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

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, 20, 22, 25, 27]. 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 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 [24, 26]. 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.

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, 14]. 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.

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.
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 \( k = 43 \lor k = 47 \lor k = 48 \lor \ldots \), 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 \( (0 \leq k \leq 42) \lor (44 \leq k \leq 46) \lor \ldots \) 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 [16]. 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 intertwining of the two axioms. For example, the branch condition at line 4 in Figure 1 is encoded in the following way, where \( b64_0, b64_1, \ldots, b64_{255} \) represent the variables associated with the concrete array reads \( b64[0], b64[1], \ldots, b64[255] \) and \( b64_k \) is the variable chosen to represent the symbolic array read \( b64[k] \):

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

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, 28]. For instance the read \( A[j] \) after a write \( A[i] = v \) is encoded as follows, where the uninterpreted functions \( \text{read}(A, i) \) and \( \text{write}(A, i) \) denote reading and writing the \( i \)th element of array \( A \), respectively, and \( \text{rte} \) denotes an if-then-else expression:

\[
\text{read} \left( \text{write}(A, i, v), j \right) \leftrightarrow \text{rte}(i = j, v, \text{read}(A, j))
\]

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 [17] 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
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 \( i \)th and \( j \)th 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 \( check[curClass] != 0 \).

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., \( i \equiv 2 \lor i \equiv 5 \lor \ldots \)) 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 \( 1 \leq currState \leq 579 \). 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, \( equivClass[0] \equiv 0 \), \( equivClass[1 \ldots 8, 11 \ldots 13, \ldots] \equiv 1 \), 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

```c
char equivClass[256] = {0, 1, 1, 20, 12, 13, 1, 1, 1};
char accept[298] = {0, 0, 41, 32, 28, 23, 2, 2, 0};
unsigned check[580] = {0, 1, 1, 141, 142, 144, 297, 297};
unsigned base[302] = {0, 0, 412, 422, 120, 395, 398};
unsigned def[302] = {297, 300, 297, 297, 297, 297};
char meta[54] = {0, 1, 1, 3, 3, 3, 3, 3, 3, 1, 1, 1};
unsigned next[580] = {0, 6, 7, 132, 133, 297, 297};

void tokenMatch(char *input)
{
    unsigned currState = 0;
    char charPtr = input;
    do
    {
        char currClass = equivClass[*charPtr];
        if (accept[currState])
            break;
        lastAcceptState = currState;
        lastAcceptPos = *charPtr;
    }
    while(check[base[currState] + currClass] != currState)
        currState = def[currState];
    if (currState >= 298)
        return;
    currClass = meta[currClass];
    currState = next[base[currState] + currClass] + *charPtr;
    while(base[currState] != 526);
}
```

Figure 2: A simplified parsing routine from BC.
the users. The tool is publicly available and open source.\footnote{https://srg.doc.ic.ac.uk/projects/klee-array/}

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 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.\footnote{https://srg.doc.ic.ac.uk/projects/klee-array/}

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

<table>
<thead>
<tr>
<th>Term</th>
<th>Original Encoding</th>
<th>Transformed Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ec0 = 0 ∧ ... ∧ ec255 = 1 ∧ *charPtr = 0 → currClass = ec0 ∧ *charPtr = 1 → currClass = ec1 ∧ currClass = equival*charPtr = 255 → currClass = ec255</code></td>
<td><code>ite( *charPtr = 0, currClass = 0</code> `ite(1 ≤ *charPtr &lt; 9 → currClass = 1<code> \</code>ite( *charPtr = 9, currClass = 2, `</td>
<td></td>
</tr>
</tbody>
</table>
| Line 13, 1st iteration | : | :
| `currClass = equival*charPtr` | : | :
| `*charPtr = 255 → currClass = ec255` | : | :
| Line 18, 1st iteration | `check0 = 0 ∧ ... ∧ check579 = 297∧ checkcurrClass ≠ 0 ∧ currClass = 0 → checkcurrClass = check0 ∧ currClass = 579 → checkcurrClass = check579` | `1 ≤ currClass ≤ 579` |

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

Figure 3: Array-oriented language.

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 IndexTrans 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 transforms 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 $\sigma: e \rightarrow v$, meaning that given the store $\sigma$, 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 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.

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-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
We propose three kinds of transformations. The first one is dynamic constant folding that is conducted by rules E-CONSTFOLDING, ARR-Rd-Cl, and Arr-Wr-Cl 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.

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 \(0 \leq v_i \leq 1 \lor 4 \leq v_i \leq 5\). 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:

\[
\text{ITE}(0 \leq v_i \leq 1 \lor 4 \leq v_i \leq 5, 0, \text{ITE}(2 \leq v_i \leq 3, 2, 3))
\]

### 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 \(0 \leq v_i \leq 1 \lor 4 \leq v_i \leq 5\). 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

---

**Algorithm 1 Value-Based Transformation**

```
1: function valTrans(A, v_i)
2:   uniqueVal = the set of unique values in A
3:   return rangeItE(uniqueVal, A, v_i)
4: end function
```

**Algorithm 2 Index-Based Transformation**

```
1: function idxTrans(A, op, v_i, c)
2:   R = the index ranges such that
3:   \(\forall v_i \in R, \text{eval}(op, A[v_i], c) = \text{true}\)
4:   C = false
5: for each [lb, ub] \in R do
6:   C = C \lor lb \leq v_i \leq ub
7: end for
8: return C
9: end function
```
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** [17] v1.06, an arbitrary-precision calculator that solves mathematical expressions written in a C-style language.
3. **Binutils** [18] 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** [19] v6.11, a collection of 89 programs for file, text, and shell manipulation.
We validate the correctness of our transformation by checking that with an Intel processor (8 cores) at 3.5 GHz and 16GB of RAM. For those returned by time-bounded search heuristics, concrete memory addresses (as relies on a wide variety of timeouts (e.g. for constraint solving), deterministically across runs. This is hard to achieve since KLEE transformations. The main challenge is to configure KLEE to behave as deterministically as possible, we use depth-first search (DFS) strategy, a sandbox for file descriptors). To force KLEE to behave as deterministically 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.

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 [23] and STP 2.1.2 as the constraint solver [16]. 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.

Table 3: Subjects where our transformations do not apply.

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

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.

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

<table>
<thead>
<tr>
<th>Program</th>
<th>Eq</th>
<th>Baseline</th>
<th>Constant</th>
<th>Index-based</th>
<th>Value-based</th>
<th>Combined</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time (s)</td>
<td>Time (s)</td>
<td>S Inv.</td>
<td>Time (s)</td>
<td>S Inv.</td>
</tr>
<tr>
<td>factor</td>
<td>✓</td>
<td>1.815.21</td>
<td>1.873.34</td>
<td>1.02x 6</td>
<td>1.02x 2</td>
<td>1.02x 2</td>
</tr>
<tr>
<td>bc</td>
<td>✓</td>
<td>1.834.82</td>
<td>1.830.64</td>
<td>1.01x 0</td>
<td>1.01x 0</td>
<td>1.01x 0</td>
</tr>
<tr>
<td>ttort</td>
<td>✓</td>
<td>1.801.62</td>
<td>1.755.57</td>
<td>1.03x 0</td>
<td>1.03x 0</td>
<td>1.03x 0</td>
</tr>
<tr>
<td>bzip2</td>
<td>✓</td>
<td>1.820.49</td>
<td>476.42</td>
<td>3.82x 584</td>
<td>1.01x 0</td>
<td>1.01x 0</td>
</tr>
<tr>
<td>expr</td>
<td>✓</td>
<td>1.799.20</td>
<td>1.819.47</td>
<td>0.99x 1</td>
<td>1.01x 0</td>
<td>1.01x 0</td>
</tr>
<tr>
<td>bandit</td>
<td>✓</td>
<td>1.790.93</td>
<td>1.739.23</td>
<td>1.03x 120</td>
<td>1.03x 120</td>
<td>1.03x 120</td>
</tr>
<tr>
<td>ar</td>
<td>✓</td>
<td>1.800.72</td>
<td>1.795.91</td>
<td>1.00x 0</td>
<td>1.00x 0</td>
<td>1.00x 0</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>57.16</td>
<td>47.84</td>
<td>1.20x 16</td>
<td>1.07x 9</td>
<td>1.07x 9</td>
</tr>
<tr>
<td>nice</td>
<td>✓</td>
<td>1.799.79</td>
<td>1.162.12</td>
<td>1.55x 1244</td>
<td>1.02x 22</td>
<td>1.02x 22</td>
</tr>
<tr>
<td>od</td>
<td>✓</td>
<td>1.823.44</td>
<td>1.546.20</td>
<td>1.18x 2813</td>
<td>1.38x 824</td>
<td>1.38x 824</td>
</tr>
<tr>
<td>users</td>
<td>✓</td>
<td>20.75</td>
<td>18.38</td>
<td>1.13x 18</td>
<td>1.18x 30</td>
<td>1.18x 30</td>
</tr>
<tr>
<td>dinname</td>
<td>✓</td>
<td>13.39</td>
<td>11.71</td>
<td>1.14x 0</td>
<td>1.25x 18</td>
<td>1.25x 18</td>
</tr>
<tr>
<td>unexpand</td>
<td>✓</td>
<td>1.824.42</td>
<td>1.187.02</td>
<td>1.54x 2746</td>
<td>1.06x 46</td>
<td>1.06x 46</td>
</tr>
<tr>
<td>as</td>
<td>✓</td>
<td>1.351.43</td>
<td>1.353.75</td>
<td>1.00x 0</td>
<td>1.28x 411</td>
<td>1.28x 411</td>
</tr>
<tr>
<td>join</td>
<td>✓</td>
<td>1.836.76</td>
<td>1.802.21</td>
<td>1.02x 0</td>
<td>1.20x 21</td>
<td>1.20x 21</td>
</tr>
<tr>
<td>fnt</td>
<td>✓</td>
<td>1.813.96</td>
<td>1.706.96</td>
<td>1.06x 5363</td>
<td>1.16x 262</td>
<td>1.16x 262</td>
</tr>
<tr>
<td>csplit</td>
<td>✓</td>
<td>1.808.87</td>
<td>1.828.12</td>
<td>0.99x 68</td>
<td>1.16x 526</td>
<td>1.16x 526</td>
</tr>
<tr>
<td>id</td>
<td>✓</td>
<td>1.863.00</td>
<td>1.851.98</td>
<td>1.01x 0</td>
<td>1.02x 6</td>
<td>1.02x 6</td>
</tr>
<tr>
<td>basename</td>
<td>✓</td>
<td>14.87</td>
<td>13.79</td>
<td>1.08x 0</td>
<td>1.08x 0</td>
<td>1.08x 0</td>
</tr>
<tr>
<td>tr</td>
<td>✓</td>
<td>1.794.21</td>
<td>2.741.41</td>
<td>1.00x 0</td>
<td>1.34x 959</td>
<td>1.34x 959</td>
</tr>
<tr>
<td>rmdir</td>
<td>✓</td>
<td>1.817.25</td>
<td>1.802.29</td>
<td>1.01x 0</td>
<td>1.11x 1199</td>
<td>1.11x 1199</td>
</tr>
<tr>
<td>comm</td>
<td>✓</td>
<td>1.830.82</td>
<td>1.814.43</td>
<td>1.01x 0</td>
<td>1.02x 12</td>
<td>1.02x 12</td>
</tr>
<tr>
<td>seq</td>
<td>✓</td>
<td>1.794.71</td>
<td>1.802.22</td>
<td>1.12x 335</td>
<td>1.03x 10</td>
<td>1.03x 10</td>
</tr>
<tr>
<td>base64</td>
<td>✓</td>
<td>1.838.75</td>
<td>1.800.52</td>
<td>1.196.109</td>
<td>1.10x 42</td>
<td>1.10x 42</td>
</tr>
<tr>
<td>sleep</td>
<td>✓</td>
<td>1.813.49</td>
<td>1.848.46</td>
<td>1.00x 0</td>
<td>1.11x 574</td>
<td>1.11x 574</td>
</tr>
<tr>
<td>link</td>
<td>✓</td>
<td>1.813.22</td>
<td>1.809.15</td>
<td>1.00x 0</td>
<td>1.06x 574</td>
<td>1.06x 574</td>
</tr>
<tr>
<td>dircolors</td>
<td>✓</td>
<td>1.825.17</td>
<td>1.796.52</td>
<td>1.02x 4837</td>
<td>1.05x 12</td>
<td>1.05x 12</td>
</tr>
<tr>
<td>mknod</td>
<td>✓</td>
<td>1.804.43</td>
<td>1.825.69</td>
<td>0.99x 0</td>
<td>1.14x 554</td>
<td>1.14x 554</td>
</tr>
<tr>
<td>ln</td>
<td>✓</td>
<td>1.848.82</td>
<td>1.875.80</td>
<td>0.99x 0</td>
<td>1.03x 368</td>
<td>1.03x 368</td>
</tr>
<tr>
<td>fold</td>
<td>✓</td>
<td>1.839.19</td>
<td>1.847.24</td>
<td>1.00x 0</td>
<td>1.10x 2</td>
<td>1.10x 2</td>
</tr>
<tr>
<td>setuidgid</td>
<td>✓</td>
<td>1.805.31</td>
<td>1.855.57</td>
<td>0.98x 534</td>
<td>1.02x 85</td>
<td>1.02x 85</td>
</tr>
<tr>
<td>whoami</td>
<td>✓</td>
<td>1.810.00</td>
<td>1.844.74</td>
<td>0.98x 0</td>
<td>1.00x 26</td>
<td>1.00x 26</td>
</tr>
<tr>
<td>llogin</td>
<td>✓</td>
<td>1.803.89</td>
<td>1.864.51</td>
<td>0.98x 0</td>
<td>1.00x 26</td>
<td>1.00x 26</td>
</tr>
<tr>
<td>readdir</td>
<td>✓</td>
<td>1.849.33</td>
<td>1.875.24</td>
<td>0.99x 0</td>
<td>1.02x 6</td>
<td>1.02x 6</td>
</tr>
<tr>
<td>tty</td>
<td>✓</td>
<td>1.816.64</td>
<td>1.860.85</td>
<td>0.98x 0</td>
<td>1.00x 12</td>
<td>1.00x 12</td>
</tr>
</tbody>
</table>

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., llogin, 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, the combined 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 counter-measures 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, 15], restricting the domain of formulas to eliminate potentially irrelevant constraints [15], splitting constraints into independent subsets [7], caching query results [7], caching solutions/counterexample amplifying and exploiting subset/superset relationships between constraint queries [6], reusing solutions across runs [31] or even between symbolic execution engines [1, 21, 29], and employing a portfolio of constraint solvers [26].

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, 30], 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 [16] 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 not 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


