A Bounded Symbolic-Size Model for Symbolic Execution

David Trabish
Tel Aviv University
Israel
davivtra@post.tau.ac.il

Shachar Itzhaky
Technion
Israel
shachari@cs.technion.ac.il

Noam Rinetzky
Tel Aviv University
Israel
maon@cs.tau.ac.il

ABSTRACT

Symbolic execution is a powerful program analysis technique which allows executing programs with symbolic inputs. Modern symbolic execution tools use a concrete modeling of object sizes, that does not allow symbolic-size allocations. This leads to concretizations and enforces the user to set the size of the input ahead of time, thus potentially leading to loss of coverage during the analysis.

We present a bounded symbolic-size model in which the size of an object can have a range of values limited by a user-specified bound. Unfortunately, this model amplifies the problem of path explosion, due to additional symbolic expressions representing sizes. To cope with this problem, we propose an approach based on state merging that reduces the forking by applying special treatment to symbolic-size dependent loops.

In our evaluation on real-world benchmarks, we show that our approach can lead in many cases to substantial gains in terms of performance and coverage, and find previously unknown bugs.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Symbolic Execution

ACM Reference Format:
David Trabish, Shachar Itzhaky, and Noam Rinetzky. 2021. A Bounded Symbolic-Size Model for Symbolic Execution. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468596

1 INTRODUCTION

Symbolic execution (SE) is a program analysis technique that has gained significant attention over the last years in both academic and industrial areas, including software engineering, software testing, programming languages, program verification, and cybersecurity. It lies at the core of many applications such as high-coverage test generation [14, 15, 32], bug finding [14, 20], debugging [23], automatic program repair [28, 30], cross checking [16, 25], and side-channel analysis [12, 13, 31]. In symbolic execution, the program is run with an unconstrained *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 [17] is used to determine the feasibility of each branch side, and the appropriate paths are further explored while updating their paths constraints with the corresponding constraints. Once the execution of a given path is completed, the solver generates a satisfying assignment using the corresponding path constraints, i.e., a concrete test case that can be used to replay that path.

In modern symbolic execution tools [14, 29, 38], 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* [6] shown in Figure 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 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. So when that happens, the call to `strchr` at line 22 results in an *out-of-bound* read, since `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, that would result in an *out-of-bound* 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 as well.

In this paper, we propose a model that supports symbolic-size allocations, and thus enabling the analysis of programs with inputs

ESEC/FSE '21, August 23–28, 2021, Athens, Greece
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468596

whose size belongs to a range of values. Designing an *unbounded* model in which the symbolic size of an object is unconstrained, imposes several difficulties. Symbolic execution tools typically use a linear address space where the base addresses of memory objects are concrete values, so address ranges 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 object has its own address space, or use symbolic base addresses with some additional constraints that will ensure the non-overlapping of address ranges. Besides, symbolic execution tools use the *QF_ABV* logic fragment [10, 11, 19] 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 objects will not be possible, as it will require an amount of memory which is not available 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 symbolic execution tools.

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, the main scenario where our model introduces additional forking.

Our main contributions can be summarized as follows:

(1) We present a bounded symbolic-size model that enables analysis with variable-size inputs.
(2) We propose a state merging based 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.

2 PRELIMINARIES

In modern symbolic execution engines, e.g., KLEE [14], 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 a solver array that tracks the values written to that memory object. The address space is then represented as a set of *non-overlapping* memory objects, i.e., every memory object has its own unique address range which does not intersect with the address ranges of other memory objects. This non-overlapping property allows identifying memory objects by addresses, i.e., a concrete address may be associated with at most one memory object.

When a pointer p is accessed, the engine needs first to resolve it, i.e., find the memory objects that p may point to. To determine if a memory object $mo = (b, s, a)$ may be pointed by p, the engine checks if the following *resolution* query is satisfiable:

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

```c
int osip_via_parse(osip_via_t *via, const char *hvalue) {
    if (hvalue == NULL)
        return OSIP_BADPARAMETER;
    const char *version = strchr(hvalue, '/');
    if (version == NULL)
        return OSIP_SYNTAXERROR;
    const char *protocol = strchr(version + 1, '/');
    if (protocol == NULL)
        return OSIP_SYNTAXERROR;
    if (protocol - version < 2)
        return OSIP_SYNTAXERROR;
    ...
    const char *host = strchr(protocol + 1, '␣');
    if (host == NULL)
        return OSIP_SYNTAXERROR;
    if (host == protocol + 1) {
        while (0 == strncmp(host, "␣", 1)) {
            host++;
            if (strlen(host) == 1)
                return OSIP_SYNTAXERROR;
        }
        // out-of-bound read
        host = strchr(host + 1, '␣');
        ...
    }
    ...
}
int osip_uri_parse_headers(osip_uri_t *url,
                           const char *headers) {
  const char *equal = strchr(headers, '=');
  // out-of-bound read
  const char *_and = strchr(headers + 1, '&');
  ...
}
```

Figure 1: Bugs found in *libosip* 5.2.0.

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

Figure 2: Unbounded symbolic size.

When p may point to mo, the offset e with which the object is accessed is given by: $p - b$. When the e^{th} byte of mo is read, its value is expressed by $select(a, e)$. If a value v is written to the e^{th} offset, the array a is replaced by a new array expressed by $store(a, e, v)$.

A symbolic state s is represented as follows: $s.pc$ denotes the path constraints, $s.m$ is a mapping between variables and expressions (concrete or symbolic), and $s.h$ is a set of memory objects.

3 TECHNIQUE

3.1 Symbolic-Size Model

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

In the concrete-size model every object has a fixed address range, so when a new object has to be allocated, the memory allocator can easily pick a new address range 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 2. Let's assume

```
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 3: Symbolic-size dependent loop.

that the first memory object (line 2) is allocated at address 7000 and has an unbounded symbolic size n. The symbolic execution engine can't allocate the second memory object (line 3) at a concrete base address after the first memory object as the resulting address range might overlap with the address range 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 property of the two memory objects:

$$[7000, 7000 + n) \cap [\beta, \beta + 10) = \emptyset$$

will be encoded using the following constraints:

$$\beta + 10 \leq 7000 \lor \beta \geq 7000 + n$$

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 solver.

Symbolic execution 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 2, the first object is allocated with `calloc`, which initializes the memory with zeros. If we don't have a concrete bound for the size of this object, i.e., the maximum value for n, then we would be forced to use some form the universal (forall) quantifier to express the side effect 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 object would be at least *linear*. That means that for large enough objects, the analysis will be impossible due to the extremely high memory usage, suggesting that the size of a given 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 and a are defined similarly to Section 2, σ is a symbolic expression describing the size of the memory object, and c is the maximum concrete value for σ, i.e., the *capacity* of the memory object. The pointer resolution process together with the read and write operations (described in Section 2) remain almost unmodified with the small change of replacing the concrete size s with the symbolic one σ.

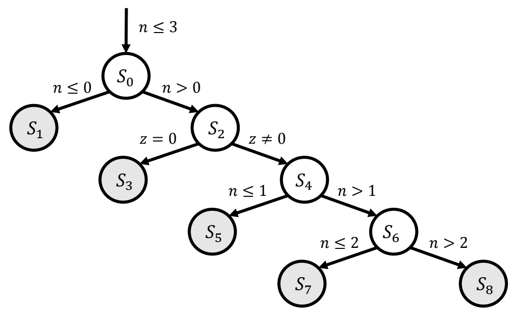

Figure 4: The execution tree of the program from Figure 3.

3.2 Mitigating Path Explosion By State Merging

Our model enables a more *complete* analysis since it supports the symbolic execution of programs with 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 objects sizes, 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 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 3, 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 [22, 26], 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 join point.

3.2.1 Merging Symbolic-Size Objects. In our model, the merging of symbolic states is defined similarly to previous works [22, 26], 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 $mo_1 = (b, \sigma, c, a_1)$ and $mo_2 = (b, \sigma, c, a_2)$ 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:

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

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 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} offset of the solver array a, the symbolic execution engine always validates that i is in-bound, i.e., $i < \sigma$. Therefore, even if i is an out-of-bound 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.

3.2.2 *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 basic blocks. 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.

3.2.3 *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 symbolic execution engine allocates an object (b, σ, c, a) with a symbolic size, we mark the (atomic) symbolic variables that build the expression 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.*

3.2.4 *Loop Merging.*

Algorithm 1 depicts the application of state merging for symbolic-size dependent loops[1]. When a symbolic state s executing a loop l reaches a branch whose condition c is tainted, we associate s with a new *merging context* (line 5), if not already set, initializing its loop, set of states, and liveness counter. We then associate the states s_t and s_f (forked at line 7) with the merging context mc, add them to mc, update the liveness counter, and finally update the searcher's worklist (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 mc finished executing the loop, we finally perform the merging: As the merging of two symbolic states is allowed only when their *program counter* points to the same location, we first split the states to groups based on the loop exit that was taken in each state (line 18), and then merge each group of states separately (line 20). Finally, we reset the merging context of the merged state m (line 21), and update the engine's searcher (line 22).

Algorithm 2 depicts the merging procedure which accepts as parameters the merging context and the set of states to merge. First, we compute the common constraints of the input states (line 2), which can be also given by the path constraints of the state that initialized the merging context. Then, we extract for each state s_i the suffix constraints pc'_i (line 3), i.e., the constraints of $s_i.pc$ that does not appear in c, which are the constraints that were added by s_i during the execution of the loop till its exit. The constraints of the merged state are set to the conjunction of c and the disjunction of all the suffix constraints (line 4). For each variable v, its value in the merged state is set to an *ite* expression (line 5), which results in the value $s_i.m(v)$ if $s_i.pc$ holds. The heap merging (lines 6-12) is performed assuming that the input states are *heap-compatible*, i.e., for every memory object in one state there is a memory object in another state which has the same base address, size, and capacity. Under this assumption, we group the memory objects according to the base addresses (line 7), such that each group corresponds to a specific memory object with a_i being the solver array tracking the contents of that memory object in s_i. For each such group, we first create a solver array (line 8) that will be used later to construct the merged memory object. Then for every offset up to the corresponding capacity, we merge the values stored at that offset separately (line 10) and update the new solver array (line 11). Finally, we add the resulting memory object to the heap (line 12).

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

As an example, consider again the program from Figure 3 whose execution tree is given in Figure 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 loop is detected as symbolic-size dependent, since the branch condition $0 < n$ is tainted. The loop at line 4 has two loop exits (lines 6,9), so once the exploration of the loop is completed, the states are split into two merging groups. The first group contains only the state s_3 whose path constraints are $n > 0 \land z = 0$, and the second group contains the rest of the states, i.e., $\{s_1, s_5, s_7, s_8\}$. In the merged state of the second group, for example, the paths constraints will be:

$$(n \leq 3) \land (c_1 \lor c_5 \lor c_7 \lor c_8)$$

where:

$$c_1 := (n \leq 0)$$
$$c_5 := (n > 0 \land z \neq 0 \land n \leq 1)$$
$$c_7 := (n > 0 \land z \neq 0 \land n > 1 \land n \leq 2)$$
$$c_8 := (n > 0 \land z \neq 0 \land n > 1 \land n > 2)$$

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

$$ite(c_1, e, ite(c_5, e, ite(c_7, e, 2)))$$

where $e = select(a, 2)$ is an uninitialized solver array value.

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 states, due to the introduction of *ite* and *disjunctive* terms. As the representation complexity has a direct impact on the complexity of the queries, and therefore on the performance of the solver itself, the merged states should be represented as compactly as possible.

3.3 Optimizations

When we merge multiple symbolic states, we generate *ite* and *disjunctive* terms that may contain duplicate or redundant terms. This happens, for example, in the program from Figure 3, when we merge the states from the merging group that corresponds to the loop exit at line 9. Let c_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 state is given by (as mentioned in Section 3.2):

$$(n \leq 3) \land (c_1 \lor c_5 \lor c_7 \lor c_8)$$

Note that there are conditions that unnecessarily repeat across the different constraints: For example, the condition $n > 0 \land z \neq 0$ appears in the last three disjuncts (c_5, c_7, and c_8), and the condition $n > 1$ appears in both c_7 and c_8. Also note that the disjunction of

Algorithm 1 Loop Merging Algorithm.

```
1:  function ON-BRANCH($s, c, l$)
2:    if IS-TAINTED($c$) then
3:      $mc \leftarrow s.mc$
4:      if $mc = null$ then
5:        $mc \leftarrow \{.loop : l,\ .states : \{s\},\ .counter : 1\}$
6:        $mc.root \leftarrow s.n \leftarrow \{.s : s,\ .c : true,\ .l = null,\ .r = null\}$
7:      $s_t, s_f \leftarrow$ FORK($s, c$)
8:      $s_t.mc \leftarrow mc,\ s_f.mc \leftarrow mc$
9:      $mc.states \leftarrow (mc.states \setminus \{s\}) \cup \{s_t, s_f\}$
10:     $mc.counter \leftarrow mc.counter + 1$
11:     $s.n.l \leftarrow s_t.n \leftarrow \{.s : s_t,\ .c : c,\ .l = null,\ .r = null\}$
12:     $s.n.r \leftarrow s_f.n \leftarrow \{.s : s_f,\ .c : \neg c,\ .l = null,\ .r = null\}$
13:     $worklist \leftarrow (worklist \setminus \{s\}) \cup \{s_t, s_f\}$
14: function ON-LOOP-EXIT($s, l$)
15:   if $s.mc.loop = l$ then
16:     $mc.counter \leftarrow mc.counter - 1$
17:     if $s.mc.counter = 0$ then
18:       $groups \leftarrow$ GROUP-BY-LOOP-EXIT($mc.states$)
19:       for $g \in groups$ do
20:         $m \leftarrow$ MERGE($s.mc, g$)
21:         $m.mc \leftarrow null$
22:         $worklist \leftarrow (worklist \setminus g) \cup \{m\}$
```

Algorithm 2 Merging Algorithm.

```
1:  function MERGE($mc, \{s_1, ..., s_n\}$)
2:    $c \leftarrow$ COMMON-CONSTRAINTS($[s_i.pc]_{i=1}^n$)
3:    $[pc'_i]_{i=1}^n \leftarrow$ [SUFFIX-CONSTRAINTS($s_i.pc, c$)]
4:    $pc \leftarrow c \wedge$ MERGE-CONSTRAINTS($[pc'_i]_{i=1}^n$)
5:    $m \leftarrow \{v \mapsto$ MERGE-VALUES($[s_i]_{i=1}^n, [pc'_i]_{i=1}^n, \{s_i \mapsto s_i.m[v]\}$)$\}$

6:    $h \leftarrow \emptyset$
7:    for $(b, \sigma, k, [a_i]_{i=1}^n) \in$ GROUP-BY-ADDRESS($\{s_i\}_{i=1}^n$) do
8:      $a \leftarrow$ NEW-SMT-ARRAY()
9:      for $0 \leq j < k$ do
10:       $e \leftarrow$ MERGE-VALUES($[s_i]_{i=1}^n, [pc'_i]_{i=1}^n, \{s_i \mapsto select(a_i, j)\}$)
11:       $a \leftarrow store(a, j, e)$
12:     $h \leftarrow h \cup (b, \sigma, k, a)$
13:   return $\{.pc : pc,\ .m : m,\ .h : h\}$
14: function MERGE-CONSTRAINTS($[c_i]_{i=1}^n$)
15:   return $c_1 \vee ... \vee c_n$
16: function MERGE-VALUES($[s_i]_{i=1}^n, [c_i]_{i=1}^n, m$)
17:   return $ite(c_1, m[s_1], ite(..., ite(c_{n-1}, m[s_{n-1}], m[s_n])))$
```

the last three constraints ($c_5 \vee c_7 \vee c_8$) is actually equivalent to ($n > 0 \wedge z \neq 0$), since:

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

Similar redundancies occur when we merge the contents of the memory object allocated at line 3.

Instead of optimizing on top of 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 constructed during the symbolic evaluation of the loop in a given merging context. Each node in the execution tree is either a leaf node that corresponds to a final 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 annotated with a corresponding state and the constraint because of which that state was forked, where the constraint of the root node is initialized to *true*. In the execution tree from Figure 4, 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 1 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, each of which is annotated with the appropriate condition c and $\neg c$, respectively.

Our constraint merging procedure is given in Algorithm 3. The procedure MERGE-CONSTRAINTS-OPT receives a merging group g from the given merging context and a node n from the execution tree. If n is a leaf node, then we return its annotated condition if its state belongs to g (lines 2-3), and $false$ otherwise. If n is an intermediate node, then we recursively generate the constraints c_l and c_r for the children nodes (lines 4-5). If f does not hold, i.e., n's sub-tree contains at least one state that does not belong to g, then we return the conjunction of the current condition with the disjunction of the children's constraints (line 10). If f holds, then we are in the case where n's sub-tree contains states from g only. The disjunction of the constraints corresponding to any complete sub-tree is always equivalent to $true$, therefore the term $c_l \vee c_r$ can be further simplified to $true$ (line 8). Note that the current condition $n.c$ is always added only once, thus avoiding duplicate occurrences. Also note that if c_l (or c_r) is $false$, i.e., the sub-tree originating from $n.l$ (or $n.r$) does not contain states from g, then only the condition c_r (or c_l) is propagated.

As an example, consider the application of Algorithm 3 with g as the merging group $\{s_1, s_5, s_7, s_8\}$, and n as the root of the execution tree from Figure 4. Without the simplification at line 8, 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 8, the constraint is further reduced to:

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

Our value merging procedure MERGE-VALUES-OPT is given in Algorithm 4. It receives as parameters a merging group g from the given merging context, a node n from the execution tree, and a mapping m which associates a state with the value of the target memory location (variable or heap object). For leaf nodes we return the value associated with $n.s$ if the latter exists in the mapping, and *null* otherwise (lines 2-3). For intermediate nodes, if a merged value is available only in one of the children (lines 9,11), 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 the values v_l and v_r (line 13).

Algorithm 3 Optimized constraint merging

```
1: function MERGE-CONSTRAINTS-OPT($g, n$)
2:   if IS-LEAF($n$) then
3:     return $true, n.c$ if $n.s \in g$ else $false, false$
4:   $f_l, c_l$ = MERGE-CONSTRAINTS-OPT($g, n.l$)
5:   $f_r, c_r$ = MERGE-CONSTRAINTS-OPT($g, n.r$)
6:   $f \leftarrow f_l \wedge f_r$
7:   if $f$ then
8:     $c \leftarrow n.c$
9:   else
10:    $c \leftarrow n.c \wedge (c_l \vee c_r)$
11:  return $f, c$
```

Algorithm 4 Optimized value merging

```
1: function MERGE-VALUES-OPT($g, n, m$)
2:   if IS-LEAF($n$) then
3:     return $m[n.s]$ if $n.s \in g$ else $null$
4:   $v_l$ = MERGE-VALUES-OPT($g, n.l, m$)
5:   $v_r$ = MERGE-VALUES-OPT($g, n.r, m$)
6:   if $v_l = null \wedge v_r = null$ then
7:     $v \leftarrow null$
8:   else if $v_l = null \wedge v_r \neq null$ then
9:     $v \leftarrow v_r$
10:  else if $v_l \neq null \wedge v_r = null$ then
11:    $v \leftarrow v_l$
12:  else
13:    $v \leftarrow ite(n.l.c, v_l, v_r)$
14:  return $v$
```

For example, when we use the original procedure to merge the value of `p[2]` for the merging group $\{s_1, s_5, s_7, s_8\}$, we get:

$$ite(c_1, e, ite(c_5, e, ite(c_7, e, 2)))$$

assuming that p's memory object is $(b, n, 3, a)$ and $e = select(a, 2)$. When applying the optimized procedure, we get:

$$ite(n \leq 0, e, ite(n \leq 1, e, ite(n \leq 2, e, 2)))$$

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

To incorporate Algorithms 3 and 4 in Algorithm 2, we replace the original invocation of MERGE-CONSTRAINTS at line 4 with:

$pc \leftarrow c \wedge \text{MERGE-CONSTRAINTS-OPT}(\{s_i\}_{i=1}^{n}, mc.root)$

and replace the invocations of MERGE-VALUES at lines 5 and 10 with:

$m \leftarrow \{v \mapsto \text{MERGE-VALUES-OPT}(\{s_i\}_{i=1}^{n}, mc.root, \{s_i \mapsto s_i.m[v]\})\}$

and

$e \leftarrow \text{MERGE-VALUES-OPT}(\{s_i\}_{i=1}^{n}, mc.root, \{s_i \mapsto select(a_i, j)\})$

respectively.

Algorithms 3 and 4 are search heuristic independent, 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 states to be merged. This is an improvement over Algorithm 2, where the worst case complexity is quadratic.

3.3.1 *Correctness.*

Without loss of generality, we assume that the initial path constraints $c = true$, and show that Algorithms 3 and 4 correctly construct the merged constraints and values:

$$(a) \quad \models \text{MERGE-CONSTRAINTS}([pc_i]_{i=1}^{k}) \leftrightarrow \text{MERGE-CONSTRAINTS-OPT}\left(\{s_i\}_{i=1}^{k}, r\right)$$

$$(b) \quad \left(\bigvee_i pc_i\right) \models \text{MERGE-VALUES}\left([s_i]_{i=1}^{k}, [pc_i]_{i=1}^{k}, m\right) = \text{MERGE-VALUES-OPT}\left(\{s_i\}_{i=1}^{k}, r, m\right)$$

where s_i is a leaf state in the execution tree, pc_i denotes the path constraints of s_i, m associates a state s_i to a corresponding value in that state, and r is the root of the execution tree.

Proof sketch. A fork creates two states whose path constraints differ by one negation, i.e., $pc_1 = r.c \wedge \neg\varphi$ and $pc_2 = r.c \wedge \varphi$. Going forward, each state will have accumulated some additional constraints: $pc_1' = r.c \wedge \neg\varphi \wedge \psi_1$ and $pc_2' = r.c \wedge \varphi \wedge \psi_2$. The standard merging algorithm MERGE-CONSTRAINTS will merge them as:

$$pc_1' \vee pc_2' = (r.c \wedge \neg\varphi \wedge \psi_1) \vee (r.c \wedge \varphi \wedge \psi_2)$$

which is equivalent, by reordering and distributivity, to:

$$r.c \wedge \left((\neg\varphi \wedge \psi_1) \vee (\varphi \wedge \psi_2)\right)$$

which is what MERGE-CONSTRAINTS-OPT would generate in this case, where $c_l \equiv \neg\varphi \wedge \psi_1$ and $c_r \equiv \varphi \wedge \psi_2$. This shows that the results are equivalent for the first forking point. If there are subsequent forks, the argument for them is identical, therefore, by induction, the same holds for the merged constraints generated for the entire merging context.

The argument regarding MERGE-VALUES-OPT is analogous. The path constraints $\{pc_i\}_{i=1}^{n}$ are disjoint (i.e., $\models \neg(pc_i \wedge pc_j)$, for $i \neq j$), so the following holds for each $1 \leq i \leq n$:

$$pc_i \models \text{MERGE-VALUES}\left([s_i]_{i=1}^{k}, [pc_i]_{i=1}^{k}, m\right) = m[s_i]$$

Similarly, we assume that the paths from the root to every s_i also represent disjoint conditions. Let:

$$\pi(s_i, n) := \bigwedge\{n'.c \mid n' \text{ on path } n \rightsquigarrow s_i\}$$

We now show that for every descendant n of r:

$$\pi(s_i, n) \models \text{MERGE-VALUES-OPT}\left(\{s_i\}_{i=1}^{k}, n, m\right) = m[s_i]$$

This can also be shown by induction. For the base case, i.e., $n.s = s_i$, MERGE-VALUES-OPT returns $m[n.s] = m[s_i]$ (line 3). Otherwise n is an intermediate node, and we assume without loss of generality that the path goes through the left node, i.e., $n.l.c$ holds, then by the induction hypothesis:

$$\pi(s_i, n.l) \models \text{MERGE-VALUES-OPT}\left(\{s_i\}_{i=1}^{k}, n.l, m\right) = m[s_i]$$

By definition, it holds that $\pi(s_i, n) \equiv n.l.c \wedge \pi(s_i, n.l)$, so the return value of MERGE-VALUES-OPT$(\{s_i\}_{i=1}^{k}, n, m)$, which is v_l or $ite(n.l.c, v_l, v_r)$, can be simplified to v_l as $n.l.c$ holds. Note that v_l is exactly MERGE-VALUES-OPT$(\{s_i\}_{i=1}^{k}, n.l, m)$, which completes the induction step. Since by construction, $pc_i \equiv \pi(s_i, r)$, it follows that:

$$pc_i \models \text{MERGE-VALUES}\left([s_i]_{i=1}^{k}, [pc_i]_{i=1}^{k}, m\right) = \text{MEDGE-VALUES-OPT}\left(\{s_i\}_{i=1}^{k}, r, m\right)$$

and from that (b) follows trivially.

3.4 Limitations

The size bound of a symbolic-size object, i.e., the maximal 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 object, it 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 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 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 solver queries.

The approaches presented in Section 3.2 and Section 3.3 have the known limitations of state merging: In practice, the number of states that can be merged while keeping the analysis efficient is limited, as complex representations affect both memory usage and constraint solving.

4 IMPLEMENTATION

We implemented our symbolic-size model on top of KLEE [14], a *state-of-the-art* symbolic executor operating on LLVM bitcode [27]. Our extension of KLEE is configured with LLVM 7.0.0 and STP 2.3.3 [19]. 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 3.1), 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 certain threshold, as applying it in such cases leads to high memory usage and poor performance of the solver. We disable state merging in loops which contain function calls, as the number of explored states in such cases is typically too high.

5 EVALUATION

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

5.1 Experimental Setup

Vanilla KLEE concretizes symbolic size expressions using the solver, which means that the user has little control over the resulting concretized size value. Throughout experimentation, we observed that such concretization strategy typically results in small size values (i.e., 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 maximal feasible value with respect to 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 object are considered, resulting in a complete exploration with respect to the specified capacity. We compare between several range modes: Under the concrete-size model, we use the eager forking mode (ForkEager) which forks at allocation time for each feasible value of the symbolic size expression. Under the symbolic-size model, we use the lazy forking mode (ForkLazy) described in Section 3.1, and the two merging modes (SM and SMOpt) described in Section 3.2 and Section 3.3, respectively.

We run 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 [2] and the associated artifact [3] available as open-source.

5.2 API Testing

The benchmarks used in this experiment are: *libtasn1* [5] v4.16.0 (15K SLOC), *libpng* [7], v1.6.37 (56K SLOC), and *libosip* [6] 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 API's that depend on variable-size objects.

In each benchmark we focused on API's whose input can be modeled using symbolic-size 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 78 API's across the different benchmarks: 17 API's from the decoding and encoding modules in *libtasn1*, 13 API's from the *pngread* and *pngwrite* modules in *libpng*, and 48 API's from the *osipparser2* module in *libosip*.

For each API, we run KLEE in the five modes (Base, ForkEager, ForkLazy, SM, and SMOpt) using the deterministic DFS search heuristic, with a one hour time limit and a 4GB memory limit. In each run we check the following metrics: analysis time, number of solver queries, line coverage computed with GCov [3], and number of explored paths.

5.2.1 Empirical Validation. In API's 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

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

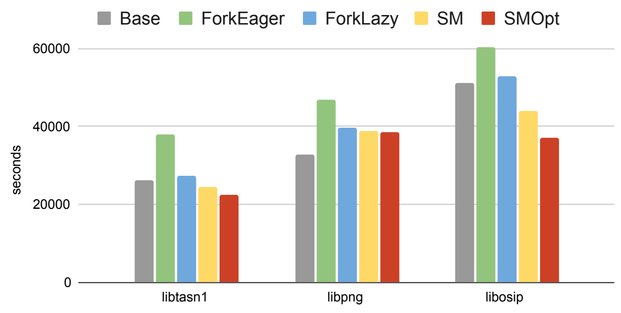

Figure 5: Total analysis time.

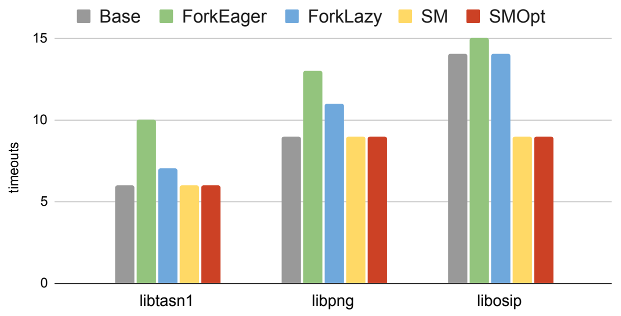

Figure 6: Number of timeouts.

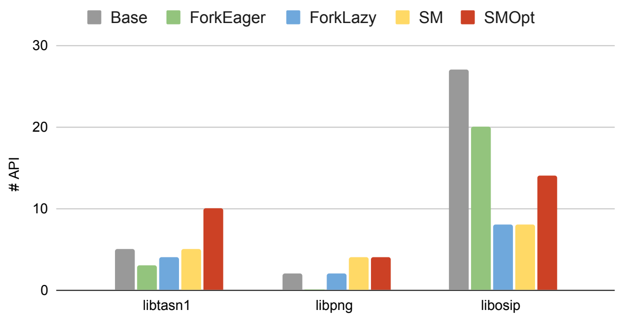

Figure 7: Analysis time scoreboard.

based comparison meaningless. As the optimizations described in Section 3.3 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.

5.2.2 Analysis Time. The Base mode uses concretization to handle symbolic-size allocations, therefore it can't 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 to ForkEager and ForkLazy. The SM and SMOpt modes use state merging, which might result in less paths compared to the other modes and faster analysis even compared to Base.

Figure 5 shows for each benchmark and mode the total time required to analyze all the API's. The Base mode was the fastest in

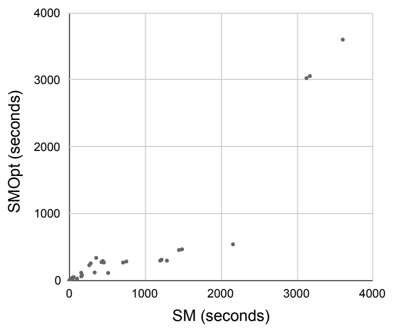

Figure 8: Analysis times of SM vs SMOpt (in *seconds*).

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 that takes place when symbolic-size objects are allocated.

As we analyze each API with a timeout of one hour, we also examine the cases which resulted in a timeout. Figure 6 shows for each benchmark and mode the number of API's 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 55 API's in which the analysis completed before the timeout at least in one of the modes. Figure 7 shows for each benchmark and mode the number of API's 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*, ForkEager and SMOpt had relatively high scores as well.

A more detailed comparison between the two merging modes is given in the scatter plot from Figure 8, where the x-axis and y-axis represent the analysis times for SM and SMOpt, respectively. Across all the API's, the speedup of SMOpt relatively to SM varies between 0.9×-4.4×, and the average is 1.6×. The SM mode was faster than the SMOpt mode only in one case, where the difference was less than 5 seconds.

5.2.3 Solver Queries. In addition to comparing the analysis times, we also compare the number of solver queries generated by each of the modes. Here, we report the number of queries that actually reached the 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 39 API's 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 3130 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 with respect to SMOpt was 17% in Base, 70% in ForkLazy, and 178% in ForkEager. When comparing between the two merging modes SM and SMOpt,

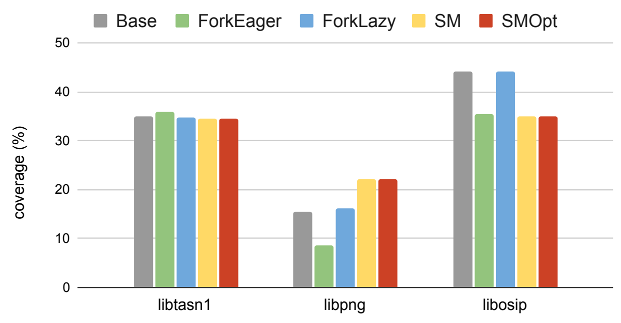

Figure 9: Total line coverage.

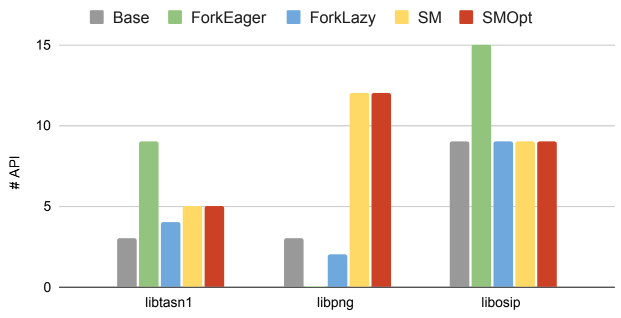

Figure 10: Line coverage scoreboard.

the number of queries is roughly the same. The slight differences between these modes originates from the different representation of the merged states, which affects the various heuristics used in KLEE's solver chain.

5.2.4 Coverage. Figure 9 shows for each benchmark and mode the total coverage generated by the test cases of all the API's. 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 26% compared to ForkLazy and the merging modes, respectively.

We now discuss in more detail the results of 39 API's in which at least one of the modes had a timeout. Figure 10 shows for each benchmark and mode the number of API's 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 the same scores.

In 24 out of the 39 API's mentioned above, SM and SMOpt had a timeout and achieved the same coverage. To further evaluate the results in these cases with those two modes, we use an additional evaluation metric: *path coverage*. Note that comparing between these two modes using this metric makes sense: The only difference between the merging modes is the representation of merged states, therefore the exploration order of the search space remains identical.

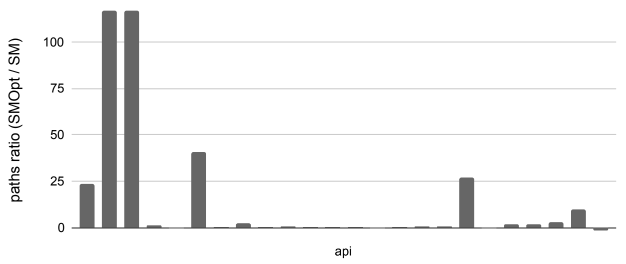

Figure 11: Increase in path coverage (%) of SMOpt vs. SM in cases where line coverage is identical.

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 11 shows the increase in the number of explored paths with SMOpt relatively to SM. The increase in path coverage is 14% on average and varies between -1% and 116%. In the 5 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.

In the 39 API's in which all the modes reached full exploration, the Base mode achieved the same coverage as the other range modes in all the cases except for two cases in *libtasn1*.

5.2.5 Merging Complexity. The main bottleneck of state merging comes from *ite* expressions and *disjunctive* constraints introduced by the merging, which propagate to the queries making constraint solving harder. We provide an additional comparison between the two merging modes (SM and SMOpt) based on the representation complexity of the states, i.e., the complexity of the constraints and the memory values resulting from the merging. In order to have a meaningful comparison, we compare only the results of 54 API's in which both of the modes reached full exploration.

Figure 12 shows for each API the ratio between the total size of all the merged constraints between SM and SMOpt[4] . The size of the constraints with SMOpt is never greater than in SM, and the average reduction is 886× (60× without *asn1_decode_simple_ber*'s 44657× outlier). Figure 13 shows for each API the ratio between the total size of all the merged values, i.e., variables and heap memory objects, between SM and SMOpt. Here again, we can observe a clear advantage for the SMOpt mode, where the average reduction with SMOpt compared to SM is 1420× (174× without *asn1_decode_simple_ber*'s 67479× outlier).

5.2.6 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 merging worked better, occurs when we have multiple independent operations on different inputs. On the other side, a scenario in which merging worked worse, occurs when we have multiple subsequent operations on the same input

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

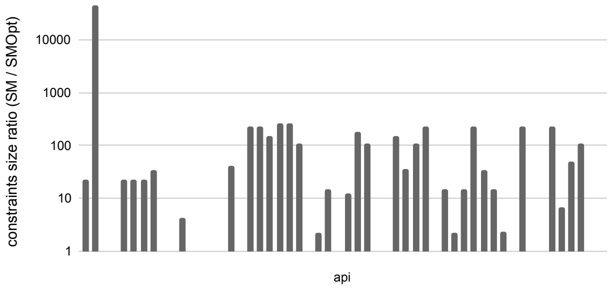

Figure 12: Decrease in constraints complexity.

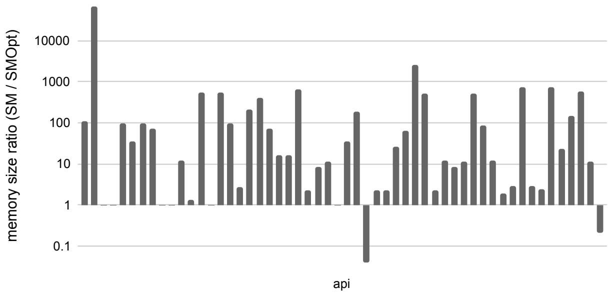

Figure 13: Decrease in memory complexity.

where each operation depends on the previous ones. Two API's that demonstrate these two scenarios are *sdp_messsage_to_str* and *osip_message_parse*. In the first case, we receive a struct describing a *SDP* message and translate it to a string representation. This struct contains several fields which are strings as well, and the translation is performed on each of them independently. In this case, SMOpt for example, completes the analysis in 39 seconds, while Base runs for 166 seconds, ForkLazy for 393 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 describing the parsed message. Here, we have a chain of *strchr* invocations on the 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* starts with a more complex state representation than the previous one. In this case, Base and ForkEager completes the analysis in 2 seconds, ForkLazy in 21 seconds, while in SM and SMOpt it takes 454 and 271 seconds, respectively.

5.2.7 Found Bugs. Throughout our experiments, we found 5 bugs in two of our benchmarks: *libosip* and *libtasn1*. In *libosip*, we found 3 *out-of-bound read* bugs and one *integer underflow* bug, all of which were triggered by symbolic-size objects, strings in this case. Table 1 shows the API's 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* [1, 2]. Note that the Base mode misses some of these bugs when the capacity is too high or too low, due to its single-size out-of-bound reasoning. In *libtasn1*, we found another *out-of-bound read* bug in the `ETYPE_OK` macro,

Table 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	`''`	$= 1$

which is used in several API's in the library. The bug happens due to an incorrect range-check of an array index, which can be triggered with a specific *element type* value. In this case, the bug was not directly triggered by a symbolic-size object, although certain size values of some API parameters make this bug unreachable.

5.3 Whole-Program Testing

As a benchmark for whole-program testing, we chose 99 programs from *GNU Coreutils* [4]. 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 object with a user-specified size. Symbolic-size allocations are not common in *GNU Coreutils*, so in order to have a more insightful evaluation in our context, we model those inputs using symbolic-size 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 5 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 3 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. Note that 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 3 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, where SMOpt had a slight advantage over SM.

5.4 Discussion

We evaluated several approaches that consider a range of object sizes: 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 *GNU 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 operates on multiple symbolic-size 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.

6 RELATED WORK

In the *segment-offset-plane* memory model [41], a memory object has its own unique address space, i.e., a segment, and an address expression is represented as a pair consisting of a segment identifier and an offset (within that segment). This model supports supposedly unbounded symbolic-size allocations, using a the two-dimensional address space where the non-overlapping property (mentioned in Section 2) is naturally supported. However, this model explicitly encodes each read and write operation, so the analysis will not scale with large enough objects due to the expected high memory consumption. Therefore we believe that the advantage of this model in supporting unbounded allocations is only theoretical. Moreover, to support fat pointers, this model encodes expressions using a more complex language, which may incur additional overhead. Šimáček [39] adopt the memory model proposed by [41] to support symbolic-size allocations with KLEE, thus inheriting the limitations discussed above. These two works don't address the problem of additional forking introduced by symbolic-size expressions, and particularly, that of symbolic-size dependent loops.

The segmented memory model [24] is an approach for handling symbolic pointers that have multiple resolutions: The memory is partitioned into segments using static pointer analysis such that every pointer is guaranteed to refer to at most one segment, thus avoiding any forks when symbolic pointers are dereferenced. Our symbolic-size model can be easily integrated with such model since every allocated object has a finite capacity. Technically, the only change required is to call the allocation function with the capacity of the allocated object instead of its size (Algorithm 3 from [24]).

Sinha [40] simplifies *ite* expressions using rewrite rules, which are not expressive enough to achieve the effect of the optimizations discussed in Section 3.3. 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 approach could be used in our context as well, but it can be applied only when there exists a *linear* dependency between the symbolic variables, which is not always the case.

Path merging [22, 26, 36] has been used in the past to scale symbolic execution. Kuznetsov et al. [26] propose dynamic state merging with a *query count estimation* heuristic that decides when merging should be applied, and MultiSE [36] proposes an alternative approach for state representation. Veritesting [9, 37] is another path merging technique which statically summarizes code regions. JavaRanger [37] extends veritesting for Java programs to support dynamically dispatched methods, by using the runtime information available during the analysis. The works mentioned above have no support for symbolic-size objects, and statically summarizing code regions that contain loops is challenging, even with the aid of runtime information.

Loop-extended symbolic execution [35] is a technique that can be used to summarize 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. Our approach is more dynamic in nature and does not depend on static analyses. Godefroid et al. [21] 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 approach has no restrictions on the loop variables and the number of iterations. Both of these works provide summaries only for scalar variables, so clearly does not support symbolic-size memory objects.

Anand et al. [8] 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. [18] model symbolic-size arrays similarly to [8], but place a bound on the number of initialized array cells instead of using an abstraction-based pruning. These works [8, 18] don't handle symbolic size *allocations* (e.g., using *malloc*) that occur directly in the program. In contrast, UC-KLEE [34] supports symbolic-size arrays in its lazy initializing algorithm as well as allocation of symbolic-size objects [33]. Essentially, it uses an approach similar to the model described in Section 3.1, where every symbolic-size object has a user-specified upper bound on its size. None of the aforementioned works investigate the tradeoffs between different approaches for enabling symbolic-size allocations, or explore strategies for managing the symbolic state space resulting from size-dependent loops.

7 CONCLUSION AND FUTURE WORK

We proposed a bounded symbolic-size model which addresses the problem of variable-size inputs in symbolic execution. We evaluated our model in terms of performance and test case generation, and found previously unknown bugs.

Finding more efficient state representations, for example, by using different logic fragments, can further improve the merging approach. To cope with the incompleteness of our modeling, one can try to adapt the capacity using static or dynamic techniques. Efficiently handling unbounded objects remains an open problem.

ACKNOWLEDGEMENTS

This research was funded by the Israel Science Foundation (ISF) grant No. 1996/18, 243/19 and 2740/19, and by the United States-Israel Binational Science Foundation (BSF) Grant No. 2018675.

REFERENCES

[1] 2021. https://git.savannah.gnu.org/cgit/osip.git/commit/?id=ef6497.

[2] 2021. https://git.savannah.gnu.org/cgit/osip.git/commit/?id=2f0380.

[3] 2021. GCov. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[4] 2021. GNU Coreutils. https://www.gnu.org/software/coreutils/.

[5] 2021. GNU libtasn1. https://www.gnu.org/software/libtasn1/.

[6] 2021. GNU oSIP. https://www.gnu.org/software/osip/.

[7] 2021. libpng. http://www.libpng.org/pub/png/libpng.html.

[8] Saswat Anand, Corina S Păsăreanu, and Willem Visser. 2006. Symbolic execution with abstract subsumption checking. In *International SPIN Workshop on Model Checking of Software*. Springer, 163–181. https://doi.org/10.1007/11691617_10

[9] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing Symbolic Execution with Veritesting. In *Proc. of the 36th International Conference on Software Engineering (ICSE'14)* (Hyderabad, India). https://doi.org/10.1145/2568225.2568293

[10] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. 1998. A Decision Procedure for Bit-Vector Arithmetic. In *Proc. of the 35th Design Automation Conference (DAC'98)* (San Francisco, CA, USA). https://doi.org/10.1145/277044.277186

[11] Aaron R Bradley, Zohar Manna, and Henny B Sipma. 2006. What's decidable about arrays?. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 427–442. https://doi.org/10.1007/11609773_28

[12] Tegan Brennan, Seemanta Saha, Tevfik Bultan, and Corina S Păsăreanu. 2018. Symbolic path cost analysis for side-channel detection. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 27–37. https://doi.org/10.1145/3213846.3213867

[13] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. 2019. CaSym: Cache aware symbolic execution for side channel detection and mitigation. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 505–521. https://doi.org/10.1109/SP.2019.00022

[14] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. 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)* (San Diego, CA, USA).

[15] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. 2006. EXE: Automatically Generating Inputs of Death. In *Proc. of the 13th ACM Conference on Computer and Communications Security (CCS'06)* (Alexandria, VA, USA). https://doi.org/10.1145/1455518.1455522

[16] Peter Collingbourne, Cristian Cadar, and Paul H.J. Kelly. 2011. Symbolic Cross-checking of Floating-Point and SIMD Code. In *Proc. of the 6th European Conference on Computer Systems (EuroSys'11)* (Salzburg, Austria). https://doi.org/10.1145/1966445.1966475

[17] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: introduction and applications. *Commun. ACM* 54, 9 (2011), 69–77. https://doi.org/10.1145/1995376.1995394

[18] Xianghua Deng, Jooyong Lee, et al. 2006. 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)*. IEEE, 157–166. https://doi.org/10.1109/ASE.2006.26

[19] Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-vectors and Arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification* (Berlin, Germany) *(CAV'07)*. Springer-Verlag, Berlin, Heidelberg, 519–531. http://dl.acm.org/citation.cfm?id=1770351.1770421

[20] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proc. of the 15th Network and Distributed System Security Symposium (NDSS'08)* (San Diego, CA, USA).

[21] Patrice Godefroid and Daniel Luchaup. 2011. Automatic Partial Loop Summarization in Dynamic Test Generation. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'11)* (Toronto, Canada). https://doi.org/10.1145/2001420.2001424

[22] Trevor Hansen, Peter Schachte, and Harald Søndergaard. 2009. State Joining and Splitting for the Symbolic Execution of Binaries. In *Proc. of the 2009 Runtime Verification (RV'09)* (Grenoble, France). https://doi.org/10.1007/978-3-642-04694-0_6

[23] Wei Jin and Alessandro Orso. 2012. BugRedux: Reproducing Field Failures for In-house Debugging. In *Proc. of the 34th International Conference on Software Engineering (ICSE'12)* (Zurich, Switzerland). https://doi.org/10.1109/ICSE.2012.6227168

[24] Timotej Kapus and Cristian Cadar. 2019. 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* (Tallinn, Estonia) *(ESEC/FSE 2019)*. ACM, 774–784. https://doi.org/10.1145/3338906.3338936

[25] Timotej Kapus, Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Cristian Cadar. 2019. 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*. 874–888. https://doi.org/10.1145/3314221.3314610

[26] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient state merging in symbolic execution. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'12)* (Beijing, China). https://doi.org/10.1145/2345156.2254088

[27] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of the 2nd International Symposium on Code Generation and Optimization (CGO'04)* (Palo Alto, CA, USA). https://doi.org/10.1109/CGO.2004.1281665

[28] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*. ACM, 691–701. https://doi.org/10.1145/2884781.2884807

[29] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1186–1189. https://doi.org/10.1109/ASE.2019.00133

[30] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proc. of the 35th International Conference on Software Engineering (ICSE'13)* (San Francisco, CA, USA). https://doi.org/10.1109/ICSE.2013.6606623

[31] Corina S Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. 2016. Multi-run side-channel analysis using Symbolic Execution and Max-SMT. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. IEEE, 387–400.

[32] Corina S. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. 2013. 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)* (Palo Alto, CA, USA).

[33] David A. Ramos. 2015. *Under-constrained symbolic execution : correctness checking for real code.* Ph.D. Dissertation. Stanford University.

[34] David A. Ramos and Dawson Engler. 2015. Under-constrained Symbolic Execution: Correctness Checking for Real Code. In *Proc. of the 24th USENIX Security Symposium (USENIX Security'15)* (Washington, D.C., USA).

[35] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. 2009. Loop-extended Symbolic Execution on Binary Programs. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'09)* (Chicago, IL, USA). https://doi.org/10.1145/1572272.1572299

[36] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. 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. https://doi.org/10.1145/2786805.2786830 ACM SIGSOFT Distinguished Paper Award.

[37] Vaibhav Sharma, Soha Hussein, Michael W. Whalen, Stephen McCamant, and Willem Visser. 2020. 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* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 123–134. https://doi.org/10.1145/3368089.3409734

[38] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'16)* (San Jose, CA, USA). https://doi.org/10.1109/SP.2016.17

[39] M Šimáček. 2018. Symbolic-size memory allocation support for Klee. Master's thesis, Masaryk University, Faculty of Informatics, Brno, 2018. (2018).

[40] Nishant Sinha. 2008. Symbolic program analysis using term rewriting and generalization. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*. IEEE Press, 19. https://doi.org/10.1109/FMCAD.2008.ECP.23

[41] Marek Trtík and Jan Strejček. 2014. Symbolic Memory with Pointers. In *Automated Technology for Verification and Analysis (ATVA)*. https://doi.org/10.1007/978-3-319-11936-6_27