Indexing Operators to Extend the Reach of Symbolic Execution

EARL T. BARR, CREST, University College London, UK
DAVID CLARK, CREST, University College London, UK
MARK HARMAN, CREST, University College London, UK
ALEXANDRU MARGINEAN, CREST, University College London, UK

Traditional program analysis analyses a program language, that is, all programs that can be written in the language. There is a difference, however, between all possible programs that can be written and the corpus of actual programs written in a language. We seek to exploit this difference: for a given program, we apply a bespoke program transformation (INDEXIFY) to convert expressions that current SMT solvers do not, in general, handle, such as constraints on strings, into equisatisfiable expressions that they do handle. To this end, INDEXIFY replaces operators in hard-to-handle expressions with homomorphic versions that behave the same on a finite subset of the domain of the original operator, and return \( \bot \) denoting unknown outside of that subset. By focusing on what literals and expressions are most useful for analysing a given program, INDEXIFY constructs a small, finite theory that extends the power of a solver on the expressions a target program builds.

INDEXIFY’s bespoke nature necessarily means that its evaluation must be experimental, resting on a demonstration of its effectiveness in practice. We have developed INDEXIFY, a tool for INDEXIFY and released it publicly. We demonstrate its utility and effectiveness by applying it to two real world benchmarks — string expressions in coreutils and floats in fdlibm53. INDEXIFY reduces time-to-completion on coreutils from Klee’s 49.5m on average to 6.0m. It increases branch coverage on coreutils from 30.10% for Klee and 14.79% for Zesti to 66.83%. When indexifying floats in fdlibm53, INDEXIFY increases branch coverage from 34.45% to 71.56% over Klee. For a restricted class of inputs, INDEXIFY permits the symbolic execution of program paths unreachable with previous techniques: it covers more than twice as many branches in coreutils as Klee.

Additional Key Words and Phrases: Symbolic Execution, Reliability, Testing

1 INTRODUCTION

Symbolic execution (symex) supports reasoning about all states along a path. It is limited by its solver’s ability to resolve the constraints that occur in a program’s execution. Different types of symex handle intractable constraints differently. Broadly, static symbolic execution abandons a path upon encountering an intractable constraint; dynamic symbolic execution concretises the variables occurring in the constraint and continues execution, retaining the generality of symex only in the variables that remain symbolic. In this paper, we improve the responses of symex for a specific program, allowing it to continue past intractable constraints without resorting to fully concretising the variables involved in that constraint.

Our approach is INDEXIFY, a general program transformation framework that re-encodes intractable expressions, in any combination of types, into tractable expressions. This is impossible in general but can be achieved to a limited (and varying) degree for any particular program, so we characterise INDEXIFY as program-centric. To transform a program, INDEXIFY homomorphically maps a finite subset of the program’s algebra of expressions to an algebra of indices (or labels), augmented with the undefined value \( \bot \).
**Fig. 1.** Standard dynamic symex compared to indexified symex; the control flow graph (CFG) node is a control point and the solver returns unknown on queries containing $\alpha$.

Indexify is a program transformation that rewrites its input program to replace operators with versions that 1) are restricted to $G$, a finite subset of their original domain and range, and 2) take and return indices over $G$. Let $\text{indexOf}$ map $G$ to $\mathbb{N}$. When Indexify replaces the operator $f$, its finite replacement $\check{f}(\text{indexOf}(x)) = \text{indexOf}(f(x))$ if $x \in G$ and $\perp$ otherwise. To build $\check{f}$, Indexify memoises the computation of $f$ over $G$. To this end, Indexify takes an input $P$, a (small) set of types $T$, and (small) set of literals, $S$, of type $T$, as seeds. In this work, we harvest constants from the program as seeds, as we explain in Section 5.1. It takes two sets of operators $B$ and $F$, not necessarily distinct, where $F$’s elements occur in $P$ and may produce intractable constraints. Indexify transforms a program $P$ in stages. First, Indexify repeatedly and recursively applies the operators in $B$ to expand $S$ to a larger set, $G$, the “Garden”. Then, it memoises $f \in F$ over $G$ to produce $\check{f}$. Finally, it rewrites $P$ to use the memoised versions of $F$.

Consider Figure 1, which depicts dynamic symbolic execution (DSE). In the figure, $\alpha$ is intractable, so the solver returns unknown when queried about $\alpha$. On the left, DSE replaces the free variables in $\alpha$ with concrete values, either drawn from a concrete execution that reaches $\alpha$ (concolic [Marinescu and Cadar 2012]) or generated using heuristics [Chen et al. 2013], collapsing the state space of $\alpha$’s variables to those concrete values, but allowing symex to proceed over the rest of the state space.

On the right, we see Indexify in action on a transformed program $P$. Over $\alpha$’s free variables, the program is restricted to the garden $G$. An indexed expression, like $\alpha$, that contains $\check{f}$ evaluates to $\perp$ when it takes as an argument either $\perp$ or an unindexed value or it evaluates to an unindexed value. Indexed operators like $\check{f}$ are lookup tables that generate constraints in equality theory as we show in Section 4.3. Indexify’s transformation permits symbolic reasoning over the algebra of indices at the cost of explicit ignorance, reified in $\perp$, about values outside the index set; it allows the symbolic execution of the the indexed program over a (previously intractable) subset of the original program’s state space, permitting the symbolic exploration of previously unreachable regions of that state space. If we indexify the problematic operators in $\alpha$, symex can continue, restricted to $G$, past $\alpha$ in Figure 1. Once we apply Indexify to a program, the problem becomes identifying a useful set of expressions whose indexification might improve our ability to find bugs.

An indexed program under-approximates the semantics of the original program in the following way: if a transformed program’s execution stays entirely within the indexed subset of values and operators, its output is either an index that is an image under the homomorphism of the original program’s output or it is undefined (returns $\perp$). The under-approximation of the semantics and its dependency on the choice of the algebra to indexify again emphasises the
typedef enum {NVidia, NVidiaCorporation, ...} string_enum;
static const char * Nv11Vendor = "NVidia Corporation";
static const string_enum Nv11Vendor = vendor11;

BOOLEAN IsVesaBiosOk(...){
... 
if (!(strncmp(Vendor, Nv11Vendor, sizeof(Nv11Vendor))))
if (!(i_strncmp(Vendor, Nv11Vendor, sizeof(Nv11Vendor))))
assert(strncmp(Vendor, Nv11Vendor, MAXLEN) == 0);
assert( i_strncmp(Vendor, Nv11Vendor, MAXLEN) == 0);
}

Fig. 2. Diff of buggy code from ReactOS, after INDEXIFY: the assertion, which we added to reify the bug, at line 55/56 can fail; this bug is difficult to reach under either pure or concolic symbolic execution; INDEXIFY reaches and triggers the bug by restricting variables to a finite set of indices (adding line 47) and by replacing types, constants (lines 48/49), and operators (lines 53/54, and 55/56) to work with these indices.

Garden: \[ G = \{ \emptyset, N, Y, i, d, ..., NV, uN, ... \} \]
IOT: \[
\text{i_strncmp}(\text{const string_enum lhs, const string_enum rhs, int count})\
\text{if}(\text{lhs} == \text{vendor} \&\& \text{rhs} == \text{vendor11} \&\& \text{count} == 4 \text{ return 0;})\
\text{else if}(\text{lhs} == \text{vendor} \&\& \text{rhs} == \text{vendor11} \&\& \text{count} == 8) \text{ return 1;})\
\ldots \text{return -1;} // -1 represents \bot
\]

Fig. 3. The garden, and the indexed operator tables IOTs (indexed initial operators) that INDEXIFY generates for the code snippet in Figure 2. We use the Kleene Closure to build this example: \( G \) is the Kleene Closure of all the strings in ReactOS. We build the IOT \( \text{i_strncmp} \) for the operator \( \text{strncmp} \). Section 4 describes how INDEXIFY achieves the above.

bespoke nature of the transformation and its dependence on the goodness of the choice. As we demonstrate later through examples, in practice, it is not difficult to make a good choice. Our main contributions follow

(1) The introduction and formalisation of a general framework for restricting operators to produce tractable constraints, allowing symbolic execution to explore some paths previously only concretely reachable (Section 3);
(2) The realisation of our framework in the tool INDEXIFY (Section 4) available at <url>;
(3) Comprehensive demonstrations of INDEXIFY’s utility (Section 6): we compare INDEXIFY with dynamic symbolic execution, i.e. Klee [Cadar et al. 2008], and concolic testing, i.e. Zesti [Marinescu and Cadar 2012]. We show that INDEXIFY achieves 66.83% branch coverage, compared to Klee’s 30.10% and Zesti’s 14.79%, on coreutils, and does this within less than a third of the time that Klee without INDEXIFY requires on average. Finally, we show it reaches bugs that Klee alone does not on a famous C bug finding benchmark.

2 MOTIVATING EXAMPLE

Figure 2 presents a code fragment that contains a real world bug in ReactOS [Developers 2016b], an open-source operating system. Commit 5926581, on 21.12.2010, changes the type of variable Nv11Vendor from array to pointer. This causes the sizeof operator to return the size of a pointer, not the length of the array. ReactOS developers fixed this bug in commit 30818df, on 18.03.2012, after ReactOS’s developer mailing list discussed it. The developers fixed the bug once after three months and it was committed.

\[1\]https://reactos.org/archives/public/ros-dev/2012-March/015516.html
Figure 2 shows only the relevant code, after using INDEXIFY, in diff format. Although INDEXIFY operates at LLVM bitcode level, we use source code here for clarity. INDEXIFY adds line 47 to define indices as an enum. Figure 3 shows the garden, and the indexed version of the operator strncmp (i_strncmp). More details about the garden, the indexed operators, and how we build them, are in Section 4.

The bug is a classic error: line 57 (lines 53/54 in Figure 2) in vbe.c of ReactOS, commit 30818df, incorrectly applies sizeof() to a string pointer, not strlen() [Wagner et al. 2000]. As a result, strncmp() compares only the first 4 characters of its operands, assuming a 32 bit pointer. A pair of strings, whose first four characters are identical then differ afterwards in at least one character, triggers the bug when bound to ‘Vendor’ and ‘Nv11Vendor’: the if condition on lines 53/54 wrongly evaluates to true and assertion on lines 55/56 fails. This bug is an error in a string expression and the theory of strings is undecidable, when the string length is unbounded [Quine 1946]. Thus, most string solvers return UNKNOWN on this constraint [Bjørner et al. 2009].

Static symbolic execution must content with intractable constraints. CBMC, for instance, errors on them [Kroening and Tautschnig 2014] and would not reach the assertion on line 55. Klee [Cadar et al. 2008] implements dynamic symbolic execution and uses bit-blasting. When Klee reaches the if on line 53, Klee internalizes strncmp to bitblast it. The strncmp function loops over the length of strings, causing Klee’s default solver to time out with its default settings (1 minute time-out). Thus, Klee does not produce an input that triggers this bug. Concolic testing searches a neighbourhood around the path executed under its concrete inputs. Upon reaching exit, concolic testing backtracks to the nearest condition, complements it, then restarts execution from the entry point [Godefroid et al. 2005; Sen et al. 2005a]. Thus, concolic testing can reach this bug only if it is given a concrete input in the neighbourhood of this bug. Unlike concolic testing, which is tethered to a single concrete execution, INDEXIFY can symbolically reason over all the values in its finite set G, broadening its exploration relative to concolic testing.

INDEXIFY finitizes operators in undecidable theories by transforming them into finite lookup tables over values of interest thereby converting potentially undecidable expressions into decidable ones. To build G, the set of interesting values, INDEXIFY harvests the constants in a program, such as the ones in Figure 2, as seeds, then concretely and repeatedly applies operators, such as Kleene closure [Kleene 1951], to the constants, up to a bounded number, to expand the set. From the constants in Figure 2, this process produces NVidia and NVidiaCorporation among others. The lookup table for strncmp memoize the concrete results of repeatedly evaluating it on pairs drown from G. In Figure 2, to index the strings INDEXIFY introduces the enum on line 47, then, on lines 48–49, it changes Nv11Vendor’s type to int and replaces the constant to which it is initialized to one of the indices introduced on line 47. INDEXIFY then replaces strncmp with i_strncmp on lines 53/54 and 55/56. Indexing these and building i_strncmp, the memoised looking table for strncmp over the values in G is sufficient to violate the assertion at line 55/56.

Figure 3 shows the garden G: the extended set of constants that we consider in our analysis; and the indexed operator table (IOT) i_strncmp, the memoization table for the operator strncmp. We discuss these concepts in details in Section 3 and in Section 4. i_strncmp contains entries for all the values in G. If i_strncmp gets parameters that are out of G we abandon the path and return ⊥. This means that the values that flowed into i_strncmp are out of our analysis.

Klee times out on the strncmp operator. When Klee runs on the indexified version of the code in Figure 2 it successfully executes the indexed strncmp operator: i_strncmp; and produces a bug triggering input. The bug-triggering constraint that INDEXIFY generates is: Vendor = 0 ∧ Nv11Vendor = 1. The solver produces the values: Vendor = 0 and Nv11Vendor = 1. Under these inputs, i_strncmp returns 0 signalling that the strings are equal. The assertion on line 56 fails, as the strings are equal only in the first 6 character but differ afterwards.
3 APPROACH

The concept of Indexify is quite general. It aims to transform a program so as to generate tractable constraints at some points at which it had previously generated intractable or undecidable constraints by restricting these constraints to a simpler theory over a finite set of values, augmented with the unknown value, ⊥. The resulting, transformed constraints should be satisfiable whenever the original constraints were satisfiable, be more tractable with regard to satisfaction, and sometimes be satisfiable when the original constraints were not; but only when restricted to the finite set of chosen values and operators; crucially, SMT solvers can handle them efficiently. This simpler theory turns out in practice to depend on the way in which Indexify is implemented, although in each case the overall approach is the same.

To reiterate: for a given type or set of types we identify a useful set of literals and a desired set of operators on the literals, then memoize the outcomes of all combinations of applications of the operators on the literals — but only up to a limit, k, on the number of applications in any one expression. This can be represented simply as a finite set of index tables, one for each operator of interest. Since the useful set of literals is not necessarily closed under applications of the desired set of operators, we need to enter undefined for some entries in the tables. The final step is to perform a program transformation by replacing all the members of the memoized sets of literals and operators, as they occur in the program syntax, with their indexified versions. The effect from the point of view of constraint solving is that of shifting between logical theories. Solving constraints containing indexified operators can, as a result, use a more tractable theory such as equality.

This approach can be applied to any source theory, but, for the purposes of presentation and examples in the present paper, we restrict ourselves to string expressions and their operators. All non-trivial fragments of theory of strings are NP-complete [Jha et al. 2009], and thus, string constraints are intractable, making Indexify highly useful.

3.1 Terminology

Logical theories can be viewed as algebras. In universal algebra, an algebra is an algebraic structure, that is, a set of literals and a set of operators on the literals, together with a set of axioms that collectively play the role of laws for the algebra. Sometimes the notion of an algebraic structure is simplified to just a set of operators and the literals appear as nullary operators. In what follows, we explain in detail the soundness requirement for the transformation in the program syntax, i.e. that it must be a partial homomorphism between two algebraic structures. This does not map logical laws between the algebras. In our setting, there is not necessarily a homomorphism for the logical laws; to see why consider that laws of the naturals and strings.

To elaborate, in order to be sound, we require that the result of applying a transformed operator to transformed arguments yields the same result as applying the original operator to the original arguments and then transforming the result, whenever the result is defined in the transformed program. This is the minimum guarantee we should expect. Without it we could (unsuccessfully) transform any type to any type, any operator to any operator, e.g. strings to integers and replace operations on strings with operations on integers arbitrarily. In this section, we specify the behaviour of the homomorphism on the operators, then show that the algorithm for the transformation constructs a homomorphism of this kind. Finally, the transformation is implemented as a rewrite system on the syntax of the program.

It would be useful to be able to show that, whenever a transformed constraint has a model in the target theory, the untransformed version either has a model (but not necessarily the same model) in the source theory or is not satisfiable in the source theory. A proof of this would rely on properties of the individual theories and is left outside of the scope
Assuming that the reason for this is that, in our experiments, only 2% of the applications of the Indexify function grow quickly:

Despite this prodigious growth rate, Indexify works well in practice given a small $B$, as we show in Section 6.1. One reason for this is that, in our experiments, only 2% of the applications of $f \in B_+$ produced a new value.

**Indexifying a Set of Operators:**

Considering the set of literal values and the set of operators on them that may occur in a program, we can partition each set into those that we indexify and those that we do not. Those that we do not indexify are left untouched by the transformation and on these the homomorphism is simply the identity. For simplicity, we require that there is no interaction between the untouched parts and the transformed values and operators. In consequence, once we identify a set of operators to indexify, we must also indexify a "sufficiently large set of values" which are of the input and output types for this chosen set of operators. We could make other choices with regard to the relationship between the indexified and unindexified operators and values but this is the simplest choice.

**Constructing a “Garden” of Literal Values to Indexify:** Here, we formally present how we target a set of type literals and a set of operators to expand the initial set (seeds) into a larger set of literals $G$ (garden).

We begin our description with some useful notation for types and operators. Let $X$ be a set of operators. Denote the subset of nullary operators (literals) of $X$ as $X_0$ and the set of non-nullary operators by $X_+ = X - X_0$. We will henceforth use the subscripts $0$ and $+$ to indicate sets of literals and sets of non-nullary operators respectively. Let $H$ be a function that takes a set and returns a set of the same type. Use $H^k(X)$ to mean the recursive application of $H$ $k$ times to the set $X$, so that $H^0(X) = X$ and $H^k(X) = H^{k-1}(H(X))$.

Suppose we have a program $P$ and want to indexify some of the operators that occur in $P$. Let $T$ be the set of types in $P$ and $\oplus$ be the set of operators used in $P$. Initially we have in mind a set of (non-nullary) operators of interest, ones out of which are presumably potentially problematic for SMT solvers. We first select a set, $S = B_0$, of nullary operators (literals) whose types are basic types that include all the argument and return types of these operators of interest. We call this set the seeds. Then we select $B_+$, a set of non-nullary operators on these seeds that we use to build a larger set of literals, the garden $G$. $B_+$ is not restricted to $\oplus$ and does not necessarily contain any of the operators of interest.

Each literal in $S = B_0$ has type $\tau$, where $\tau = \tau_1 \uplus \tau_2 \uplus \cdots \uplus \tau_n$. In other words, $\tau$ is a disjoint union of the types of nullary operators and each literal has one of those types.

Let $f \in B_+ \Rightarrow f : \tau \rightarrow \cdots \rightarrow \tau \rightarrow \tau$, i.e. the argument and return types of $f$ are in $\tau$.

We define a function, $H : \tau \rightarrow \tau$ on a set of nullary operators, $Z$, as follows.

$$H(Z) = Z \cup \{ f(x_1, x_2, \ldots, x_n) \mid f \in Z, x_1 \in Z_0, \hat{f}(x_1, x_2, \ldots, x_n) \text{ is defined} \} \neq \perp$$

For simplicity of presentation, we have ignored all the "side information" about elements of $Z$ as arguments to $H$ in the type of $H$ so as to focus on its application as an iterative step in growing the garden. We can then define $G$, the garden resulting from $k$ applications of $H$, as

$$G = G_k = H^k(B_0) \quad (1)$$

Assuming $\tau = \tau'$ and that every possible non-nullary operator application to nullary operators is defined and returns a fresh literal, $G$ grows quickly:

$$|G_k| = \sum_{m=2}^{k+2} \sum_{f \in B_+} \left( \frac{|G_{m-1}|}{\text{arity}(f)} \right) - \left( \frac{|G_{m-2}|}{\text{arity}(f)} \right) \quad (2)$$

Despite this prodigious growth rate, Indexify works well in practice given a small $B$, as we show in Section 6.1. One reason for this is that, in our experiments, only 2% of the applications of $f \in B_+$ produced a new value.
\[ \tau \in T \] \quad \Rightarrow \quad \text{”in } \hat{x}” \\
\[ l \in G \] \quad \Rightarrow \quad \text{”T”} \\
\[ l \not\in G \land \text{typeof}(l) \in T \] \quad \Rightarrow \quad \text{”l”} \\
\[ f \in F_+ \] \quad \Rightarrow \quad \text{”f(a_1, \ldots, a_i, \ldots)”} \\
\[ \hat{f} \in \hat{F}_+ \land \exists a_i \text{ s.t. } \neg \delta^7(a_i) \] \quad \Rightarrow \quad \text{”\hat{f}(a_1, \ldots, a_i)”} \\
\[ f \in F_+ \land \exists a_i \text{ s.t. } \delta^7(a_i) \] \quad \Rightarrow \quad \text{”f(a_1, \ldots, a_i)”} \\
\[ f \not\in F_+ \land \exists a_i \text{ s.t. } \delta^7(a_i) \] \quad \Rightarrow \quad \text{”x := f(a_1, \ldots, a_n)”} \\
\[ \hat{f} \in \hat{F}_+ \land \neg \delta^7(x) \land \exists a_i \text{ s.t. } \neg \delta^7(a_i) \] \quad \Rightarrow \quad \text{”x := f(a_1, \ldots, a_n)”} \\

Equation 7 to Equation 10 rewrite function calls, which lack type annotations. For this reason, they do not overlap with Equation 3. They handle flows between indexed regions of the programs and unindexed ones. Equation 7 wraps unindexed arguments to an indexed call in \( \delta \) to convert them to indices. Equation 8 it is the complement of Equation 7: it unindexes indexed variables that flow into unindexed functions. There are four different combinations of the two conditions in Equation 7 and Equation 8. The two combinations that we treat are flows across indexed to unindexed conditions in Equation 7 and Equation 8. The two combinations that we treat are flows across indexed to unindexed variables that flow into unindexed functions. There are four different combinations of the two

---

Fig. 4. \( \Phi_S \), Indexify’s rewriting schema: \( x \in F; \hat{x} \in \hat{F}; \hat{f} \in \hat{F}_+ \). We underline the redexes.

Having described the construction of the garden of literals, \( G \), that becomes indexified inputs and outputs for the indexified operators, we return to the set of operators of interest that occur in \( P \) and that we wish to indexify. Let \( F_0 \) be this set and let \( F = G \cup F_0 \) so that \( G = F_0 \), the set of literals or nullary operators of interest. Note again that \( F_+ \cap B_+ \) may be empty. The index function: specify \( \delta: F \rightarrow \hat{F} \) as an isomorphism that maps operators in \( F \) to fresh names for the indexed version of the operator that takes and returns indices over \( G^2 \). Generally, we cannot index all the operators in a program, because some variables or operators can be defined externally.

Figure 4 shows the rewriting schema that Indexify implements to transform an input program. For \( x \in F \), let

\[ \delta_\perp(x) = \begin{cases} 1 & \text{if } \delta_0(x) = 1 \\ \perp & \text{otherwise} \end{cases} \]

In Figure 4, each \( a_i \) is an argument expression and the \( \delta^7 \) function checks if its argument has been directly indexed via Equation 3 or converted to an index because Equation 7 has wrapped it in a call to \( \delta \); when \( e \) is an expression and \( \hat{x} \in \hat{F} \), its definition is:

\[ \delta^7(t) = \begin{cases} T & \text{if } t = \text{”x”} \lor t = \text{”}\delta_\perp(e)” \\ F & \text{otherwise} \end{cases} \]

Equation 3 indexes variables and function declarations, in the latter case through repeated applications on a function’s arguments and its return type name pair. \( \Phi_S \) only changes the parts in the program that appear in redexes in Figure 4. For the rest of them, the identity is implicit. Equation 4 replaces literals in \( G \) with their index under the homomorphism \( \phi \). Equation 5 handles the error case, where a constant has an indexified type but is not in \( G \) by replacing it with \( \perp \). For each function call on a function to index, \( i.e. \forall f \in F_+ \), Equation 6 replaces the call’s function identifier with the name of its indexed variant.

Equation 7 to Equation 10 rewrite function calls, which lack type annotations. For this reason, they do not overlap with Equation 3. They handle flows between indexed regions of the programs and unindexed ones. Equation 7 wraps unindexed arguments to an indexed call in \( \delta \) to convert them to indices. Equation 8 it is the complement of Equation 7: it unindexes indexed variables that flow into unindexed functions. There are four different combinations of the two conditions in Equation 7 and Equation 8. The two combinations that we treat are flows across indexed to unindexed

---

\(^{\text{a}}\text{We use } \delta \text{ to denote our indexing function, because ”δεικτήσεω” means ”index” in Greek and starts with } \delta; \text{ it replaces the self-explanatory, but longer and less elegant name } indexify \text{ which we used in the introduction.} \)
boundaries. The other two combinations do not require transformation as no flows across indexed and unindexed boundaries occur in them. Equation 9 indexes returns from unindexed \((f \notin F_+)\) functions into indexed variables \((\delta^T(x))\). The conditions \(\exists a_i \text{ s.t. } \delta^T(a_i)\) and \(\exists a_i \text{ s.t. } \neg \delta^T(a_i)\) in these two rules sequence their application, so the last two rules only trigger after exhausting the the first six rules, as \(\delta^T(a_i)\) checks if \(a_i\) has been already indexed. Without the ordering, the rules overlap and we would not be able to prove the confluence of INDEXIFY’s TRS in Theorem 1. Similar to the previous two rules, Equation 10 complements Equation 9: it unindexes returns from indexed functions \((\tilde{f} \in F_+)\) that flow into an unindexed variable \((\neg \delta^T(x))\). As in the previous case, we only need rules for two out of four condition combinations (of the first two conjuncts of the guard), as only in these two we have flows across indexed and unindexed regions.

The schema can interact. Consider Equation 8 and Equation 9. Assume we have a call to \(f \notin F_+\) that returns into a indexed variable and takes two indexed arguments. Two applications of Equation 8 and one of Equation 9, in any order, would rewrite this call. Correctness requires INDEXIFY’s rewriting to be confluent.

**Theorem 1 (Confluence).** All instantiations of the term rewriting schema \(\Phi_S\) into term rewriting systems are confluent.

**Proof Sketch 1.** Of the rules in Figure 4, Equation 3 and Equation 4 share \(G\); Equation 6, Equation 8, and Equation 9 share \(F_+\); and Equation 7 and Equation 10 share \(\tilde{F}_+.\) The other rules cannot overlap because their guards restrict them to distinct sets. Equation 3 and Equation 4 do not overlap because their guards partition \(G\). Similarly Equation 6 is defined over a different part of \(F_+\) than Equation 7 and Equation 8. The guards of Equation 7 and Equation 10 guarantee that they do not overlap: \(\exists a_i \text{ s.t. } \neg \delta^T(a_i)\) for Equation 7 and \(\exists a_i \text{ s.t. } \neg \delta^T(a_i)\) for Equation 8. Finally, \(\delta^T\) prevents Equation 8 from being applied to eligible calls until Equation 7 has rewritten every parameter within it. Thus, none of the rewriting schemas have overlapping terms on the left hand side and \(\Phi_S\) is non-overlapping. \(\Phi_S\) is also left-linear: no variable occurs more than once in the left hand sides of the rules in Figure 4. We use substitute to instantiate the term rewriting schema \(\Phi_S\) into the term rewriting \(\Phi_i\) for a particular program: \(\forall r_S \in \Phi_S, \forall t \in T : \Phi_i = \Phi_S \cup r_S(t/\tau)\). For example, let \(T = \{\text{float, string}\}\). Then we generate the following rewriting rules from the rewriting schema in Equation 3, like “float x” \(\rightarrow “\text{int } \hat{x}”\) and “string x” \(\rightarrow “\text{int } \hat{x}”\). Since \(\Phi_S\) is left-linear and non-overlapping and the substitution does not violate either property, so is \(\Phi_i\). Rosen [Rosen 1973] proved that left-linear and non-overlapping systems are confluent and so, each \(\Phi_i\) is confluent, so \(\Phi_S\) is confluent.

**4 IMPLEMENTATION**

Given a set of operators (or functions) that can form undecidable expressions, INDEXIFY memoizes a finite part of their behaviour, maps the rest to unknown \(\bot\), then transforms a program to use them. The resulting program produces decidable constraints using these indexed operators for a subset of its original state space. We implemented INDEXIFY on top of LLVM [Developers 2016a] for the C language to produce LLVM IR for the symbolic execution (symex) engine Klee [Cadar et al. 2008] to execute. We use Klee’s default solver, STP [Ganesh and Dill 2007], because Dong et al. showed Klee performs best with STP [Dong et al. 2015].

Figure 5 shows the architecture of INDEXIFY. The two main components of INDEXIFY are its **Indexer** and **Rewriter**. To use INDEXIFY, the user specifies \(P\), the program to indexify, and \(F_+\), the signature of the functions (or operators) to index. It is the user’s responsibility to ensure to specify \(F_+\) so that it contains all the operators needed to guarantee that indexify\((P)\) produces decidable constraints, at least along the paths the user wishes to explore. Directly specifying \(F_+\) is tedious. Section 4.4 details how we enable the user to indirectly specify \(F_+\) in a simpler way.

If the user does not provide \(S\), the indexer populates \(S\) with constants in \(P\). Then, it computes \(G\), the set of values to index and builds indexed operator tables. The rewriter rewrites \(P\)’s IR to use indexed operators and values, including
Fig. 5. The architecture of INDEXIFY; $G$, $k$, and $F_\tau$ are optional inputs.

Algorithm 1 \textit{driver}($\delta_0, B_+, F_+, k$) = $\delta'_0, \hat{F}_+$
This algorithm first calls Algorithm 2 to build the garden $G$, then calls Algorithm 3, using $G$ (as carried within $\delta_0$) to build the indexed operator tables.

\textbf{Input:} $\delta_0$, an indexed set of nullary operators
$B_+$, a set of non-nullary functions for building $G$
$F_+$, a non-nullary set of functions to index
$k$ limits applications of $b \in B_+$

\textbf{Output:} $\delta'_0$, an extension of $\delta_0$
$\hat{F}_+$, the image of $F_+$ under the homomorphism $\phi$

\begin{enumerate}
\item $\forall i \in [1..k]$ do 
\item $\forall b \in B_+$ do 
\item $\delta_0 := \text{extend}(\delta_0, b)$
\item $\forall f \in F_+$ do 
\item $\hat{F}_+ := \hat{F}_+ \cup \text{memoise}(\delta_0, f)$
\item return $\delta_0, \hat{F}_+$
\end{enumerate}

This algorithm first calls Algorithm 2 to build the garden $G$, then calls Algorithm 3, using $G$ (as carried within $\delta_0$) to build the indexed operator tables.

\textbf{Input:} $\delta_0$, an indexed set of nullary operators
$B_+$, a set of non-nullary functions for building $G$
$F_+$, a non-nullary set of functions to index
$k$ limits applications of $b \in B_+$

\textbf{Output:} $\delta'_0$, an extension of $\delta_0$
$\hat{F}_+$, the image of $F_+$ under the homomorphism $\phi$

\begin{enumerate}
\item $\forall i \in [1..k]$ do
\item $\forall b \in B_+$ do
\item $\delta_0 := \text{extend}(\delta_0, b)$
\item $\forall f \in F_+$ do
\item $\hat{F}_+ := \hat{F}_+ \cup \text{memoise}(\delta_0, f)$
\item return $\delta_0, \hat{F}_+$
\end{enumerate}

This algorithm first calls Algorithm 2 to build the garden $G$, then calls Algorithm 3, using $G$ (as carried within $\delta_0$) to build the indexed operator tables.

\textbf{Input:} $\delta_0$, an indexed set of nullary operators
$B_+$, a set of non-nullary functions for building $G$
$F_+$, a non-nullary set of functions to index
$k$ limits applications of $b \in B_+$

\textbf{Output:} $\delta'_0$, an extension of $\delta_0$
$\hat{F}_+$, the image of $F_+$ under the homomorphism $\phi$

\begin{enumerate}
\item $\forall i \in [1..k]$ do
\item $\forall b \in B_+$ do
\item $\delta_0 := \text{extend}(\delta_0, b)$
\item $\forall f \in F_+$ do
\item $\hat{F}_+ := \hat{F}_+ \cup \text{memoise}(\delta_0, f)$
\item return $\delta_0, \hat{F}_+$
\end{enumerate}

4.1 Indexer
For each $\tau \in T$, we must index a subset of $\tau$’s values ($G_\tau$), but which subset? INDEXIFY is \textit{useless} without a good selection of values from $\tau$. For instance, if all the values in $G_\tau$ take the same path (e.g., immediately exit), we learn nothing about the program under analysis. To build $G_\tau$, the indexer uses functions from a set of constructors ($B_+$) to extend an initial set of seeds ($S_\tau$); obviously, the quality of $B_+$ and $S = \bigcup_{\tau \in T} S_\tau$ determine the quality of $G = \bigcup_{\tau \in T} G_\tau$.

In Section 6.1, we show that constants harvested from the source of the input program $P$ make a surprisingly effective $S$ and, in Section 5, that users unfamiliar with $P$ do not improve on these constants much. Supplementing $S$ with constants from developer artefacts other than the program text, like test cases or documentation, is future work. If the user does not specify $B_+$, the indexer defaults to using $F_+$, \textit{i.e.} $B_+ = F_+$. Section 5.3 evaluates this choice of default for $B_+$.

Adding new operators in $B_+$ to evaluate using INDEXIFY is straightforward: starting with the initial definitions, we automatically compute their indexed versions (Indexed Operator Table), by executing that operators, and memoising the results. We just require access to call the initial definition while memoising.
Algorithm 2 extend$(\delta_0, f) = \delta'_0$

**Input:** $\delta_0$, an indexed set of nullary operators (constants)
$f : \tau_0 \rightarrow \tau_1$, a function to evaluate to extend $\delta_0$

**Output:** $\delta'_0$, $\delta_0$ extended using $f$

1. let $m := \text{arity}(f)$ in
2. $G := \{g \mid (g, n) \in \delta_0\}$  # Snapshot $G$ before changing it
3. for all $g^m \in G^m \land \text{typeof}(g^m) = \tau_0$ do
4.   let $u := f(g^m)$ in
5.   if $\delta_{\bot}(u) = \bot$ then
6.     $\delta_0 := \delta_0 \cup \{(u, |\delta_0| + 1)\}$
7. return $\delta_0$

Algorithm 3 memoise$(\delta_0, f) = \hat{f}$

This algorithm builds the image of $f$ under $\delta_0$.

**Input:** $\delta_0$, an indexed set of nullary operators
$f : \tau_0 \rightarrow \tau_1$, a non-nullary function to index

**Output:** $\hat{f}$, the image of $f$ under the homomorphism $\phi$

1. let $m := \text{arity}(f)$ in
2. $G = \{g \mid (g, n) \in \delta_0\}$
3. let $\hat{f} = \emptyset$ in  # $\hat{f}$ is the image of $f$
4. for all $g^m \in G^m \land \text{typeof}(g^m) = \tau_0$ do
5.   let $u := f(g^m)$ in
6.   $\hat{f} := \hat{f} \cup \{(\text{map } \delta_0 g^m, \delta_0(u))\}$
7. return $\hat{f}$

Starting from $S$, INDEXIFY first constructs $G$ using the functions in $B_+$, as specified in Algorithm 1. It loops over number of applications $k$ and over $B_+$, calling Algorithm 2. Later calls to Algorithm 2 are applied to the results of early calls. In this section, we split $\delta$ (Section 3) into $\delta_0$ for the nullary operators and $\hat{F}_+$ for the rest of the operators. Algorithm 2 builds $\delta_0 = \phi|_{F_0}$ and Algorithm 3 builds $\hat{F}_+$.

Algorithm 2 simply enumerates $G^m$, filtering out type invalid permutations. In future work, we plan to experiment with sampling $G^m$. Given $S = \{a, b\}$, $K = 2$, and $B_+ = \{\text{strcat, strstr}\}$, INDEXIFY produces $G = \{\emptyset, a, b, aa, bb, ab, ba\}$. Here, $G$ contains all the values that can be obtained when applying the functions $\text{strcat}$ and $\text{strstr}$ twice first on $S$, then on $S$ and the result of the first application. As discussed in Section 3, $G$ grows quickly. Despite this forbidding growth rate, our experiments in Section 6.1 show that INDEXIFY performs effectively when restricted to small values of $k$ and $|B|$.

Next, Algorithm 3 indexes $F_+$, the functions operating over $\tau \in T$. If the user inputs $T$, we harvest $F_+$ from the intersection of functions used in the program, and a relevant library, i.e. $\text{string} . h$, when indexing strings, and $\text{maths} . h$, when indexing floats or double. For each $f : T \rightarrow \tau \in F_+$, the indexer encodes $\hat{f}$ (or $\_f$ in ASCII), the indexed version of $f$, as a sequence of if statements as follows:

```
1  i_f(x, y) { }
2    if x = 1 \land y = 1 return 1;
3    if x = 1 \land y = 2 return 3;
4    ...;
5  return -1; // return \bot when x \notin G \lor y \notin G
```
Indexed operators (or functions) memoize the result of \( f \in P_+ \) on values in terms of the indices defined by \( \delta \). When the result of \( f \)'s computation is not in \( \hat{G} \), \( \hat{f} \) returns \( \perp \). Thus, \( \delta_{-} \) above contains \( \text{str} = 1 \wedge y = 3 \) \( \text{return} \ 57 \); because \( f(\delta^{-1}(1), \delta^{-1}(3)) = \delta^{-1}(57) \). The number of if statements in an indexed operator is \(|G|^n + 1\), but Sections 5.2 and 6.1 show experimentally that Indexify was effective using parameter settings that only doubled the size of its inputs on average. The indexer injects the definition of each \( \hat{f} \in \hat{P}_1 \) into \( P \).

Algorithms 1–3 build \( \delta \) and exactly realise Equation 1. Algorithm 1 is the driver algorithm that calls Algorithm 1 and Algorithm 2 to build \( \delta = \delta_0 \cup \delta_+ \) in two parts. Algorithm 2 simultaneously extends \( \delta_0 \) and constructs the garden \( G \) using the builders \( B_+ \) applied to \( G \) as it expands. Algorithm 1 calls Algorithm 2 only over \( B_+ \) on lines 1–3. Algorithm 2 calls each \( b \in B_+ \) on the values currently in \( G \). If the resulting string is not currently in \( G \), Algorithm 2 assigns it a fresh index and updates \( \delta_0 \). Algorithm 3 constructs \( \delta_+ = \hat{F} - \hat{\delta}_0 \), the union of all the indexed operator tables for the functions in \( F_+ \). Algorithm 1 calls Algorithm 3 only over \( F_+ \) on lines 4–5. Algorithm 2 and Algorithm 3 perform no other computations.

### 4.2 Rewriter

The **rewriter** is a straightforward implementation of the term rewriting schema in Figure 4. We generate wrapper code that unindexes things on the fly. We initially index every occurrence of a variable of the indexed type, and further we unindex them on the fly, when the variables flow into unindexed operations. In the presence of aliasing, we unindex such values when they go into a different data type (structure), and later unindex it back, when it flows into an indexed data type / operator. To handle casts we use the index, and unindex function. When the program casts a indexed variable \( x \) to a different type, we first unindex \( x \) to the initial data type, and then we cast the variable of the initial data type, as in the initial program. Similarly, when the program casts a variable of an unindexed type to an indexed type, we index the result of the cast applied on the initial (unindexed) data type.

Figure 6 shows an example of applying \( \phi_\delta \). When \( G = \{ \text{"foo\bar{bar}"}, \text{"oob\bar{ar}"}, \text{"ba\bar{r}"}, \text{"\oo\"} \} \), Indexify replaces "\bar{bar}" on line 6 (Figure 6a) with \( \delta(\text{"bar"}) = 2 \) (Figure 6b), but ignores the constants \( \text{a} \) and \( \text{foo} \) on line 5 (Figure 6a). Equation 6 replaces \( \text{strstr} \) on line 6 in Figure 6a with \( \text{i_strstr} \) on line 14 in Figure 6b. Equation 8 unindexes indexed variables that flow into unindexed functions. In Figure 6b, Equation 8 unindexes the indexed variable \( \text{S1} \) on line 13, as the function \( \text{puts} \) is unindexed. Finally, Equation 9 indexes the return from unindexed functions into indexed variables in Figure 6b on line 11 since \( \text{strcat} \) is not indexed, but returns into \( S2 \), which is.

The rewriter uses LLVM’s API for IR manipulations and transformations\(^3\). In LLVM, types are immutable, so we cannot change them in place. Instead, Indexify outputs the indexed version of the IR of \( P \) in a new file. The **rewriter** is a visitor that walks the original IR of \( P \); when it reaches an IR element of a type in \( T \), it creates a new instruction with \( T \) changed to index; otherwise, it simply echoes the instruction. In LLVM, a global variable and its initializer must have the same type and LLVM forbids casts or function calls in initializers. Thus, the visitor replaces values with indices in initializers. We execute Klee on this indexed version of the LLVM bitcode. To support indexing multiple types at time, Indexify keeps multiple gardens, one for each indexed data type. For example, one might want to index both the strings and the floats in a program. For this, Indexify keeps \( G_S \), the set of string values in our domain restriction, and \( G_F \), the set of float values in our domain restriction. The indexes in \( G_S \) and \( G_F \) do not overlap. Further, Indexify proceeds normally, but when indexing a type it uses indexes from the corresponding garden: \( G_S \) if the type to index is a string; \( G_F \) if the type to index is a float.

\(^3\)http://llvm.org
int main()
{
    char S1[3], S2[5];
    klee_make_symbolic(S1, sizeof(S1), "S1");
    puts(strlen(S1));
    S2 = strcat("a", "foo");
    if(strstr(S1, "bar")) return 1; // return 1
    else return 0;
}

(a) P, the program to indexify.

(b) P’ = indexify(P, string); puts and strcat are not in $F_+$. 

Fig. 6. Indexification: INDEXIFY also injects $\delta$ and $\delta^{-1}$ into $P'$, although not shown.

INDEXIFY allows an indexed operator to take both indexed and unindexed types. In this case, INDEXIFY does not simply replace the function’s body with an IOT; instead, it indexes the function’s body, replacing any internal calls to indexed operators with their indices versions and inserting index-casts to (un)index values, including the return value, as needed. Currently INDEXIFY does not use Equation 5. INDEXIFY does not need Equation 5 because for us $S$ is the set of the literals in the program. Thus, all the literals in the program are in $G$.

4.3 Indexed Constraint Construction

To understand how an indexified program constructs constraints, consider the program $P$ in Figure 6a. Here, the string $S$ is symbolic; and “bar” is a constant string. The call to klee_make_symbolic makes the program variable $S$ symbolic. When $G = \{ \text{“foobar”, “oobar”, “bar”, “oo”} \}$, indexify --type string P.c --garden path_to_G --F_+ path_to_F_+ produces $P'$ in Figure 6b, whose behaviour is restricted to $G$ over string operations. In $P'$, $\hat{f}_{strstr} = i_{strstr}$.

INDEXIFY produces $P'$. When we symbolically execute $P'$, we reach the call to i_strstr on line 12 (Figure 6b). For i_strstr, Klee delays calling the internal solver [Cadar et al. 2008], encoding i_strstr into an indexed operator table (IOT) as disjunctions for the branches of the if statement (Figure 6b). The constraints that i_strstr generates are:

$(s1 = 0 \land s1 = 2 ) \lor (s1 = 0 \land s2 = 3 ) \lor (s1 = 3 \land s2 = 2 ) \lor \ IOT$.

The constraints in the IOT of $\hat{f} \in \hat{F}_+$ are in the theory of equality by construction; an IOT is a lookup table that disjuncts equalities over the values of its parameters. These constraints are solvable by an integer solver equipped with equality theory in polynomial time. Barrett et al. [Barrett et al. 2009] show this and provide a linear time solving algorithm. Our implementation currently cannot fully exploit this fact because it extends Klee, which builds constraints directly in the bitvector theory with arrays. In our current implementation, INDEXIFY first indexes the program then uses Klee for symbolic execution. Klee takes the indexed program and bitblasts it. In this case, INDEXIFY still improves performance whenever bitblasting integers results in shorter constraints than bitblasting the initial data type. For example, a string of length $N$ requires $N \cdot \text{sizeof}(\text{char})$ bits. After indexing, the same string uses only $\text{sizeof}(\text{int})$ bits.

INDEXIFY takes advantage of Klee’s internal query optimisations. Klee removes unsatisfiable constraints from the path conditions via simplifications [Cadar et al. 2008; Dillig et al. 2010]. Under these optimizations, Klee selects only the relevant subset of $\hat{f}_{strstr}$ in the context of the predicate in the if statement on line 15 in Figure 6b: only the entries where

---

4This is an actual, if pedagogical example. We provide the constraints in SMTLIB format, as generated by Klee, at <url>. 

12
the second parameter is $\delta(\text{bar}) = 2$. It does so prior to sending the query to the internal solver. Thus, the constraints that Klee actually sends to STP for $\text{i_strstr}$ are: $(s1 = 0 \land s1 = 2) \lor (s1 = 3 \land s2 = 2) \lor IOT|_{s2=2}$.

Section 6.7 compares the number of clauses that Klee and \textsc{Indexify} generate on coreutils. Without optimisation, \textsc{Indexify} generates three times more clauses than Klee. The results drastically change, when we look at the number of clauses that reach the solver, after Klee’s internal optimisations: indexified programs send $\frac{1}{10}$ as many constraints to the solver. \textsc{Indexify}'s \textit{IOT}, by construction, are particularly amenable to the syntactic constraint simplifications that Klee employs.

4.4 Usage

Once Klee is successfully installed, \textsc{Indexify} is easy to use: issuing \texttt{indexify --type string yourprogram.c} indexifies yourprogram.c, and then runs Klee on the indexified program. \textsc{Indexify} takes optional parameters. When runs with its \texttt{--outputIndexedIR} flag, \textsc{Indexify} outputs the indexed IR. By default, \textsc{Indexify} harvests $S$, the set of seed values to index from the input program. The switch \texttt{--addSeeds <file>} specifies a file containing seeds to add to the harvested constants; \texttt{--seeds <file>} specifies a file containing $S$ and prevents constant harvesting. By default, \textsc{Indexify} automatically constructs the operator definitions, using Algorithm 3. The switch \texttt{--indexOpDefs <file>} specifies the LLVM bitcode file that contains indexed operators to allow their reuse across runs, amortizing the cost of their construction.

Specifying $F_+$, the functions in a program to index, is tedious. We allow the user to specify only $T$, which triggers the indexer to populate $F_+$ from a header file. For $T = \text{string}$, \textsc{Indexify} indexes all the functions in \texttt{string.h}; for $T = \text{float}$, \textsc{Indexify} indexes all the functions in \texttt{maths.h}. For an arbitrary data type, the user needs to specify the name of the desired header file.

5 EXPERIMENTAL SETUP

\textsc{Indexify} operates on an input program $P$ in two phases. First, its indexer constructs the “garden” of values $G$, then it builds operator tables for the target operators in $P$ over $G$. Finally, \textsc{Indexify} transforms $P$ to $P'$ and symbolically executes it. As detailed in the previous section, the indexer takes four parameters: the types to index or a list of functions to index $F_+$, the seed values $B_0 = S$, the functions for building the garden from $S$, and $k$, a bound on the applications of the builder functions. In this section, we explore various setting for $S$, $B_+$, and $k$ in order to fix them, leaving only $F_+$ to vary, in Section 6.

Corpus Our experimental corpus contains two benchmarks: GNU Coreutils\textsuperscript{5}, and fdlibm\texttt{53}\textsuperscript{6}. GNU Coreutils contains the basic file, shell and text manipulation utilities of the GNU operating system, such as \texttt{echo}, \texttt{rm}, \texttt{cp} and \texttt{chmod}. We include all the 89 Coreutils in our experiments, and use \textsc{Indexify} (\texttt{string}) on them. We use the second benchmark, fdlibm\texttt{53}, for exploring the capabilities of \textsc{Indexify}, when indexifying floats. We picked these benchmarks because \textsc{Indexify}, Klee and Zesti can execute all the benchmarks in our corpus.

We select Klee (\texttt{KLEE 1.1.0 47a97ce; LLVM 2.7}), as a dynamic symbolic execution engine, and Zesti (the beta version on the Zesti’s website\textsuperscript{7}) as a concolic testing engine, for the comparison with \textsc{Indexify}. We selected these 2 tools, as \textsc{Indexify} is based on Klee, as is the case for Zesti. \textsc{Indexify} first indexifies the program, and further calls Klee on the indexified program. We also decided to compare \textsc{Indexify} with Klee and Zesti, because of the fact that we support

\textsuperscript{5}http://www.gnu.org/software/coreutils/coreutils.html
\textsuperscript{6}http://www.validlab.com/software/fdlibm53.tar.gz
\textsuperscript{7}http://arg.doc.ic.ac.uk/zesti/zesti.tar.gz
symbolic execution on C code. Klee is the most famous symbolic execution engines for C code. We called all the three tools (INDEXIFY, Klee, and Zesti) with exactly the same parameters as used in the initial Klee paper [Cadar et al. 2008].

5.1 Human-Provided Seeds

INDEXIFY constructs $G$, the subspace of value to which INDEXIFY restricts a program’s behaviour, from $S$. Thus, defining $S$ is crucial to INDEXIFY’s effectiveness. INDEXIFY defaults $S$ to the constants of $T$ in $P$. This extraction method exploits the domain knowledge embedded in these constants to bias $B$, and therefore $G$, to values that $P$ is more likely to compute.

Here, we ask whether human intuition can help INDEXIFY by augmenting $S$ with values that a developer considers interesting? To decide if human intervention in seed selection is effective, we compare the manual effort to discover seeds against the coverage and execution time gains of the augmented set of seeds. For this comparison, we uniformly selected 10 programs from coreutils: cat, expand, fold, mknod, mktemp, runcon, shred, shred, tsort, unexpand, and wc.

One of the authors spent no more than ten minutes to construct strings that he thought might allow symbolic execution of the indexed program to explore new paths or corner cases and added them to $S$. We ran INDEXIFY with $T$ set to strings, $B_+$ set to string functions occurring in $P$ and uclibc, and $k = 3$ on this set.

The results were identical: the human-augmented seeds provided no discernable improvement. Given INDEXIFY’s overall effectiveness, we take this as a testament to the effectiveness of INDEXIFY’s constant harvesting heuristic.

5.2 Bounding $k$ to Operator Chain Length

INDEXIFY’s core indexer algorithm (Algorithm 1) is computationally and memory expensive as a function of $k$, the number of applications of the functions in $B$ (Equation 2). Setting $k$ is a trade-off between the power of analysis (i.e. the size of $G$) and the computational cost of running INDEXIFY. First, we identify the value of $k$ in our corpus, then we observe the performance when using the identified $k$.

From our corpus, we infer $k$ from the lengths of chains of applications over $F_+$: we define chains have nonzero length. In our first experiment, we statically symbolically execute our corpus to collect symbolic state, from which we extract operator chain lengths from def-use chains in killed expressions. Figure 7 shows the lengths of operator chains $k$ in our corpus. The first boxplot, labelled All.coreutils, reports the distribution of all operator chain lengths in coreutils: the median is 11; the minimum is 1; and the maximum is 2833. The percentage of outliers is 11.66% upward and 0% downward, since 1 is the first quartile and we do not have 0s. A manual exploration of uniformly picked outliers reveals that loops cause them.

Figure 7 also reports boxplots for string and float operators. For string operators, the median value is 2, with 3.64% outliers; for float operators, the median value is 1, with 15.53% outliers. Setting $k$ to the third quartile of operator chain lengths in our corpus reduces the probability that symbolically executing an indexified benchmark will escape $G$ and bind $\perp$ to a variable, merely through operator applications. Thus, we fix $k = 3$.

5.3 Finding a Good Builder

INDEXIFY uses the functions in $B_+$ to build the garden $G$ from $S$. Which functions should we use? For strings, we consider two different builders. $B^+_\text{str} = \{\ast\}$ contains only Kleene closure and $B^+_\text{str} = F_+$, the indexed functions in $P$. To build $G$, we apply the operators in $B_+$ up to $k$ times. For example, let $S = \{a, b\}$ and $k = 2$. Under $B^+_\text{str}$, $G = \{0, a, b, aa, bb, ab, ba\}$. For floats, $B^+_\text{fl}$ applies all the float operators in $P$ including maths.h on all combinations of float constants in the program text up to $k$ times. Each application of $B_+$ potentially generates a new value in $G$.  

14
To compare $B_+^k$ and $B_0^k$ on strings up to $k = 3$, we executed INDEXIFY on coreutils, then ran Klee on the resulting indexed programs. We constrained the $G$ that $B_+^k$ constructed: only adding a generated value to $G$ if its length was less than or equal to 8. We experimented with different maximum sizes, up to 20. We observed that the branch and statement coverages are not improving any more when considering strings with lengths greater than 8. Our experimental corpus, Coreutils, does not computes during its execution strings longer than 8 character. This is because the most of the programs in Coreutils do not use very often the concatenation operator over strings strcat. This is the only string operator that increases the sizes of strings during execution. The most of the strings that the programs in Coreutils use are flags, or file names, which are in general short in size. For different experimental programs, higher values of maximal string lengths might be required. Table 1’s “Time” row shows that $B_+^k$ executes more than three times slower than $B_0^k$. At $k = 3$, $B_+^k$ covers 55.32% of the instructions in coreutils (“ICov”), while $B_0^k$ covers 57.67% of them. The branch coverage (“BCov”) is 69.73% for $B_+^k$ and 66.83% for $B_0^k$. These results show a trade-off between execution time and coverage. $B_+^k$ covers more strings, increasing both BCov and the execution time. The difference in BCov is only 3% and $B_+^k$ achieves 2% higher ICov. Thus, we set $k = 3$ and $B_+ = B_+^3$ in Section 6.

When using $k$ greater than 3, we observed an increase in runtime and a decrease in code coverage. This is due to the fact that the number of conjunctions in the constraints that INDEXIFY generates grows exponentially, and Klee’s internal solver time-out is reached. When this happens, Klee abandons the path with the constraint that time-outs. We ran $k$ up to 10. The branch coverage decreased smoothly from 69.73% when $k = 3$ to 65.00% when $k = 10$. The execution time increased smoothly from 1008 seconds when $k = 3$ to 3150 when $k = 10$. 

![Graph](image)

**Fig. 7.** Length of operator chains in coreutils; we use these values to set $k$.

**Table 1. Measures of Algorithm 1 as a function of $k$.**

|   | $k = 1$ | $k = 2$ | $k = 3$ |
|---|---------|---------|---------|
| $B_+^k$ | $B_0^k$ | $B_+^k$ | $B_0^k$ |
| Time(seconds) | 324 | 123 | 544 | 263 |
| $|G| | 10792 | 69068 | 78920 | 163378 |
| ICov(%) | 26.22 | 52.21 | 34.09 | 55.65 |
| BCov(%) | 21.93 | 59.74 | 39.98 | 62.37 |

To compare $B_+^k$ and $B_0^k$ on strings up to $k = 3$, we executed INDEXIFY on coreutils, then ran Klee on the resulting indexed programs. We constrained the $G$ that $B_+^k$ constructed: only adding a generated value to $G$ if its length was less than or equal to 8. We experimented with different maximum sizes, up to 20. We observed that the branch and statement coverages are not improving any more when considering strings with lengths greater than 8. Our experimental corpus, Coreutils, does not compute during its execution strings longer than 8 character. This is because the most of the programs in Coreutils do not use very often the concatenation operator over strings strcat. This is the only string operator that increases the sizes of strings during execution. The most of the strings that the programs in Coreutils use are flags, or file names, which are in general short in size. For different experimental programs, higher values of maximal string lengths might be required. Table 1’s “Time” row shows that $B_+^k$ executes more than three times slower than $B_0^k$. At $k = 3$, $B_+^k$ covers 55.32% of the instructions in coreutils (“ICov”), while $B_0^k$ covers 57.67% of them. The branch coverage (“BCov”) is 69.73% for $B_+^k$ and 66.83% for $B_0^k$. These results show a trade-off between execution time and coverage. $B_+^k$ covers more strings, increasing both BCov and the execution time. The difference in BCov is only 3% and $B_+^k$ achieves 2% higher ICov. Thus, we set $k = 3$ and $B_+ = B_+^3$ in Section 6.

When using $k$ greater than 3, we observed an increase in runtime and a decrease in code coverage. This is due to the fact that the number of conjunctions in the constraints that INDEXIFY generates grows exponentially, and Klee’s internal solver time-out is reached. When this happens, Klee abandons the path with the constraint that time-outs. We ran $k$ up to 10. The branch coverage decreased smoothly from 69.73% when $k = 3$ to 65.00% when $k = 10$. The execution time increased smoothly from 1008 seconds when $k = 3$ to 3150 when $k = 10$. 

![Graph](image)
Table 1 shows that, for both $B^e_+$ and $B^e_*$, coverage increases with $k$ up to 3, but execution time does not become unreasonable. Even $k = 3$ generates an immense garden, as Equation 2, in Section 3.1 shows and $|G|$ in Table 1 confirms. How does Indexify manage to be effective, despite Equation 2? We hypothesized that $k > 3$ is feasible because very few elements are unique in practice. Especially with strings, many operators return substrings of existing elements. We evaluated our hypothesis on $B^e_+$. Our results show that from the total of generated values, only 1.97% are unique; only the unique values are indexed.

6 EVALUATION

Here, we demonstrate that Indexify can improve existing automated testing techniques by trading space to reduce time and increase solution coverage. When used to generate test data, dynamic symbolic execution and concolic testing aim to achieve the highest structural coverage possible. Since the theoretical space complexity of Indexify is worse than exponential, an essential goal of this evaluation is to demonstrate that its space consumption, in practice, is manageable. Furthermore, we evaluate the degree to which this manageable increase in space consumption reduces time and improves solution coverage. We evaluate whether Indexify can catch bugs that are out of reach to traditional symex. Then, we assess Indexify ability to make the constraints easier for the underlying SMT solver by restricting the domain of some operators. Finally, we evaluate Indexify when indexing floats.

6.1 Trading Space for Time and Coverage

Table 2. Overall results for Indexify (String). We fixed the bound $k = 3$; we used $B_e = B^e_+$. The trends support our core insight: Indexify increases memory consumption reducing execution time and increasing code coverage.

| Metric      | Indexify | Klee | Zesti |
|-------------|----------|------|-------|
| Time(min)   | 6.05     | 49.52| 22.70 |
| Memory(MB)  | 900.45   | 241.16| 3000.00|
| ICov(%)     | 57.67    | 41.24| 20.60 |
| BCov(%)     | 66.83    | 30.10| 14.79 |

Indexify trades memory for reduced execution time and increased code coverage. Memory has become cheaper and more abundant, but our core algorithm consumes exponential memory in the worst case. Can we significantly reduce execution time or increase code coverage at reasonable memory cost?

To answer this question, we compare Indexify to dynamic symbolic execution using Klee and to concolic testing using Zesti. Zesti uses concrete inputs to kick-off concolic testing. It searches a neighbourhood around the path executed under the concrete inputs. For each concrete input, Zesti follows the concrete execution that the input generates and records the path condition. When reaching exit, Zesti backtracks to the nearest condition, complements it, generates a new input that obeys the new path condition, and restarts execution from the entry point [Godefroid et al. 2005; Sen et al. 2005a].

We compare their performance in terms of run time, memory consumption, instruction, and branch coverages. We index strings in the input programs when running Indexify, i.e. $T = \text{string}$.

To configure Indexify, Klee, and Zesti, we use the same settings that Klee used on the coreutils benchmark [Cadar et al. 2008]. The time-out is 3600 seconds; the maximum memory usage allowed is 1000 MB; and the maximum time spent on one query is 30 seconds. Although the maximum memory consumption is 1000 MB, Zesti might use more memory than this, as it composes multiple symbolic execution runs. We use $k = 3$, as we explain in Section 5.2.

We construct $IOT$ and $G$ online. We automatically harvest the seeds $S$ from each program’s text. $F_e$ contains the functions involving strings in the input program; we use $B^e_+3$ (Section 5) to build $G$. 

Fig. 8. Uniquely covered branches by: Klee, INDEXIFY with $B^*_x$, and INDEXIFY with $B^e_x$.

We report the instruction and branch coverage on LLVM bitcode, for the final linked bitcode file. This bitcode combines tool and library code. Achieving 100% coverage of this bitcode is usually impossible, because programs tend to use very little library code, relegating the rest to dead code. For example, `printf("Hello!")` does not cover `printf`’s format specification handling code [Cadar et al. 2008].

Table 2 reports our experimental results. As expected, these results (row “Memory”) show that INDEXIFY consumes more memory than Klee. Averaged over all coreutils, Klee requires 241.16 MB, while INDEXIFY requires 900.45 MB. INDEXIFY’s memory consumption is reasonable relative to Zesti, consuming $\frac{1}{3}$ the memory (3000MB) Zesti does. Zesti exceeds the 1000MB memory limit on a single run of Klee, as it runs Klee multiple times.

INDEXIFY has compensatory advantages to set against its memory consumption. First, INDEXIFY improves time performance. Over all, INDEXIFY requires 6.05 minutes mean time, while Zesti requires 22.70 minutes and Klee requires 49.5 minutes. Second, INDEXIFY outperforms the existing state of the art in terms of coverage achieved: INDEXIFY achieves 57.67% instruction coverage and 66.83% branch coverage. By contrast, Klee achieves 41.24% instruction coverage and 30.10% branch coverage while Zesti achieves 20.60% instruction coverage and 14.79% branch coverage.

While there is much debate in the testing community over the value coverage [Gligoric et al. 2013; Lakhotia et al. 2009], there is little doubt that greater coverage is to be strongly preferred over lower coverage. We argue that the increased coverage and reduced run-time are more significant than the increase in memory consumption. For an increase of 659.29 MB of memory for INDEXIFY, we get 16% increase in instruction coverage; a 36% increase in branch coverage; and a 43 minute reduction in run time, on average.

6.2 Unique Statements Coverage

We explore how many previously unreachable branches INDEXIFY brings in the reach of symex and compare how many branches Klee and INDEXIFY uniquely cover. In Figure 8, we show the percentages of branches that $B^e_x$, $B^*_x$, and Klee uniquely cover. Klee uniquely covered 3.02% of branches in coreutils’s IR. Both $B^e_x$ and $B^*_x$ missed these branches: they are infeasible in the indexed programs, because our garden $G$ does not contain the strings that these branches use. To cover these branches, a greater $K$ might help.

Figure 8 shows that although 2.54% of statements are covered only by Klee, INDEXIFY enables symbolic execution to cover far more previously out of reach branches: on total, 66.58% are only covered by $B^*_x$, or $B^e_x$. $B^*_x$ covers uniquely the most branches: 13%. $B^e_x$ covers uniquely 3.68%. More branches are uniquely covered by the $B^*_x$ because the Kleene
Table 3. Bugs that Klee, and INDEXIFY can reach.

| Project | Bug             | Size | Klee | INDEXIFY |
|---------|-----------------|------|------|----------|
| ncompress | stack smash     | 1935 | NO   | YES      |
| gzip    | buffer overflow | 4653 | NO   | NO       |
| man     | buffer overflow | 2805 | NO   | YES      |
| polymorph | buffer overflow | 240  | YES  | YES      |

closure contains strings that are not the result of operator applications. Some branches are uniquely covered by $B^c_k$ because $K$ applications of string operators may lead to strings longer than the bound for the Kleene closure.

6.3 Bug Finding

The most compelling evaluation for any testing technique is the ability to reveal bugs that other techniques cannot. Accordingly, we investigate whether INDEXIFY can identify bugs not detected by Klee. We evaluated INDEXIFY on a set of bugs from Bugbench, a famous benchmark for C bug finding tools [Lu et al. 2005]. We consider only the bugs that the default oracle in Klee can detect — bugs that cause programs to crash. For Klee, experiments [Cadar et al. 2008] on coreutils set the time-out to 60 minutes. coreutils average 434 LOC. Our corpus averages 2408 LOC, a 5-fold linear increase.

Although Klee does not scale linearly, we increased the time budget 5-fold to five hours, for both Klee and INDEXIFY.

Table 3 reports our results: INDEXIFY catches two bugs that Klee cannot catch: the bugs in ncompress and man; both Klee and INDEXIFY catch the bug in polymorph. Neither tool catches the bug in gzip. Not catching a bug means that the tool reaches the five hours timeout without generating a test case to reveal the bug in our benchmark [Lu et al. 2005]. Our results show that INDEXIFY reaches bugs previously unreachable for Klee in two cases out of four.

6.4 Domain Restriction

INDEXIFY restricts the domain of operators involving $T$ to a subset of the initial supported values, the one that $G$ contains, to make them simpler: INDEXIFY rewrites constraints involving types in $T$ into the theory of equality over integers. We report the number of constraints on which STP times out: How many SMT queries return “unknown” before and after the application of INDEXIFY?

When running Klee on all coreutils, STP time-outs on 354 constraints. After indexifying the corpus, this number drops to 0. In 53 out of the 89 coreutils programs, Klee times out before finishing; only 6 programs time out for INDEXIFY. These results show INDEXIFY generates indexed programs that produce simpler constraints than Klee does when run directly on the original program.

We then divided the 89 coreutils programs into 3 categories: projects with at least 5% of the expressions indexed after applying INDEXIFY; projects with at least 10% of expressions indexed; and projects with at least 15% of expressions indexed. These 3 categories overlap, as we wanted to assess the performance of INDEXIFY as a function of the percentage of indexed expressions. There were no important differences in the performance of INDEXIFY between these three groups. INDEXIFY is not dependent on the proportion of program expressions that it converts, but rather on how much of the code in the program the paths that become reachable due to indexing make available for symex.

6.5 INDEXIFYing Floats

To assess how well INDEXIFY performs when indexing data types other than strings, we indexed the type float in our second benchmark: fdlibm53. For $B$, we used the intersection between the functions involving floats in fdlibm53 and
the ones in math.h. We fixed \( k = 3 \), as the third quartile value of \( k \), the length of chains of floating point operators in our corpus (Figure 7).

Table 4 shows INDEXIFY’s general effectiveness. INDEXIFY increased instruction coverage 33% (row “ICov”) and branch coverage 37% (row “BCov”). INDEXIFY consumes ten times more memory than Klee (row “Memory”). INDEXIFY’s increase in the run time from 0.84 seconds in the case of Klee to 10.45 seconds. This increase has two causes. The first is a very small execution times for fdlibm53. Running INDEXIFY has an additional runtime cost because of: building the IOTs, changing the target type to indexes in the LLVM IR, and replacing the calls to the functions to be indexed, with their corresponding indexed versions. When the runtime to symex the initial program is big enough, the runtime reduction that our simpler constraints generate is bigger than the computational cost for INDEXIFY. When the runtime to symex the initial program is very small, as is the case for our float benchmark, the preprocessing runtime to indexify the program is bigger than runtime reduction that the simpler constraints cause. The second is the fact that more of the indexed program’s constraints are solvable allows Klee to go further down paths, increasing coverage but incurring state space explosion.

### 6.6 The Size of Indexified Programs

Indexification encodes the IOTs in the IR representation of \( P \). Although the disk space is cheap, it is important that the size of indexed programs are manageable in practice.

We used `wc -l` to report the LOCs of the IR before and after indexification for coreutils. The total size in text lines for the initial coreutils is 1,731,007 lines; the total size for the indexified coreutils is 3,735,435. In the LLVM IR file format\(^8\) every instruction appears on a new line. Thus, we considered the newlines in LLVM IR as a good proxy for the number of instructions in that program.

Indexification increased the total size of the IR for coreutils by 215%. Although in the worst case, the size of the IOT grows exponentially in the average arity of the operators, this increase is manageable in practice. We speculate two reasons for this in coreutils. The programs in coreutils do not use all the string operators in string.h: for example, the Unix tool `yes` does not use any string operators. INDEXIFY adds additional code only for the IOTs. We construct an IOT only for an operator that we index. Second, INDEXIFY encodes IOT as a sequence of if statements, one per value in the garden. LLVM translates each branch of an if statement into only 2 lines of IR: the predicate on one line, and the label for the true branch.

### 6.7 Number of Clauses

The number of clauses in queries affects the performance of symbolic execution. We compared the numbers of clauses between each original program and its indexified version. We ran Klee, and INDEXIFY with the option: `--use-query-log solver:all,pc`. This reports all the queries sent to the underlying solver, after Klee’s internal optimisations.

After the application of INDEXIFY, symbolically executing indexed programs generated 2,920,246,627 clauses in total, while the unindexed programs generated 997,389,929. Klee on indexed programs sent 44,138,715 clauses to STP, while it sent 401,289,490 on unindexed programs.

\(^8\)http://releases.llvm.org/2.7/docs/LangRef.html
INDEXIFY generated three times as many constraints as Klee because of IOTs are encoded into queries sent to the solver. Klee’s optimisations include simplifications that remove infeasible clauses from the queries, prior to sending them to the solver. These optimisations hugely benefit INDEXIFY, removing irrelevant IOT entries. After the optimisations, indexed programs generates \( \frac{1}{10} \) the constraints that unindexed programs do. INDEXIFY is very effective at simplifying constraints and Klee’s solver chain is very effective at dealing with the simple constraints that INDEXIFY produces.

7 RELATED WORK

Symbolic execution (symex) binds symbols to variables during execution. When it traverses a path, it constructs a path condition that define inputs (including environmental interactions) that cause the program to take that path. Symex has some well known limitations [Cadar and Sen 2013] including path explosion, coping with external code, and out-of-theory constraints.

Harman et al. [Harman et al. 2004] introduced the concept of testability transformations, source-to-source program transformations whose goal is to improve test data generation. Following Harman et al., Cadar speculated that program transformations might improve the scalability of symbolic execution in a position paper [Cadar 2015]. INDEXIFY realizes, in theory and practice, such a program transformation, rewriting a program to allow symbolic execution to cover portions of the program’s state space that current symex engines cannot efficiently reach, as we show in Section 6.2. INDEXIFY transforms expressions in a program that produce out-of-theory constraints into expressions that produce in-theory constraints. The transformed (indexed) program underapproximates the input program’s behaviour: it is precise over every component of the program’s state it binds to a value in \( G \) and reifies its ignorance of component’s actual value when it binds \( \perp \) to that component.

7.1 Handling Unknown via Concretization

Solver unknowns bedevil Symex. One way to handle them is to resort to concrete execution. Dynamic Symbolic Execution (DSE) [Cadar et al. 2008] does this by lazily concretizing the subset of the symbolic state for which the solver return unknown. Concolic testing [Godefroid et al. 2005; Sen et al. 2005b; Marinescu and Cadar 2012] or white-box fuzzing [Godefroid et al. 2008] is an extension to DSE that initially follows the path that a concrete input executes. Further, concolic testing searches a neighbourhood around the path executed under its concrete input by negating the values of the current branch conditions from the current followed path. First concolic testing flips the closest branch to the end of execution and then, it continues to do so upwards to the entry point of the program. To flip the path condition, concolic testing uses an SMT solver.

Whenever reaching a constraint that the solver cannot solve, or for which the solver times out, the concretization methods, such as DSE and concolic testing, get a concrete value that can be either random, or obtained under different heuristics [Marinescu and Cadar 2012]. In contrast INDEXIFY keeps the value of the unsolvable constraint symbolic, but restricted to the garden. This allows INDEXIFY to obtain higher code coverage, by considering more paths than the concretization methods, as our direct comparison with Klee [Cadar et al. 2008] (DSE) and Zesti [Marinescu and Cadar 2012] (concolic testing) shows in Section 6.1. Additionally, INDEXIFY does not require concrete inputs from the user, as concolic testing does: INDEXIFY’s builder automatically builds the set of values that it will consider in the analysis.

INDEXIFY generalizes the concretization techniques: our garden \( G \) allows us to reason about a finite set of values from \( G \), instead of a single concrete value, as in the case of the concretization methods. Like concolic testing, INDEXIFY generates test inputs from an initial set of values. For INDEXIFY, this initial set is the set of seeds harvested from constants; for concolic testing and white-box fuzzing, the initial set is the input test suite. INDEXIFY and concolic testing generate
these inputs differently. Concolic testing uses an SMT solver to negate the path conditions of the initial inputs; Indexify constructs its garden using repeated applications of builder functions. Our intuition is that the constants in a program are discrete points in the state space of the program: the constants appear in the predicates of the decisions points in the program. By considering these points in G, we explore the behaviour of the program on the true branch of the predicate, the discrete point in the state space of the program. For example, in the predicate if (strstr(S,"foo")), we would like to consider the value foo for S. Under Indexify foo will certainly be considered, as Indexify will automatically harvest into its seed set. Concolic testing tends to generate inputs that follow paths in the program near one of its initial inputs. Indexify, in contrast, is not tethered to a set of initial paths, so it can cover widely varying program paths.

7.2 Constraint Encoding
During its execution, a program under Symex produces constraints in the different domains of its data types, such as constraints over strings, or floats. Before calling the solver, a Symex engine needs to encode the constraints in a language that the solver understands. Some of these constraints are difficult for the solver: the literature provide us a couple of constraint encoding mechanism to cope with the difficult kinds of constraints, in particular constraints over string, or floating point values.

**Strings** Quine proved that the first-order theory of string equations is undecidable [Quine 1946]. Since then different techniques have emerged implementing decision procedures over fragments of string theory. All these approaches have limitations: either they do not scale; they support a fragment of string theory that is not well-aligned with string expressions developers write; they require fixed length strings; or they require a maximum length and loop through all possible lengths up to that maximum. There are three main categories of string solvers: automata, bit-vector, and word-based.

Automata-based solvers [Veanes et al. 2010] use regular languages or context-free grammars to encode the string constraints. The idea is to construct a finite-state automata that accepts all the strings that satisfy the path conditions in a program. Building this automata requires handcrafted building algorithms for the set of string operations that we intend to support and the set of values that the program accepts as inputs [Hooimeijer and Weimer 2009]. When a new string constraint is added to the path condition, these approaches refine the automaton to not accept the strings that violate the newly added constraint. The refinement process is automatic: we remove from the set of values that the automata accepts the ones that are not valid for the new constraint. Infeasible paths construct automata that do not accept any strings. For string constraints, the automaton becomes the solver. Automata-based string solvers tend not do not combine strings with other data types [Li and Ghosh 2013], as combining string automatas with other data types and operations over them, requires handcrafted initial automatas for the particular data types and operations that the program under analysis uses in conjunction with the string operations. Yu *et al.* [Yu *et al.* 2009] tackles the problem of using an automaton to handle both string and integer constraints, but no other data types. Within the bounds of the values it handles, Indexify combines different data types, including strings, by automatically translating them into the theory of integer equality up to its garden, the finite set of values over it is defined.

Bit-vector based symex engines convert string constraints into the domain of bit-vectors [Ganesh and Dill 2007]. The bit-vector solvers require a maximum string length and lack scalability. Specifying a maximum length per each query that symex sends to the solver would require comprehensive annotations, so symex engines use a global maximum. In general, string length is domain-specific and specifying a maximum length for an entire program is problematic: for example, a database query might be hundreds of characters long, while a command line flag can occupy only one character. The string length restriction means that users must specify a single length for all strings in the program. When too big, this
length slows the solver; when too small, it limits the analysis to small strings. Thus, these engines typically execute a program over different length strings [Ganesh and Dill 2007]. The bit-vector encoding of values causes exponential blow up in model size, $2^n$ where $n$ is the length of a bit-vector, since each bit becomes a propositional variable for the underlying SAT solver and hampers scalability. For a restricted set of values, INDEXIFY encodes constraints into table lookups, not bit-vectors. Running Klee on programs transformed by INDEXIFY covers more than twice the number of branches that Klee manages on the unindexed programs (Section 6). While Klee still encodes the values in an indexed program’s queries into bit-vectors, these are short bit-vector encodings of integers, not long bit-vector encodings of strings.

Word-based string solvers define a subtheory that uses rewriting rules and axioms tailored to a fragment of string theory. Word-based string solvers escape Quine’s result by not handling all string expressions. CVC4 [Liang et al. 2014] and S3 [Trinh et al. 2014] support constraints over unbounded strings restricted only to length and regular expression membership operators. Z3-STR [Zheng et al. 2013] supports unbounded strings together with the concatenation, string equality, substring, replace, and length operators. INDEXIFY does not constrain the syntax of string expressions; instead, it constrains their values. To do so, it finitzes string operators, restricting their definition to a finite portion of their domain and range. As Section 4 describes, INDEXIFY converts its finitized string expressions into constraints over integer equality, then forwards them to an SMT solver.

Floating Point  

Bit-blasting is the most widely used technique for solving floating-point constraints. Bit-blasting converts floats into bit-vectors and encodes floating point operations into formulae over these bit-vectors. Bit-blasting floating-point constraints generates formulae that require a huge number of variables. For example, Brillout et al. [Brillout et al. 2009] showed that when using a precision of only 5 (mantissa width) addition or subtraction over floating point variables requires a total of 1035 propositional variables to be encoded in bit-vector theory. In a similar scenario, multiplication or division requires a total of 1048 propositional variables. Because of this, bit-blasting for floating point constraints does not scale to large programs [Darulova and Kuncak 2014].

Testing is also used to solve floating point constraints. Microsoft’s Pex symex engine can use the FloPSy [Lakhotia et al. 2010] floating point solver. FloPSy transforms floating point (in)equalities into objective functions. For example, the predicate $\text{if}(\text{Math.Log}(a) == b)$ becomes the objective function $|\text{Math.Log}(a) - b|$ for which we want to find values of $a$ and $b$ that yield 0. FloPSy uses hill climbing [Clarke 1976] to find values for $a$ and $b$. Fu et al. proposes Mathematical Execution (ME) [Fu and Su 2016]. They capture the testing objective through a function that is minimised via mathematical optimisation to achieve the testing objective. Given a program under test $P$, they derive another program $P_R$ called representing function. $P_R$ represents how far an input $x \in \text{dom}(P)$ is from reaching the set $x|P(x)$ is wrong. $P_R$ returns non-negative results and $P_R$ is the distance between the current input and an error triggering input. Further, they minimise $P_R$. Souza et al. [Souza et al. 2011] integrate CORAL, a meta-heuristic solver designed for mathematical constraints, into PathFinder [Visser et al. 2004].

Klee-FP [Collingbourne et al. 2011] and Klee-CL [Collingbourne et al. 2014] replaces floating-point instructions with uninterpreted functions. Klee-CL and Klee-FP apply a set of canonizing rewritings and further do a syntactic match of floating-point expressions trees. Both Klee-CL and Klee-FP solve floating point constraints by proving that them are equivalent (or not) with an integer only version of the program. They do this by matching the equivalent expression trees. Thus, the main limitation is the requirement of having the both version of the programs available: the floating point and the integer implementation. INDEXIFY does not require any existing integer implementation.

Other approaches use constraint programming [Michel et al. 2001] to soundly remove intervals of float numbers that cannot make the path condition true. Botella et al. [Botella et al. 2006] solve floating point constraints using interval
propagation. Interval propagation tries to contract the interval domains of float variables without removing any value that might make the constraint true. These approaches are either imprecise or do not scale. INDEXIFY translates floating point arithmetic into equality theory over integers that is solvable in polynomial time [Barrett et al. 2009].

### 7.3 The State Of the Art in Symbolic Execution

Significant research has tackled the problem of applying symex to real world programs [Cadar and Sen 2013]. INDEXIFY complements this work: INDEXIFY is a program transformation that is applied prior to symex; and thus can be applied in conjunction with any technique that improves symex.

Some approaches aim to improve a symex engine’s interactions with the constraint solver. Klee [Cadar et al. 2008] and Green [Visser et al. 2012] cache solved constraints. Lazy initialization delays the concretization of symbolic memory thereby avoiding the concretization of states that additional constraints make infeasible [Rosner et al. 2015]. Memoized symex [Yang et al. 2012] and directed incremental symbolic execution [Person et al. 2011] reuse query results across different symex runs.

Other approaches improve the scalability of SymEx by mitigating the path explosion problem. Veritesting /citeavgerinos2014enhancing proposes a path merging technique that reduces the number of paths being considered as a result of reasoning about the multiple merged paths simultaneously. MultiSE [Sen et al. 2015] perform symbolic execution per method, rather than per the entire input program. MultiSE merges different symex paths into a value summary, or a conditional execution state. Further, we can symex each method in a program starting from its value summary, rather than from the program’s entry point.

For an unsolvable constraint, symex concretizes its variables, causing incompleteness: it cannot reason on the rest of the state space. Pasareanu et al. [Păsăreanu et al. 2011] delay concretization to limit the incompleteness. They divide the clauses into simple and complex ones. When a simple clause is unsatisfiable, the entire PC becomes unsatisfiable. This can avoid reasoning about the complex clauses. Khurshid et al. [Khurshid et al. 2003] concretizes objects only when they need to access them.

A recent study [Dong et al. 2015] show that 33 optimization flags in LLVM decrease symex’s coverage on coreutils. Overify [Wagner et al. 2013] proposes a set of compiler optimisations to speed symex. Sharma et al. [Sharma 2014] exploit these results and introduce undefined behaviour to trigger various compiler optimisations that speed up symex [Cadar 2015]. Under-Constrained SymEx [Ramos and Engler 2015] operates on each function in a program individually. Abstract subsumption [Anand et al. 2006] checks for symbolic states that subsume other ones and remove the subsumed ones. Ariadne transforms numerical programs to explicitly check exception triggering conditions in the case of floats [Barr et al. 2013]. As INDEXIFY is a program transformation step prior to symex, we can take advantage of these optimisations by running INDEXIFY in conjunction with them.

### 8 CONCLUSION

We introduced indexification, a novel technique that rewrites a program to constrain its behaviour to a subset of its original state space. Over this restricted space, the rewritten program under-approximates the original; its symbolic execution generates tractable constraints. We realized indexification in INDEXIFY and show that it automatically harvests program constants to define restricted space permitting symex to explore paths and find bugs that other symbolic execution techniques do not reach.
REFERENCES

Saswat Anand, Corina S Păsăreanu, and Willem Visser. 2006. Symbolic execution with abstract subsumption checking. In International SPIN Workshop on Model Checking of Software. Springer, 163–181.

Earl T Barr, Thanh Vo, Vu Le, and Zhendong Su. 2013. Automatic detection of floating-point exceptions. In ACM SIGPLAN Notices, Vol. 48. ACM, 549–560.

Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. 2009. Satisfiability Modulo Theories. Handbook of satisfiability 185 (2009), 825–885.

Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. 2009. Path feasibility analysis for string-manipulating programs. In Tools and Algorithms for the Construction and Analysis of Systems. Springer, 307–321.

Bernard Botella, Arnaud Gotlieb, and Claude Michel. 2006. Symbolic execution of floating-point computations. Software Testing, Verification and Reliability 16, 2 (2006), 97–121.

Angelo Brillout, Daniel Kroening, and Thomas Wahl. 2009. Mixed abstractions for floating-point arithmetic. In Formal Methods in Computer-Aided Design, 2009. FMCAD 2009. IEEE, 69–76.

Cristian Cadar. 2015. Targeted program transformations for symbolic execution. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ACM, 906–909.

Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In OSDI Vol. 8. 209–224.

Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. Commun. ACM 56, 2 (2013), 82–90.

Ting Chen, Xiao-song Zhang, Shi-ze Guo, Hong-yuan Li, and Yue Wu. 2013. State of the art: Dynamic symbolic execution for automated test generation. Future Generation Computer Systems 29, 7 (2013), 1758–1773.

Lori A. Clarke. 1976. A system to generate test data and symbolically execute programs. IEEE Transactions on software engineering 3 (1976), 215–222.

Peter Collingbourne, Cristian Cadar, and Paul HJ Kelly. 2011. Symbolic crosschecking of floating-point and SIMD code. In Proceedings of the sixth conference on Computer systems. ACM, 315–328.

Peter Collingbourne, Cristian Cadar, and Paul HJ Kelly. 2014. Symbolic crosschecking of data-parallel floating-point code. IEEE Transactions on Software Engineering 40, 7 (2014), 710–737.

Eva Darulova and Viktor Kuncak. 2014. Sound compilation of reals. Acm Sigplan Notices 49, 1 (2014), 235–248.

LLVM Developers. 2016a. LLVM. http://llvm.org/. (2016). Accessed: 2016-07-02.

ReactOS Developers. 2016b. ReactOS. http://www.reactos.org/. (2016). Accessed: 2016-07-02.

Ilis Dillig, Thomas Dillig, and Alex Aiken. 2010. Small formulas for large programs: On-line constraint simplification in scalable static analysis. In International Static Analysis Symposium. Springer, 236–252.

Shiyu Dong, Oswaldo Olivo, Lingming Zhang, and Sarfraz Khurshid. 2015. Studying the influence of standard compiler optimizations on symbolic execution. In Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on. IEEE, 205–215.

Zhouli Fu and Zhendong Su. 2016. Mathematical Execution: A Unified Approach for Testing Numerical Code. arXiv preprint arXiv:1610.01133 (2016).

Vijay Ganesh and David L Dill. 2007. A decision procedure for bit-vectors and arrays. In International Conference on Computer Aided Verification. Springer, 519–531.
Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. 2013. Comparing non-adequate test suites using coverage criteria. In *International Symposium on Software Testing and Analysis (ISSTA 2013)*, Mauro Pezzè and Mark Harman (Eds.). ACM, Lugano, Switzerland, 302–313.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Programming Language Design and Implementation (PLDI 2005)*, Vivek Sarkar and Mary W. Hall (Eds.). ACM, 213–223.

Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated Whitebox Fuzz Testing.. In *NDSS*, Vol. 8. 151–166.

Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. 2004. Testability transformation. *IEEE Transactions on Software Engineering* 30, 1 (2004), 3–16.

Pieter Hoimeijer and Westley Weimer. 2009. A decision procedure for subset constraints over regular languages. *ACM Sigplan Notices* 44, 6 (2009), 188–198.

Susmit Jha, Sanjit A Seshia, and Rhishikesh Limaye. 2009. On the computational complexity of satisfiability solving for string theories. *arXiv preprint arXiv:0903.2825* (2009).

Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. 2003. Generalized symbolic execution for model checking and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 553–568.

Stephen Cole Kleene. 1951. *Representation of events in nerve nets and finite automata*. Technical Report. DTIC Document.

Daniel Kroening and Michael Tautschnig. 2014. CBMC–C bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 389–391.

Kiran Lakhotia, Phil McMinn, and Mark Harman. 2009. Automated Test Data Generation for Coverage: Haven’t We Solved This Problem Yet?. In *4th Testing Academia and Industry Conference — Practice And Research Techniques (TAIC PART’09)*. Windsor, UK, 95–104.

Kiran Lakhotia, Nikolai Tillmann, Mark Harman, and Jonathan De Halleux. 2010. Flopsy-search-based floating point constraint solving for symbolic execution. In *IFIP International Conference on Testing Software and Systems*. Springer, 142–157.

Guodong Li and Indradeep Ghosh. 2013. PASS: String solving with parameterized array and interval automaton. In *Haifa Verification Conference*. Springer, 15–31.

Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. 2014. A DPLL (T) theory solver for a theory of strings and regular expressions. In *International Conference on Computer Aided Verification*. Springer, 646–662.

Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. 2005. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, Vol. 5.

Paul Dan Marinescu and Cristian Cadar. 2012. Make test-zesti: A symbolic execution solution for improving regression testing. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 716–726.

Claude Michel, Michel Rueher, and Yahia Lebbah. 2001. Solving constraints over floating-point numbers. In *International Conference on Principles and Practice of Constraint Programming*. Springer, 524–538.

Corina S Păsăreanu, Neha Rungta, and Willem Visser. 2011. Symbolic execution with mixed concrete-symbolic solving. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 34–44.

Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. 2011. Directed incremental symbolic execution. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 504–515.

Willard V Quine. 1946. Concatenation as a basis for arithmetic. *The Journal of Symbolic Logic* 11, 04 (1946), 105–114.
David A Ramos and Dawson R Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In USENIX Security. 49–64.

Barry K Rosen. 1973. Tree-manipulating systems and Church-Rosser theorems. Journal of the ACM (JACM) 20, 1 (1973), 160–187.

Nicolás Rosner, Jaco Geldenhuys, Nazareno M Aguirre, Willem Visser, and Marcelo F Frias. 2015. BLISS: Improved symbolic execution by bounded lazy initialization with sat support. IEEE Transactions on Software Engineering 41, 7 (2015), 639–660.

Koushik Sen, Darko Marinov, and Gul Agha. 2005a. CUTE: a concolic unit testing engine for C. In ACM SIGSOFT Software Engineering Notes, Vol. 30. ACM, 263–272.

Koushik Sen, Darko Marinov, and Gul Agha. 2005b. CUTE: a concolic unit testing engine for C. In 10th European Software Engineering Conference and 13th ACM International Symposium on Foundations of Software Engineering (ESEC/FSE ‘05), Michel Wermelinger and Harald Gall (Eds.). ACM, 263–272.

Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: Multi-path symbolic execution using value summaries. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ACM, 842–853.

Asankhaya Sharma. 2014. Exploiting undefined behaviors for efficient symbolic execution. In Companion Proceedings of the 36th International Conference on Software Engineering. ACM, 727–729.

Matheus Souza, Mateus Borges, Marcelo di?Amorim, and Corina S Păsăreanu. 2011. CORAL: solving complex constraints for symbolic pathfinder. In NASA Formal Methods Symposium. Springer, 359–374.

Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2014. S3: A symbolic string solver for vulnerability detection in web applications. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, 1232–1243.

Margus Veanes, Nikolaj Bjørner, and Leonardo De Moura. 2010. Symbolic automata constraint solving. In International Conference on Logic for Programming Artificial Intelligence and Reasoning. Springer, 640–654.

Willem Visser, Jaco Geldenhuys, and Matthew B Dwyer. 2012. Green: reducing, reusing and recycling constraints in program analysis. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. ACM, 58.

Willem Visser, Corina S Păsăreanu, and Sarfraz Khurshid. 2004. Test input generation with Java PathFinder. ACM SIGSOFT Software Engineering Notes 29, 4 (2004), 97–107.

David Wagner, Jeffrey S Foster, Eric A Brewer, and Alexander Aiken. 2000. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In NDSS. 2000–02.

Jonas Wagner, Volodymyr Kuznetsov, and George Canadea. 2013. -Overify: Optimizing Programs for Fast Verification. In 14th Workshop on Hot Topics in Operating Systems (HotOS XIV).

Guowei Yang, Corina S Păsăreanu, and Sarfraz Khurshid. 2012. Memoized symbolic execution. In Proceedings of the 2012 International Symposium on Software Testing and Analysis. ACM, 144–154.

Fang Yu, Tevfik Bultan, and Oscar H Ibarra. 2009. Symbolic string verification: Combining string analysis and size analysis. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 322–336.

Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: A z3-based string solver for web application analysis. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, 114–124.