Accelerating Array Constraints in Symbolic Execution

David M. Perry[*]
Purdue University
USA
perry74@purdue.edu

Andrea Mattavelli
Imperial College London
United Kingdom
amattave@imperial.ac.uk

Xiangyu Zhang
Purdue University
USA
xyzhang@cs.purdue.edu

Cristian Cadar
Imperial College London
United Kingdom
c.cadar@imperial.ac.uk

ABSTRACT

Despite significant recent advances, the effectiveness of symbolic execution is limited when used to test complex, real-world software. One of the main scalability challenges is related to constraint solving: large applications and long exploration paths lead to complex constraints, often involving big arrays indexed by symbolic expressions. In this paper, we propose a set of semantics-preserving transformations for array operations that take advantage of contextual information collected during symbolic execution. Our transformations lead to simpler encodings and hence better performance in constraint solving. The results we obtain are encouraging: we show, through an extensive experimental analysis, that our transformations help to significantly improve the performance of symbolic execution in the presence of arrays. We also show that our transformations enable the analysis of new code, which would be otherwise out of reach for symbolic execution.

CCS CONCEPTS

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

KEYWORDS

Symbolic execution, Constraint solving, Theory of arrays

ACM Reference format:
David M. Perry, Andrea Mattavelli, Xiangyu Zhang, and Cristian Cadar. 2017. Accelerating Array Constraints in Symbolic Execution. In *Proceedings of 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10-14, 2017 (ISSTA'17),* 11 pages.
https://doi.org/10.1145/3092703.3092728

1 INTRODUCTION

Symbolic execution is an approach at the core of many modern techniques to software testing, automatic program repair, and reverse engineering [6, 10, 19, 21, 24, 26]. At a high-level, symbolic execution provides an automated mechanism for exploring multiple paths in a program by constructing and solving symbolic path conditions. Symbolic execution employs satisfiability-modulo theory (SMT) constraint solvers to determine the feasibility of a path condition and to generate concrete solutions for it. Despite the recent significant improvements in SMT solvers [12], constraint solving is still one of the main bottlenecks of symbolic execution. First, symbolic executors issue a huge number of queries to the constraint solver—for example those generated at every branch that depends on symbolic inputs—and second, symbolic execution generates large and complex constraints when applied to real-world programs. As a result, constraint solving dominates runtime for the majority of non-trivial programs, in which often more than 90% of the time is spent in constraint solving activities [23, 25]. Authors of symbolic execution engines recognize this major bottleneck and have invested significant effort in reducing the impact of constraint solving using various techniques, such as caching of solutions and optimizing queries with subset/superset relations [6]. Nevertheless, constraint solving still remains a showstopper for the application of symbolic execution to many real-world software.

```
1 char b64[256] = {−1, −1, −1,  ...  62, −1, −1, −1, 63, 52, 53, 54,
        55, 56, 57, 58, 59, 60, 61, −1, −1, −1, −1,  ...,  −1 };
2
3 unsigned isBase64(unsigned char k) {
4   if (b64[k] >= 0)
5       return 1;
6   else
7       return 0;
8 }
```

Figure 1: A simplified excerpt from the *base64* decoding routine in Coreutils. Array b64 has positive values at offsets 43, 47–57, 65–90, and 97–122.

Some of the most expensive constraints in symbolic execution involve arrays. Many programs take as inputs various forms of arrays, for example strings are encoded as arrays of characters, and developers extensively use arrays to implement various data structures (e.g. hash tables and vectors). Pointer operations in low-level code are often modelled using arrays [6?, 7]. As a result, arrays are prevalent in symbolic path conditions.

While array accesses with concrete indexes can be expressed and handled similarly to scalar variables, array accesses with symbolic indexes are more difficult to manage, as they could refer to potentially every position in the array. Consider the code in Figure 1, where the function isBase64 computes if a given integer input represents a value in the *base64* encoding. The access $b64[k]$ at line 4 can refer to any of the 256 positions in the array. This implies that all the array values have to be communicated to the constraint solver to determine the feasibility of $b64[k] \geq 0$. Therefore, an SMT formula over arrays requires creating a variable for each offset of the array and asserting the indexed value (more in §2.1 and §2.2). Such formulaic representation rapidly leads to inefficiencies

[*]The first two authors contributed equally to this paper.

in the constraint solving procedures that result in slowdowns and timeouts, thus hindering symbolic execution.

In this paper, we propose a fully automatic technique to optimize constraints involving arrays, which are one of the major bottlenecks in symbolic execution. Our technique employs orthogonal semantics-preserving transformations for array operations, by taking advantage of contextual information during symbolic execution. Our transformations lead to optimized constraint encodings, ultimately improving the efficiency of symbolic execution. They rely on the key observation that when the array contains concrete values, we can transform the array operations into semantically-equivalent ones involving only the indexes satisfying the condition, or the unique values in the array. For instance, the operation $b64[k] \geq 0$ in Figure 1 can be transformed into the equivalent query $k = 43 \lor k = 47 \lor k = 48 \lor ..$ that involves only the integer index values satisfying the condition, for which constraint solvers are highly optimized. Moreover, an array read with a symbolic index can be transformed to index range check operations that group the repeated values in the array, reducing the size of the formula in the solver. For example, the array indexing $b64[k]$ can be rewritten using an *if-then-else* construct so that the indexes in $(0 \leq k \leq 42) \lor (44 \leq k \leq 46) \lor ..$ result in reading -1 from the array, and so on.

We develop a prototype implementation of our transformations within the KLEE symbolic executor [6]. Our experimental evaluation demonstrates that the transformations can indeed lead to significant improvements in the performance of symbolic execution. In summary, we make the following contributions:

(1) We propose novel semantics-preserving transformations for array operations to achieve better encodings of arrays.
(2) We precisely define where and how these transformations are applied during symbolic execution, which requires distinguishing scenarios such as concrete arrays versus symbolic arrays and concrete indexing versus symbolic indexing.
(3) We develop a prototype implementation inside the symbolic executor KLEE and make it publicly available.
(4) We report the results on 57 real-world programs. Our technique improves symbolic execution performance by a factor of up to 27.46x when applicable. For inapplicable cases, it does not harm performance. We also show that our technique enables symbolic execution to analyze code unreachable otherwise.

2 OVERVIEW

This section provides the necessary background on the theory of arrays (§2.1), motivates our work using the popular GNU BC tool (§2.2), and then gives an overview of our contributions (§2.3).

2.1 Background

Arrays are prevalent in programs. Many popular data structures, such as vectors, strings, and hash tables, are internally represented as arrays. As such, the theory of arrays is one of the primitive theories supported by SMT solvers. Next, we briefly explain the array theory to provide the context for later discussion.

Inside an SMT solver, an array is typically represented as a set of (scalar) symbolic variables, one for each array element. Array reads/writes are realized by selecting the corresponding elements and performing the operation. Such selection may be very costly in situations where both concrete and symbolic array indexes are involved, requiring enumerating all possible concrete index values.

The array encoding in SMT solvers is generally founded on two axioms [15]. The first axiom enforces the *offset-to-value* correspondence in the array by creating a variable for each offset of the array and asserting that it is equal to the corresponding value in the array. Note that the values may be symbolic or concrete, depending on whether the array is updated by some symbolic expression at runtime. The second axiom enforces the *index-to-offset* correspondence, which implies that if the symbolic index of the array is equal to a specific offset, then the result of the read is the one enforced by the offset-to-value axiom. As such, the formulaic representation referring to an array read (with a symbolic index) is an intertwinement of the two axioms. For example, the branch condition at line 4 in Figure 1 is encoded in the following way, where $b64_0, b64_1, ..., b64_{255}$ represent the variables associated with the concrete array reads $b64[0], b64[1], ..., b64[255]$ and $b64_k$ is the variable chosen to represent the symbolic array read b64[k]:

$$(b64_0 = -1) \land (b64_1 = -1) \land \cdots \land (b64_{255} = -1) \land$$
$$(b64_k \geq 0) \land$$
$$(k = 0 \rightarrow b64_k = b64_0) \land \ldots \land (k = 255 \rightarrow b64_k = b64_{255}))$$

Array writes are not independently encoded. Instead, they are represented together with the enclosing read operations through the so-called *read-over-write* transformation [3, 27]. For instance the read $A[j]$ after a write $A[i] = v$ is encoded as follows, where the uninterpreted functions $read(A, i)$ and $write(A, i)$ denote reading and writing the ith element of array A, respectively, and ITE denotes an *if-then-else* expression:

$$read(write(A, i, v), j) \leftrightarrow \textsc{ite}(i = j, v, read(A, j))$$

That is, when a read has the same index as a preceding write, the value obtained is the updated value. When the indexes differ, the value obtained is the one stored in the array before the update.

In the next section, we discuss how the array theory can become the performance bottleneck for symbolic execution.

2.2 Motivation

GNU BC [16] is an arbitrary-precision calculator that solves expressions written in a C-style language. While performing symbolic execution on BC using KLEE, we observed that the analysis was encountering severe performance issues, in fact, it was still within one of the first functions after an hour of execution. Through manual inspection, we found that the symbolic execution was stuck in a code region whose simplified version is shown in Figure 2.

The code is a token matching algorithm performed in the function yylex in BC. The algorithm performs the lexical analysis step of the input script by implementing a Deterministic Finite Automaton (DFA). The goal of the DFA is to recognize the tokens defined in the language grammar of the calculator, which represent digits, variable names, and mathematical operators. To implement the DFA, programmers make use of several constant arrays whose values represent both state information and lexer actions. The user-supplied input is then used as an index for the arrays, and the subsequent result of the access is used to drive the behavior of the lexer. In

```
 1 char equivClass[256] = {0,  1,  1,  ..  10,  11,  12,  13  ..  1,  1,  1};
 2 char accept[298]     = {0,  0,  ..  41,  32,  28,    ..     23,  2,  2,  0};
 3 unsigned check[580]  = {0,  1,  1,  ..  141,  142,  144,  ..  297,  297};
 4 unsigned base[302]   = {0,  0,  ..  412,  422,  ..  120,  395,  398};
 5 unsigned def[302]    = {0,  297,    ..  63,  300,     ..  297,  297,  297};
 6 char meta[54]        = {0,  1,  1,  ..  1,  3,  3,  ..  3,  3,  1,  1,  1};
 7 unsigned next[580]   = {0,  6,  7,  ..  132,  133,    ..  297,  297};
 8
 9 void tokenMatch(char *input) {
10   unsigned currState = 0;
11   char charPtr = input;
12   do {
13     char currClass = equivClass[*charPtr];
14     if(accept[currState]) {
15       lastAcceptState = currState;
16       lastAcceptPos = charPtr;
17     }
18     while(check[base[currState] + currClass] != currState) {
19       currState = def[currState];
20       if(currState >= 298)
21         currClass = meta[currClass];
22     }
23     currState = next[base[currState] + currClass];
24     ++charPtr;
25   } while(base[currState] != 526);
26 }
```

Figure 2: A simplified parsing routine from BC.

particular, the array read at line 13 finds what character class an individual character belongs to. At line 23, nested array reads are used to look up the next state of the DFA using the current state and the input class. We observed that lines 13, 18, and 23 lead to constraints that are very expensive, many taking more than 10 seconds to resolve, whereas most other constraints only take a few milliseconds. As a result, the analysis became very slow when exploring paths in the function.

The first two columns of Table 1 present how the array operations at lines 13 and 18 are encoded inside the SMT solver, following the array theory presented in Section 2.1. Variables ec_i and $check_j$ denote the simple variables introduced to represent the ith and jth elements of arrays equivClass and check, respectively. We show the encodings on the first iteration of the loop, where the while condition on line 18 becomes simply check[currClass] != 0.

We can observe that the number of clauses in each constraint is based on the size of the array. When symbolic execution goes beyond the first iteration of the while loop at line 18, variable currState becomes symbolic and hence the nested array reads at line 18 yield a constraint that may be as large as 580 (the array size of *check*) $\times$ 302 (the array size of *base*) clauses, because the solver has to assert each index combination of the two arrays. A similar encoding happens for line 23. These very large constraints substantially degrade the performance of the solver. To make the situation worse, this piece of code is executed almost immediately once the program is started and is executed by most paths in the program. Moreover, since the path condition of exiting the do-while loop is the aggregation of the expensive array read operations (e.g. at lines 14, 18, and 23), solving the corresponding constraint causes timeouts. In other words, the analysis is stuck in this function.

Unfortunately, such performance bottlenecks caused by array operations are common in real-world applications. In fact, in our experimentation on real-world programs, for example taken from Binutils and Coreutils, we observed 25 programs that contained large arrays which determined the paths taken during execution.

2.3 Technique Overview

Array operations inevitably lead to the application of the (expensive) array theory, and thus the overarching idea of our technique is *to reduce as many array operations as possible through constraint transformations.* Such transformations are conducted on-the-fly to leverage runtime information to achieve better reduction. More exactly, we perform the following three transformations.

First, we perform *dynamic constant folding* on array reads such that we can preclude both symbolic variables used to denote the constant values of individual array elements, and formulas passed to the solver asserting the constant values for individual indexes.

Second, we perform *index-based transformation* for predicates whenever the arrays are constant. In essence, we transform predicates involving array reads (e.g. $A[i] < 10$) to equivalence checks involving only the index variables (e.g. $i \equiv 2 \vee i \equiv 5 \vee \ldots$) and completely remove the array operations.

Third, we observe that many arrays contain repeated values, for instance the meta array in Figure 2 contains a lot of repeated 1's. We perform dynamic range analysis to determine the ranges of indexes that lead to identical values, and then apply *value-based transformation.* This transformation replaces array reads with unique values guarded by range checks, substantially reducing the number of formulas passed to the solver. Essentially, this transformation allows the solver to explore multiple possible indexes simultaneously. Therefore, the solver algorithm has to branch and backtrack much less frequently, resulting in large performance improvements. This transformation is even applicable to partially symbolic arrays.

Table 1 column 3 shows the constraints after our transformations. Variables ec_i and $check_j$ are replaced with concrete values through constant folding. Furthermore, at line 18 the complex condition was replaced by a simple constraint on the index, namely $1 \leq currClass \leq 579$. At line 13 the original assertions for individual indexes are replaced with ITE expressions that return the unique values of the array based on a series of range checks. For example, according to the source code in Figure 2, $equivClass[0] \equiv 0$, $equivClass[1-8, 11-13, \ldots] \equiv 1$, and so on, leading to the transformed encoding for this line. After the transformation, the symbolic execution of BC improves significantly: Within one hour, KLEE completes the exploration of 828 paths, whereas the original run explored only 20 paths.

The index-based transformation (i.e. replacing conditions over array values with conditions over indexes) can lead to significant speed-ups, because (1) it is no longer needed to communicate to the solver all the values in the arrays and (2) the constraints on indexes are typically much simpler than those on the array values, as the constraints establishing the *index-to-offset* correspondence are not added anymore in the formula.

The speed-up produced by the value-based transformation is achieved by dramatically reducing the number of cases that the solver needs to examine. Note that nested array reads lead to combinatorial explosion of such cases. For simplicity, let us assume that there are k layers of nesting of array reads (e.g. $k = 3$ for $A[B[C[i]]]$), and each array read has n branches (i.e. the size of the array). The solver may need to consider $O(n^k)$ cases. However, after our transformation, if we assume that there are m ranges, the number of cases to examine is reduced to $O(m^k)$. The reduction is

Table 1: Original and optimized constraints for the motivating example in Figure 2.

Term	Original Encoding	Transformed Encoding
Line 13, 1st iteration currClass = equivClass[*charPtr]	$ec_0 = 0 \wedge \ldots \wedge ec_{255} = 1 \wedge$	$\textsc{ite}$(*charPtr = 0, currClass = 0
	*charPtr = 0 $\rightarrow$ currClass = ec_0 $\wedge$	$\textsc{ite}$(1 $\leq$ *charPtr < 9 $\vee \ldots$, currClass = 1
	*charPtr = 1 $\rightarrow$ currClass = ec_1 $\wedge$	$\textsc{ite}$(*charPtr = 9, currClass = 2,
	$\vdots$	$\vdots$
	*charPtr = 255 $\rightarrow$ currClass = ec_{255}	
Line 18, 1st iteration check[currClass] != 0	$check_0 = 0 \wedge \ldots \wedge check_{579} = 297 \wedge$	1 $\leq$ currClass $\leq$ 579
	$check_{currClass} \neq 0$	
	currClass = 0 $\rightarrow$ $check_{currClass} = check_0 \wedge$	
	$\vdots$	
	currClass = 579 $\rightarrow$ $check_{currClass} = check_{579}$	

thus $O((n/m)^k)$. In other words, the improvement is exponential in the number of nested layers.

Our technique is implemented inside KLEE and transparent to the users. The tool is publicly available and open source.[1]

3 DESIGN

A plausible design is to statically transform the subject program such that an off-the-shelf symbolic execution engine (e.g. KLEE) can directly produce cost-effective array encoding from the transformed code. However, our experience suggested that there are limited opportunities to apply such transformations due to the conservative nature of static analysis. For example, an array may be constant along many paths but symbolic along others. A static analysis will thus conservatively assume that it is symbolic. As such, transformations that require a constant array cannot be applied. Therefore, we decided to develop our technique as part of the symbolic execution runtime to exploit the path-sensitivity of array properties (e.g. an array is constant along a specific path).

The proposed transformations not only require extending the symbolic execution runtime, but are also closely coupled with the symbolic execution process itself. In particular, during symbolic execution we need to distinguish the different scenarios where the transformations can be applied, such as when the arrays are constant, symbolic, or mixed (i.e. some array elements are constant while others are symbolic); when the indexes of array reads/writes are constant or symbolic; and when the non-array operands involved in an array operation (e.g. x in $A[i] > x$) are constant or symbolic. In addition, the transformations change the overall encoding process of array-related operations. To precisely describe our design, we use a highly simplified array-oriented language and its symbolic execution semantics to explain our extension to the runtime, the array encodings, and our proposed transformations.

3.1 Semantics

Language. The array-oriented language is presented in Figure 3. The language allows explicitly annotating symbolic variables, i.e. variables on which symbolic constraints are built. They are usually input variables. The language supports simple assignments, array reads and writes, conditional statements and loops. Despite its simplicity, it models all the features related to our design and

[1] https://srg.doc.ic.ac.uk/projects/klee-array/

Program	π	$::= s*$
Statement	s	$::= x = e \mid x = A[e] \mid A[e_i] = e_v \mid$ **if** e **then** s_1 **else** s_2 $\mid$ **while** (e) s
Expression	e	$::= x \mid \hat{x} \mid c \mid e_1\ op\ e_2$
Variable $x, y, \ldots$	Array $A, B, \ldots$	SymVar $\hat{x}, \hat{y}, \ldots$
Constant	c	$::= \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \mid$ **true** $\mid$ **false**
Operator	op	$::= + \mid - \mid \ldots \mid > \mid \geq \mid = \mid \neq \mid \wedge \mid \vee$

Figure 3: Array-oriented language.

SymValue $v ::= c \mid \hat{x} \mid v_1\ op\ v_2 \mid \textsc{ite}(v_c, v_1, v_2)$
SymStore $\sigma ::= (\textit{Variable} \mid \textit{Array}[\textit{Constant}]) \rightarrow \textit{SymValue}$
PathCond $p ::= v \mid p_1 \wedge p_2$
eval(op, c_1, c_2) : evaluate the concrete value of $(c_1\ op\ c_2)$
SAT(p) : determine if a path denoted by the path condition p is feasible
$\textsc{valTrans}(A, v_i)$: value-based transformation for array read $A[v_i]$ with v_i the symbolic index expression
$\textsc{idxTrans}(A, >, v_i, c)$: index-based transformation for predicate $A[v_i] > c$ with v_i the symbolic index expression

Figure 4: Definitions for semantic rules.

is sufficient for our discussion. Note that our prototype is implemented within KLEE and hence supports all the real-world language features that KLEE supports.

Figure 4 introduces a few definitions and functions for the semantic rules. Specifically, *SymValue* denotes a symbolic expression that may be a constant, a symbolic variable, a compound expression, or an ITE (*if-then-else*) expression. Intuitively, one can consider these as the symbolic values held by program variables during symbolic execution. *SymStore* maps a program variable or an array element to its symbolic expression. Its functionality in symbolic execution is analogous to memory in concrete execution. Most symbolic execution engines, including KLEE, have internal data structures that serve as *SymStore*. Function *eval* evaluates an operator (e.g. integer addition/subtraction). Functions VALTRANS and IDXTRANS perform the value-based transformation and index-based transformation, which we will discuss in details in Section 3.2. The former returns a transformed symbolic expression, whereas the latter returns a transformed predicate (i.e. a boolean formula).

Semantic Rules. The rules in Figure 5 describe how we handle array-related operations during symbolic execution. They are self-contained, to some extent, such that the overall symbolic execution procedure is illustrated, and thus the conditions and the applications of our transformations can be perceived in the appropriate contexts.

The rules consist of two subsets. The first subset defines rules for evaluating expressions, while the second for evaluating statements. The evaluation of expressions has the form of $\sigma : e \xrightarrow{e} v$, meaning that given the store σ, an expression e is evaluated to a symbolic expression v. Rule E-Const defines that a constant is evaluated to itself. A program variable x is evaluated to its symbolic expression in the store (E-Var), and a symbolic input variable $\hat{x}$ is evaluated to itself (E-SymVar). In case of a binary operation, if the two operands are evaluated to constants, the binary operation is performed, which subsequently yields a constant outcome (E-ConstFolding). For instance, $x + 2$ is evaluated to 5 if x has a constant value 3 in the current path, e.g. , the preceding write to x is $x = 3$. Such constant folding is only performed for the current path explored by the symbolic execution engine, and hence the name *dynamic constant folding.* Note that in the aforementioned example, a compiler cannot perform static constant folding if there is another preceding write $x = 4$ along a different path. However, dynamic constant folding can yield constants $x = 5$ and $x = 6$ when the respective paths are explored. A binary expression with at least one symbolic operand yields a symbolic expression (E-SymExpr). For instance, $\hat{x} + 2$ is evaluated to $\hat{x} + 2$. Such symbolic values will be used to build symbolic path constraints.

Statement rules are of the form $\langle s,\sigma,p\rangle \xrightarrow{s} \langle s',\sigma',p'\rangle$ with the symbols defined in Figures 3 and 4. In other words, each step of a statement evaluation may update the statement s, the store σ, and the path condition p. Rule Arr-Wr-CI specifies that if the index of an array write is evaluated to a constant, then the corresponding array element in the store is mapped to the right-hand-side symbolic value v_2. Note that whether v_2 is constant is irrelevant. As a side note, when an array is created, the entries for individual array elements are also allocated in the store. When the index is symbolic (Arr-Wr-SI), we do not know which array element needs to be updated. We hence update all the elements in the array with an ite expression. For example, a statement $A[i] = 3$ leads to the expression $\sigma[A[0] \mapsto \text{ITE}(i = 0, 3, \sigma[A[0]])]$, $\sigma[A[1] \mapsto \text{ITE}(i = 1, 3, \sigma[A[1]])]$, and so on. Intuitively, this means that depending on the symbolic index value, each element may either be updated to the new value, or retain its original value. We note that these symbolic values are still inside the symbolic execution engine and not yet passed to the solver. We also note that this representation could be stored more compactly by the symbolic executor via lists of array writes [7].

Rule Arr-Rd-CI specifies that an array read with a constant index results in simply loading the symbolic value from the store and assigning it to the left-hand-side variable. This rule is crucial for array constant folding because if the array element is constant, the array read is completely precluded and replaced with a constant. For example, in Figure 1 variables ec_0, ..., ec_{255} (column *Original Encoding*) are replaced with constants after dynamic constant folding (column *Transformed Encoding*). It is also worth noting that dynamic constant folding requires tracking and evaluating all constant operations during symbolic execution (e.g. by rule E-ConstFolding). In other words, the encoding of a statement may require the global view of the operations that contributed to the operand values of the statement. Many symbolic execution engines, for simplicity of implementation, adopt a statement-local encoding scheme that encodes a statement independently from the others (e.g. encoding line 13 currClass=equivClass[*charPtr] in Figure 2 without referring to the definitions of equivClass), and hence introduce redundancy caused by constants.

Expression Rule: $\boxed{\sigma : e \xrightarrow{e} v}$

$$\sigma : c \xrightarrow{e} c \quad (\text{E-Const}) \qquad \sigma : x \xrightarrow{e} \sigma[x] \quad (\text{E-Var})$$

$$\sigma : \hat{x} \xrightarrow{e} \hat{x} \quad (\text{E-SymVar}) \qquad \frac{\sigma : e_1 \xrightarrow{e} c_1 \quad \sigma : e_2 \xrightarrow{e} c_2}{\sigma : e_1\ op\ e_2 \xrightarrow{e} eval(op, c_1, c_2)} \ (\text{E-ConstFolding})$$

$$\frac{\sigma : e_1 \xrightarrow{e} v_1 \quad e_2 \xrightarrow{e} v_2 \quad \neg isConst(v_1) \lor \neg isConst(v_2)}{\sigma : e_1\ op\ e_2 \xrightarrow{e} v_1\ op\ v_2} \ (\text{E-SymExpr})$$

Statement Rule: $\boxed{\langle s, \sigma, p\rangle \xrightarrow{s} \langle s', \sigma', p'\rangle}$

$$\frac{\sigma : e_1 \xrightarrow{e} c_1 \quad \sigma : e_2 \xrightarrow{e} v_2}{\langle A[e_1] = e_2; s, \sigma, p\rangle \xrightarrow{s} \langle s, \sigma[A[c_1] \mapsto v_2], p\rangle} \ (\text{Arr-Wr-CI})$$

$$\frac{\sigma : e_1 \xrightarrow{e} v_1 \quad \sigma : e_2 \xrightarrow{e} v_2 \quad \neg isConst(v_1)}{\langle A[e_1] = e_2; s, \sigma, p\rangle \xrightarrow{s} \langle s, \sigma[\forall i\ A[i] \mapsto \mathbf{ite}(i \equiv v_1, v_2, \sigma[A[i]])], p\rangle} \ (\text{Arr-Wr-SI})$$

$$\frac{\sigma : e_1 \xrightarrow{e} c_1}{\langle x = A[e_1]; s, \sigma, p\rangle \xrightarrow{s} \langle s, \sigma[x \mapsto \sigma[A[c_1]]], p\rangle} \ (\text{Arr-Rd-CI})$$

$$\frac{\sigma : e_1 \xrightarrow{e} v_1 \quad \neg isConst(v_1) \quad \forall i\ isConst(\sigma[A[i]])}{\langle x = A[e_1]; s, \sigma, p\rangle \xrightarrow{s} \langle s, \sigma[x \mapsto \text{valTrans}(A, v_1)], p\rangle} \ (\text{Arr-Rd-SI-RangeChk})$$

$$\frac{\begin{array}{c}\sigma : e_1 \xrightarrow{e} v_1 \quad \neg isConst(v_1) \quad \exists i,\ \neg isConst(\sigma[A[i]]) \\ v' = \mathbf{ite}(v_1 \equiv 0, \sigma[A[0]], \mathbf{ite}(v_1 \equiv 1, \sigma[A[1]], \mathbf{ite}(\ldots)))\end{array}}{\langle x = A[e_1]; s, \sigma, p\rangle \xrightarrow{s} \langle s, \sigma[x \mapsto v'], p\rangle} \ (\text{Arr-Rd-SI})$$

$$\frac{\begin{array}{c}\sigma : e_1 \xrightarrow{e} v_1 \quad \neg isConst(v_1) \\ \sigma : e_2 \xrightarrow{e} c_2 \quad \forall i\ isConst(\sigma[A[i]]) \\ p' = p \land \text{idxTrans}(A, >, v_1, c_2) \quad SAT(p')\end{array}}{\langle \mathbf{if}\ A[e_1] > e_2\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2; s, \sigma, p\rangle \xrightarrow{s} \langle s1; s, \sigma, p'\rangle} \ (\text{Pred-GT-T})$$

$$\frac{\begin{array}{c}\sigma : e_1 \xrightarrow{e} v_1 \quad \neg isConst(v_1) \\ \sigma : e_2 \xrightarrow{e} c_2 \quad \forall i\ isConst(\sigma[A[i]]) \\ p' = p \land \text{idxTrans}(A, \leq, v_1, c_2) \quad SAT(p')\end{array}}{\langle \mathbf{if}\ A[e_1] > e_2\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2; s, \sigma, p\rangle \xrightarrow{s} \langle s_2; s, \sigma, p'\rangle} \ (\text{Pred-GT-F})$$

Figure 5: Semantic rules.

For an array read with a symbolic index, we have two ways to proceed. If the array is a constant array, e.g. all elements are constants like in the example in Figure 2, we apply rule Arr-Rd-SI-RangeChk to invoke function valTrans and transform the array read to a set of range check operations, which yield much simpler constraints. The details of this transformation are discussed in Section 3.2. If the array is not constant (Arr-Rd-SI), we leverage the *index-to-offset* and *offset-to-value* axioms (§2.1) and use a nested ite expression to denote the value of x. Intuitively, it means $x = A[0]$ if e_1 evaluates to 0; $x = A[1]$ if e_1 evaluates to 1; and so on. In

Algorithm 1 Value-Based Transformation

```
function valTrans($A$, $v_i$)
    $uniqueVal$ = the set of unique values in $A$
    return rangeIte($uniqueVal$, $A$, $v_i$)
end function

function rangeIte($uniqueVal$, $A$, $v_i$)
    if $|uniqueVal| \equiv 1$ then
        return $uniqueVal.pop()$
    end if
    $t = uniqueVal.pop()$
    $R$ = the index ranges such that $\forall i \in R,\ A[i] \equiv t$
    $C = false$
    for each $[lb, ub] \in R$ do
        $C = C \lor lb \le v_i \le ub$
    end for
    return ite($C$, $t$, rangeIte($uniqueVal$, $A$, $v_i$))
end function
```

the implementation, the *read-over-write* axiom is further applied to simplify the encoding. Since this procedure is standard, the details are elided.

Rules Pred-GT-T and Pred-GT-F illustrate how we handle predicates in the form of $A[e_1] > e_2$. Rule Pred-GT-T describes the scenario where the *then* branch is feasible and specifies that when (1) *the array is constant*, (2) *the index is symbolic*, and (3) *the value in the array read is compared with is a constant (after unfolding)*, we will invoke idxTrans to transform the predicate to simpler checks of the index expression and suppress the array read. Details will be discussed in Section 3.2. The rule also specifies that the transformed predicate is added to the path condition p. The solver is queried with the resulted path condition through the SAT primitive presented in Figure 4. If the condition is feasible, the true branch is taken. Observe that p accumulates the predicates along the path. Rule Pred-GT-F describes the scenario where the *else* branch is feasible. Note that in symbolic execution, it is possible that both branches are feasible and thus both rules can apply. The evaluation rules for other comparison operations such as equality or less than are treated similarly and hence elided. In case the aforementioned three conditions are not simultaneously satisfied, index-based transformation will not be triggered. However, value-based transformation may still be triggered. The evaluation of other statements such as *while* statements is standard and hence elided.

3.2 Transformations

We propose three kinds of transformations. The first one is dynamic constant folding that is conducted by rules E-ConstFolding, Arr-Rd-CI, and Arr-Wr-CI in Figure 5. In this section, we focus on discussing the value-based and the index-based transformations.

Value-Based Transformation. As described by rule Arr-Rd-SI-RangeChk, the value-based transformation is triggered upon an array read with a constant array and a symbolic index. Informally, the transformation turns an array read to a switch-case kind of structure that returns the unique values of the array based on index ranges, thus avoiding the application of the expensive array theory.

Algorithm 2 Index-Based Transformation

```
function idxTrans($A$, $op$, $v_i$, $c$)
    $R$ = the index ranges such that
                        $\forall i \in R,\ eval(op, A[i], c) \equiv true$
    $C = false$
    for each $[lb, ub] \in R$ do
        $C = C \lor lb \le v_i \le ub$
    end for
    return $C$
end function
```

The procedure is described in Algorithm 1. Function valTrans takes the array and the index expression, and returns another expression denoting the value of the array read. The algorithm starts acquiring all the unique values of the array (line 2). For instance, given the array $\{0, 0, 2, 2, 0, 0, 3\}$, $uniqueVal = \{0, 2, 3\}$. Note that such information can be precomputed and reused. The algorithm then invokes rangeIte to generate a range check-based ite expression (line 3). Function rangeIte is recursive. At each recursion, if the number of values in *uniqueVal* is greater than 1, it pops a value t from *uniqueVal* (line 9) and computes the set of index ranges within which the array elements have the value t (line 10). In the previous example, for value 0 the set is $R = \{[0, 1], [4, 5]\}$. The loop at lines 12–14 constructs a range check condition from R. For example, the condition for value 0 is $0 \le v_i \le 1 \lor 4 \le v_i \le 5$. The algorithm finally constructs an ite expression dictating that if the condition C holds, the value of the expression is t, otherwise another ite expression is constructed from the remainder of the *uniqueVal* (line 15). The recursive procedure terminates when there is only one value left in *uniqueVal* (lines 6–7), and we simply return the value. For our example, the resulting expression is the following:

$$\text{ITE}(0 \le v_i \le 1 \lor 4 \le v_i \le 5,\ 0,\ \text{ITE}(2 \le v_i \le 3,\ 2,\ 3))$$

Index-Based Transformation. As illustrated by rule Pred-GT-T, the index-based transformation is triggered upon a comparison operation of an array read that satisfies the three conditions specified in the description of the rule (§3.1). The transformation turns the comparison operation into a conjunctive formula that compares index ranges without the explicit array read, avoiding the application of the array theory.

The procedure is described in Algorithm 2. The algorithm scans the array to identify the index ranges that satisfy the comparison operation (lines 2–3). For example, given $A = \{0, 0, 3, 3, 0, 0, 4\}$ and the comparison $A[v_i] < 3$, $R = \{[0, 1],\ [4, 5]\}$. The loop at lines 4–6 constructs a formula that checks if the given symbolic index v_i falls into these ranges as a disjunction of the index ranges. For our example, the condition is $0 \le v_i \le 1 \lor 4 \le v_i \le 5$. Note that the index-based transformation is different from the value-based transformation as the values of the array are not explicitly present in the transformed formula. The index-based transformation also leads to more concise encodings since it completely replaces the original predicate.

Interplay of Transformations. The transformations we propose can achieve synergy. Dynamic constant folding can be combined with the other two under all circumstances. Consider the encodings of line 18 in Figure 1 (i.e. the first iteration of the loop while (check[base[currState]+currClass]!=currState)). Dynamic constant folding replaces base[currState] with 0 and currState with 0. The resulting comparison check[currClass]!=0 is then further transformed by the index-based transformation to $1 \leq currClass \leq 579$.

The value- and index-based transformations also work together. Our strategy is to try to apply the index-based transformation first, and if unsuccessful try the value-based transformation. For instance, on line 18 in Figure 1, in the first iteration we can apply the index-based transformation, while in the second iteration, when *currState* becomes symbolic, we apply the value-based transformation.

3.3 Handling Mixed Arrays

The value-based transformation and the index-based transformation described previously can only work when every value in an array is concrete. While we have found this is true in many programs, arrays frequently contain a mix of both concrete and symbolic values. It is hence highly desirable to optimize encodings for those arrays. Next, we discuss our extension to the value-based transformation to support mixed arrays. Recall that the transformation turns an array read to a nested ITE expression with each branch denoting a range of indexes that lead to a unique value. A plausible solution is to perform the same transformation to the concrete part of the array and then include the original array read in the last else branch of the nested ITE to denote all the remaining symbolic array elements. For instance, an array read $A[v_i]$ with $A = \{0, 0, v_1, v_2, 2, 2\}$ (v_1 and v_2 symbolic expressions) is transformed to $\text{ITE}(0 \leq v_i \leq 1,\ 0,\ \text{ITE}(4 \leq v_i \leq 5,\ 2,\ A[v_i]))$. The method tries to leverage the constant elements as much as possible and only resorts to the array read when necessary. While the transformed expression is equivalent to the original read, we found that the presence of $A[v_i]$ in the ITE nonetheless triggers the expensive array theory, which prevents any improvement.

The key is thus to avoid any explicit array operation. Therefore, our solution is to enumerate individual symbolic array elements in separate ITE branches. For example, the above array read example is transformed to:

$$\text{ITE}(0 \leq v_i \leq 1,\ 0,\ \text{ITE}(4 \leq v_i \leq 5,\ 2,\ \text{ITE}(v_i = 2,\ v_1,\ v_2)))$$

4 IMPLEMENTATION

We implemented our transformations inside the KLEE symbolic execution engine [6]. The dynamic constant folding and value-based transformations can be applied as soon as the symbolic executor reaches an instruction involving an array read. More precisely, we implemented these constraint transformations as program expression optimizations using KLEE's APIs.

In the case of the dynamic constant folding transformation, we rewrite the array read into a simple constant value whenever the array and the index are both constants. Since the transformation is path-sensitive, we check that no previous write operation to the array occurs with either a symbolic index, or with a symbolic value at the same array location. In the case of the value-based transformation, we generate an ITE expression when the array is concrete and the index is a symbolic expression. As discussed in Section 3.3, the value-based transformation can be applied even if the array is only partially concrete.

While the first two transformations are generally applicable, the index-based transformation can only be employed for boolean conditions. We thus invoke the index-based transformation during the execution of a branch instruction. In this case, it is crucial to implement the evaluation function *eval* in Algorithm 2 without relying on the constraint solver, which would become a major performance bottleneck for large arrays. Instead, the *eval* function is invoked for each possible concrete index of the array, and evaluated using KLEE's constant folding features. In this way, we are able to handle even large arrays in a few microseconds.

Integrating the value- and index-based transformations requires careful design. If we allow the value-based transformation to optimize an array read as soon as it is performed, it then becomes difficult to apply the index-based transformation, as the read is now a large ITE expression. Therefore, we first try to apply the index-based transformation, and if it fails, we then use the value-based one (§3.2). The main disadvantage of such approach is that we need to implement the value-based transformation at the level of boolean conditions rather than array reads, which requires extra work to scan the reads encountered in boolean conditions.

5 EXPERIMENTAL VALIDATION

We conduct an experimental analysis to validate our transformations. We consider three validation questions. The first research question (RQ1) is about *correctness*: We want to make sure that our transformations yield new constraints that are semantically equivalent to the original ones.

The second research question (RQ2) is about *effectiveness*: We want to ensure that our transformations lead to improved performance. Specifically, we want to verify that symbolic execution explores programs faster when our transformations are applied, without disrupting performance when they cannot be applied.

The third research question (RQ3) is about *significance*: We want to validate that the improvements we obtain with our transformation can make an important difference on code coverage, that is they enable the symbolic executor to explore parts of a program that would have been unreachable otherwise.

5.1 Evaluation Setup

We consider 104 programs written in C from five different subjects that can be successfully analyzed by the KLEE symbolic executor. For each program, we manually derive the set of symbolic arguments to use in combination with KLEE to maximize the exploration of the symbolic executor. When possible, we use the same argument configuration as in prior work [6]. The subjects are:

1. **Bandicoot** [2] v6, a programming system with a set-based programming language. We focus on the compiler.
2. **BC** [16] v1.06, an arbitrary-precision calculator that solves mathematical expressions written in a C-style language.
3. **Binutils** [17] v2.27, a collection of 12 programs to analyze, link, and manipulate binaries from the `binutils` directory.
4. **Bzip2** [4] v1.0.6, a compression utility.
5. **Coreutils** [18] v6.11, a collection of 89 programs for file, text, and shell manipulation.

Table 2: Subjects considered in the evaluation with number of lines of code, conditional branches, and arrays.

Subject	C SLOC	LLVM Branches	Arrays	
			No.	Bytes
Bandicoot	9,695	1,582	42	11,351
BC	10,150	1,513	18	10,163
Binutils	1,582,690	48,326	290	214,056
Bzip2	5,823	1,608	1	4,096
Coreutils	15,065	62,713	659	39,149

Table 2 briefly characterizes the subject programs. For each subject (*Subject*) we report the number of C source lines of code measured by cloc [11] (*C SLOC*), the number of LLVM bitcode conditional branches as reported by KLEE (*LLVM Branches*), and the number (*No.*) and size (*Bytes*) of the arrays statically found in the subjects (*Arrays*). To obtain the data on arrays, we rely on a static analysis on the LLVM bitcode that focuses on global arrays in the subject only. The analysis thus misses both array definitions local to functions, and dynamically allocated arrays.

We use KLEE commit ad22e84 compiled with LLVM 3.4.2 [22] and STP 2.1.2 as the constraint solver [15]. We conduct our experiments on two identical servers running Ubuntu 14.04, equipped with an Intel processor (8 cores) at 3.5 GHz and 16GB of RAM. For brevity we omit other details of the KLEE setup but we make the experiments available for further analysis.[2]

5.2 Correctness (RQ1)

We validate the correctness of our transformation by checking that symbolic execution follows identical paths with and without our transformations. The main challenge is to configure KLEE to behave deterministically across runs. This is hard to achieve since KLEE relies on a wide variety of timeouts (e.g. for constraint solving), time-bounded search heuristics, concrete memory addresses (as those returned by malloc), and other environmental values (e.g. file descriptors). To force KLEE to behave as deterministically as possible, we use depth-first search (DFS) strategy, a sandbox for file system operations, and rely on a deterministic memory allocator.

We proceed with the following evaluation process:

(1) We create a set of subject programs that KLEE symbolically executes deterministically as follows:
   (a) We run KLEE with the DFS strategy and a time limit of 30 minutes, and we record the number of instructions executed as well as the execution trace.
   (b) We run KLEE again with DFS and we limit the number of instructions executed to the value obtained in step (a).
   (c) We compare the execution traces between steps (a) and (b). We consider the program for further analysis if and only if there is no mismatch.
(2) We then activate our transformations in KLEE one at a time on the set of subject programs selected at step (1). We configure KLEE with DFS strategy and we limit the executed instructions to the value obtained at step (1) (a).
(3) We compare for each run the execution traces against those obtained at step (1) (b). We report any mismatch as a failure of our transformation in preserving semantics.

[2]https://srg.doc.ic.ac.uk/projects/klee-array/

Table 3: Subjects where our transformations do not apply.

cat	cksum	echo	env	nl	printenv
runcon	tee	yes	cxxfilt	elfedit	nm
objcopy	objdump	readelf	size	strings	strip

Results. We report in Table 3 and Table 4 the 57 programs selected through the selection process described at step 1 (column *Subject*). The selection process discards 47 programs where the behavior of KLEE is not deterministic. Table 3 lists the benchmarks for which execution was deterministic, but our transformations were never triggered, likely because they do not use (largely) constant arrays during execution. The runtime overhead caused by our technique is always less than 1% for these programs.

Table 4 shows the benchmarks for which our transformations were triggered. We report the result on correctness (equivalence) among runs in column *Eq*, where we consider the experiment a success (✓) if and only if all transformations execute the same instructions as the Baseline run, and a failure (✗) otherwise.

As can be seen, our transformations produce semantically equivalent constraints that lead to the exploration of identical paths and instructions in all but two cases: ld and ptx. In ld, the unoptimized version reaches an expensive array query and times out, while KLEE with our transformations can successfully solve the queries. Due to such a timeout in the unoptimized version, the two executions diverge. However, in this case, this exploration divergence highlights the benefits of our transformations, rather than indicating issues in the semantic equivalence of the constraints.

In ptx, KLEE explores different paths when run with the combined transformations because the unoptimized version is forced to concretize symbolic data due to a user-defined limit on the size of symbolic array reads. However, when run with the combined transformation, KLEE optimizes some constraints at the beginning of the execution, and can proceed without concretizing the read expression. Again, this divergence highlights the benefits of our transformations.

5.3 Effectiveness (RQ2)

We evaluate the effectiveness of our transformations by assessing their ability to speed-up symbolic execution. We report the results in Table 4. The column Baseline shows the runtime for the unmodified KLEE runs in seconds. Notice that for a few programs, KLEE terminates to explore all feasible paths within the time limit (e.g. basename). Moreover, although we run KLEE with a 1800 seconds limit, the actual execution times slightly differ due to both additional checks performed to limit the exploration to a given number of instructions, and variations in constraint solving time.

For each transformation, we report the runtime in seconds (*Time*), the speed-up over the unmodified run computed as the ratio between the Baseline runtime and the runtime with the transformation enabled (S), and the number of invocations of each transformation (*Inv.*). Benchmarks are sorted by the speed-up achieved by the combined transformation (Column Combined).

From the results, we can notice that our transformations indeed improve performance significantly in many cases. In particular, in 12 cases our combined transformations speed-up symbolic execution by more than 1.35x, and in 8 cases by over 2.50x. The largest improvements are seen for factor and BC with speed-ups of 27.46x

Table 4: Runtime of different versions of KLEE running the same instructions: Baseline is the unmodified KLEE, Constant, Index-based, and Value-based represent KLEE running with each of our transformations, and Combined is KLEE running with all of them. For each transformation, S is the speedup over the Baseline, and Inv is the number of successful invocations of each transformation. Eq has a checkmark for the programs on which all transformations preserve semantics.

Program	Eq	Baseline	Constant			Index-based			Value-based			Combined		
		Time (s)	Time (s)	S	Inv.	Time (s)	S	Inv.	Time (s)	S	Inv.	Time (s)	S	Inv.
factor	✓	1,815.21	1,873.34	1.02x	1,626	78.84	24.20x	171	136.90	13.94x	159	69.48	27.46x	171
bc	✓	1,834.82	1,830.64	1.01x	0	472.62	3.90x	354	260.33	7.12x	312	218.61	7.78x	632
tsort	✓	1,801.62	1,755.57	1.03x	0	570.62	3.16x	13	568.36	3.17x	340	433.60	4.16x	572
bzip2	✓	1,820.49	476.42	3.82x	58,448	1,805.12	1.01x	0	1,805.19	1.01x	0	482.63	3.77x	59,515
expr	✓	1,799.20	1,819.47	0.99x	0	697.60	2.58x	11	1,785.36	1.01x	107	479.44	3.75x	259
bandicoot	✓	1,790.93	1,739.23	1.03x	120	1,202.22	1.49x	78	175.93	10.18x	562	497.73	3.60x	651
ar	✓	1,800.72	1,795.91	1.00x	0	815.05	2.21x	147	232.57	7.74x	341	511.68	3.52x	476
[	✓	57.16	47.84	1.20x	16	34.32	1.67x	9	56.39	1.01x	5	22.82	2.50x	51
nice	✓	1,799.79	1,162.12	1.55x	12,444	1,763.22	1.02x	22	1,056.71	1.70x	16	1,053.71	1.71x	910
od	✓	1,823.44	1,546.20	1.18x	28,136	1,318.24	1.38x	824	1,489.90	1.22x	27,425	1,085.17	1.68x	1,150
users	✓	20.75	18.38	1.13x	0	17.57	1.18x	30	18.24	1.14x	0	15.17	1.37x	30
dirname	✓	13.39	11.71	1.14x	0	10.70	1.25x	18	13.27	1.01x	0	9.88	1.36x	18
unexpand	✓	1,824.42	1,187.02	1.54x	2,747,586	1,726.17	1.06x	46	1,797.74	1.01x	0	1,383.94	1.32x	384
as	✓	1,351.43	1,353.75	1.00x	0	1,058.72	1.28x	4,811	648.33	2.08x	2,706	1,038.53	1.30x	5,172
join	✓	1,836.76	1,802.21	1.02x	0	1,535.66	1.20x	21	1,807.86	1.02x	0	1,507.14	1.22x	21
fmt	✓	1,813.96	1,706.96	1.06x	53,664	1,567.08	1.16x	2,622	1,807.46	1.00x	0	1,560.29	1.16x	3,038
csplit	✓	1,808.87	1,828.12	0.99x	68	1,562.78	1.16x	526	1,802.06	1.00x	25	1,570.10	1.15x	786
id	✓	1,863.00	1,851.98	1.01x	0	1,822.29	1.02x	6	1,821.81	1.02x	0	1,621.89	1.15x	6
basename	✓	14.87	13.79	1.08x	0	13.80	1.08x	6	14.59	1.02x	0	13.17	1.13x	6
tr	✓	1,794.26	1,721.41	1.04x	0	1,675.34	1.07x	959	1,665.11	1.08x	4,125	1,592.90	1.13x	1,823
rmdir	✓	1,817.25	1,802.29	1.01x	0	1,640.48	1.11x	1,199	1,821.13	1.00x	0	1,620.63	1.12x	1,199
comm	✓	1,830.82	1,814.43	1.01x	0	1,797.61	1.02x	12	1,824.62	1.00x	0	1,686.12	1.09x	12
seq	✓	1,794.71	1,602.25	1.12x	3,315	1,744.43	1.03x	10	1,652.69	1.09x	0	1,653.37	1.09x	10
base64	✓	1,838.75	1,800.52	1.02x	1,196,109	1,710.60	1.07x	42	1,858.64	0.99x	71,049	1,724.70	1.07x	42
sleep	✓	1,813.49	1,818.46	1.00x	0	1,711.49	1.06x	574	1,818.79	1.00x	0	1,709.58	1.06x	574
link	✓	1,813.22	1,809.15	1.00x	0	1,706.63	1.06x	574	1,815.61	1.00x	0	1,704.86	1.06x	574
dircolors	✓	1,825.17	1,790.52	1.02x	481,704	1,745.48	1.05x	12	1,814.46	1.01x	0	1,731.96	1.05x	12
mkfifo	✓	1,804.43	1,825.65	0.99x	0	1,715.44	1.05x	574	1,823.02	0.99x	0	1,719.93	1.05x	574
expand	✓	1,825.98	1,818.62	1.00x	0	1,775.52	1.03x	181	1,815.85	1.01x	0	1,775.12	1.03x	181
wc	✓	1,835.07	1,847.77	0.99x	0	1,766.08	1.04x	16	1,789.70	1.03x	0	1,775.23	1.03x	16
sort	✓	1,825.29	1,791.46	1.02x	0	1,769.29	1.03x	1	1,777.55	1.03x	0	1,769.31	1.03x	1
mknod	✓	1,808.77	1,802.50	1.00x	0	1,781.57	1.02x	118	1,814.71	1.00x	0	1,748.34	1.03x	118
ln	✓	1,848.82	1,875.80	0.99x	0	1,794.97	1.03x	368	1,818.30	1.02x	0	1,808.79	1.02x	368
fold	✓	1,839.19	1,847.24	1.00x	0	1,819.46	1.01x	2	1,810.92	1.02x	2	1,812.61	1.01x	2
setuidgid	✓	1,805.31	1,835.57	0.98x	534	1,777.02	1.02x	85	1,817.13	0.99x	0	1,783.46	1.01x	85
whoami	✓	1,810.00	1,844.74	0.98x	0	1,804.21	1.00x	26	1,814.63	1.00x	0	1,794.64	1.01x	26
logname	✓	1,803.89	1,846.51	0.98x	0	1,803.70	1.00x	26	1,817.37	0.99x	0	1,802.20	1.00x	26
readlink	✓	1,849.33	1,875.24	0.99x	0	1,810.44	1.02x	6	1,821.32	1.02x	0	1,849.94	1.00x	6
tty	✓	1,816.64	1,860.85	0.98x	0	1,819.59	1.00x	12	1,820.17	1.00x	0	1,818.15	1.00x	29
ld	✗	Unoptimized run experiences solver timeout												
ptx	✗	Unoptimized run concretizes symbolic data												

and 7.78x respectively. Both programs contain complex nested reads on large arrays, where the benefits of our transformations are exponential (§ 2.3). In absolute terms, the runtime of factor and BC decreases from 30 minutes to just 1 and 4 minutes, respectively.

Other examples of programs that greatly benefit from our transformations are tsort, bzip2, expr, ar, and Bandicoot, all with over 3.50x speed-up. All the aforementioned subject programs contain large arrays, which result in complex constraints during symbolic execution. The large improvements in all these subject programs demonstrate that our transformations are successful in causing efficient array constraint encoding and solving.

The results also show that our transformations do not negatively impact performance. Indeed, while there are benchmarks for which we see no (e.g. logname, or tty) or insignificant speed-ups (e.g. 1.01x for whoami), there are no benchmarks on which our combined transformations cause noticeable performance degradation.

The results also show that the different transformations are largely complementary to each other. For example, the dynamic constant folding transformation is mainly responsible for the speed-up in bzip2 and unexpand; the value-based transformation for the speed-up in ar, Bandicoot, and nice; the index-based transformation for the speed-up in expr and factor.

In general, the transformations combine well with each other, with the combined version usually performing better than the individual transformations (24 out of 39 cases). However, there are cases in which the individual transformations perform better. For instance, unexpand and seq show the largest improvement for the dynamic constant folding transformation, ar and as for the value-based transformation, and csplit for the index-based transformation.

In the cases of unexpand and seq, when the individual transformations are applied, value-based transformation and index-based transformation get precedence over the dynamic constant folding. In those circumstances, the array reads are transformed before constant folding could be applied. Given the results obtained in the evaluation, we plan to extend our dynamic constant folding transformation to handle such missed opportunities.

In the cases of ar, as, and csplit, the performance degradation by the combination happens when the cases where index-based transformation can be applied are a super set of those where value-based transformation can be applied. Hence in the combined transformation, these cases went through both transformations, but yielded the benefits of just one transformation. We plan to develop a lightweight analysis to identify undesirable interference.

5.4 RQ3: Significance

We evaluate the significance of our transformations by analyzing the improvement in code coverage achieved on programs where the performance of symbolic execution is limited due to complex constraints over arrays. We use the combined transformation as it is the most generally applicable, and obtains the largest increases in performance. We conduct the experiment on BC.

We run KLEE unmodified and KLEE with the combined transformations for 6 hours each and measure the resulting code coverage. In this experiment, we use the default search strategy of KLEE, which employs heuristics to steer execution towards uncovered code. To allow symbolic execution to achieve significant coverage, we need to use large symbolic buffers as input for BC. However, it is well-known that large symbolic buffers cause search space explosion. To limit the search space, we concretize portions of the symbolic input buffers. This strategy is very useful when performing analysis on large real-world applications as it essentially allows the user to guide the analysis towards specific parts of code they are interested in testing. For example, this modification allows us to explore code in BC responsible for parsing function and array declarations. Such code is only executed when specific keywords of the BC language are used in a specific order. It is thus unlikely that pure random exploration of the program would reach these functions in a reasonable amount of time.

Our results show that our combined transformations significantly improve statement coverage from 35% to 59% (+68%), and branch coverage from 39% to 67% (+71%). The unmodified KLEE run quickly reaches code portions with large and complex constraints over arrays. As a result, solving constraints becomes extremely expensive, and the constraint solver reaches the timeout limit 10 times even with a budget of 200 seconds. Conversely, KLEE with our transformations benefits from the optimized array constraint encoding and never suffers from timeouts.

This experiment on BC and the results obtained on ld and ptx in Section 5.3 support our hypothesis that our transformations indeed improve the performance of constraint solving over arrays, and enable symbolic execution to explore parts of the code that would be unreachable otherwise.

5.5 Threats to Validity

We acknowledge potential problems that might limit the validity of our experimental results. Here we briefly discuss the countermeasures we adopted to mitigate such threats. The internal validity depends on the correctness of our prototype implementation, and may be threatened by the evaluation setting and the execution of the experiments. We carefully tested our prototype with respect to the original KLEE baseline. To validate the correctness of our transformation we also added research question RQ1.

Threats to external validity may derive from the selection of case studies. We validated our transformations on 57 real-world subjects. Different results could be obtained for different subjects. By construction, our approach does not produce any valuable result on systems that do not include any arrays at all, or symbolic arrays only. However, our experiments show that our transformations do not harm performance in these cases.

6 RELATED WORK

Constraint solving is widely-acknowledged as one of the main scalability challenges in symbolic execution [8, 9]. Prior work has proposed several constraint solving optimizations, such as simplifying expressions via standard arithmetic transformations [7, 26], restricting the domain of formulas to eliminate potentially irrelevant constraints [14], splitting constraints into independent subsets [7], caching query results [7], caching solutions/counterexamples and exploiting subset/superset relationships between constraint queries [6], reusing solutions across runs [30] or even between symbolic execution engines [1, 20, 28], and employing a portfolio of constraint solvers [25].

Prior work also studied the impact on symbolic execution of both semantics-preserving and semantics-altering transformations. Our work is inspired by the idea paper [5] and is orthogonal to the approaches in [13, 29], with which it could be combined.

While the optimizations previously proposed can benefit constraints involving arrays, none of them are specifically targeted toward such constraints. As far as we know, we are the first to propose optimizations focused on constraints involving arrays, operating at the level of the symbolic execution engine. However, optimizations targeting arrays have been proposed at the level of SMT solvers. In particular, the STP solver [7, 15] proposed two such optimizations, one targeting arrays with constant indexes and the other arrays with symbolic indexes. Note that KLEE uses STP as its default solver (and we used STP during our experiments) so it already benefits from these optimizations. Nevertheless, our transformations provide additional speed-ups that are out of reach at the solver level because program semantics and execution contexts are largely invisible to the solver. By the time the arrays are sent to the solver, it is to some extent already too late to optimize the query—the extra constraints communicating the array values have already been created.

7 CONCLUSION

Arrays are prevalent in real-world code, and represent one of the main challenges in constraint solving for symbolic execution. In this paper, we propose a novel technique to speed-up array constraints, based on transformations that replace array operations with simpler operations on their indexes and values. As such, the expensive array theory is not triggered inside the solver, leading to substantial reductions in constraint solving time. Our results on a large set of real-world programs show that our technique is applicable to a significant portion of these programs and the performance gains can be as large as 27x. This in turn enables testing of parts of the code which would be unreachable without our transformations.

ACKNOWLEDGMENTS

We thank Frank Busse, Timotej Kapus, Dan Liew, and Luís Pina for their useful comments. This work was supported in part by the EPSRC through grants EP/N007166/1 and EP/L002795/1.

REFERENCES

[1] A. Aquino, F. A. Bianchi, C. Meixian, G. Denaro, and M. Pezzè. Reusing constraint proofs in program analysis. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'15)*, July 2015.

[2] Bandicoot. http://bandilab.org, May 2017.

[3] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proc. of the 14th International Conference on Computer-Aided Verification (CAV'02)*, July 2002.

[4] Bzip2. http://www.bzip.org, May 2017.

[5] C. Cadar. Targeted program transformations for symbolic execution. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering, New Ideas Track (ESEC/FSE NI'15)*, Aug.-Sept. 2015.

[6] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, Dec. 2008.

[7] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically Generating Inputs of Death. In *Proc. of the 13th ACM Conference on Computer and Communications Security (CCS'06)*, Oct.-Nov. 2006.

[8] C. Cadar, P. Godefroid, S. Khurshid, C. Pasareanu, K. Sen, N. Tillmann, and W. Visser. Symbolic Execution for Software Testing in Practice—Preliminary Assessment. In *Proc. of the 33rd International Conference on Software Engineering, Impact Track (ICSE Impact'11)*, May 2011.

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

[10] V. Chipounov and G. Candea. Reverse engineering of binary device drivers with revnic. In *Proc. of the 5th European Conference on Computer Systems (EuroSys'10)*, Apr. 2010.

[11] CLOC - count lines of code. http://cloc.sourceforge.net/, May 2017.

[12] L. De Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the Association for Computing Machinery (CACM)*, 54(9):69–77, Sept. 2011.

[13] S. Dong, O. Olivo, L. Zhang, and S. Khurshid. Studying the influence of standard compiler optimizations on symbolic execution. In *Proc. of the 26th International Symposium on Software Reliability Engineering (ISSRE'15)*, Nov. 2015.

[14] I. Erete and A. Orso. Optimizing constraint solving to better support symbolic execution. In *Proc. of the Workshop on Constraints in Software Testing, Verification, and Analysis (CSTVA'11)*, Mar. 2011.

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

[16] GNU BC. https://www.gnu.org/software/bc, May 2017.

[17] GNU Binutils. https://www.gnu.org/software/binutils, May 2017.

[18] GNU Coreutils. www.gnu.org/software/coreutils, May 2017.

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

[20] X. Jia, C. Ghezzi, and S. Ying. Enhancing reuse of constraint solutions to improve symbolic execution. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'15)*, July 2015.

[21] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, Apr. 2003.

[22] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of the 2nd International Symposium on Code Generation and Optimization (CGO'04)*, Mar. 2004.

[23] T. Liu, M. Araújo, M. d'Amorim, and M. Taghdiri. A comparative study of incremental constraint solving approaches in symbolic execution. In *Proc. of the Haifa Verification Conference (HVC'14)*, Nov. 2014.

[24] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program repair via semantic analysis. In *Proc. of the 35th International Conference on Software Engineering (ICSE'13)*, May 2013.

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

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

[27] A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for an extensional theory of arrays. In *Proc. of the 16th IEEE Symposium on Logic in Computer Science (LICS'01)*, June 2001.

[28] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: reducing, reusing and recycling constraints in program analysis. In *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'12)*, Nov. 2012.

[29] J. Wagner, V. Kuznetsov, and G. Candea. -Overify: Optimizing programs for fast verification. In *Proc. of the 14th (HotOS'13)*, May 2012.

[30] G. Yang, C. S. Păsăreanu, and S. Khurshid. Memoized symbolic execution. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'12)*, July 2012.