State Merging with Quantifiers in Symbolic Execution

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

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

Sharon Shoham
Tel Aviv University
Tel Aviv, Israel
sharon.shoham@cs.tau.ac.il

Vaibhav Sharma
University of Minnesota
Minneapolis, USA
vaibhav@umn.edu

ABSTRACT

We address the problem of constraint encoding explosion which hinders the applicability of state merging in symbolic execution. Specifically, our goal is to reduce the number of disjunctions and *if-then-else* expressions introduced during state merging. The main idea is to dynamically partition the symbolic states into merging groups according to a similar uniform structure detected in their path constraints, which allows to efficiently encode the merged path constraint and memory using quantifiers. To address the added complexity of solving quantified constraints, we propose a specialized solving procedure that reduces the solving time in many cases. Our evaluation shows that our approach can lead to significant performance gains.

CCS CONCEPTS

• **Software and its engineering**;

KEYWORDS

Symbolic Execution, State Merging

ACM Reference Format:
David Trabish, Noam Rinetzky, Sharon Shoham, and Vaibhav Sharma. 2023. 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.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616287

1 INTRODUCTION

Symbolic execution is a powerful 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 [19, 20, 42], bug finding [19, 31], debugging [34], automatic program repair [39, 40], cross checking [22, 35], and side-channel analysis [17, 18, 41]. 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 the symbolic input, an SMT solver [26] is used to determine the feasibility of each branch side, and the feasible paths are further explored while updating their path constraints with the corresponding constraints. Once the execution of a given path is completed, the solver provides a satisfying assignment for the corresponding path constraints, from which a concrete test case that replays that path can be generated.

A key remaining challenge in symbolic execution is path explosion [21]. State merging [33, 37] 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 results in a symbolic state where 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 states being merged is high. Consider, for example, the function `memspn` from Section 1 which is based on the implementation of `strspn` in *uClibc* [54].[1] `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 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 `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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3616287

[1] `strspn` receives null-terminated buffers, slightly complicating the presentation.

```
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 }
```

Figure 1: Motivating example.

preserving soundness and completeness w.r.t. standard symbolic execution. At a high level, our approach takes as an input the execution tree [36], 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 us to partition them into merging groups of 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 that leverages the particular structure of the generated quantified queries, and resort to the generic method only if our approach fails.

We implemented our approach on top of KLEE [19] and evaluated it on real-world benchmarks. Our experiments show that our approach can have significant performance gains compared to state merging and standard symbolic execution.

2 PRELIMINARIES

State Merging. A *symbolic state* s consists of (i) a *path constraint* $s.pc$, (ii) a *symbolic store* $s.mem$ that associates variables[2] V with symbolic expressions obtained from the symbolic inputs, (iii) and an *instruction counter* $s.ic$. Symbolic states are *merge-compatible* if they have the same instruction counter and contain the same variables in their stores.

Definition 2.1. 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:

$$s.ic \triangleq s_1.ic,\ s.pc \triangleq \bigvee_{i=1}^n s_i.pc,$$
$$s.mem \triangleq \lambda v \in V.\ merge_var(\{s_i\}_{i=1}^n, v)$$

where the merged value of a variable v is defined by:

$$merge_var(\{s_i\}_{i=1}^n, v) \triangleq ite(s_1.pc, s_1.mem(v), ite(\ldots, ite(s_{n-1}.pc, s_{n-1}.mem(v), s_n.mem(v))))$$

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

State merging is 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 [19] of the symbolic execution engine to continue the exploration.

Execution Trees. An *execution tree* [36] 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. For example, consider the symbolic execution of `memspn` (Section 1) with a symbolic buffer `s`, a symbolic size `n`, and `"a"`, where `n` is bounded by 3. The corresponding execution tree is depicted in Figure 2, where the symbolic condition associated with each node is depicted on the incoming edge of the node. The node n_1 corresponds to the initial symbolic state (i.e., $n_1.s.pc \triangleq n \leq 3$), the nodes n_2, n_6 n_{10}, and n_{14} correspond to paths where `s` is comprised of only `a` characters, and the nodes n_5, n_9, and n_{13} correspond to paths where `s` contains a non-`a` character. For now, ignore the color of the nodes.

Given an execution tree t with root r, we denote the sequence of nodes on the path from node n_1 to node n_k in t by $\pi_t(n_1, n_k)$ and write $\pi_t(n_k)$ when n_1 is the root r. Given a path $\pi_t(n_1, n_k) = [n_1, n_2, ..., n_k]$ in t, we define its *tree path condition* (*tpc*) and *tree path condition tail* ($\overline{tpc}$):

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

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. For example, in the execution tree depicted in Figure 2:

$$\pi(n_3, n_7) \triangleq [n_3, n_4, n_7]$$
$$tpc(n_3, n_7) \triangleq n > 0 \land s[0] = 97 \land n > 1$$
$$\overline{tpc}(n_3, n_7) \triangleq s[0] = 97 \land n > 1$$

An execution tree t with root r is *valid* if $n.s.pc = r.s.pc \land tpc(n)$ for every node n. Note that $r.s$ is not necessarily the initial symbolic state of the whole program, so $tpc(n)$ is a suffix of the path constraints. From now on, we assume that all trees are valid.

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. A formula is either an application of a predicate symbol to terms or obtained by applying boolean connectives or quantifiers to formulas. Let φ, φ' be formulas and m a model. We write $\varphi \equiv \varphi'$ to note that φ and φ' are semantically equivalent and $\varphi \doteq \varphi'$ to note that they are syntactically equal. We write $m \models \varphi$ to note that m is a model of φ. For a term t, we denote by $m(t)$ the value assigned by m to t, and we write $t_1 \equiv t_2$ to denote that $m(t_1) = m(t_2)$ in any model m. We use the standard theory of arrays [51] and write $a[e]$ as a shorthand for $select(a, e)$.

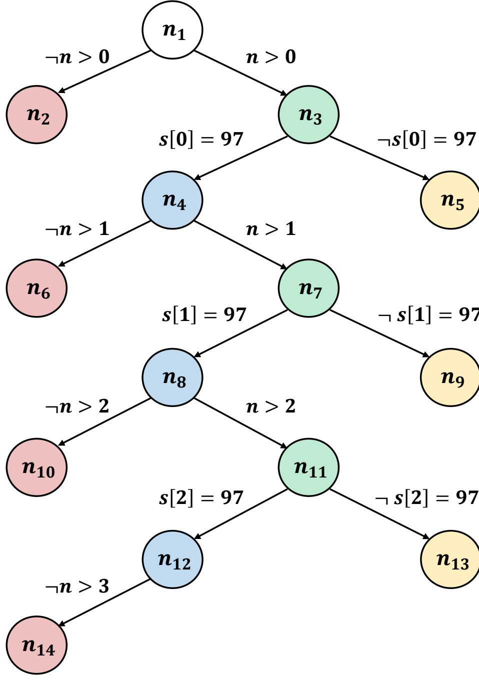


Figure 2: The execution tree of the loop from Section 1 when `chars` is set to "`a`". (Recall that the ASCII code of `a` is 97.)

3 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 states associated with the nodes n_5, n_9, and n_{13} from the execution tree in Figure 2, whose tree path conditions, i.e., , $tpc(n_5)$, $tpc(n_9)$, and $tpc(n_{13})$, are:

$$n > 0 \land \neg s[0] = 97$$
$$n > 0 \land s[0] = 97 \land n > 1 \land \neg s[1] = 97$$
$$n > 0 \land s[0] = 97 \land n > 1 \land s[1] = 97 \land n > 2 \land \neg s[2] = 97$$

The path constraint of the initial symbolic state ($n_1.s$) is $n \leq 3$, so applying standard state merging (Definition 2.1) on the symbolic states of the nodes above will result in a symbolic state whose path constraint is equivalent to:

$$n \leq 3 \land (tpc(n_5) \lor tpc(n_9) \lor tpc(n_{13}))$$

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 $n > _ \land s[_] = 97$ to encode that the size of the buffer (n) is big enough to contain k consecutive occurrences of `a` characters, and another formula $n > k \land \neg s[k] = 97$. This uniformity is exposed when rewriting each disjunct using universal quantifiers as follows:

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

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

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

The auxiliary variable allows us to achieve similar savings in the encoding of the 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 3.1). Then, for each partition, we attempt to synthesize the merged symbolic state using universal quantifiers (Sections 3.2 and 3.3), and resort to standard state merging if that fails.

3.1 Partitioning 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 3.1. 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:

Definition 3.2. The hash of a path $\pi(n_1, n_k) = [n_1, ..., n_k]$ in t is defined as follows:

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

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

Definition 3.3. 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 leaf nodes $\{n_j\}_{j=1}^{n}$ in t, and numbers $\{k_j\}_{j=1}^{n} \subseteq \mathbb{N}$, we say that $\{(n_j, k_j)\}_{j=1}^{n}$

[3] Note that $(k = 0 \lor k = 1 \lor k = 2)$ can be rewritten as $0 \leq k \leq 2$.

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

Table 1: A regular partitioning of the leaf nodes of the execution tree in Figure 2, and the resulting merged states.

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

match the regular pattern $(\omega_1, \omega_2, \omega_3)$ if for every $j = 1, ..., n$: $h(\pi(n_j)) = \omega_1 \omega_2^{k_j} \omega_3$.

Definition 3.4. 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 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 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:*

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

A (possible) regular partitioning of the leaf nodes in Figure 2 is given in Table 1, which shows in the two leftmost columns the regular patterns and their corresponding regular partitions.

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

3.2 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 3.5. 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 $j = 1, ..., n$:

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

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

Definition 3.6. Let $\{n_j\}_{j=1}^n$ be a set of leaf nodes in t such that $\{n_j.s\}_{j=1}^n$ are merge-compatible and $\{(n_j, k_j)\}_{j=1}^n$ 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, $s.pc$, is:

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

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 $t[k_j/x] \doteq n_j.s.mem(v)$ for every $j = 1, \ldots, n$, then the value of v is encoded as $s.mem(v) \triangleq t[k/x]$. Otherwise, $s.mem(v) \triangleq merge_var(\{n_j.s\}_{j=1}^n, v)$ (Definition 2.1).

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

Theorem 3.7. *Under the premises of Definition 3.6, 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 2.1). The following holds for any model m:*

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

Example 2. *Consider the regular partition* $\{n_5, n_9, n_{13}\}$ *shown in the first row of Table 1. The formula pattern* $(true, n > x - 1 \land s[x - 1] = 97, n > x \land \neg s[x] = 97)$ *is matched by* $(\{(n_5, 0), (n_9, 1), (n_{13}, 2)\}$. *The merged symbolic state induced by that formula pattern is shown in the rightmost column in Table 1 (pc and mem). 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.

3.3 Synthesizing Formula Patterns

So far, we have yet to discuss how formula patterns are obtained. We now describe an approach that attempts to synthesize a formula pattern given a regular pattern and its associated regular partition. As explained in Section 3.2, this enables us to perform state merging with quantifiers.

Our hash function h, which we assume to be valid for t (Definition 3.1), has the following useful property:

Lemma 3.8. *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.

Accordingly, we define:

Definition 3.9. 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 and n_2 in t. Then we define:

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

which gives us the tree path condition tails associated with the paths $\pi(n_1)$ and $\pi(n_1, n_2)$, respectively. (Note that Lemma 3.8 ensures that n_1 and n_2 are uniquely determined by ω_1 and ω_2.)

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

Lemma 3.10. *Suppose that $\{(n_j, k_j)\}_{j=1}^{n}$ match the regular pattern $(\omega_1, \omega_2, \omega_3)$. Let $(\varphi_1, \varphi_2(x), \varphi_3(x))$ be 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 $(\varphi_1, \varphi_2(x), \varphi_3(x))$.

Based on Lemma 3.10, 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 lemma.) Each synthesis task has the form:

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

where (i) $\varphi(x)$ is the formula to be synthesized (i.e., φ_2 or φ_3), (ii) 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), (iii) $\{\psi_\ell\}_{\ell=1}^{p}$ are formulas (obtained from the extracted path constraints), and (iv) $\{d_\ell\}_{\ell=1}^{p}$ 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 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:*

$\varphi_1 \doteq true$	$extract(W)$
$\varphi_2[1/x] \doteq n > 0 \land s[0] = 97$	$extract(W, WGB)$
$\varphi_2[2/x] \doteq n > 1 \land s[1] = 97$	$extract(WGB, WGBGB)$
$\varphi_3[0/x] \doteq n > 0 \land \neg s[0] = 97$	$extract(W, WGY)$
$\varphi_3[1/x] \doteq n > 1 \land \neg s[1] = 97$	$extract(WGB, WGBGY)$
$\varphi_3[2/x] \doteq n > 2 \land \neg s[2] = 97$	$extract(WGBGB, WGBGBGY)$

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

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

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

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

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

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

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

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:

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

Such terms are synthesized similarly to formula patterns.

For each regular partition shown in Table 1, we automatically synthesize the formula pattern and the induced merged symbolic state using the technique above.

The proofs for Theorem 3.7 and the other lemmas are given in [53, Section A].

4 INCREMENTAL STATE MERGING

When symbolically analyzing code fragments that contain disjunctive conditions, the number of generated 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 internal nodes during the construction of the tree. This allows to compress the tree as it is constructed. Once the construction of the 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 internal 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* [11]. We additionally require that n_1 is unreachable from

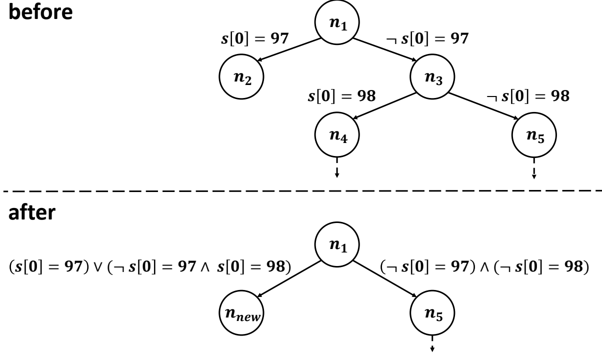


Figure 3: Execution tree transformation when `memspn` is called with `chars` set to `"ab"`.

n_2 to avoid infinite sequences of merges. If such a node n_2 is found, we replace n_1 and n_2 (and their subtrees) with a single merged node n_{new} that is added as a child of their lowest common ancestor, n_{lca}. We fix $n_{new}.c \triangleq \overline{tpc}(n_{lca}, n_1) \lor \overline{tpc}(n_{lca}, n_2)$ and $n_{new}.s$ is the merged 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 $n.c \land p.c$. As we merge internal 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 4. *Consider again the function* `memspn` *from Section 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 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 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 appropriately update the condition of n_5. This results in the execution tree shown in the lower part of Figure 3. After applying similar steps in the subsequent iterations of the loop, the final execution tree is similar to the one from Figure 2, and can be obtained from it by replacing $s[i] = 97$ and $\neg s[i] = 97$ with $s[i] = 97 \lor (\neg s[i] = 97 \land s[i] = 98)$ and $\neg s[i] = 97 \land \neg s[i] = 98$, respectively (for $i = 0, 1, 2$). Now, pattern-based state merging can be applied similarly to the example given in Section 3.*

The incremental state merging approach uses a standard liveness analysis [11] 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.

5 SOLVING QUANTIFIED QUERIES

In general, the quantified queries generated by our approach (Section 3) can be solved using an SMT solver that supports quantified formulas, e.g., Z3 [24]. In practice, however, we observed that the generic method employed by Z3[5] 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 that leverages the particular structure of the generated quantified formulas, and resort to the generic method if our approach fails.

Our solving procedure assumes a closed formula $\varphi = \bigwedge c$ where each clause c is either a quantifier-free formula or a universal formula of the form $\forall i.\ 1 \le i \le k \to \psi$ where ψ is a quantifier-free formula with a free variable i. Our solving procedure works in four stages:[6]

(1) Quantifier stripping. We weaken φ into a quantifier-free formula φ_{QF} by replacing quantified clauses with implied quantifier-free clauses. Technically, each quantified clause $\forall i.\ 1 \le i \le k \to \psi$ in φ is replaced with the conjunction of the following two quantifier-free formulas[7]:

$$(1)\ k \ge 1 \to \psi[1/i] \quad (2)\ \bigwedge \{\neg(1 \le t \le k) \mid (\neg\psi[t/i]) \in \varphi\}$$

Intuitively, the former provides a quantifier-free clause which partially preserves the properties imposed by the quantified clause, and the latter reduces the chances that the SMT solver computes a model of φ_{QF} that does not satisfy φ: if $1 \le t \le k$ then $\forall i.\ 1 \le i \le k \to \psi$ demands that $\psi[t/i]$ holds in any model of φ. If the SMT solver fails to find a model for φ_{QF}, then φ is also unsatisfiable. If a model was found, we check whether it is also a model of φ.

Example 5. *Consider the following query, a simplification of a representative query from our experiments:*

$$\varphi \triangleq (s[n] = 0) \land (1 \le k \le 10) \land (s[k-1] = 8) \land$$
$$(\forall i.\ 1 \le i \le k \to s[i-1] \ne 0)$$

Note that (a) the instantiation of the quantified formula using $i = 1$ results in $k \ge 1 \to s[0] \ne 0$, and (b) $s[n] = 0$ is obtained by substituting $\neg(s[i-1] \ne 0)[n+1/i]$. Thus, the weakened query obtained by quantifier stripping is given by:

$$\varphi_{QF} \triangleq (s[n] = 0) \land (1 \le k \le 10) \land (s[k-1] = 8) \land$$
$$(k \ge 1 \to s[0] \ne 0) \land \neg(1 \le n+1 \le k)$$

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, it is not a model of φ.

(2) Assignment Duplication. If m is not a model of φ, we modify m into a model m_d which assigns to every array cell accessed by a quantified clause a value v of a cell in that array that was explicitly constrained by φ_{QF}. Technically, for every array a accessed with an offset that depends on the quantified variable i we do the following: (1) pick an accessed offset o of a in ψ such that o depends on i, (2) evaluate the value of $(a[o])[1/i]$ in m, namely v, and (3) compute the concrete offsets obtained by evaluating $o[j/i]$ in m (for $2 \le j \le$

[5] CVC5 [15] and Yices [28] failed to solve most of our queries.
[6] For the interested reader, a complete pseudo code of the solving procedure is given in [53, Section B].
[7] We write $c \in \varphi$ to note that c is one of the clauses of φ.

$m(k)$) and modify m such that the values of a at these offsets are set to v. Recall that the accessed cells of a in $\psi[1/i]$ were explicitly constrained in φ_{QF}, so v is a good candidate to fill in all the other cells of a constrained in φ. However, this duplication is rather naive and might result in a model that does not even satisfy φ_{QF}.

Example 6. *Continuing Example 5, 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 *(specifically, the clause* $s[k-1] = 8$ *is violated).*

(3) Model Repair. If m_d is not a model of φ, we 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}. For every quantified clause $\forall i.\ 1 \leq i \leq k \rightarrow \psi$, we collect all the accesses $a[o]$ where o depends on i. For each such access and for each $2 \leq j \leq m(k)$, we compute the concrete offset obtained by evaluating $o[j/i]$ in m_d and strengthen φ_{QF} with the instantiation $\psi[j/i]$ *if* that offset appears in the concrete offsets of a violated quantifier-free clause (or a violated instantiation). Rather than computing from scratch a new model for the strengthened query, we fix the values of all the array cells (and variables) according to their interpretation in m_d except for the arrays that are accessed with i, those for which a new interpretation is sought. If the resulting query has a model, we apply assignment duplication on it. This time, to avoid overwriting, the duplication is not applied to the offsets involved in violations.

Example 7. *Continuing Example 6, 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 and its possible model are:*

$$\varphi_{QF} \land (s[6] \neq 0) \land (n = 7) \land (k = 7)$$
$$\{n \mapsto 7, k \mapsto 7, s \mapsto [1,0,0,0,0,0,8,0]\}$$

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*:*

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

(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.

6 IMPLEMENTATION

We implemented our state merging approach on top of the KLEE [19] symbolic execution engine, configured with LLVM 7.0.0 [38]. Our approach generates quantified queries over arrays and bit vectors, so we use Z3 [25] (version 4.8.17) as the underlying SMT solver. We extended KLEE's expression language to support quantified formulas, and modified some parts of the solver chain accordingly. We implemented our solving procedure (Section 5) as an additional component in the solver chain. To implement the *hash* function used by the pattern-based state merging approach (Section 3), we relied on the expression hashing utility of KLEE and modified it by assigning a pre-defined hash value to all constants. To extract the regular patterns from the execution trees, we used a basic regular expression matching algorithm. If our hash function is not valid for a given generated execution tree (Definition 3.1), or the number of extracted regular patterns in that tree exceeds a user-specified threshold, then we fallback to standard state merging (Definition 2.1). Our implementation is available at [1].

7 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* memory model [52]. This model supports bounded symbolic-size objects, i.e., objects whose size can have a range of values, limited by a user-specified *capacity* bound.[8] It was observed in [52] that loops operating on symbolic-size 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 in [52] avoids the need for manual annotations. We emphasize, however, that our technique is independent of the symbolic-size memory model itself (see Section 7.7). That said, the symbolic-size memory 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 3 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 4) and the solving procedure (Section 5) are enabled. The *CFG* mode is the state merging approach discussed above (*SMOpt* mode from [52]), 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 [29] (disjunctions and *ite* expressions). The *BASE* mode is the forking approach used in vanilla KLEE [19].

The following research questions guide our evaluation:

(RQ1) Does *PAT* improve standard state merging (*CFG*)?
(RQ2) Does *PAT* improve standard symbolic execution (*BASE*)?
(RQ3) Do all components contribute to the performance of *PAT*?

7.1 Benchmarks

The benchmarks used in our evaluation are listed in Table 2. These benchmarks were chosen as they are challenging for symbolic execution and provide numerous opportunities for applying state merging. In each benchmark, we analyzed a set of subjects (APIs and whole programs) whose inputs (parameters, command-line

[8]This is in contrast to the standard *concrete-size* model where every object has a concrete size.

arguments, etc.) can be modeled using symbolic-size objects, i.e., arrays and strings. In *libosip* [7], *libtasn1* [6], and *libpng* [10], the test drivers for the APIs were taken from [52].[9] In *wget* [8], a library for retrieving files using widely used internet protocols (HTTP, etc.), 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* [12] (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* [9], a library for decoding and encoding JSON objects, we constructed test drivers for APIs that manipulate string objects. In *busybox* [4], 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.

7.2 Setup

We run every mode under the symbolic-size memory model [52] 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 2.[10]

In every experiment, we use the following metrics to compare between the modes: analysis time and line coverage computed with GCov [5]. 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 the two modes, we measure the relative speedup and the relative increase in coverage for each subject. 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. In [53, Section C.1.1], we additionally report the average (and median) when computed over all the subjects.

We ran our experiments on several machines (Intel i7-6700 @ 3.40GHz with 32GB RAM) with Ubuntu 20.04.

[9] We noticed that some of the APIs from *libosip* that were used in [52] are similar, i.e., different APIs with the same internal functionality. The analysis of such APIs leads to the same results, therefore, we excluded them from the evaluation to avoid redundancy.

[10] In *libosip*, *libtasn1*, and *libpng*, the capacity settings were set similarly to the experiments from [52].

Table 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

7.3 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 3 and Figure 4.

Analysis Time. Column *Speedup* in Table 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 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*, *PAT* was slower only in one API. In this 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 5) 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 3 shows the difference between *PAT* and *CFG* in terms of the total time required to analyze all the subjects. Note that the time difference is interpreted as zero in subjects where both modes are timed out. In *libosip*, *wget*, *apr*, and *busybox*, *PAT* achieved a considerable reduction of roughly 8, 4, 1, and 3 hours, respectively. In *json-c*, *PAT* achieved a reduction of roughly 20 minutes, and in *libtasn1* and *libpng*, the time difference was minor. Figure 4b breaks down the time difference between *PAT* and *CFG* per subject.

Coverage. Column *Coverage* in Table 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. Again, 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 cases, as it did not find regular (and formula) patterns. Therefore, the results were similar to those of *CFG*, and coverage was not improved. 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 this 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, resulting in identical explorations.) Column *Diff.* in Table 3 shows the difference between *PAT* and *CFG* in terms of the total number of covered lines across all the subjects. Again, note that there is no difference in coverage in subjects where both modes terminated. 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 4c breaks down the coverage improvement of *PAT* over *CFG* per subject.

Scaling. The main obstacle in applying state merging originates from the introduction of disjunctive constraints and *ite* expressions, especially when the number of 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 objects. When the capacity is increased, the size of the symbolic-size objects is potentially increased as well. This typically leads to additional forks, for example, in loops that operate on symbolic-size 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 states. In this experiment, we run each API in each of the state merging modes (*PAT* and *CFG*) under several different capacity settings. The results are shown in Table 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, the coverage improvement was less significant under the highest capacity settings (100 and 200) 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 states.

RQ1 Answer: *PAT outperforms CFG in many cases and scales better in executing complex state merging operations.*

7.4 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 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 nine 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 and the complex constraints that were introduced during the state merging. Regarding 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 modes covered most 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.

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.

For space reasons, the breakdown of the improvement of *PAT* over *BASE* per subject is shown in [53, Section C.1.2].

RQ2 Answer: *PAT outperforms BASE in many cases, however, the known tradeoff between state merging and forking remains.*

7.5 Results: Component Breakdown

Now, we evaluate the significance of the components used in our pattern-based state merging approach (i.e., *PAT*).

7.5.1 Solving Procedure. To evaluate our solving procedure (Section 5), 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.

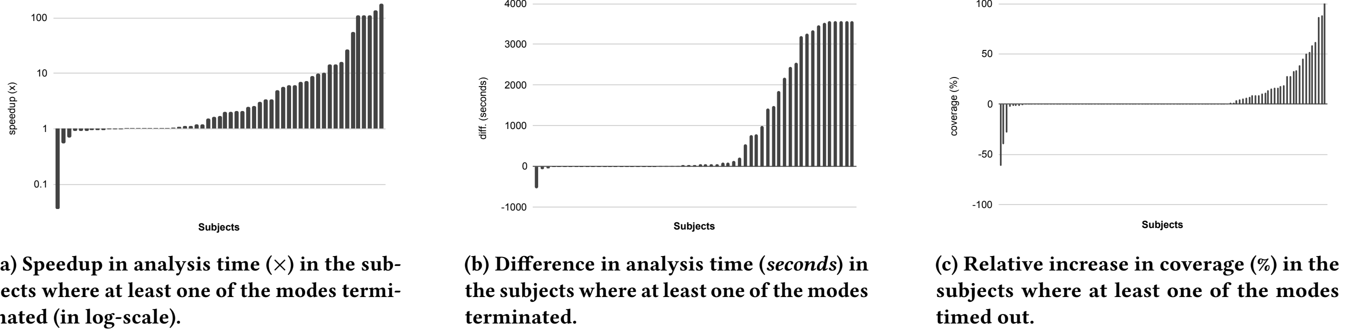
To evaluate the impact of the solving procedure, we show in Table 6 its effect on analysis time and coverage in the relevant subsets. Here, the two modes have the same exploration order, so we use the path coverage metric as well. In *libosip*, *wget*, *apr*, *json-c*, and *busybox*, our solving procedure generally leads to lower analysis times and higher (line or path) coverage. The results were mostly similar in *libtasn1* and *libpng* 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%.

7.5.2 Incremental State Merging. To evaluate the incremental state merging approach (Section 4), 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 7.

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 higher line coverage. In *wget*, there were four APIs where incremental state merging could be applied, and in two of these cases, the coverage

Table 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



(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 4: Breakdown of the improvement of *PAT* over *CFG* per subject.

Table 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

Table 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

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 in the execution tree.

Table 6: 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

Table 7: 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

RQ3 Answer: *All the components contribute.*

7.6 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* [54]. We reported the bugs, which were confirmed and fixed by the official maintainers [2]. We note that these bugs were detected by *PAT* and *BASE*, but were not found by *CFG* due to a timeout.

7.7 Threats to Validity

First, our implementation may have bugs. To validate its correctness, we performed a separate experiment where each subject was 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 3.7. 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, we chose a diverse set of real-world benchmarks used in prior work [35, 47, 52]. 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 [52]. 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 memory model. In this experiment, we set the concrete sizes of the input objects according to the capacity configuration in Table 2, and apply state merging in loops whose conditions depend on these sizes, as we do in our original experiments. The results, shown in [53, Section C.3], 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 using the default search heuristic in KLEE. The results, shown in [53, Section C.4], are comparable.

7.8 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 synthesized with our approach, for example, aggregate values such as the sum of array contents. As a result, relatively few formula patterns are detected. Nevertheless, in these cases, our approach still preserves the benefits of standard state merging w.r.t. standard symbolic execution.

8 RELATED WORK

Compact symbolic execution [50] 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 unbounded repetitions of *some* of the cyclic paths in the program. In contrast, we seek regularity at the level of the constraints and, therefore, do not rely on uniformity in the control flow graph. In `memspn` (Section 1), for example, 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 states. Furthermore, [50] solves quantified queries using a standard solver as opposed to our specialized solving procedure.

Godefroid et al. [32] propose a dynamic approach for inferring invariants in input-dependent loops, which allows them to partially summarize the loop's effect on induction variables. Loop-extended symbolic execution [46] summarizes input-dependent loops. It uses static analysis to infer linear relations between variables and trip count variables tracking the number of iterations in the loop. In contrast, our approach does not rely on induction variables or the number of loop iterations. Kapus et al. [35] summarize string loops by synthesizing calls to standard string functions. S-Looper [55] introduces string constraints that can be solved by solvers that support the string theory. Our approach is not restricted to string loops and does not require a solver supporting string theory.

Veritesting [13] improves the performance of symbolic execution by merging similar execution paths. Given a symbolic branch, veritesting summarizes side effects from both branch sides to avoid path explosion. Java Ranger [49] extends veritesting of Java programs to support dynamically dispatched methods, by using the runtime information available during the analysis. MultiSE [48] summarizes updates to values by efficiently guarding each value with a path predicate. Kuznetsov et al. [37] merge symbolic states based on a query count heuristic that estimates if the merging would reduce the solving time in the future. Trabish et al. [52] perform state merging in loops that depend on objects whose size is symbolic. They reduce the size of the encoding in the resulting merged states using the execution tree, but still rely on disjunctions and *ite* expressions, therefore unable to achieve the reduction obtained with our approach. We explicitly compared our technique with theirs (referred to as *CFG* in Section 7) and show that our approach performs better in many cases. The works mentioned above do not address the encoding explosion problem caused by using disjunctions and *ite* expressions.

There are many works on handling quantified formulas [14, 16, 23, 27, 30, 43–45]. Our solving procedure (Section 5) mainly targets satisfiable queries, and adapts ideas from E-matching [23] and model-based quantifier instantiation [30] to our specific needs.

9 CONCLUSIONS AND FUTURE WORK

We propose a state merging approach that significantly reduces the encoding complexity of merged symbolic states and show through our evaluation that this is a promising direction toward scaling state merging in symbolic execution.

Our approach automatically detects regular patterns to partition similar symbolic states into merging groups. For each group, we synthesize a formula pattern that 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.

Acknowledgements. This research was partially funded by the Israel Science Foundation (ISF) grants No. 1996/18 and No. 1810/18 and by Len Blavatnik and the Blavatnik Family foundation.

10 DATA AVAILABILITY

Our replication package is available at [3]. It contains a Docker image with all the resources needed to run the experiments.

REFERENCES

[1] 2023. https://github.com/davidtr1037/klee-quantifiers.
[2] 2023. https://github.com/klee/klee-uclibc/pull/47.
[3] 2023. https://doi.org/10.6084/m9.figshare.21990386.v8.
[4] 2023. busybox. https://busybox.net/.
[5] 2023. GCov. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.
[6] 2023. GNU libtasn1. https://www.gnu.org/software/libtasn1/.
[7] 2023. GNU oSIP. https://www.gnu.org/software/osip/.
[8] 2023. GNU Wget. https://www.gnu.org/software/wget/.
[9] 2023. json-c. https://github.com/json-c/json-c/.
[10] 2023. libpng. http://www.libpng.org/pub/png/libpng.html.
[11] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison Wesley.
[12] APR. 2023. *Apache Portable Runtime.*
[13] 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
[14] Kshitij Bansal, Andrew Reynolds, Tim King, Clark Barrett, and Thomas Wies. 2015. 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.* Springer, 87–105.
[15] 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. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *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*, Dana Fisman and Grigore Rosu (Eds.). Springer. https://doi.org/10.1007/978-3-030-99524-9_24
[16] 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
[17] 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
[18] 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
[19] 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).
[20] 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
[21] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90. https://doi.org/10.1145/2408776.2408795
[22] Peter Collingbourne, Cristian Cadar, and Paul H.J. Kelly. 2011. Symbolic Crosschecking 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
[23] Leonardo de Moura and Nikolaj Bjørner. 2007. Efficient E-Matching for SMT Solvers. In *Automated Deduction – CADE-21*, Frank Pfenning (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 183–198.
[24] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.
[25] Leonardo de Moura and Nikolaj Bjørner. 2008. 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)* (Budapest, Hungary).
[26] 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
[27] David Detlefs, Greg Nelson, and James B Saxe. 2005. Simplify: a theorem prover for program checking. *Journal of the ACM (JACM)* 52, 3 (2005), 365–473.
[28] Bruno Dutertre. 2014. Yices 2.2. In *Proc. of the 26th International Conference on Computer-Aided Verification (CAV'14)* (Vienna, Austria).
[29] 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
[30] Yeting Ge and Leonardo Mendonça de Moura. 2009. Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5643)*, Ahmed Bouajjani and Oded Maler (Eds.). Springer, 306–320. https://doi.org/10.1007/978-3-642-02658-4_25
[31] 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).
[32] 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
[33] 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
[34] 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
[35] 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 *Proc. of the Conference on Programing Language Design and Implementation (PLDI'19)* (Phoenix, AZ, USA). https://doi.org/10.1145/3314221.3314610
[36] James C. King. 1976. Symbolic execution and program testing. *Communications of the Association for Computing Machinery (CACM)* 19, 7 (1976), 385–394.
[37] 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
[38] 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
[39] 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
[40] 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
[41] 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.
[42] 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).
[43] Andrew Reynolds, Haniel Barbosa, and Pascal Fontaine. 2018. 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.* Springer, 112–131.
[44] Andrew Reynolds, Cesare Tinelli, and Leonardo De Moura. 2014. Finding conflicting instances of quantified formulas in SMT. In *2014 Formal Methods in Computer-Aided Design (FMCAD).* IEEE, 195–202.
[45] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstić, Morgan Deters, and Clark Barrett. 2013. 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.* Springer, 377–391.
[46] 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
[47] Daniel Schemmel, Julian Büning, Frank Busse, Martin Nowack, and Cristian Cadar. 2022. A Deterministic Memory Allocator for Dynamic Symbolic Execution. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)* (Berlin, Germany). 9:1–9:26.
[48] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: Multi-Path Symbolic Execution Using Value Summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy)

(ESEC/FSE 2015). Association for Computing Machinery, New York, NY, USA, 842–853. https://doi.org/10.1145/2786805.2786830

[49] 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

[50] Jiri Slaby, Jan Strejcek, and Marek Trtík. 2013. Compact Symbolic Execution. In *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8172)*, Dang Van Hung and Mizuhito Ogawa (Eds.). Springer, 193–207. https://doi.org/10.1007/978-3-319-02444-8_15

[51] Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. 2001. A Decision Procedure for an Extensional Theory of Arrays. In *Proc. of the 16th IEEE Symposium on Logic in Computer Science (LICS'01)* (Boston, MA, USA).

[52] David Trabish, Shachar Itzhaky, and Noam Rinetzky. 2021. 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*. Association for Computing Machinery, New York, NY, USA, 1190–1201. https://doi.org/10.1145/3468264.3468596

[53] David Trabish, Noam Rinetzky, Sharon Shoham, and Vaibhav Sharma. 2023. State Merging with Quantifiers in Symbolic Execution. arXiv:arXiv:2308.12068

[54] uClibc 2022. uClibc. https://www.uclibc.org/.

[55] Xiaofei Xie, Yang Liu, Wei Le, Xiaohong Li, and Hongxu Chen. 2015. S-looper: Automatic Summarization for Multipath String Loops. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'15)* (Baltimore, MD, USA).

Received 2023-02-02; accepted 2023-07-27