge (s[k-1] = 8) \wedge (\forall i.\ 1 \leq i \leq k \rightarrow s[i-1] \neq 0)$$

Note that (a) the instantiation of the quantified formula using $i = 1$ results in:

$$k \geq 1 \rightarrow s[0] \neq 0$$

and (b) $s[n] = 0$ is obtained by substituting $\neg(s[i-1] \neq 0)[n+1/i]$. Thus, the weakened query obtained by quantifier stripping is given by:

$$\varphi_{QF} \triangleq (s[n] = 0) \wedge (1 \leq k \leq 10) \wedge (s[k-1] = 8) \wedge (k \geq 1 \rightarrow s[0] \neq 0) \wedge \neg(1 \leq n+1 \leq k)$$

The following model, for example, is a model of φ_{QF}:

$$m \triangleq \{n \mapsto 7, k \mapsto 7, s \mapsto [1, 0, 0, 0, 0, 0, 8, 0]\}$$

but, unfortunately, is not a model of φ.

Note that if we would consider a different model of φ_{QF}:

$$m \triangleq \{n \mapsto 1, k \mapsto 1, s \mapsto [8, 0]\}$$

then we could get a satisfying model of φ.

Algorithm 7 A specialized solving procedure

function *strip*(φ)
  $\varphi_s \leftarrow \mathit{true}$
  for $(\forall i.\ 1 \le i \le k \to \psi) \in q(\varphi)$ **do**
    $\varphi_s \leftarrow \varphi_s \wedge (k \ge 1 \to \psi[1/i]) \wedge \big(\bigwedge\{\neg(1 \le t \le k) \mid (\neg\psi[t/i]) \in \mathit{qf}(\varphi)\}\big)$
  return $\big(\bigwedge \mathit{qf}(\varphi)\big) \wedge \varphi_s$
function *duplicate*($\varphi, m, \mathit{conflicts}$)
  for $(\forall i.\ 1 \le i \le k \to \psi) \in q(\varphi)$ **do**
    for $a \in \mathit{qarrays}(\psi)$ **do**
      $r \leftarrow \{(a', e) \in \mathit{vreads}(\psi, i) \mid a' = a\}$
      let $(a, e) \in r$
      $\tilde{v} \leftarrow m[i \mapsto 1](a[e])$
      for $2 \le n \le m(k)$ **do**
        $(\tilde{a}, \tilde{o}) \leftarrow m[i \mapsto n](a, e)$
        if $(\tilde{a}, \tilde{o}) \notin \mathit{conflicts}$ **then**
          $m(\mathit{select})(\tilde{a}, \tilde{o}) \leftarrow \tilde{v}$
  return m
function *repair*(φ, m)
  $\mathit{conflicts} \leftarrow \emptyset$, $\mathit{map} \leftarrow \{\}$
  for $(\forall i.\ 1 \le i \le k \to \psi) \in q(\varphi)$ **do**
    for $1 \le n \le m(k)$ **do**
      if $m[i \mapsto n] \not\models \psi$ **then**
        $\mathit{conflicts} \leftarrow \mathit{conflicts} \cup m[i \mapsto n](\mathit{vreads}(\psi, i))$
  for $\theta \in \mathit{qf}(\varphi)$ **do**
    if $m \not\models \theta$ **then**
      $\mathit{conflicts} \leftarrow \mathit{conflicts} \cup m(\mathrm{reads}(\theta))$
  for $(\forall i.\ 1 \le i \le k \to \psi) \in q(\varphi)$ **do**
    for $1 \le n \le m(k)$ **do**
      for $(\tilde{a}, \tilde{o}) \in m[i \mapsto n](\mathit{vreads}(\psi, i)) \cap \mathit{conflicts}$ **do**
        $\mathit{map}[(\tilde{a}, \tilde{o})] \leftarrow \mathit{map}[(\tilde{a}, \tilde{o})] \cup \{\psi[n/i]\}$
  $\mathit{terms} \leftarrow \{a[e] \mid (a, e) \in \mathit{reads}(\varphi) \wedge a \notin \mathit{qarrays}(\varphi)\} \cup \mathit{uconsts}(\varphi)$
  $\varphi' \leftarrow \mathit{strip}(\varphi) \wedge \big(\bigwedge_{(\tilde{a},\tilde{o}) \in \mathit{conflicts}} \mathit{map}[(\tilde{a}, \tilde{o})]\big) \wedge \big(\bigwedge_{t \in \mathit{terms}} t = m(t)\big)$
  $m' \leftarrow \mathit{smt\text{-}compute\text{-}model}(\varphi')$
  if $m' = \bot$ **then return** $\bot$
  return $\mathit{duplicate}(\varphi, m', \mathit{conflicts})$
function *compute-model*(φ)
  $\varphi_{QF} \leftarrow \mathit{strip}(\varphi)$
  $m \leftarrow \mathit{smt\text{-}compute\text{-}model}(\varphi_{QF})$
  if $m = \bot \vee m \models \varphi$ **then return** m
  $m_d \leftarrow \mathit{duplicate}(\varphi, m, \emptyset)$
  if $m_d \models \varphi$ **then return** m_d
  $m_r \leftarrow \mathit{repair}(\varphi, m_d)$
  if $m_r \neq \bot \wedge m_r \models \varphi$ **then return** m_r
  return $\mathit{smt\text{-}compute\text{-}model}(\varphi)$

6.4.2.2 Assignment Duplication

If m is not a model of φ, we use the function *duplicate* (line 39) to modify m into in a model m_d which assigns to every array cell accessed by a quantified clause a value of a cell in that array that was explicitly constrained by φ_{QF}. To do so, *duplicate* iterates over each of the quantified clauses $\forall i.\ 1 \leq i \leq k \rightarrow \psi$ of φ, and attempts to obtain a satisfying model for them based on m. If $m(k) < 1$, then the quantified clause is trivially satisfied. Otherwise, for every array a that ψ accesses using the quantified variable i, *duplicate* (1) records in r the set of access pairs coming from such accesses (line 9), (2) non-deterministically chooses one of these access pairs (a, e) (line 10), and (3) determines the value $\tilde{v}$ stored in a at the chosen index e when i is substituted by 1. Recall that the accessed cells of a in $\psi[1/i]$ were explicitly constrained by φ_{QF} due to the added instantiations (line 4), so the value $\tilde{v}$ assigned to them by m is a good candidate to fill in all the other array cells of a constrained by φ. Accordingly, the interpretation of *select* in m is modified such that every semantic access pair pertaining to the access pair (a, e) is mapped to $\tilde{v}$ (line 15). We explain the role of *conflicts* in the next stage, and for now, assume that it is an empty set. The duplication, however, is rather naive and might result in a model which does not even satisfy φ_{QF}.

Example 6.4.2. Continuing Example 6.4.1, we pick from the quantified clause the accessed offset $i - 1$ of the array s, and update the value of $s[j]$ to $m(s[i - 1][1/i])$ for each $1 \leq j \leq 6$. This results in the following model:

$$m_d \triangleq \{n \mapsto 7, k \mapsto 7, s \mapsto [1, 1, 1, 1, 1, 1, 1, 0]\}$$

The model m_d helps to satisfy the quantified clause, but does not satisfy φ due to the violation of the clause $s[k - 1] = 8$.

Note that if we would consider a different model of φ_{QF} in the stripping stage:

$$m \triangleq \{n \mapsto 7, k \mapsto 7, s \mapsto [8, 0, 0, 0, 0, 0, 8, 0]\}$$

then the model m_d obtained after assignment duplication could be:

$$m_d \triangleq \{n \mapsto 7, k \mapsto 7, s \mapsto [8, 8, 8, 8, 8, 8, 8, 0]\}$$

which does satisfy φ.

6.4.2.3 Model Repair

If m_d is not a model of φ, we invoke the function *repair* (line 41) to further modify m_d into another model, m_r, which, much like m_d, attempts to satisfy the constraints on the contents of arrays that are imposed by φ but omitted in φ_{QF}. However, it does so in a more principled way than *duplicate*: First, *repair* collects a set of semantic access pairs, called *conflicts*, from clauses that are not satisfied by m (lines 19-25). This set is used both to identify quantifier-free constraints that need to be added to φ_{QF}, and to later avoid overwriting "good" array contents. Second, *repair* iterates again over the quantified clauses of φ and collects for each semantic access pair $(\tilde{a}, \tilde{o})$ in *conflicts* the set of all instantiations that constrain it (lines 26-29). These instantiations are implied by φ in all models that agree with m on the value of k, which are our focus. Third, we strengthen $\mathit{strip}(\varphi)$ with the collected instantiations (line 31), but rather than computing a model for the strengthened query from scratch, we fix the values of array cells (and variables) according to their interpretation in m except for arrays that are accessed with i, those for which a new interpretation is sought. To do so, we add constraints that force the interpretation of closed terms to agree with their interpretation in m (line 31). Finally, if a model m' is found, then duplication is applied on m' (line 34). However, this time the semantic access pairs in *conflicts*, which were explicitly constrained when computing m', are excluded from the duplication in order to avoid their overwriting.

Example 6.4.3. Continuing Example 6.4.2, the violated clause in the model m_d is:

$$s[k-1] = 8$$

and its concrete access is $s[6]$. The concrete access in the instantiation $(s[i-1] \neq 0)[7/i]$ that was omitted in φ_{QF} is also $s[6]$, so we add it to φ_{QF}. In addition, we concretize the values of n and k according to m_d. The resulting strengthened query is given by:

$$\varphi_{QF} \wedge (s[6] \neq 0) \wedge (n = 7) \wedge (k = 7)$$

and its possible model is given by:

$$\{n \mapsto 7, k \mapsto 7, s \mapsto [1, 0, 0, 0, 0, 0, 8, 0]\}$$

Then, we duplicate again, but this time while skipping over the cell $s[6]$. Similarly to the first duplication, v is set to 1, but the value of $s[j]$ is updated only for $1 \leq j \leq 5$, thus avoiding the original violation. The resulting model indeed satisfies φ:

$$m_r \triangleq \{n \mapsto 7, k \mapsto 7, s \mapsto [1, 1, 1, 1, 1, 1, 8, 0]\}$$

6.4.2.4 Fallback

If no model m_r is found, or if it does not satisfy φ, we ask the SMT solver to find a model for φ (line 43).

6.5 Implementation

We implemented our state merging approach on top of KLEE [39], a *state-of-the-art* symbolic executor that operates on LLVM bitcode [78]. As our approach generates quantified queries over arrays and bit vectors, we use Z3 [48] as the underlying SMT solver. We extended KLEE's expression language to support quantified formulas, and modified accordingly the various parts of the solver chain. We implemented our solving procedure (Section 6.4) as an additional component in the solver chain. To implement the *hash* function used by the pattern-based state merging approach (Section 6.2), we relied on the expression hashing utility of KLEE and modified it by assigning a predefined hash value to all constants. To extract the regular patterns from the execution trees, we used a simple regular expression matching algorithm, which tries to detect repetitions in path hashes using forward and backward scanning. We dynamically check that the hash function is valid for the generated execution trees (Definition 6.2.1), and if that is not the case, then we fallback to standard state merging (Definition 2.5.1). In our experiments, however, we did not encounter such cases. If the number of extracted regular patterns in a given execution tree exceeds a user-specified threshold, then we also fallback to standard state merging.

6.6 Evaluation

Evaluating a state merging approach requires determining the desired merging points, i.e., the code segments where state merging should be applied. In our case, this translates to identifying code segments that produce merging operations that involve

many symbolic states. To do so, we evaluate our approach in the context of the *symbolic-size* model (Chapter 5). This model supports bounded symbolic-size memory objects, i.e., memory objects whose size can have a range of values, limited by a user-specified *capacity* bound.[7] In Section 5.4, it was observed that loops operating on symbolic-size memory objects typically produce many symbolic states, and state merging was suggested to combat the ensued state explosion problem. Thus, this memory model provides a suitable basis for evaluating our state merging approach. Furthermore, the automatic detection of merging points used in Section 5.2.2 helps avoiding manual annotations. We emphasize, however, that our technique is independent of the symbolic-size model itself (see Section 6.6.8). That said, the symbolic-size model does have the potential to produce more challenging merging operations than the concrete-size model as it considers a larger state space.

The following modes are the main subjects of comparison: The *PAT* mode is the pattern-based state merging approach described in Section 6.2 which partitions the symbolic states into merging groups based on regular patterns in the execution tree, and uses quantifiers to encode the merged path constraints. In the *PAT* mode, the incremental state merging approach (Section 6.3) and the solving procedure (Section 6.4) are enabled. The *CFG* mode is the state merging mode *SMOpt* described in Section 5.4, which partitions the symbolic states into merging groups according to their exit point from the loop in the CFG, and uses the standard *QFABV* encoding [59] (disjunctions and *ite* expressions). The *Base* mode is the forking approach used in vanilla KLEE [39].

The following research questions guide our evaluation:

1. Does *PAT* improve (optimized) state merging (*CFG*)?
2. Does *PAT* improve standard symbolic execution (*Base*)?
3. Do all components contribute to the performance of *PAT*?

6.6.1 Benchmarks

The benchmarks used in our evaluation are listed in Table 6.2. These benchmarks were chosen as they are challenging for symbolic execution and provide numerous

[7] This is in contrast to the standard *concrete-size* model where every memory object has a concrete size.

opportunities for applying state merging. In each benchmark we analyzed a set of subjects (APIs and whole programs) whose inputs can be modeled using symbolic-size memory objects, i.e., arrays and strings. In *libosip* [12], *libtasn1* [14], and *libpng* [13], the test drivers for the APIs were taken from Section 5.4. In *wget* [19], a library for retrieving files using widely used internet protocols (e.g., HTTP), we reused the test drivers from the existing fuzzing test suite whenever possible, and for other APIs we constructed the test drivers manually. In *apr* [23] (Apache Portable Runtime), a library that provides a platform-independent abstraction of operating system functionalities, we constructed test drivers for APIs from several modules (*strings*, *file_io* and *tables*) which manipulate strings, file-system paths, and data structures. In *json-c* [18], a library for decoding and encoding JSON objects, we constructed test drivers for APIs that manipulate string objects. In *busybox* [17], a software suite that provides a collection of Unix utilities, we focused on utilities whose input comes from command-line arguments and files, which can be symbolically modeled using KLEE's *posix* runtime. We did not analyze utilities whose behavior depends on the state of system resources (process information, permissions, file-system directories, etc.), since KLEE has no symbolic modeling for those. To prevent the symbolic executor from getting stuck in `getopt()`, the routine used in *busybox* to parse command line arguments, we added the restriction that symbolic command line arguments do not begin with a '`-`' character.

6.6.2 Setup

We run every mode under the symbolic-size model with the following configuration: a DFS search heuristic, a one hour time limit, and a 4GB memory limit. The capacity settings in each of the benchmarks are shown in Table 6.2. In each benchmark, we set the capacity to be high enough to produce complex merging operations. However, the capacity should not be too high, otherwise the analysis will be too complex with each of the modes.

In every experiment, we use the following metrics to compare between the modes: analysis time and line coverage computed with GCov [11]. When the compared modes have the same exploration order, we additionally use the path coverage metric, i.e., the number of explored paths.

Each benchmark consists of multiple subjects, so when comparing between two modes, we measure for each subject the relative speedup and the relative increase in

Table 6.2: Benchmarks.

	Version	**SLOC**	**#Subjects**	**Capacity**
libosip	5.2.1	18,783	35	10
wget	1.21.2	100,785	31	200
libtasn1	4.16.0	15,291	13	100
libpng	1.6.37	56,936	12	200
apr	1.6.3	60,034	20	50
json-c	0.15	8,167	5	100
busybox	1.36.0	198,500	30	100

coverage. Note that when we measure the average (and median) speedup, for example, the speedup in the subjects where both modes timed out is always 1×. Similarly, when we measure coverage, the coverage in the subjects where both modes terminated, i.e., completed the analysis before hitting the timeout, is always identical. To separate the subjects where the results are trivially identical, we report the average (and median) over a *subset* of the subjects depending on the evaluated metric: When measuring analysis time, we consider the subset of the subjects where at least one of the modes terminated. When measuring coverage, we consider the subset of the subjects where at least one of the modes timed out.

We performed our experiments on several machines with Intel i7-6700, 32 GB of RAM, and Ubuntu 20.04 as the operating system. We make our implementation[8] and the associated replication package[9] available as open-source.

6.6.3 Results: *PAT* vs. *CFG*

In this experiment, we compare between the performance of the state merging modes: *PAT* and *CFG*. The results are shown in Table 6.3 and Figure 6.4.

Analysis Time Column *Speedup* in Table 6.3 shows the (average, median, minimum, and maximum) speedup of *PAT* compared to *CFG* in the subjects where at least one of the modes terminated. Column # shows the number of considered subjects out of the total number of subjects. In *libosip*, *wget*, *apr*, *json-c*, and *busybox*, *PAT* was significantly faster in many subjects, and in *libtasn1* and *libpng*, the analysis times were roughly identical. Figure 6.4a breaks down the speedup of *PAT* compared to *CFG* per subject. Overall, there were 12 subjects where *PAT* was slower than *CFG*. In *libosip*,

[8] https://github.com/davidtr1037/klee-quantifiers
[9] https://doi.org/10.6084/m9.figshare.21990386

PAT was slower only in one API. In that case, the slowdown of 0.03× (from 20 to 554 seconds) was caused by a small number of queries (9) that our solving procedure (Section 6.4) failed to solve, and whose solving using the SMT solver required most of the analysis time. In *wget*, *PAT* was slower in two APIs. In one case, the slowdown was caused by the computational overhead of the incremental state merging approach. In the other case, the slowdown was caused by a relatively high number of queries that our solving procedure failed to solve. In *libtasn1*, *PAT* was slower in seven APIs, but the time difference in these cases was rather minor (roughly 10 seconds). In *libpng*, *PAT* was slightly slower in one API due to the computational overhead of extracting regular patterns. In *busybox*, *PAT* was slower in one utility with a minor time difference of two seconds. Column *Diff.* in Table 6.3 shows the difference between *PAT* and *CFG* in terms of the total time required to analyze all the subjects. Note that in subjects where both modes timed out, the time difference is interpreted as zero. In *libosip*, *wget*, *apr*, and *busybox*, *PAT* achieved a considerable reduction of roughly eight, four, one, and three hours, respectively. In *json-c*, *PAT* achieved a reduction of roughly 20 minutes, and in *libtasn1* and *libpng*, the time difference was minor. Figure 6.4b breaks down the time difference between *PAT* and *CFG* per subject.

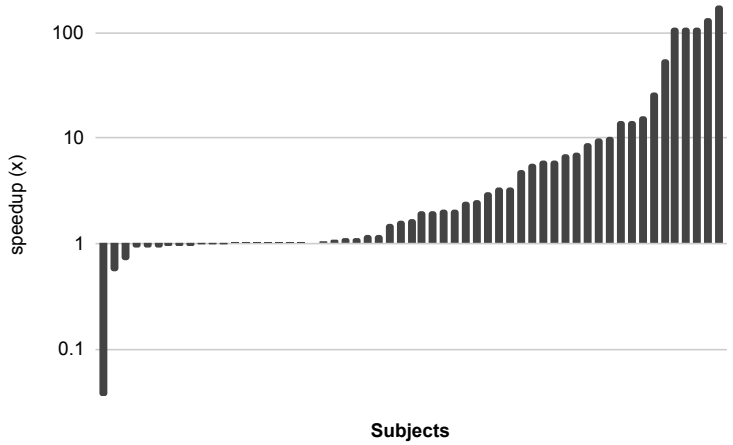
Coverage Column *Coverage* in Table 6.3 shows the (average, median, minimum, and maximum) relative increase in line coverage of *PAT* over *CFG* in the subjects where at least one of the modes timed out. As before, column # shows the number of considered subjects. In *libosip* and *wget*, *PAT* achieved higher coverage in many cases. In *libtasn1*, *PAT* resorted to standard state merging in most of the cases, as it did not find regular (and formula) patterns. Therefore, the results were similar to those of *CFG*, and there was no improvement in coverage. In *libpng*, the coverage was roughly identical in all the APIs except for two APIs where *PAT* achieved an improvement of 8.69% and 18.33%. In *apr*, the coverage was identical in all the APIs except for two cases where *PAT* had an increase of 16.62% and a decrease of 2.12%. In *json-c*, there was only one API where one of the modes timed out, and in that case, *CFG* achieved higher coverage. In *busybox*, there were 23 cases where at least one of the modes timed out. In four cases *PAT* achieved an improvement of 3.98%-15.45%, and in two cases *CFG* achieved an improvement of 1.15% and 61.78%. In the remaining 17 cases, the coverage was identical. (In most of these cases, *PAT* did not find formula patterns, which resulted

Table 6.3: Comparison of *PAT* vs. *CFG*.

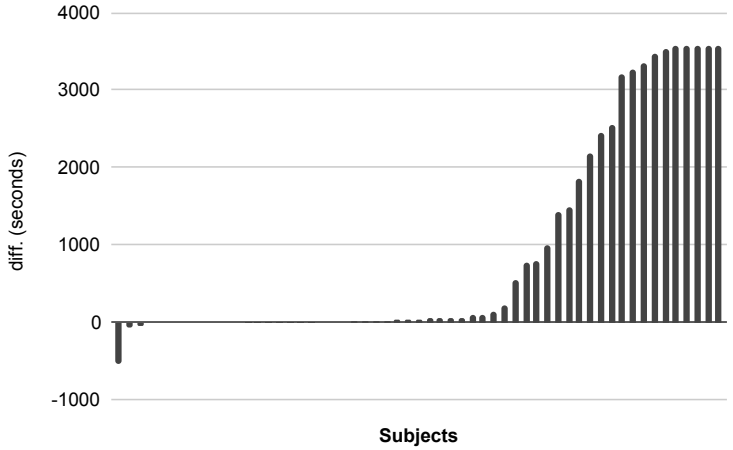
	Time						**Coverage (%)**					
	Speedup (×)					Diff. (seconds)						Diff. (lines)
	#	Avg.	Med.	Min.	Max.		#	Avg.	Med.	Min.	Max.	
libosip	16/35	7.18	5.50	0.03	180.00	27668	28/35	20.45	9.00	0.00	88.63	291
wget	11/31	2.69	1.67	0.54	14.69	12942	24/31	15.02	0.00	-40.00	300.00	89
libtasn1	7/13	0.94	0.95	0.90	0.96	-41	6/13	0.00	0.00	0.00	0.00	0
libpng	1/12	0.70	0.70	0.70	0.70	-9	11/12	2.03	0.00	-2.88	18.33	104
apr	10/20	3.50	1.63	1.00	138.46	4375	11/20	1.31	0.00	-2.12	16.62	0
json-c	4/5	3.16	2.97	2.00	5.76	1149	1/5	0.81	0.81	0.81	0.81	1
busybox	8/30	1.68	1.07	0.92	16.20	10100	23/30	-1.08	0.00	-61.78	15.45	74

in identical explorations.) Column *Diff.* in Table 6.3 shows the difference between *PAT* and *CFG* in terms of the total number of covered lines across all the subjects. Again, note that in subjects where both modes terminated, there is no difference in coverage. It is possible to have an improvement in average coverage but not in total line difference (*apr*), and vice versa (*busybox*). This happens due to shared code that is covered by only one mode in one subject but covered by the other mode in other subjects. Figure 6.4c breaks down the coverage improvement of *PAT* over *CFG* per subject. There were three cases where *CFG* achieved notably more coverage compared to *PAT*. In these cases, the merging operations that occurred at the beginning of the analysis resulted in different merging groups in each of the modes. As the partitioning into merging groups affects the exploration, each mode eventually covered different parts of the program.

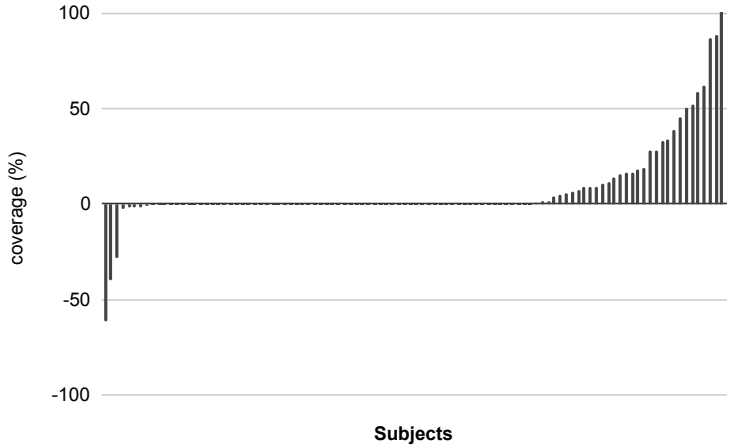
Scaling The main obstacle in applying state merging originates from the introduction of disjunctive constraints and *ite* expressions, especially when the number of symbolic states to be merged is high. We evaluate the ability of our approach to cope with a particular aspect of this challenge where the states are generated by loops iterating over large data objects, a frequent situation in our experience. Technically, we conducted a case study on *libosip*, one of our benchmarks, where we gradually increase the *capacity* of symbolic-size memory objects. When the capacity is increased, the size of the symbolic-size memory objects is potentially increased as well. This typically leads to additional forks, for example, in loops that operate on symbolic-size memory objects. As we apply state merging in such loops, this eventually results in more complex merging operations. Thus, increasing the capacity allows us to measure how each mode scales w.r.t the number of merged symbolic states. In this experiment, we run each API in each of the state merging modes (*PAT* and *CFG*) under several capacity settings. The



(a) Speedup in analysis time (×) in the subjects where at least one of the modes terminated (in log-scale).



(b) Difference in analysis time (*seconds*) in the subjects where at least one of the modes terminated.



(c) Relative increase in coverage (%) in the subjects where at least one of the modes timed out.

Figure 6.4: Breakdown of the improvement of *PAT* over *CFG* per subject.

Table 6.4: Comparison of *PAT* vs. *CFG* under different capacity settings (column *Capacity*) in *libosip*.

Capacity	**Speedup (×)**			**Coverage (%)**		
	#	Avg.	Med.	#	Avg.	Med.
10	16/35	7.18	5.50	28/35	20.45	9.00
20	13/35	4.58	5.53	29/35	23.41	19.29
50	12/35	1.99	2.43	30/35	15.19	10.63
100	5/35	2.99	2.75	30/35	10.23	2.32
200	5/35	4.81	6.11	30/35	4.22	0.00

results are shown in Table 6.4.

As can be seen, *PAT* achieved better results than *CFG* in all the capacity settings. In general, when the capacity is increased, there are typically more forks and queries, which makes the analysis of size-dependent loops harder for both modes. Therefore, under the highest capacity settings (100 and 200), the coverage improvement was less significant compared to the lower capacity settings. Note also that under those capacity settings, there were only five APIs in which at least one of the modes terminated. We observed that in these APIs the analysis time increased in both modes when the capacity was increased. However, with *CFG* the analysis time increased more significantly, so the speedup under the highest capacity setting (200) was greater. This indicates that our approach is less sensitive to the input capacity, and hence to the resulting number of merged symbolic states.

> *PAT* outperforms *CFG* in many cases and scales better in executing complex state merging operations.

6.6.4 Results: *PAT* vs. *Base*

In this experiment, we compare the performance of *PAT* and *Base*, i.e., standard symbolic execution that uses the forking approach. The results are shown in Table 6.5.

Column *Speedup* shows the (average and median) speedup of *PAT* compared to *Base* in the subjects where at least one of the modes terminated. As can be seen, *PAT* achieved a considerable speedup in the majority of the benchmarks. Overall, there were 9 subjects in which *PAT* was slower than *Base*. In three of these cases, the time difference was minor (roughly 5 seconds). In the other cases, the slowdown was caused by the computational overhead of the incremental state merging approach

Table 6.5: Comparison of *PAT* vs. *Base*.

	Speedup (×)			**Coverage (%)**		
	#	Avg.	Med.	#	Avg.	Med.
libosip	17/35	11.21	3.10	28/35	11.43	1.88
wget	12/31	2.75	3.72	24/31	-2.32	0.00
libtasn1	7/13	4.94	9.30	7/13	1.49	0.00
libpng	1/12	2.46	2.46	11/12	23.59	7.14
apr	10/20	8.40	3.91	14/20	-0.15	0.00
json-c	4/5	1.36	3.09	2/5	0.82	0.82
busybox	9/30	2.43	2.51	22/30	-2.76	0.00

and the complex constraints that were introduced during the state merging. Figure 6.5a breaks down the speedup of *PAT* compared to *Base* per subject, and Figure 6.5b breaks down the time difference between *PAT* and *Base* per subject. In terms of timeouts, there were 20 subjects in which *Base* timed out and *PAT* terminated, and only one subject in which *PAT* timed out and *Base* terminated.

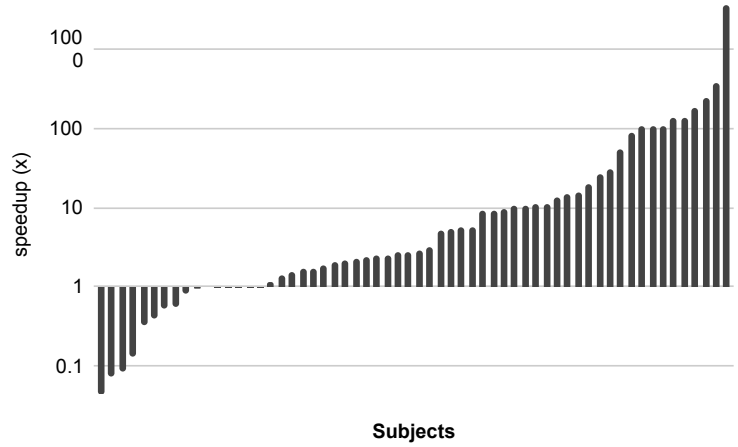
Column *Coverage* shows the (average and median) relative increase in line coverage of *PAT* over *Base* in the subjects where at least one of the modes timed out. *PAT* achieved higher coverage in many subjects, especially in *libosip* and *libpng*. In most of the cases in *libtasn1*, *apr*, and *json-c*, both of the modes covered the majority of the reachable lines in a relatively early stage, so the coverage was similar. In *wget* and *busybox*, *PAT* achieved higher coverage in some of the cases, but there were also cases in which *Base* achieved higher coverage. In general, this is a consequence of the known tradeoff between forking and state merging: The forking approach explores more paths but generates less complex constraints. Figure 6.5c breaks down the coverage improvement of *PAT* over *Base* per subject.

In addition, we observed that there were four subjects in which *Base* ran out of memory. In two of these cases, *Base* finished the analysis before *PAT*, but its analysis was incomplete since KLEE prunes the search space once the memory limit is reached.

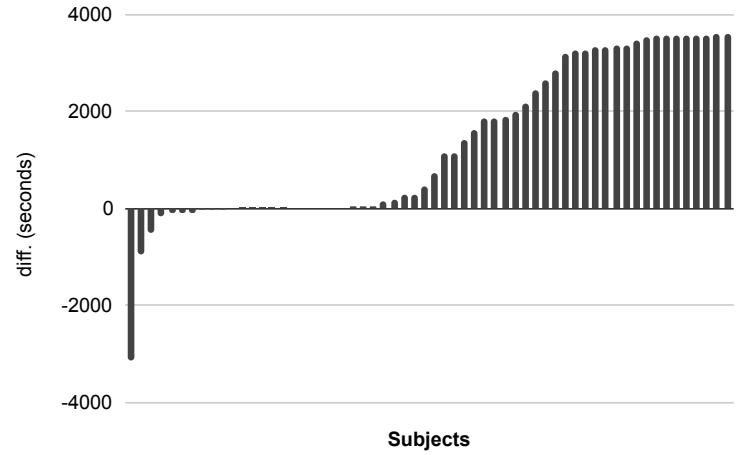
> *PAT* outperforms *Base* in many cases, however, the known tradeoff between state merging and forking remains.

6.6.5 Results: Component Breakdown

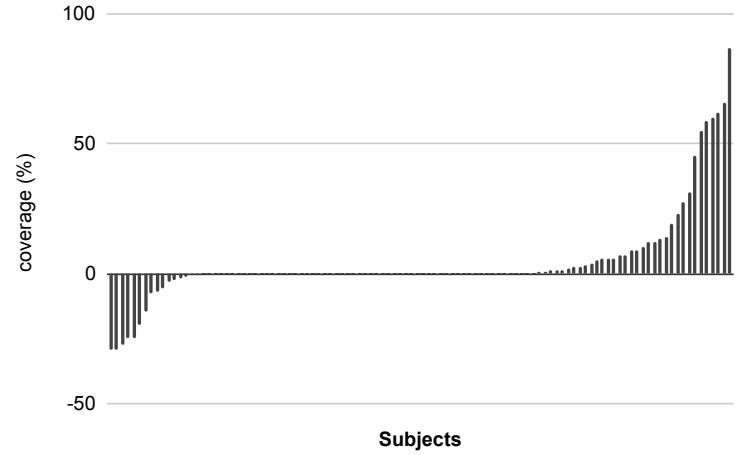
Now, we evaluate the significance of the components used in our pattern-based state merging approach (*PAT*).



(a) Speedup in analysis time (×) in the subjects where at least one of the modes terminated (in log-scale).



(b) Difference in analysis time (*seconds*) in the subjects where at least one of the modes terminated.



(c) Relative increase in coverage (%) in the subjects where at least one of the modes timed out.

Figure 6.5: Breakdown of the improvement of *PAT* over *Base* per subject.

Table 6.6: Effectiveness of solving procedure.

	Total	**Solved (%)**
libosip	517026	94
wget	208535	98
libtasn1	44	100
libpng	2411	86
apr	187700	99
json-c	7390	98
busybox	58013	98

6.6.5.1 Solving Procedure

To evaluate our solving procedure (Section 6.4), we ran each subject in two versions of *PAT*: one that relies only on the SMT solver (vanilla Z3) and another one that uses our solving procedure. Both modes are run with the incremental state merging approach enabled.

To evaluate the effectiveness of our solving procedure, we show its success rate in Table 6.6. Column *Total* shows the total number of generated quantified queries, and column *Solved* shows the percentage of queries that were solved by our solving procedure. The results show that the solving procedure was able to handle most of the generated queries. In addition, we measured the individual contribution of the different stages of our solving procedure: quantifier stripping (S), assignment duplication (D), and model repair (R). Table 6.7 shows the number of solved quantified queries in each of the stages: quantifier stripping only (S), quantifier stripping and assignment duplication ($S + D$), and the complete algorithm ($S + D + R$). Column *Fallback* shows the number of quantified queries that our solving procedure failed to solve. The results indicate that each stage plays a part in the overall efficacy of the procedure.

To evaluate the impact of the solving procedure, we show in Table 6.8 its effect on analysis time and coverage in the relevant subsets. Here, the two modes have the same exploration order, so we use the path coverage metric as well. In *libosip*, *wget*, *apr*, *json-c*, and *busybox*, our solving procedure generally leads to lower analysis times and higher (line or path) coverage. In *libtasn1* and *libpng*, the results were mostly similar since the number of quantified queries was relatively low. The only exception was one of the APIs in *libpng* where the path coverage was increased by 39.51%.

Table 6.7: The number of solved queries in the different stages of the solving procedure.

	S	$S + D$	$S + D + R$	Fallback
libosip	453688	1361	33868	28109
wget	187175	3759	15241	2360
libtasn1	44	0	0	0
libpng	2077	2	0	332
apr	135664	48014	3829	193
json-c	6205	246	859	80
busybox	51917	4837	260	999

Table 6.8: Impact of solving procedure.

	Speedup (×)			**Coverage (%)**				
					Line		Path	
	#	Avg.	Med.	#	Avg.	Med.	Avg.	Med.
libosip	16/35	1.55	1.57	19/35	0.26	0.00	89.31	72.82
wget	11/31	4.28	3.62	27/31	14.81	0.00	110.17	30.94
libtasn1	7/13	0.99	0.99	6/13	0.00	0.00	-0.74	-0.24
libpng	1/12	1.03	1.03	11/12	-0.23	0.00	2.62	0.00
apr	10/20	2.86	3.49	10/20	0.00	0.00	38.31	5.57
json-c	4/5	2.89	2.33	1/5	0.00	0.00	79.49	79.49
busybox	8/30	1.29	1.09	23/30	0.52	0.00	9.53	1.65

6.6.5.2 Incremental State Merging

To evaluate the incremental state merging approach (Section 6.3), we run each subject in two versions of *PAT*: one that disables incremental state merging and another one that enables it. The results are shown in Table 6.9.

In *libosip*, there were relatively many loops where incremental state merging was successfully applied, i.e., reduced the number of explored paths. This resulted in a significant speedup and in higher line coverage. In *wget*, there were four APIs where incremental state merging could be applied, and in two of these cases the coverage was improved by 33.33% and 300.00%. In *apr*, there were four APIs where incremental state merging could be applied, and in one of these cases the analysis time was reduced by 138.46× and the coverage was improved by 16.62%. In *busybox*, there were two utilities where incremental state merging could be applied, and in these cases the coverage was improved by 11.33% and 15.45%. In *libtasn1*, *libpng*, and *json-c*, there were no loops where incremental state merging could be applied. In some cases, this resulted in a minor performance penalty due to the computational overhead of the approach, which mainly comes from the need to maintain the snapshots of the non-active symbolic states

Table 6.9: Impact of incremental state merging.

	Speedup (×)			**Coverage (%)**		
	#	Avg.	Med.	#	Avg.	Med.
libosip	16/35	6.78	2.80	28/35	18.98	5.83
wget	11/31	0.97	0.97	20/31	16.66	0.00
libtasn1	7/13	0.96	0.98	6/13	0.00	0.00
libpng	1/12	0.96	0.96	11/12	2.35	0.00
apr	11/20	1.60	1.00	11/20	1.71	0.00
json-c	4/5	1.01	1.01	1/5	0.00	0.00
busybox	8/30	0.98	1.00	20/30	0.76	0.00

in the execution tree.

> All the components contribute to the performance of *PAT*.

6.6.6 Additional Experimental Results

To support the generality and robustness of our approach, we perform additional experiments and report additional results.

6.6.6.1 Concrete-Size Model

Recall that we evaluated our approach in the context of the symbolic-size model. To show that our approach may be beneficial in other contexts as well, we performed an additional experiment using the concrete-size model. In this experiment, we set the concrete sizes of the input memory objects according to the capacity settings in Table 6.2, and apply state merging in loops whose conditions depend on these sizes, as we do in our original experiments. The results are shown in Table 6.10. As can be seen, *PAT* achieved better results than *CFG* in most of the benchmarks.

As for the comparison between the speedup (of *PAT* compared to *CFG*) obtained with the concrete-size and symbolic-size (Table 6.3) models: In *libosip*, we achieved more speedup with the concrete-size model, and in the other benchmarks, we achieved more speedup with the symbolic-size model. As for the comparison between the relative increase in coverage (of *PAT* over *CFG*) obtained with the concrete-size and symbolic-size (Table 6.3) models: In *wget*, we had more improvement with the concrete-size model, in *libosip*, we had more improvement with the symbolic-size model, and in the other benchmarks the results were similar.

Table 6.10: Comparison of *PAT* vs. *CFG* without the symbolic-size model.

	Speedup (×)			**Coverage (%)**		
	#	Avg.	Med.	#	Avg.	Med.
libosip	28/35	10.08	8.13	20/35	16.82	2.32
wget	14/31	2.01	1.43	21/31	25.78	0.00
libtasn1	7/13	0.96	0.96	6/13	0.00	0.00
libpng	3/12	0.96	0.98	9/12	-0.55	0.00
apr	10/20	1.97	1.23	11/20	2.67	0.00
json-c	4/5	0.95	1.50	1/5	0.00	0.00
busybox	10/30	1.29	1.00	23/30	-1.33	0.00

> Our approach is not restricted to the symbolic-size model, and achieves comparable improvements with the concrete-size model.

6.6.6.2 Default Search Heuristic

Recall that we performed our experiments with the DFS search heuristic. To show that our approach does not require a specific search heuristic in order to be beneficial, we performed an additional experiment using KLEE's default search heuristic.[10] The results are shown in Table 6.11 and Table 6.12.

In general, when the analysis achieves full exploration with one search heuristic, the analysis time with other search heuristics is usually similar. Indeed, as can be seen from the results, the analysis times here are very similar to those obtained with the DFS search heuristic (Table 6.3 and Table 6.5).

In terms of coverage, the results here are comparable to those obtained with the DFS search heuristic. In some cases (for example, *busybox* and *libpng*), we had more improvement, and in other cases (for example, *libosip* and *wget*), we had less improvement.

> Our approach is not restricted to a specific search heuristic, and achieves comparable improvements using KLEE's default search heuristic.

[10] When state merging is enabled, the default search heuristic is set using the command-line option: `-search=nurs:covnew`.

Table 6.11: Comparison of *PAT* vs. *CFG* with the *nurs:covnew* search heuristic.

	Speedup (×)			**Coverage (%)**		
	#	Avg.	Med.	#	Avg.	Med.
libosip	17/35	6.69	4.00	28/35	15.12	2.11
wget	10/31	2.85	1.91	25/31	11.66	0.00
libtasn1	7/13	0.94	0.96	6/13	-0.42	0.00
libpng	1/12	0.74	0.74	11/12	9.39	13.04
apr	10/20	3.60	1.72	11/20	1.55	0.00
json-c	4/5	3.10	2.84	1/5	0.82	0.82
busybox	8/30	1.69	1.14	23/30	3.03	0.00

Table 6.12: Comparison of *PAT* vs. *Base* with the *nurs:covnew* search heuristic.

	Speedup (×)			**Coverage (%)**		
	#	Avg.	Med.	#	Avg.	Med.
libosip	18/35	9.58	3.02	27/35	8.52	0.00
wget	10/31	2.87	5.29	26/31	-5.30	0.00
libtasn1	7/13	4.80	9.29	7/13	0.65	0.00
libpng	1/12	4.80	4.80	11/12	2.11	3.50
apr	10/20	8.12	4.51	15/20	-0.14	0.00
json-c	4/5	1.36	3.59	2/5	0.41	0.41
busybox	10/30	2.11	2.01	23/30	5.57	0.00

6.6.7 Found Bugs

We found two bugs during our experiments with *busybox*. In both cases, a null-pointer dereference occurred in the implementation of `realpath` in *klee-uclibc*, KLEE's modified version of *uClibc* [117]. We reported the bugs and they were confirmed and fixed by the official maintainers.[11] We note that these bugs were detected by *PAT* and *Base*, but were not found by *CFG* due to a timeout.

6.6.8 Threats to Validity

First, our implementation may have bugs. To validate its correctness, we performed a separate experiment where each subject is run in the *PAT* mode with a timeout of one hour. During these runs, we validated that every executed state merging operation is correct w.r.t. Theorem 6.2.7. In addition, for every query that our solving procedure was able to solve, we validated the consistency of the reported result w.r.t. the underlying SMT solver.

Second, our choice of benchmarks might not be representative enough. That said,

[11] https://github.com/klee/klee-uclibc/pull/47

we chose a diverse set of real-world benchmarks that were used in prior work [75, 99]. In addition, we used benchmarks that process inputs of both binary and textual formats.

Third, we evaluated our approach in the context of the symbolic-size model. To address the threat that our approach may be beneficial only in the context of that particular memory model, we performed an additional experiment using the standard concrete-size model. The results, shown in Section 6.6.6.1, lead to conclusions similar to the ones drawn from the original experiments.

Fourth, the search heuristic might affect the coverage when the exploration does not terminate. To address the threat that our results may be valid only for the DFS search heuristic, we performed an additional experiment with the default search heuristic in KLEE. The results, shown in Section 6.6.6.2, are comparable.

6.6.9 Discussion

Taking a high-level view of the experiments, we observe that our approach brings significant gains w.r.t. both baselines in most of the benchmarks (*libosip*, *wget*, *apr*, *json-c*, and *busybox*). This is because these benchmarks contain an abundant number of size-dependent loops that generate expressions that are linearly dependent on the number of repetitive parts in the path constraints, which leads to the detection of many regular (and formula) patterns. In *libtasn1* and *libpng*, however, most of the size-dependent loops generate expressions that cannot be expressed using the linear terms that our synthesis can produce, for example, loops in which conditions depend on aggregate values such as the sum of array contents. As a result, relatively few formula patterns are detected. Nevertheless, our approach still preserves in these cases the benefits of standard state merging w.r.t. standard symbolic execution.

Chapter 7

Related Work

In this section, we discuss past work that relates to the ideas proposed in this thesis.

7.1 Memory Models

In Section 2.2, we described the standard memory model. However, other memory models for symbolic execution have been proposed in the past, and here, we briefly describe some of those memory models that mostly relate to our context.

7.1.1 MemSight

Coppa et al. [57] propose the memory model *MemSight*, which models the symbolic memory as a set of tuples, where each tuple associates an address expression to a value expression, along with a timestamp, and a condition. When a write is performed, the memory is updated with a new tuple containing the corresponding address and value expressions. When a read is performed, the memory is scanned to determine the tuples that match the given address expression. The read value is then expressed using an *ite* expression, which is built using the matching tuples. This approach attempts to improve the merging memory model used in ANGR [103], by avoiding concretizations of symbolic pointers when they are encountered in reads or writes.

In our memory model presented in Chapter 3, memory access operations, i.e., read and write, are handled by using array theory. In addition, *MemSight* does not address the challenge of handling address-dependent queries (Chapter 4), and does not support symbolic-size memory objects (Chapter 5).

7.1.2 Segmented Memory Model

Segmented memory model (SMM) [73] is a technique for handling symbolic pointers that have multiple resolutions, which was discussed in more detail in Section 3.2.3.1. At a high level, it partitions the memory into segments using static pointer analysis, such that each pointer refers to a single segment, thus avoiding forks when symbolic pointers are dereferenced.

Similarly to SMM, our memory model presented in Chapter 3 is focused on mitigating the path explosion caused by symbolic pointers. In contrast to SMM, which performs the memory partitioning ahead of time, our approach performs the memory partitioning on the fly. This allows us to obtain smaller segments, which result in smaller SMT arrays and less complex array-theory constraints.

In addition, SMM does not address the problem of caching address-dependent queries (Chapter 4). Memory objects can still be allocated in different segments or allocated in different offsets within the same segment, which will result in the same problem that happens when base addresses differ. Moreover, the challenge of caching address-dependent queries exists not only when symbolic pointers have multiple resolutions, but also when they are resolved to a single memory object (or segment).

Finally, SMM does not support symbolic-size memory objects. However, our symbolic-size model (Chapter 5) can be easily integrated with the segmented memory model since every allocated memory object has a finite capacity. Technically, the only modification needed is calling the function *HandleAlloc* (See Algorithm 3, line 29 from [73]) with the capacity of the allocated memory object instead of its size.

7.1.3 Segment-Offset-Plane

In the *segment-offset-plane* memory model [116], a memory object has its own unique address space, i.e., a segment. This memory model uses *fat* pointers, i.e., a pointer is represented as a pair consisting of a segment identifier and an offset (within that segment). This model supports unbounded symbolic-size allocations using a two-dimensional address space where the non-overlapping property (Section 2.2) is naturally supported. This model supports *multi-writes*, i.e., operations that write to symbolic number of memory locations at once, but in general, it explicitly represents the value of each byte in a memory object.

Because of the explicit representation used in their model, their analysis will not

scale with large enough memory objects due to the expected high memory consumption. For example, if an `strchr`-like loop is executed with a string of unbounded length, then every iteration generates a constraint on a different byte in the string. In our bounded symbolic-size model (Chapter 5), we set a bound on symbolic-size expressions, which gives us control over the memory consumption. Their model, in contrast to our state merging approach presented in Chapter 5, does not address the problem of additional forking introduced by symbolic-size expressions, and in particular, that of symbolic-size dependent loops. Their model associates every segment with a single memory object, so it does not support static ([73]) or dynamic (Chapter 3) memory partitioning. In addition, to support *fat* pointers, this model encodes expressions using a more complex language, which may incur additional overhead. Šimáček [104] implements the memory model proposed by [116] on top of KLEE, thus inheriting the limitations discussed above.

7.1.4 CUTE

CUTE [100] is a concolic execution [65] approach which represents pointers using symbolic values. The pointer constraints are maintained together with the path constraints, so when CUTE generates a new test input, it obtains a new memory graph. CUTE represents the memory graphs using *logical* addresses, which abstract the physical addresses. By using logical addresses, CUTE is more likely to generate similar memory graphs, which eventually result in similar path constraints that can be solved incrementally. In contrast to our memory model presented in Chapter 3, where pointer constraints may involve arbitrary expressions over bit-vectors and arrays, the pointer constraints in CUTE can contain only equality (or inequality) between pointers.

7.1.5 UC-KLEE

UC-KLEE [93] supports symbolic-size arrays in its lazy initializing algorithm as well as allocations of symbolic-size memory objects [92]. Essentially, it uses an approach similar to the model described in Chapter 5, where every symbolic-size memory object has a user-specified upper bound on its size.

Their work, in contrast to our state merging approach presented in Chapter 5, does not address the problem of additional forking introduced by symbolic-size dependent loops. In addition, we investigated the tradeoffs between different approaches for

modeling symbolic-size allocations (Section 5.4.1).

7.1.6 Memory Abstraction Techniques

Anand et al. [21] model symbolic-size arrays as part of the lazy initialization algorithm. Here, arrays are modeled as linked lists with symbolic length, where each node has a symbolic index and a symbolic value. An abstraction-based subsumption is used for state pruning and for bounding the number of initialized array cells, thus potentially leading to missed feasible behaviors. Deng et al. [50] model symbolic-size arrays similarly to [21], but place a bound on the number of initialized array cells instead of using an abstraction-based pruning.

Similarly to previously mentioned approaches [93, 116], these works [21, 50] do not address the problem of additional forking introduced by symbolic-size expressions.

7.1.7 Memory Partitioning

The idea of using memory partitioning for improving program analysis has been explored before. In the context of bounded model checking, partitioned models have been used based on various complementary analyses such as points-to analysis [22, 110, 111], data structure analysis (DSA) [79], and type based analysis [36, 43]. CBMC [41] and ESBMC [44] also use points-to analysis to refine their memory models. SeaHorn [67] uses a context-insensitive variant of DSA [79].

The memory partitioning used in prior work is computed ahead of time, while our memory model presented in Chapter 3 does not require additional pre-computations, and enables a *path-specific* memory partitioning during runtime, thus resulting in a more accurate partitioning.

7.1.8 Other Memory Models

The idea of modeling addresses *not* as constant values was proposed in the past. Hajdu et al. [68] model address values in smart contracts as uninterpreted symbols as in this context addresses can be only queried for equivalence. Our memory model presented in Chapter 3 allows for arbitrary queries over symbolic addresses.

7.2 Constraint Solving

The idea of scaling constraint solving by reusing previously solved results has been investigated in the past: KLEE [39] uses counter-example caching, which stores results into a cache that maps constraint sets to concrete variable assignments. Using these mappings, KLEE can solve several types of similar queries, involving subsets and supersets of the constraint sets already cached. Green [119] is a framework that enables reusing constraints results within a single run as well as across different runs and programs. To enable efficient reuse, the technique uses slicing to reduce the complexity of the constraints, and canonization to store the constraints in a normal form. GreenTrie [71] is an extension of Green that detects implications between constraints to improve caching for satisfiability queries. Cashew [33] is a caching framework for model-counting queries built on top of Green [119], which introduces an aggressive normalization scheme and parameterized caching. Eiers et al. [55] use subformula caching to improve the performance of model counting constraint solvers in the context of quantitive program analysis.

The approaches mentioned above do not address the challenge of solving address-dependent queries (Chapter 4), as they fail to detect equisatisfiable address-dependent queries when they are syntactically distinct, due to address constants.

Other approaches have been proposed to scale constraint solving in the context of symbolic execution: arithmetic transformations [38, 100], splitting constraints into independent sets [38, 39], multiple solvers support [87], interval-based solving [52], and fuzzing-based solving [80, 88]. Perry et al. [91] focus on reducing the cost of array-theory constraints using several semantics-preserving transformations. These transformations attempt to eliminate array constraints as much as possible by replacing them with constraints over their indices and values. Modern SMT solvers such as CVC4 [28], Z3 [47], and Yices [54], have support for *incremental solving*, which enables learning lemmas that can be later reused for solving similar queries.

The approaches mentioned in the previous paragraph are orthogonal to our query caching approach presented in Chapter 4, with which they could be combined.

There are many works on solving quantified queries [26, 31, 45, 51, 61, 94–96]. Our specialized solving procedure (Section 6.4) is mainly designed to solve satisfiable queries, and resorts to the standard solving procedure when it fails. It adapts ideas from

E-matching [45] and model-based quantifier instantiation [61] to our specific needs.

7.3 State Merging

State merging [69, 77, 101] has been used in the past to scale symbolic execution. Kuznetsov et al. [77] propose dynamic state merging with a *query count estimation* heuristic that decides when merging should be applied. MultiSE [101] proposes an alternative approach for state representation. where updates to values are summarized by guarding each value with a path predicate. Veritesting [24, 102] is another state merging technique which statically summarizes code regions. JavaRanger [102] extends veritesting for Java programs to support dynamically dispatched methods, by using the runtime information available during the analysis.

These works have no support for symbolic-size memory objects, and statically summarizing code regions that contain loops is challenging, even with the aid of runtime information. Moreover, in contrast to our state merging approach presented in Chapter 6, the works mentioned above do not address the encoding explosion problem caused by using disjunctions and *ite* expressions.

7.4 Loop Summaries

Loop-extended symbolic execution [98] summarizes input-dependent loops. It uses static analysis to infer linear relations between variables and trip count variables which track the number of iterations in the loop.

In contrast, our approaches presented in Chapters 5 and 6 are more dynamic in nature. The approach in Chapter 5 does not depend on static analysis, and the approach in Chapter 6 only partially depends on liveness analysis (Section 6.3).

Godefroid et al. [64] propose a dynamic approach that can infer partial invariants in input-dependent loops. This approach can be applied only in loops where all the variables depend on induction variables, and only when the loop iteration is executed at least three times.

In contrast, our approaches presented in Chapters 5 and 6 have no restrictions on the loop variables or the number of iterations. In addition, both [98] and [64] provide summaries only for scalar variables, so clearly does not support symbolic-size memory objects, as opposed to our memory model presented in Chapter 5.

Kapus et al. [75] summarize string loops by synthesizing calls to standard string functions. S-Looper [120] introduces string constraints that can be solved by SMT solvers that support the string theory. The approaches presented in Chapters 5 and 6 are not restricted to string loops and do not require an SMT solver with support for string theory.

Sinha [105] simplifies *ite* expressions using rewrite rules, which are not expressive enough to achieve the effect of the optimizations discussed in Section 5.2.3. The same work also proposes a technique for generalizing *ite* expressions generated during the analysis of loops, which generates parametric expressions based on pattern matching. This could be used in our approaches (Chapters 5 and 6) to simplify merged values.

7.5 Encoding with Quantifiers

Compact symbolic execution [106] uses quantifiers to encode the path conditions of cyclic paths that follow the *same* control flow path in each iteration and update all the variables in a regular manner. This allows them to encode the effect of an unbounded repetitions of *some* of the cyclic paths in the program.

In contrast, our approach presented in Chapter 6 seeks regularity at the level of the constraints and therefore does not rely on uniformity in the control flow graph. For example, in `memspn` (Figure 6.1) they can only summarize the paths in which either all the characters of `s` are matched with the first character of `chars` (the *then* branch) or the first character of `s` is unmatched (the *else* branch). In contrast, our approach can summarize *all* paths up to a given bound using two merged symbolic states. Furthermore, [106] solves quantified queries using a standard SMT solver as opposed to our specialized solving procedure (Section 6.4). We attempted to compare their implementation to ours, but, as was confirmed by the authors, their tool is now inoperable.

7.6 Static Analysis

Our incremental state merging approach (Section 6.3) uses liveness analysis to detect symbolic states that have the same symbolic store w.r.t. live variables. Boonstoppel et al. [30] use liveness analysis for a different purpose, pruning the state space. If two symbolic states differ only in program values that are not subsequently read, then they

treat them as identical and prune one of them.

Chapter 8

Conclusions

In Chapter 3, we proposed a novel addressing model where the base addresses are symbolic values rather than concrete ones. First, this model provides the ability to reshape the underlying layout of the address space, which we exploit in two applications: *inter-object* partitioning, which helps to improve upon the existing segmented memory model by dynamically relocating memory objects, and *intra-object* partitioning, which helps to reduce the cost of constraint solving by dynamically splitting large memory objects into smaller ones. Second, this model provides the ability to distinguish address expressions from non-address expressions, which helps to perform efficient query caching of queries that depend on address expressions (Chapter 4). In Chapter 5, we proposed a novel memory model where the size of a memory object can be symbolic and not only concrete. In this model, a memory object can have a size that ranges over a set of values and is bounded by a user-specified capacity. To reduce the additional forking imposed by this model, we proposed a state merging approach that is applied in loops that are involved with symbolic-size memory objects. To further scale the state merging approach mentioned above, whose effectiveness is limited in the presence of complex *ite*'s and disjunctions, we proposed a novel state merging approach in Chapter 6. In this approach, we reduce the encoding complexity of merged symbolic states by using quantifiers instead of the standardly used *ite*'s and disjunctions, and propose a specialized solving procedure to efficiently handle the resulting quantified constraints.

Chapter 9

Future Work

In our relocatable addressing model (Chapter 3), the non-overlapping property of the address space is preserved in a rather restrictive way, by using a set of address constraints that bind each symbolic base address to a concrete value. Unfortunately, this can make the analysis incomplete, as can be illustrated by the following example:

```
1 char *p = malloc(10);
2 if (p == 0x80000000) {
3     // ...
4 }
```

Suppose that the allocated memory object is (β, s, a) and its address constraint is:

$$\beta = 0x70000000$$

Recall that the address constraints are substituted before a query is passed to the SMT solver, so the query that corresponds to the branch condition of the *if* statement will be:

$$0x70000000 = 0x80000000$$

which is clearly unsatisfiable. As a result, the block of the *if* statement will be uncovered. To overcome this, one needs to preserve the non-overlapping property in a more general way, for example, by adding constraints that ensure the disjointness of all address intervals without using specific concrete values. However, efficiently solving such constraints might be challenging.

In our dynamic intra-object partitioning (Section 3.2.3), we used a rather simple strategy that splits memory objects into smaller memory objects of equal size. Exploring

more sophisticated strategies, or applying different strategies for different memory objects, might further improve the performance. In our experiments (Section 3.4), we evaluated inter-object partitioning (Section 3.2.3) and intra-object partitioning (Section 3.2.4) separately, but one can apply those approaches simultaneously. In addition, it was observed that applying these approaches is not always beneficial, so one can try to predict when an application of these approaches is likely to payoff.

Our query caching approach (Chapter 4) cannot be applied to queries which are not address-agnostic. Therefore, coming up with an approach that can efficiently handle such queries can further improve the performance, especially in programs where symbolic pointer resolution is expensive. In addition, our query caching approach supports *satisfiability* and *validity* queries, but does not support *model* queries, i.e., queries that provide satisfying models which are used to generate test cases. Another research direction is applying a similar approach for such queries, including address-dependent ones.

Our symbolic-size model (Chapter 5) may lead to an incomplete analysis, since every memory object has a user-specified capacity. Modeling unbounded memory objects in a way that will allow scalable analysis remains a major challenge.

In our experiments (Section 5.4), the capacity settings were chosen manually. Given a program to be analyzed, one can try to use static analysis to find a capacity setting that will allow achieving high coverage while making the analysis as simple as possible. In addition, the capacity setting in our experiments is global, i.e., it applies for all the symbolic-size allocations in the program. One can try to set the capacity in a more refined manner, for example, by tuning it according to the allocation context.

Our state merging approach (Chapter 6) uses quantified constraints to encode merged symbolic states. We believe that this opens up opportunities for experimenting with other applications of symbolic execution where quantified constraints might be naturally used.

Our state merging approach (Chapter 6) automatically detects regular patterns to partition similar symbolic states into merging groups. For each group, we synthesize a formula pattern which enables an efficient encoding of the merged symbolic state using quantifiers. Extracting more complex patterns, e.g., beyond linear formulas, can further improve the applicability of our approach.

Our solving procedure (Section 6.4) is mainly designed to solve satisfiable quantified

queries. Reducing the solving time of unsatisfiable quantified queries can further improve our approach.

The memory models proposed in Chapters 3 and 5 were presented and evaluated separately. However, we believe that these memory models can be unified into a single memory model.

Bibliography

[1] libyaml. https://github.com/yaml/libyaml. 72

[2] GNU Bash. https://www.gnu.org/software/bash. 71

[3] *Communications of the Association for Computing Machinery (CACM)*. 156, 159

[4] expat. https://libexpat.github.io. 71

[5] json-c. https://github.com/json-c/json-c. 71

[6] libxml2. http://www.xmlsoft.org. 46, 71

[7] GNU oSIP. https://www.gnu.org/software/osip. 72

[8] July 2009. 157, 162

[9] July 2015. 159, 165

[10] GNU Coreutils. https://www.gnu.org/software/coreutils/, 2021. 102

[11] Gcov. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, 2021. 95, 126

[12] Gnu osip. https://www.gnu.org/software/osip/, 2021. 78, 94, 126

[13] libpng. http://www.libpng.org/pub/png/libpng.html, 2021. 94, 126

[14] Gnu libtasn1. https://www.gnu.org/software/libtasn1/, 2021. 94, 126

[15] https://git.savannah.gnu.org/cgit/osip.git/commit/?id=ef6497, 2021. 101

[16] https://git.savannah.gnu.org/cgit/osip.git/commit/?id=2f0380, 2021. 101

[17] busybox. https://busybox.net/, 2023. 126

[18] json-c. https://github.com/json-c/json-c/, 2023. 126

[19] Gnu wget. https://www.gnu.org/software/wget/, 2023. 126

[20] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, 2006. ISBN 0321486811. 116, 117

[21] Saswat Anand, Corina S Păsăreanu, and Willem Visser. Symbolic execution with abstract subsumption checking. In *International SPIN Workshop on Model Checking of Software*, pages 163–181. Springer, 2006. doi: 10.1007/11691617_10. 143

[22] Lars Ole Andersen. Program analysis and specialization for the C programming language. Technical report, 1994. 143

[23] APR. Apache Portable Runtime. https://apr.apache.org/, 2019. 46, 71, 126

[24] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In *Proc. of the 36th International Conference on Software Engineering (ICSE'14)*, May 2014. doi: 10.1145/2568225.2568293. 145

[25] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), may 2018. ISSN 0360-0300. doi: 10.1145/3182657. URL https://doi.org/10.1145/3182657. 1

[26] Kshitij Bansal, Andrew Reynolds, Tim King, Clark Barrett, and Thomas Wies. Deciding local theory extensions via e-matching. In *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II 27*, pages 87–105. Springer, 2015. 144

[27] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and*

Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. Springer, 2022. doi: 10.1007/978-3-030-99524-9_24. 117

[28] Clark Barrett, Christopher Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proc. of the 23rd International Conference on Computer-Aided Verification (CAV'11)*, July 2011. 144

[29] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In *Proc. of the 35th Design Automation Conference (DAC'98)*, June 1998. doi: 10.1145/277044.277186. 80

[30] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. Rwset: Attacking path explosion in constraint-based test generation. In *IN TACAS'08: INTERNATIONAL CONFERENCE ON TOOLS AND ALGORITHMS FOR THE CONSTRUCTIONS AND ANALYSIS OF SYSTEMS*, 2008. 146

[31] Aaron R Bradley, Zohar Manna, and Henny B Sipma. What's decidable about arrays? In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 427–442. Springer, 2006. doi: 10.1007/11609773_28. 80, 144

[32] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation*, pages 427–442, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. 64

[33] Tegan Brennan, Nestan Tsiskaridze, Nicolás Rosner, Abdulbaki Aydin, and Tevfik Bultan. Constraint normalization and parameterized caching for quantitative program analysis. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 535–546, 2017. 58, 144

[34] Tegan Brennan, Seemanta Saha, Tevfik Bultan, and Corina S Păsăreanu. Symbolic path cost analysis for side-channel detection. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 27–37, 2018. doi: 10.1145/3213846.3213867. 1

[35] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. Casym: Cache aware symbolic execution for side channel detection and mitigation. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 505–521. IEEE, 2019. doi: 10.1109/SP.2019.00022. 1

[36] Rodney M Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine intelligence*, 7(23-50):3, 1972. 143

[37] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, February 2013. ISSN 0001-0782. doi: 10.1145/2408776.2408795. URL http://doi.acm.org/10.1145/2408776.2408795. 1, 104

[38] 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. doi: 10.1145/1455518.1455522. 1, 12, 32, 39, 41, 58, 144

[39] 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. 1, 4, 5, 12, 15, 18, 26, 30, 31, 32, 35, 41, 45, 56, 58, 60, 70, 78, 93, 106, 124, 125, 144

[40] Vitaly Chipounov and George Candea. Reverse engineering of binary device drivers with revnic. In *Proc. of the 5th European Conference on Computer Systems (EuroSys'10)*, April 2010. 1

[41] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Proc. of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, March-April 2004. 143

[42] 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. doi: 10.1145/1966445.1966475. 1

[43] Jeremy Condit, Brian Hackett, Shuvendu K Lahiri, and Shaz Qadeer. Unifying type checking and property checking for low-level code. In *ACM SIGPLAN Notices*, volume 44, pages 302–314. ACM, 2009. 143

[44] 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. ISSN 0098-5589. 143

[45] Leonardo de Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In Frank Pfenning, editor, *Automated Deduction – CADE-21*, pages 183–198, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-73595-3. 144, 145

[46] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3. 117

[47] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. 144

[48] 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. 124

[49] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. In *Communications of the Association for Computing Machinery (CACM)* cac [3], pages 69–77. 1

[50] Xianghua Deng, Jooyong Lee, et al. Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 157–166. IEEE, 2006. doi: 10.1109/ASE.2006.26. 143

[51] David Detlefs, Greg Nelson, and James B Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM (JACM)*, 52(3):365–473, 2005. 144

[52] Oscar Soria Dustmann, Klaus Wehrle, and Cristian Cadar. Parti: a multi-interval theory solver for symbolic execution. 2018. 144

[53] Bruno Dutertre. Yices 2.2. In *Proc. of the 26th International Conference on Computer-Aided Verification (CAV'14)*, July 2014. 117

[54] Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2(2):1–2, 2006. 144

[55] William Eiers, Seemanta Saha, Tegan Brennan, and Tevfik Bultan. Subformula caching for model counting and quantitative program analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 453–464. IEEE, 2019. 144

[56] 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)* iss [8]. 15, 56

[57] 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. 140

[58] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 519–531, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-73367-6. URL http://dl.acm.org/citation.cfm?id=1770351.1770421. 64, 70

[59] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 519–531, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-73367-6. URL http://dl.acm.org/citation.cfm?id=1770351.1770421. 5, 25, 26, 32, 34, 80, 93, 125

[60] GAS. GNU Assembler. https://sourceware.org/binutils/docs/as/, 2019. 46

[61] Yeting Ge and Leonardo Mendonça de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 306–320. Springer, 2009. doi: 10.1007/

978-3-642-02658-4_25. URL https://doi.org/10.1007/978-3-642-02658-4_25. 144, 145

[62] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 1069–1084, 2019. 63

[63] GNU. GNU Coreutils. https://www.gnu.org/software/coreutils/, 2021. URL https://www.gnu.org/software/coreutils/. 46, 72

[64] 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. doi: 10.1145/2001420.2001424. 145

[65] 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. 142

[66] 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. 1, 32, 70

[67] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. The seahorn verification framework. In *International Conference on Computer Aided Verification*, pages 343–361. Springer, 2015. 143

[68] Ákos Hajdu and Dejan Jovanović. solc-verify: A modular verifier for solidity smart contracts. *arXiv preprint arXiv:1907.04262*, 2019. 143

[69] Trevor Hansen, Peter Schachte, and Harald Søndergaard. State joining and splitting for the symbolic execution of binaries. In *Proc. of the 2009 Runtime Verification (RV'09)*, June 2009. doi: 10.1007/978-3-642-04694-0_6. 19, 21, 84, 104, 145

[70] 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. 32, 63

[71] Xiangyang Jia, Carlo Ghezzi, and Shi Ying. Enhancing reuse of constraint solutions to improve symbolic execution. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'15)* iss [9]. 56, 144

[72] 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. doi: 10.1109/ICSE.2012.6227168. 1

[73] Timotej Kapus and Cristian Cadar. A segmented memory model for symbolic execution. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 774–784. ACM, 2019. ISBN 978-1-4503-5572-8. doi: 10.1145/3338906.3338936. 13, 32, 39, 40, 47, 52, 72, 141, 142

[74] 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 *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 874–888, 2019. doi: 10.1145/3314221.3314610. 1

[75] 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. doi: 10.1145/3314221.3314610. 139, 146

[76] James C. King. Symbolic execution and program testing. In *Communications of the Association for Computing Machinery (CACM)* cac [3], pages 385–394. 1, 27, 105

[77] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'12)*, June 2012. doi: 10.1145/2345156.2254088. 19, 21, 84, 104, 145

[78] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of the 2nd International Symposium on Code Generation and Optimization (CGO'04)*, March 2004. doi: 10.1109/CGO.2004.1281665. 93, 124

[79] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'07)*, June 2007. 143

[80] Daniel Liew, Cristian Cadar, Alastair F Donaldson, and J Ryan Stinnett. Just fuzz it: solving floating-point constraints using coverage-guided fuzzing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 521–532, 2019. 144

[81] M4. GNU M4. https://www.gnu.org/software/m4/, 2019. 46, 71

[82] Make. GNU Make. https://www.gnu.org/software/make/, 2019. 46, 71

[83] 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. 1

[84] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*, pages 691–701. ACM, 2016. doi: 10.1145/2884781.2884807. 1

[85] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE, 2019. doi: 10.1109/ASE.2019.00133. 4, 15, 18, 26, 56, 78

[86] 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. doi: 10.1109/ICSE.2013.6606623. 1

[87] 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. 56, 144

[88] Awanish Pandey, Phani Raj Goutham Kotcharlakota, and Subhajit Roy. Deferred concretization in symbolic execution via fuzzing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 228–238, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362245. doi: 10.1145/3293882.3330554. URL https://doi.org/10.1145/3293882.3330554. 144

[89] 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. 1, 12, 32

[90] Corina S Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. Multi-run side-channel analysis using symbolic execution and max-smt. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 387–400. IEEE, 2016. 1

[91] David M. Perry, Andrea Mattavelli, Xiangyu Zhang, and Cristian Cadar. Accelerating array constraints in symbolic execution. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'17)*, July 2017. 56, 144

[92] David A. Ramos. *Under-constrained symbolic execution : correctness checking for real code*. Ph.d. dissertation, Stanford University, 2015. 142

[93] 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. 1, 142, 143

[94] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstić, Morgan Deters, and Clark Barrett. Quantifier instantiation techniques for finite model finding in smt. In *Automated Deduction–CADE-24: 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings 24*, pages 377–391. Springer, 2013. 144

[95] Andrew Reynolds, Cesare Tinelli, and Leonardo De Moura. Finding conflicting instances of quantified formulas in smt. In *2014 Formal Methods in Computer-Aided Design (FMCAD)*, pages 195–202. IEEE, 2014.

[96] Andrew Reynolds, Haniel Barbosa, and Pascal Fontaine. Revisiting enumerative instantiation. In *Tools and Algorithms for the Construction and Analysis of Systems: 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II 24*, pages 112–131. Springer, 2018. 144

[97] Barbara G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *In Proc. of the 12th International Conference on Compiler Construction (CC'03)*, 2003. 32

[98] 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)* iss [8]. doi: 10.1145/1572272.1572299. 145

[99] Daniel Schemmel, Julian Büning, Frank Busse, Martin Nowack, and Cristian Cadar. A deterministic memory allocator for dynamic symbolic execution. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, pages 9:1–9:26, 6 2022. 139

[100] 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. 142, 144

[101] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. Multise: Multi-path symbolic execution using value summaries. In *10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)*. ACM, 2015. doi: 10.1145/2786805.2786830. ACM SIGSOFT Distinguished Paper Award. 145

[102] Vaibhav Sharma, Soha Hussein, Michael W. Whalen, Stephen McCamant, and Willem Visser. Java ranger: Statically summarizing regions for efficient symbolic

execution of java. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 123–134, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370431. doi: 10.1145/3368089.3409734. 145

[103] 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. doi: 10.1109/SP.2016.17. 4, 15, 18, 31, 32, 56, 58, 60, 70, 78, 140

[104] M Šimáček. Symbolic-size memory allocation support for klee. master's thesis, masaryk university, faculty of informatics, brno, 2018. 2018. 142

[105] Nishant Sinha. Symbolic program analysis using term rewriting and generalization. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, page 19. IEEE Press, 2008. doi: 10.1109/FMCAD.2008.ECP.23. 146

[106] Jiri Slaby, Jan Strejcek, and Marek Trtík. Compact symbolic execution. In Dang Van Hung and Mizuhito Ogawa, editors, *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, volume 8172 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 2013. doi: 10.1007/978-3-319-02444-8_15. URL https://doi.org/10.1007/978-3-319-02444-8_15. 146

[107] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Found. Trends Program. Lang.*, 2(1):1–69, April 2015. ISSN 2325-1107. 32

[108] SQLite. SQLite Database Engine. https://www.sqlite.org/, 2019. 46, 71

[109] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. *Alias Analysis for Object-Oriented Programs*, pages 196–232. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. 32

[110] B. Steensgaard. Points-to analysis in almost-linear time. In *Principles of Programming Languages (POPL)*, 1996. 143

[111] Bjarne Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *International Conference on Compiler Construction*, pages 136–150. Springer, 1996. 143

[112] David Trabish and Noam Rinetzky. Relocatable addressing model for symbolic execution. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 51–62, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380089. doi: 10.1145/3395363.3397363. URL https://doi.org/10.1145/3395363.3397363. 14, 31

[113] David Trabish, Shachar Itzhaky, and Noam Rinetzky. A bounded symbolic-size model for symbolic execution. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 1190–1201, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385626. URL https://doi.org/10.1145/3468264.3468596. 20, 78

[114] David Trabish, Shachar Itzhaky, and Noam Rinetzky. Address-aware query caching for symbolic execution. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 116–126. IEEE, 2021. 17, 56

[115] David Trabish, Noam Rinetzky, Sharon Shoham, and Vaibhav Sharma. State merging with quantifiers in symbolic execution. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA*, New York, NY, USA, 2023. Association for Computing Machinery. URL https://doi.org/10.1145/3611643.3616287. 24

[116] Marek Trtík and Jan Strejček. Symbolic memory with pointers. In *Automated Technology for Verification and Analysis (ATVA)*, November 2014. doi: 10.1007/978-3-319-11936-6_27. 141, 142, 143

[117] uClibc. uClibc. https://www.uclibc.org/, 2022. 104, 138

[118] Arnaud Venet and Guillaume Brat. Precise and efficient static array bound checking for large embedded c programs. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, page 231–242, 2004. 63

[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. 15, 56, 58, 144

[120] 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)* iss [9]. 146

Appendix A

Proofs

A.1 Address-Aware Query Caching

A.1.1 Proof of Lemma 4.4.8

Lemma A.1.1. *If φ_1 and φ_2 are formulas in $L \supseteq L_1$ which are address-agnostic, then $\neg\varphi_1, \varphi_1 \wedge \varphi_2, \varphi_1 \vee \varphi_2$ are address-agnostic as well.*

Proof. Suppose that φ_1 and φ_2 are address-agnostic. Let S_1 and S_2 be address spaces, let m_1 and m_2 be models, and suppose that $m_1 =_{Int} m_2$, m_1 respects S_1, m_2 respects S_2, and m_1 and m_2 are consistent w.r.t. S_1 and S_2.

Negation: If $\neg\varphi_1$ respects S_1 in m_1, then φ_1 respects S_1 in m_1. We know that φ_1 is address-agnostic, so $m_1 \models \varphi_1 \iff m_2 \models \varphi_1$. Therefore, $m_1 \models \neg\varphi_1 \iff m_2 \models \neg\varphi_1$.

Conjunction: If $\varphi_1 \wedge \varphi_2$ respects S_1 in m_1, then φ_1 respects S_1 in m_1 and φ_2 respects S_1 in m_1. We know that φ_1 and φ_2 are address-agnostic, so $m_1 \models \varphi_1 \iff m_2 \models \varphi_1$ and $m_1 \models \varphi_2 \iff m_2 \models \varphi_2$. Therefore, $m_1 \models \varphi_1 \wedge \varphi_2 \iff m_2 \models \varphi_1 \wedge \varphi_2$.

Disjunction: Similar to the proof for conjunction. □

A.1.2 Proof of Lemma 4.4.9

Lemma A.1.2. *Every formula φ in L_1 is address-agnostic.*

Proof. The proof will be done by indcution on the structure of φ.

Base case 1: The case where φ is a comparison $(=, <, \leq)$ between terms of sort *Int* is trivial, since $m_1 =_{Int} m_2$ and both t_1 and t_2 contain only terms of sort *Int*.

Base case 2: Suppose that $\varphi \triangleq t_1 = t_2$ where $t_1, t_2 : Ptr$. For each term t_i, we need to consider two cases: (1) $t_i \triangleq p_i$ where p_i is an uninterpreted constant, and (2) $t_i \triangleq p_i + n_i$ where p_i is an uninterpreted constant and $n_i : Int$.

We will show how to handle the case where $t_1 \triangleq p_1 + n_1$ and $t_2 \triangleq p_2 + n_2$ (other cases are similar). We need to show that:

$$m_1 \models t_1 = t_2 \iff m_2 \models t_1 = t_2$$

If $m_1(p_1) = m_1(p_2)$, then since m_1 respects S_1, we conclude that either:

$$p_1 = p_2$$

or:

$$m_1(p_1) = 0,\ m_1(p_2) = 0$$

If $p_1 = p_2$, then:

$$(p_1 + n_1 = p_2 + n_2) \equiv (n_1 = n_2)$$

The formula $n_1 = n_2$ does not contain pointers, so:

$$m_1 \models n_1 = n_2 \iff m_2 \models n_1 = n_2$$

since $m_1 =_{Int} m_2$.

Otherwise, $m_1(p_1) = 0$ and $m_1(p_2) = 0$. Since m_1 and m_2 are consistent w.r.t. S_1 and S_2, then $m_2(p_1) = 0$ and $m_2(p_2) = 0$. As a result:

$$m_1 \models p_1 + n_1 = p_2 + n_2 \iff m_1 \models n_1 = n_2$$

$$m_2 \models p_1 + n_1 = p_2 + n_2 \iff m_2 \models n_1 = n_2$$

and since $m_1 =_{Int} m_2$ and the formula $n_1 = n_2$ does not contain pointers, we conclude:

$$m_1 \models n_1 = n_2 \iff m_2 \models n_1 = n_2$$

Now, suppose that $m_1(p_1) \neq m_1(p_2)$. Both $p_1 + n_1$ and $p_2 + n_2$ respect S_1 in m_1,

so there exist disjoint intervals $[c_1, e_1] \in S_1$ and $[c_2, e_2] \in S_1$ such that:

$$m_1(p_1 + n_1) \in [c_1, e_1], \quad m_1(p_2 + n_2) \in [c_2, e_2]$$

and therefore:

$$m_1 \not\models p_1 + n_1 = p_2 + n_2$$

As $m_1(p_1) \neq m_1(p_2)$, it must hold that $p_1 \neq p_2$. Then, since m_2 respects S_2, it must hold that $m_2(p_1) \neq m_2(p_2)$. If that was not the case, then:

$$m_2(p_1) = 0, \; m_2(p_2) = 0$$

and since m_1 and m_2 are consistent w.r.t. S_1 and S_2, it would yield that:

$$m_1(p_1) = 0, \; m_1(p_2) = 0$$

which contradicts the assumption that $m_1(p_1) \neq m_1(p_2)$. Since m_2 respects S_2, m_1 and m_2 are consistent w.r.t. S_1 and S_2, and $m_1 =_{Int} m_2$, then $p_1 + n_1$ and $p_2 + n_2$ respect S_2 in m_2. Therefore, there exist disjoint intervals $[c_1, e_1] \in S_2$ and $[c_2, e_2] \in S_2$ such that:

$$m_2(p_1 + n_1) \in [c_1, e_1], \quad m_2(p_2 + n_2) \in [c_2, e_2]$$

and as a result:

$$m_2 \not\models p_1 + n_1 = p_2 + n_2$$

Induction step: These cases follow directly from Lemma 4.4.8. □

A.1.3 Proof of Lemma 4.4.12

Lemma A.1.3. *A guard constraint is address-agnostic.*

Proof. A *guard* constraint $\gamma(p, a, n)$ is given by:

$$p \geq a \wedge p < a + n$$

Under the assumptions of Definition 4.4.7, if $\gamma(p, a, n)$ respects S_1 in m_1, then $m_1(a)$ and $m_1(a + n)$ reside in the same interval in S_1. Since m_1 and m_2 are consistent w.r.t. S_1 and S_2, then $m_1(a)$ and $m_2(a)$ reside in intervals of the same size. Both $m_1(a)$ and

$m_2(a)$ are beginnings of intervals, and $m_1(n) = m_2(n)$, so $m_2(a)$ and $m_2(a+n)$ reside in the same interval in S_2. As $p : Ptr$ is a term in L_1, then either $p \triangleq b$ or $p \triangleq b + t$, where b is a Ptr term (uninterpreted constant) and t is an Int term.

First, suppose that a and b are the same constants. If $p \triangleq b$, then:

$$p \geq a \wedge p < a + n \equiv 0 < n$$

and if $p \triangleq b + t$, then:

$$p \geq a \wedge p < a + n \equiv t \geq 0 \wedge t < n$$

Since $m_1 =_{Int} m_2$, and both t and n are Int terms, then in both cases:

$$m_1 \models p \geq a \wedge p < a + n \iff m_2 \models p \geq a \wedge p < a + n$$

Otherwise, if a and b are the different constants, then $m_1(a)$ and $m_2(b)$ reside in different intervals in S_1. Since p respects S_1 in m_1, then $m_1(p)$ cannot reside in the interval of a in S_1:

$$m_1 \not\models p \geq a \wedge p < a + n$$

Recall that m_2 respects S_2, and m_1 and m_2 are consistent w.r.t. S_1 and S_2, so $m_2(a)$ and $m_2(b)$ reside in different intervals in S_2, and $m_2(p)$ cannot reside in the interval of a in S_2:

$$m_2 \not\models p \geq a \wedge p < a + n$$

□

A.1.4 Proof of Lemma 4.4.13

Lemma A.1.4. *Let ψ be a formula in L_2, $t : Int$ be a term in L_2, and $n : Int$ be a term in L_1. If $t \equiv n$ and $\psi[n/t]$ is address-agnostic, then ψ is address-agnostic as well.*

Proof. Under assumptions of Definition 4.4.7, suppose that ψ respects S_1 in m_1. We need to prove that:

$$m_1 \models \psi \iff m_2 \models \psi$$

Since $t \equiv n$, then $\psi \equiv \psi[n/t]$, so:

$$m_1 \models \psi \iff m_1 \models \psi[n/t]$$

We know that ψ respects S_1 in m_1, $t \equiv n$, and n does not contain Ptr terms, so $\psi[n/t]$ respects S_1 in m_1. We assumed that $\psi[n/t]$ is address-agnostic, so:

$$m_1 \models \psi[n/t] \iff m_2 \models \psi[n/t]$$

Again, since $t \equiv n$, then:

$$m_2 \models \psi[n/t] \iff m_2 \models \psi$$

□

A.1.5 Proof of Lemma 4.4.14

Lemma A.1.5. *Let $\gamma \wedge \psi$ be a formula in L_2, where γ is a guard constraint, i.e., $\gamma(p, a, n)$, and ψ is in L_2. If $p, p' : Ptr$, and there exists $k : Int$ such that:*

- $\gamma \models p - p' = k$
- $\psi[k/(p - p')]$ *is address-agnostic*

then $\gamma \wedge \psi$ is address-agnostic.

Proof. Under the assumptions of Definition 4.4.7, suppose that $\gamma \wedge \psi$ respects S_1 in m_1. We need to show that:

$$m_1 \models \gamma \wedge \psi \iff m_2 \models \gamma \wedge \psi$$

We know that $\gamma \models p - p' = k$, so:

$$m_1 \models \gamma \wedge \psi \iff m_1 \models \gamma \wedge \psi[k/(p - p')]$$

We assumed that $\psi[k/(p - p')]$ is address-agnostic, and we already know from Lemma 4.4.12 that γ is address-agnostic, so using Lemma 4.4.8 we conclude that $\gamma \wedge \psi[k/(p - p')]$ is address-agnostic as well. Since $\gamma \wedge \psi$ respects S_1 in m_1, and $k : Int$, we conclude that $\gamma \wedge \psi[k/(p - p')]$ respects S_1 in m_1. Therefore, by Definition 4.4.7:

$$m_1 \models \gamma \wedge \psi[k/(p - p')] \iff m_2 \models \gamma \wedge \psi[k/(p - p')]$$

Then, since $\gamma \models p - p' = k$, we conclude that:

$$m_2 \models \gamma \wedge \psi[k/(p - p')] \iff m_2 \models \gamma \wedge \psi$$

□

A.1.6 Proof of Theorem 4.4.15

Theorem A.1.6. *Let pc be some path constraints generated by the SE engine. Then pc is address-agnostic.*

Proof. The proof is done by induction on the size of the path constraints.

Base case: This case is trivial since $pc \triangleq true$.

Induction step: We assume that pc is address-agnostic, and we need to prove that $pc \wedge \varphi$ is address-agnostic as well, where φ originates from a branch condition.

If φ in L_1, then $pc \wedge \varphi$ is address-agnostic according to Lemma 4.4.8. Otherwise, φ contains a pointer subtraction term $p - p'$, and since every term is finite, we can assume that both p and p' are in L_1.

If $p - p'$ was generated as a result of a pointer dereference, then it must be the case:

$$p' \triangleq \beta$$

where $\beta : Ptr$ is some uninterpreted constant. Recall that if a pointer p is resolved to a memory object (β, s, a), then the constraint $p \geq \beta \wedge p < \beta + s$ is added to the path constraints.[1] Therefore, pc must contain the *guard* constraint:

$$pc \triangleq \gamma \wedge pc', \quad \gamma \triangleq p \geq \beta \wedge p < \beta + s$$

Since p is a term in L_1, then there exists a term $k : Int$ such that $\gamma \models p - \beta = k$. If $\varphi[k/(p - \beta)]$ in L_1, then we can apply Lemma 4.4.14 to conclude that $\gamma \wedge \varphi$ is address-agnostic, and then $pc \wedge \varphi$ is address-agnostic according to Lemma 4.4.8, since:

$$pc \wedge \varphi \equiv (\gamma \wedge pc') \wedge (\gamma \wedge \varphi)$$

Otherwise, $\varphi[k/(p - \beta)]$ in L_2, and we can apply again the same substitution steps as before, until all the pointer subtraction terms in φ are substituted.

[1] When p is resolved to a single memory object, then the constraint $p \geq \beta \wedge p < \beta + s$ is already implied by the path constraints. For simplicity, we can assume that this constraint is added to the path constraints in such cases as well.

If $p - p'$ was generated as a result of a program statement, then it must be the case:

$$p \triangleq \beta + n, \quad p' \triangleq \beta + m$$

where $\beta : Ptr$ is an uninterpreted constant and both n and m are Int terms in L_1. If $\varphi[(n-m)/(p-p')]$ in L_1, then it is address-agnostic. The term $n - m$ is in L_1 and:

$$p - p' \equiv n - m$$

so φ is address-agnostic according to Lemma 4.4.13, and therefore $pc \wedge \varphi$ is address-agnostic as well. Otherwise, $\varphi[(n-m)/(p-p')]$ in L_2, and we can apply similar substitution steps until all the pointer subtraction terms in φ are substituted. □

A.2 State Merging Optimizations

In this section, we prove that Algorithms 5 and 6 correctly construct the merged constraints and values. To do so, we first define:

Lemma A.2.1. *Let $\{n_i\}_{i=1}^k$ be leaf nodes in an execution tree t, let n be a node in t, and let f and φ be defined as follows:*

$$f, \varphi \triangleq \textit{merge-conditions-internal}(\{n_i.s\}_{i=1}^k, n)$$

Then:

$$\varphi \equiv \textit{merge-conditions}(\{\textit{tpc}(n, n_i)\}_{i=1}^k)$$

and if $f = \textit{true}$, then:

$$\varphi \equiv n.c$$

Proof. The proof will be done by induction on the structure of the execution tree originating at the node n.

Base case: Let n be a leaf node. First, suppose that $n \in \{n_i\}_{i=1}^k$. Then n is one of the nodes n_j (for some $1 \leq j \leq k$). By definition of *tpc*:

$$\textit{tpc}(n, n_j) \triangleq n.c$$

and for every $1 \leq i \leq k$ such that $i \neq j$:

$$\textit{tpc}(n, n_i) \triangleq \textit{false}$$

and therefore:

$$\textit{merge-conditions}(\{\textit{tpc}(n, n_i)\}_{i=1}^k) \equiv n.c$$

On the other end, by definition of *merge-conditions-internal*:

$$\textit{merge-conditions-internal}(\{n_i.s\}_{i=1}^k, n) \triangleq \textit{true}, n.c$$

Now, suppose that $n \notin \{n_i\}_{i=1}^k$. Then by definition of *tpc*, for every $1 \leq i \leq k$:

$$\textit{tpc}(n, n_i) \triangleq \textit{false}$$

and therefore:

$$\textit{merge-conditions}(\{\textit{tpc}(n, n_i)\}_{i=1}^{k}) \equiv \textit{false}$$

On the other end, by definition of *merge-conditions-internal*:

$$\textit{merge-conditions-internal}(\{n_i.s\}_{i=1}^{k}, n) \triangleq \textit{false}, \textit{false}$$

Induction step: Let n be an intermediate node. According to *merge-conditions-internal*, we have:

$$f_l, \varphi_l \triangleq \textit{merge-conditions-internal}(g, n.l)$$
$$f_r, \varphi_r \triangleq \textit{merge-conditions-internal}(g, n.r)$$
$$f \triangleq f_l \wedge f_r$$

By the induction hypothesis:

$$\textit{merge-conditions}(\{\textit{tpc}(n.l, n_i)\}_{i=1}^{k}) \equiv \varphi_l$$
$$\textit{merge-conditions}(\{\textit{tpc}(n.r, n_i)\}_{i=1}^{k}) \equiv \varphi_r$$

By definition of *tpc*:

$$\textit{merge-conditions}(\{\textit{tpc}(n, n_i)\}_{i=1}^{k}) \equiv$$
$$n.c \wedge (\textit{merge-conditions}(\{\textit{tpc}(n.l, n_i)\}_{i=1}^{k}) \vee \textit{merge-conditions}(\{\textit{tpc}(n.r, n_i)\}_{i=1}^{k}))$$

and therefore:

$$\textit{merge-conditions}(\{\textit{tpc}(n, n_i)\}_{i=1}^{k}) \equiv n.c \wedge (\varphi_l \vee \varphi_r)$$

First, suppose that $f = \textit{true}$. Then $f_l = \textit{true}$ and $f_r = \textit{true}$. By the induction hypothesis:

$$\varphi_l = n.l.c, \ \varphi_r = n.r.c$$

so:

$$\textit{merge-conditions}(\{\textit{tpc}(n, n_i)\}_{i=1}^{k}) \equiv n.c \wedge (n.l.c \vee n.r.c) \equiv n.c$$

The last equivalence is valid since the conditions of two sibling nodes are complementary.

On the other end:

$$\textit{merge-conditions-internal}(\{n_i.s\}_{i=1}^k, n) \triangleq \textit{true}, n.c$$

Now, suppose that $f = \textit{false}$. As before:

$$\textit{merge-conditions}(\{\textit{tpc}(n, n_i)\}_{i=1}^k) \equiv n.c \wedge (\varphi_l \vee \varphi_r)$$

On the other end, by the definition of *merge-conditions-internal*:

$$\varphi \triangleq n.c \wedge (\varphi_l \vee \varphi_r)$$

□

Lemma A.2.2. *Let $\{n_i\}_{i=1}^k$ be leaf nodes in an execution tree t, let n be a node in t, and let $\{v_i\}_{i=1}^k$ be a set of terms. If n_j is reachable from n, then:*

$$\textit{tpc}(n, n_j) \models \textit{merge-values-opt}(\{n_i.s\}_{i=1}^k, n, \{n_i.s \mapsto v_i\}_{i=1}^k) = v_j$$

Proof. The proof will be done by induction on the length of the path originating from n.

Base case: Let n be a leaf node. In that case, the state n is n_j. By definition:

$$\textit{merge-values-opt}(\{n_i.s\}_{i=1}^k, n, \{n_i.s \mapsto v_i\}_{i=1}^k) \triangleq v_j$$

Induction step: Let n be an intermediate node, and suppose without loss of generality that the path to n_j goes through n's left child. According to *merge-values-opt*:

$$v_l \triangleq \textit{merge-values-opt}(\{n_i.s\}_{i=1}^k, n.l, \{n_i.s \mapsto v_i\}_{i=1}^k)$$

By definition:

$$\textit{tpc}(n, n_j) \equiv n.c \wedge \textit{tpc}(n.l, n_j)$$

Note that n_j is reachable from $n.l$, so by the induction hypothesis:

$$\textit{tpc}(n.l, n_j) \models \textit{merge-values-opt}(\{n_i.s\}_{i=1}^k, n.l, \{n_i.s \mapsto v_i\}) = v_j$$

and therefore:

$$tpc(n.l, n_j) \models v_l = v_j$$

It is not possible that v_l is *null*, otherwise the leaf node n_j would be unreachable from $n.l$. Therefore, *merge-values-opt* returns v_l or $ite(n.l.c, v_l, v_r)$. We know that if $tpc(n, n_j)$ holds then $n.l.c$ holds as well, so in both cases *merge-values-opt* returns v_l, so:

$$tpc(n, n_j) \models \textit{merge-values-opt}(\{n_i.s\}_{i=1}^k, n, \{n_i.s \mapsto v_i\}_{i=1}^k) = v_l$$

Finally, since:

$$tpc(n, n_j) \models tpc(n.l, n_j)$$

we get that:

$$tpc(n, n_j) \models v_l = v_j$$

and thus we can conclude that:

$$tpc(n, n_j) \models \textit{merge-values-opt}(\{n_i.s\}_{i=1}^k, n, \{n_i.s \mapsto v_i\}_{i=1}^k) = v_j$$

□

Theorem A.2.3. *Let $\{n_i\}_{i=1}^k$ be leaf nodes in an execution tree t, let r be the root of t, and let ψ_i be the suffix of n_i's path constraints in t:*

$$\psi_i \triangleq tpc(r, n_i)$$

Then:

$$(a) \quad \textit{merge-conditions}(\{\psi_i\}_{i=1}^k) \equiv \textit{merge-conditions-opt}(\{n_i.s\}_{i=1}^k, r)$$

Let $\{v_i\}_{i=1}^k$ be terms, then:

$$(b) \quad (\bigvee_i \psi_i) \models \textit{merge-values}(\{\psi_i\}_{i=1}^k, \{v_i\}_{i=1}^k) = \textit{merge-values-opt}(\{n_i.s\}_{i=1}^k, r, \{n_i.s \mapsto v_i\})$$

Proof. By applying Lemma A.2.1 with n as the root r, we obtain (a). Now, if $(\bigvee_i \psi_i)$ holds, then there exists some $1 \leq j \leq k$, such that ψ_j holds. The constraints $\{\psi_i\}_{i=1}^n$

are pairwise unsatifiabile, so:

$$\psi_j \models \textit{merge-values}(\{\psi_i\}_{i=1}^k, \{v_i\}_{i=1}^k) = v_j$$

Recall that $\psi_j \triangleq \textit{tpc}(r, n_j)$, so according to the previous lemma:

$$\psi_j \models \textit{merge-values-opt}(\{n_i.s\}_{i=1}^k, r, \{n_i.s \mapsto v_i\}) = v_j$$

so:

$$\psi_j \models \textit{merge-values}(\{\textit{tpc}(r, n_i.s)\}_{i=1}^k, \{v_i\}_{i=1}^k) = \textit{merge-values-opt}(\{n_i.s\}_{i=1}^k, r, \{n_i.s \mapsto v_i\})$$

□

A.3 Pattern-Based State Merging

A.3.1 Proof of Theorem 6.2.7

Theorem A.3.1. *Under the premises of Definition 6.2.6, let s be the pattern-based merged symbolic state of $\{n_j.s\}_{j=1}^{n}$, and let s' be their merged symbolic state obtained with standard state merging (Definition 2.5.1). The following holds for any model m:*

1. *$m \models s'.pc$ iff $m[k \mapsto \tilde{k}] \models s.pc$ for some $\tilde{k} \in \mathbb{N}$.*

2. *If $m[k \mapsto \tilde{k}] \models s.pc$ for some $\tilde{k} \in \mathbb{N}$, then $m(s'.vars(v)) = m[k \mapsto \tilde{k}](s.vars(v))$ for every variable v.*

Proof. Let r be the root of the execution tree t. According to the validity of t (Section 2.4), the following holds for every $j = 1, ..., n$:

$$n_j.s.pc \equiv r.s.pc \wedge tpc(n_j)$$

Without loss of generality, we assume that $r.s.pc \equiv true$. According to Definition 2.5.1:

$$s'.pc \triangleq \bigvee_{j=1}^{n} tpc(n_j)$$

We assume that $\{(n_j, k_j)\}_{j=1}^{n}$ match the formula pattern $(\varphi_1, \varphi_2(x), \varphi_3(x))$, so:

$$tpc(n_j) \doteq \varphi_1 \wedge \big(\bigwedge_{i=1}^{k_j} \varphi_2[i/x]\big) \wedge \varphi_3[k_j/x]$$

According to Definition 6.2.6:

$$s.pc \triangleq (\bigvee_{j=1}^{n} k = k_j) \wedge \varphi_1 \wedge (\forall i.\ 1 \leq i \leq k \rightarrow \varphi_2[i/x]) \wedge \varphi_3[k/x]$$

First, we prove (1).

$\Rightarrow$:

If $m \models s'.pc$, then there exists $1 \leq j \leq n$ such that:

$$m \models tpc(n_j)$$

Let $m' \triangleq m[k \mapsto k_j]$. We will show now that $m' \models s.pc$. Clearly, $m' \models k = k_j$, so:

$$m' \models \bigvee_{j=1}^{n} k = k_j$$

Note that $tpc(n_j)$ can be rewritten as follows:

$$tpc(n_j) \equiv \varphi_1 \wedge (\forall i.\ (1 \leq i \leq k_j \rightarrow \varphi_2[i/x])) \wedge \varphi_3[k_j/x]$$

As $m \models tpc(n_j)$ and $m' \models k = k_j$, and k does not appear in $\varphi_1, \varphi_2(x), \varphi_3(x)$:

$$m' \models \varphi_1 \wedge (\forall i.\ (1 \leq i \leq k \rightarrow \varphi_2[i/x])) \wedge \varphi_3[k/x]$$

and consequently:

$$m' \models (\bigvee_{j=1}^{n} k = k_j) \wedge \varphi_1 \wedge (\forall i.\ (1 \leq i \leq k \rightarrow \varphi_2[i/x])) \wedge \varphi_3[k/x]$$

$\Leftarrow$:

If there exists $\tilde{k} \in \mathbb{N}$ such that $m[k \mapsto \tilde{k}] \models s.pc$, then there exists $1 \leq j \leq n$ such that:

$$m[k \mapsto \tilde{k}](k) = m[k \mapsto \tilde{k}](k_j)$$

so:

$$m[k \mapsto \tilde{k}] \models \varphi_1 \wedge (\forall i.\ (1 \leq i \leq k_j \rightarrow \varphi_2[i/x])) \wedge \varphi_3[k_j/x]$$

and in particular (as k does not appear in the formula above):

$$m \models \varphi_1 \wedge (\forall i.\ (1 \leq i \leq k_j \rightarrow \varphi_2[i/x])) \wedge \varphi_3[k_j/x]$$

As mentioned before:

$$tpc(n_j) \equiv \varphi_1 \wedge (\forall i.\ (1 \leq i \leq k_j \rightarrow \varphi_2[i/x])) \wedge \varphi_3[k_j/x]$$

so $m \models tpc(n_j)$ and therefore:

$$m \models \bigvee_{j=1}^{n} tpc(n_j)$$

Second, we prove (2). Suppose that $m[k \mapsto \tilde{k}] \models s.pc$, and let v be a variable in the

symbolic store. The interesting case is when the merged value of v is encoded without *ite* expressions. That is, when there exists a term $t(x)$ with a free variable x such that:

$$t[k_j/x] \doteq n_j.s.vars(v) \quad \text{(for every } j = 1, \ldots, n)$$

and the value of v in s is encoded as:

$$s.vars(v) \triangleq t[k/x]$$

We already proved that if $m[k \mapsto \tilde{k}] \models s.pc$ then there must exist $1 \leq j \leq n$ such that:

$$m[k \mapsto \tilde{k}] \models tpc(n_j), \quad m[k \mapsto \tilde{k}](k) = m[k \mapsto \tilde{k}](k_j)$$

Recall that $s'.vars(v)$ is defined by:

$$ite(tpc(n_1), n_1.vars(v), ite(tpc(n_2), n_2.vars(v), \ldots))$$

which can be rewritten as:

$$ite(tpc(n_1), t[k_1/x], ite(tpc(n_2), t[k_2/x], \ldots))$$

Recall that $\{tpc(n_j)\}_{j=1}^{n}$ correspond to path conditions in the execution tree t, which are pairwise unsatisfiable, so:

$$m[k \mapsto \tilde{k}](s'.vars(v)) = m[k \mapsto \tilde{k}](t[k_j/x])$$

and since $m[k \mapsto \tilde{k}](k) = m[k \mapsto \tilde{k}](k_j)$, we get:

$$m[k \mapsto \tilde{k}](t[k_j/x]) = m[k \mapsto \tilde{k}](t[k/x])$$

Finally, the term $s'.vars(v)$ does not contain the term k, so:

$$m[k \mapsto \tilde{k}](s'.vars(v)) = m(s'.vars(v))$$

□

A.3.2 Proof of Lemma 6.2.8

Lemma A.3.2. *The following holds for any two nodes n_1 and n_2 in t:*

1. *If $h(\pi(n_1)) = h(\pi(n_2))$ then $n_1 = n_2$.*
2. *If $h(\pi(n_1))$ is a prefix of $h(\pi(n_2))$, then there is a single path $\pi(n_1, n_2)$ in t.*

Proof. First, we prove (1):
The proof will be done by induction on the length of the hash, which is a sequence of numbers.

Base case: The length of the hash sequences is one, so it must be the case where the paths $\pi(n_1)$ and $\pi(n_2)$ start from the root r. Therefore, it must hold that:

$$n_1 = n_2 = r$$

Induction step: We assume that:

$$h(\pi(n_1)) = h(\pi(n_2)) = h_1 \dots h_{n-1} h_n \text{ where } h_i \in \mathbb{N}$$

Let n_1' and n_2' be the nodes preceding n_1 and n_2, respectively:

$$\pi(n_1) = \pi(n_1'); n_1, \;\; \pi(n_2) = \pi(n_2'); n_2$$

We know that:

$$h(\pi(n_1')) = h(\pi(n_2')) = h_1 \dots h_{n-1}$$

so by the induction hypothesis:

$$n_1' = n_2'$$

Thus, we can conclude that n_1 and n_2 have the same parent node, i.e., n_1 and n_2 are sibling nodes. Note that:

$$h(\pi(n_1)) = h(\pi(n_1'))h(n_1), \quad h(\pi(n_2)) = h(\pi(n_2'))h(n_2)$$

so it must hold that:

$$h(n_1) = h(n_2)$$

We assumed that h is valid for t (Definition 6.2.1), so if $n_1 \neq n_2$, then $h(n_1) \neq h(n_2)$.

Therefore, $n_1 = n_2$.

Second, we prove (2):

If $h(\pi(n_1))$ is a prefix of $h(\pi(n_2))$, then there exists $\omega \in \mathbb{N}^*$ such that:

$$h(\pi(n_2)) = h(\pi(n_1)); \omega$$

According to the definition of h, there exists a node n on the path $\pi(n_2)$ such that:

$$h(\pi(n)) = h(\pi(n_1))$$

According to (1), this means that n must be n_1, so there is a path $\pi(n_1, n_2)$. Each edge in the execution tree t goes from the parent node to the child node, so the path $\pi(n_1, n_2)$ is unique.

□

A.3.3 Proof of Lemma 6.2.10

Lemma A.3.3. *Let n be a leaf node in an execution tree t, and suppose that:*

$$h(\pi(n)) = \omega_1\omega_2...\omega_j$$

Then:

$$\begin{aligned} tpc(n) \doteq\ & extract(\omega_1) \wedge \\ & extract(\omega_1, \omega_1\omega_2) \wedge \\ & ... \\ & extract(\omega_1...\omega_{j-1}, \omega_1...\omega_{j-1}\omega_j) \end{aligned}$$

Proof. The proof is done by induction on j, i.e., the length of $h(\pi(n))$.

Base case: If $j = 1$, then:

$$h(\pi(n)) = \omega_1$$

and by the definition of *extract*:

$$tpc(n) \doteq extract(\omega_1)$$

Induction step: We assume that:

$$h(\pi(n)) = \omega_1 \dots \omega_{n-1}\omega_n$$

By the definition of h, there exists a node n' such that:

$$h(\pi(n')) = \omega_1 \dots \omega_{n-1}$$

According to the induction hypothesis:

$$\begin{aligned} tpc(n') \doteq\ & extract(\omega_1) \wedge \\ & extract(\omega_1, \omega_1\omega_2) \wedge \\ & \dots \wedge \\ & extract(\omega_1 \dots \omega_{n-2}, \omega_1 \dots \omega_{n-1}) \end{aligned}$$

and according to the definition of *extract*:

$$\overline{tpc}(n', n) \doteq extract(\omega_1 \dots \omega_{n-1}, \omega_1 \dots \omega_n)$$

Finally, from the definition of *tpc*:

$$\begin{aligned} tpc(n) &\doteq tpc(n') \wedge \overline{tpc}(n', n) \\ &\doteq extract(\omega_1) \wedge \\ &\quad extract(\omega_1, \omega_1\omega_2) \wedge \\ &\quad \dots \wedge \\ &\quad extract(\omega_1 \dots \omega_{n-2}, \omega_1 \dots \omega_{n-1}) \\ &\quad extract(\omega_1 \dots \omega_{n-1}, \omega_1 \dots \omega_n) \end{aligned}$$

□

A.3.4 Proof of Theorem 6.2.11

Theorem A.3.4. *Given an execution tree t and a set $\{n_j\}_{j=1}^n$ of leaf nodes in t, suppose that $\{(n_j, k_j)\}_{j=1}^n$ match the regular pattern $(\omega_1, \omega_2, \omega_3)$. If $(\varphi_1, \varphi_2(x), \varphi_3(x))$*

is a formula pattern that satisfies:

$$\varphi_1 \doteq extract(\omega_1)$$
$$\varphi_2[i/x] \doteq extract(\omega_1\omega_2^{i-1}, \omega_1\omega_2^{i}) \quad (i = 1, ..., max\{k_j\}_{j=1}^{n})$$
$$\varphi_3[k_j/x] \doteq extract(\omega_1\omega_2^{k_j}, \omega_1\omega_2^{k_j}\omega_3) \quad (j = 1, ..., n)$$

then $\{(n_j, k_j)\}_{j=1}^{n}$ *match* $(\varphi_1, \varphi_2(x), \varphi_3(x))$.

Proof. We assumed that $\{(n_j, k_j)\}_{j=1}^{n}$ match $(\omega_1, \omega_2, \omega_3)$, so:

$$h(\pi(n_j)) \triangleq \omega_1\omega_2^{k_j}\omega_3 \quad (j = 1, ..., n)$$

According to Lemma 6.2.10:

$$\begin{aligned} tpc(n_j) \doteq\ & extract(\omega_1) \wedge \\ & ... \\ & extract(\omega_1\omega_2^{k_j-1}, \omega_1\omega_2^{k_j}) \wedge \\ & extract(\omega_1\omega_2^{k_j}, \omega_1\omega_2^{k_j}\omega_3) \end{aligned}$$

and we also assumed that:

$$\varphi_1 \doteq extract(\omega_1)$$
$$\varphi_2[i/x] \doteq extract(\omega_1\omega_2^{i-1}, \omega_1\omega_2^{i}) \quad (i = 1, ..., max\{k_j\}_{j=1}^{n})$$
$$\varphi_3[k_j/x] \doteq extract(\omega_1\omega_2^{k_j}, \omega_1\omega_2^{k_j}\omega_3) \quad (j = 1, ..., n)$$

so:

$$tpc(n_j) \doteq \varphi_1 \wedge \bigwedge_{i=1}^{k_j} \varphi_2[i/x] \wedge \varphi_3[k_j/x] \quad (j = 1, ..., n)$$

and therefore $\{(tpc(n_j), k_j)\}_{j=1}^{n}$ match $(\varphi_1, \varphi_2(x), \varphi_3(x))$. □