Green: Reducing, Reusing and Recycling Constraints in Program Analysis

Willem Visser
Stellenbosch University
Stellenbosch, South Africa
wvisser@cs.sun.ac.za

Jaco Geldenhuys
Stellenbosch University
Stellenbosch, South Africa
jaco@cs.sun.ac.za

Matthew B. Dwyer
University of Nebraska - Lincoln
Lincoln, NE, USA
dwyer@cse.unl.edu

ABSTRACT

The analysis of constraints plays an important role in many aspects of software engineering, for example constraint satisfiability checking is central to symbolic execution. However, the norm is to recompute results in each analysis. We propose a different approach where every call to the solver is wrapped in a check to see if the result is not already available. While many tools use some form of results caching, the novelty of our approach is the persistence of results across runs, across programs being analyzed, across different analyses and even across physical location. Achieving such reuse requires that constraints be distilled into their essential parts and represented in a canonical form.

In this paper, we describe the key ideas of our approach and its implementation, the Green solver interface, which reduces constraints to a simple form, allows for reuse of constraint solutions within an analysis run, and allows for recycling constraint solutions produced in one analysis run for use in other analysis runs. We describe how we integrated Green into two existing symbolic execution tools and demonstrate the reuse we achieve in the different settings.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Verification, Experimentation

Keywords

symbolic execution, path feasibility, constraint solving, NoSQL

1. INTRODUCTION

Trading off space for time is a classic approach to improving the time efficiency of (software) analysis tools. The logic is simple: if you can retrieve a result faster than it takes to recompute, and, you don't pay too high a penalty for the storage, then storing and reusing will improve analysis time. Needless to say, the benefit becomes greater when applying the approach to problems that are computationally expensive. Satisfiability checking, the world's most famous NP-complete problem, where the problem runs in time exponential in the size of the constraints being checked, is a prime example of where storing the result is better than recomputing it.

Using symbolic execution to find errors and generate tests is an active area of research [3, 11, 21, 27]. Satisfiability checking is the cornerstone of symbolic execution. Storing the result of satisfiability checking during symbolic execution and then reusing it later is not novel [3]. However, this is most commonly done within one analysis, where results are cached and then reused later in the same analysis. How about caching results across analyses? In this paper we consider many variations for reusing such persistent results: across runs of the *same* program, *similar* programs, different programs, but also the same program being analyzed with different tools. We show that the reuse of results is substantial even across different programs and different analyses. We also consider storing even more expensive computations, namely model counting [16] results, that are used to calculate path execution probabilities [10].

Our framework also supports reuse across different locations, by allowing the user to configure a list of stores to check if the result is not found locally. Once the result of an analysis is known it is *pushed* to all the stores on the list. As the underlying technology to handle storage we use the Redis[1] key-value store, an in-memory database in the NoSQL style. Since Redis is in-memory, local lookups are extremely fast, but over the network lookups can pay a network latency penalty.

In this paper, we only address symbolic execution as the technology we try to improve with the constraint results storage, but we consider both classic symbolic execution and dynamic symbolic execution (DSE). Symbolic execution builds a *path condition* that is a conjunction of all the branching conditions (in terms of the program inputs) to reach a particular point in the code; this path condition (or a slightly modified version in the case of DSE) is checked for satisfiability. Here we only consider linear integer arithmetic constraints, for which satisfiability is decidable, but our approach will also work for other theories, even undecidable theories (and the more expensive the check, the better). If

[1]`http://redis.io`

we check all the branching conditions to reach a location, one would think there will be very limited reuse across runs from different programs (and by definition, no reuse in the same program, since all path conditions are unique). To address this we firstly obtain the *minimal* path condition (by a process we refer to as slicing) required to check satisfiability, and secondly we transform path conditions into a canonical form. The canonical form contains a renaming phase, which allows us to match constraints that only differ in variable names. For example, the canonized constraints for the expressions `x > 0 && x < 10` and `y > 0 && y < 10` will be identical.

We evaluate our approach to reuse by considering a number of programs (all previously studied in the symbolic execution literature) and measuring how much reuse we get in a number of different settings. To measure reuse across similar programs we consider mutations of each program and then also one actual regression example (where we corrected a bug in one of our examples). We then evaluate reuse across different programs by considering a sample of random orders of execution between programs and measuring the reuse. Lastly we show that without slicing and canonization this approach of persisting results is futile.

We make the following contributions in this paper:

- Show that constraint analysis results reuse can be generalized from in-program caching to persistent stores that allows reuse across programs and across analyses

- Show that constructing the minimal path condition and creating a canonical form for path conditions enable more results reuse

- Describe an implementation of the Green solver interface to which existing program analyses and constraint solvers can be attached. Green is freely available[2] and has been adopted by existing projects, for example, it has been integrated with Symbolic PathFinder (SPF) [21].

2. OVERVIEW

In this section, we provide an overview of the Green framework for reusing constraint solving results, which is sketched in Figure 1, and we illustrate several different forms of reuse that it makes possible.

Green is designed to work with a wide range of program analyses, and other algorithmic techniques, that formulate a query constraint c_0 that is to be evaluated in a context defined by a conjunctive constraint $c_1 \wedge c_2 \wedge \ldots c_n$. Analyses typically take the approach of simply conjoining the query and context constraints, translating them to existing solvers, such as CVC3 [1], Choco [5], or Yices [30], and invoke the solver to determine the satisfiability of the constraint and to extract related information, such as a model of a satisfying assignment. In contrast our framework consists of four phases: slicing, canonization, reuse, and translation.

First, we calculate the minimal set of context constraints on which the query depends. This constraint *slice* has the potential to significantly reduce both the number of constraints and the number of variables in the problem. Second, constraints are *canonized* by reformulating each individual

[2]`http://green-solver.googlecode.com`

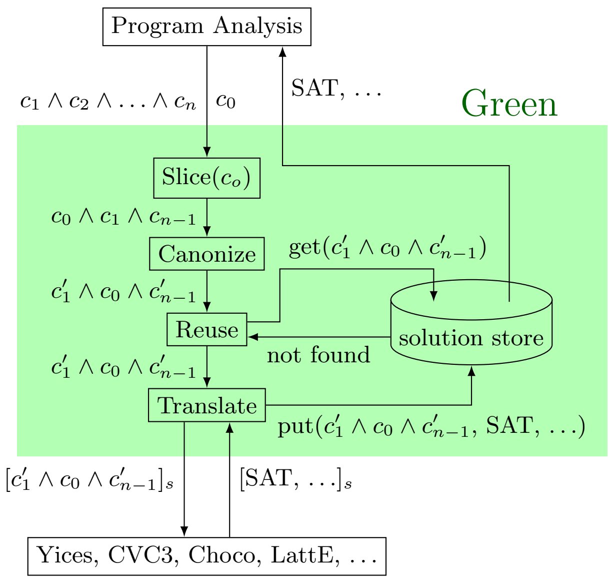
Program Analysis
$c_1 \wedge c_2 \wedge \ldots \wedge c_n$
c_0
SAT, ...
Green
Slice(c_o)
$c_0 \wedge c_1 \wedge c_{n-1}$
Canonize
get($c'_1 \wedge c_0 \wedge c'_{n-1}$)
$c'_1 \wedge c_0 \wedge c'_{n-1}$
Reuse
solution store
not found
$c'_1 \wedge c_0 \wedge c'_{n-1}$
Translate
put($c'_1 \wedge c_0 \wedge c'_{n-1}$, SAT, ...)
$[c'_1 \wedge c_0 \wedge c'_{n-1}]_s$
[SAT, ...]$_s$
Yices, CVC3, Choco, LattE, ...

Figure 1: Path-sensitive Analysis with Solver Reuse

constraint into a canonical form, indicated as primed constraints in the figure, and by reordering constraints. The canonization process also renames variables. Third, a *solution store* is queried with the canonized constraint. If a solution is found it is returned immediately to the invoking analysis, thereby avoiding the need to call a solver. Fourth, if a solution is not found, the canonized constraint is *translated* into the format of a selected solver and passed to the solver; the format for solver s is denoted $[\ldots]_s$. A solver returns its results which are translated back to correspond to the canonized constraint and those results are inserted into the solution store and returned to the invoking analysis.

One might imagine that a simpler solution for reusing solver results is possible, but the nature of path-sensitive analyses is to construct distinct constraints representing distinct program paths. As we illustrate below, and demonstrate through our evaluation in Section 4, both slicing and canonization are essential for achieving significant reuse and analysis performance gains.

2.1 Reuse Within an Analysis Run

To illustrate the operation of our framework we consider the small example program in Figure 2 and the application of a symbolic execution analysis.

The bottom of the figure shows the symbolic execution tree where nodes are labeled by the branch conditions in the source code and edges are labeled by symbolic constraints encoding conditions under which branch outcomes are taken. We represent the input values for `x` and `y` with $\mathcal{X}$ and $\mathcal{Y}$, respectively. The filled shapes represent the exit points of the method; we do not explicitly show the effects of individual statements here, but their effects are accounted for in the symbolic constraints.

For a call $m(-10, -10)$ the program would take the path ending at the star node. The path condition (PC) for this path is

$$\mathcal{X} < 0 \wedge \mathcal{Y} < 0 \wedge \neg(-\mathcal{X} < 10) \wedge 9 < -\mathcal{Y}$$

The execution of statement `x = -x;` on this path gives rise to $-\mathcal{X}$ in subsequent constraints along the path – the same goes for `y`.

There are 12 paths in this program each with a distinct path condition. To determine whether these paths are feasible, a symbolic execution tool will pass the conjunction of constraint accumulated along each path prefix to a satisfiability (SAT) solver – a SAT result implies path feasibility. This will lead to 22 solver calls for this example.

While modern solvers have increased dramatically in terms of performance in recent years, they still are quite expensive and many analyses that use them have had to adopt a number of innovative and custom optimizations in order to scale [4]. One of the goals of our framework is to provide generic support for some of those optimizations and to go further by reusing previous solver results.

Consider the true outcome of initial branch `x<0`. The PC is $\mathcal{X} < 0$ and there are two potential outcomes of the next branch `y<0` : $\mathcal{Y} < 0$ and $\neg(\mathcal{Y} < 0)$. Rather than simply conjoin a branch constraint with the PC and check for satisfiability, as explained in Figure 1, our framework slices the PC with respect to the branch constraint. In either case, this eliminates $\mathcal{X} < 0$, since $\mathcal{X} < 0$ is independent of $\mathcal{Y}$. The resulting sliced formula, e.g., $\mathcal{Y} < 0$, is canonized which involves restructuring each clause into a canonical form and then reordering those clauses and renaming variables. In the case of the constraints described above both $\mathcal{X} < 0$ and $\mathcal{Y} < 0$ would yield the canonized constraint $v_0 + 1 \leq 0$. Slicing and canonization expose a wide range of constraint equivalences as *syntactic equivalence* and this permits our framework to look up existing results and reuse them.

When our solver is used to symbolically execute the example in Figure 2 just 6, rather than 22, solver calls are needed. For the 6 PCs that require solving in the first two levels of the tree in Figure 2, only 2 solver calls are made. At the third level, the 8 new PCs lead to 2 new solver calls. At the fourth, again just 2 solver calls are needed to handle the 8 new PCs.

The path leading to the star node illustrates how the longer paths are able to be reused. Solving for the third constraint on that path yields a sliced PC of $\mathcal{X} < 0 \wedge \neg(-\mathcal{X} < 10)$ which is canonized to $v_0 + 10 \leq 0 \wedge v_0 + 1 \leq 0$. Solving for the fourth constraint on that path yields a sliced PC of $\mathcal{Y} < 0 \wedge 9 < -\mathcal{Y}$ which is canonized to the same constraint as encountered one step earlier on the path, so the solution to that constraint is reused.

A final, but important, benefit of slicing is that when a constraint solution cannot be reused the constraint that must be solved is often significantly smaller and has fewer variables.

2.2 Reuse Across Programs, Analyses, Solvers

The simple example in Figure 2 illustrates what might be achieved by results *caching* within a single analysis. The solution store in our framework is, by default, persisted when an analysis run exits. This makes the constraint solver results accumulated from a specific analysis on a specific program using a specific solver available for subsequent reuse. Importantly, the canonical forms in our framework are independent of analysis, program, or solver, and so a wide range of reuse scenarios are possible.

One obvious scenario that offers the potential for significant reuse is to improve the efficiency of techniques that perform path-sensitive regression analysis, such as DiSE [20].

```
int m(int x, int y) {
  if (x<0) x = -x;
  if (y<0) y = -y;
  if (x<10) {
    return 1;
  } else if (9<y) {
    // modification: (10<y)
    return -1;
  } else {
    return 0;
  }
}
```

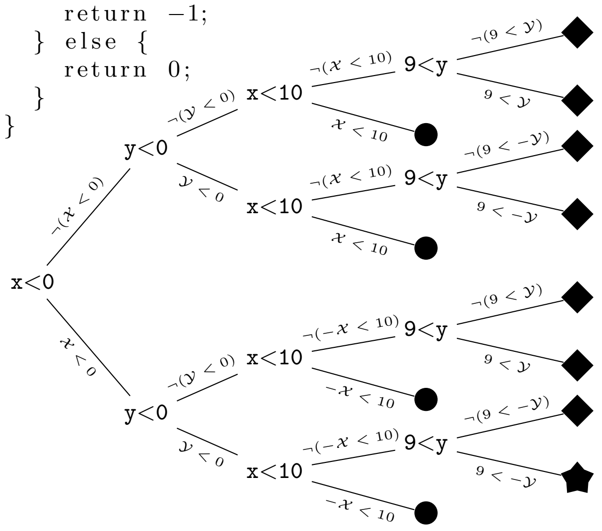

Figure 2: Example and Symbolic Execution Tree

Consider a modification of the branch `9<y` in the example above to `10<y`. This would change each of the constraints at depth 4 in the symbolic execution tree, but all 14 constraints solved at depth 3 or less would be reused from the analysis of the original program. Moreover, for those 8 new constraints only 4 solver calls are needed because of reuse within the analysis of the modified program. In total, for analyzing this modified program only 4 of the 22 possible solver calls are needed.

In Section 4, we will present results that demonstrate that even for different programs there are significant reuse opportunities. For example, we studied three different container implementations and found that more than half of the solver calls can be reused from an analysis of one of the others. We also present our experience refactoring a dynamic symbolic execution technique developed by other researchers to use our framework. The refactoring was straightforward and immediately sped up the other analysis.

It is a matter of configuration to switch Green among a set of different solvers. We have experimented primarily with CVC3 and Choco, but Green provides an opportunity for expensive solvers to be integrated into more common use since their results can be computed once and reused many times. We have demonstrated this benefit in our latest work on probabilistic symbolic execution [10] where we have refactored a caching scheme for single runs of our analysis to use Green. This has proven to dramatically reduce analysis times which are strongly dependent on the execution of costly model counting solvers.

In the next section, we present more of the details of the Green framework and the slicer and canonizer that it includes. Section 4 presents a broad evaluation of the potential benefits of Green and discusses opportunities for future enhancement. We discuss related work in Section 5 and conclude in Section 6.

3. STORING RESULTS

The results that we compute must be stored so that they can be reused. For this we employ two techniques: slicing of path conditions (Section 3.1) and path condition canonization (Section 3.2). Although we focus mostly on the problem of satisfiability checking, both these techniques are also used to enable more reuse for other calculations such as model counting [10]. A design goal was to make our storage-centric approach as simple as possible to integrate into existing tools. To this end we discuss what is required for the back-end storage component (Section 3.3) and the interface our system provides to enable integration (Section 3.4).

3.1 Path Condition Slicing

Any path condition (conjunction of constraints) can be split into a part for which the satisfiability result is known (`knownPC`) to hold and a new part (`newPC`) that when added to the other the satisfiability result is unknown. For example in classic symbolic execution every branch introduces a new constraint (`newPC`) to be added to the existing path condition (`knownPC`, that is known to be satisfiable). Whereas in dynamic symbolic execution one negates one of the constraints (`newPC`) from the path condition and the rest is the known satisfiable part (`knownPC`). Slicing is based on the fact that the satisfiability of adding the `newPC` is not dependent on all of the `knownPC`. Slicing makes sure that only the part that is required is used in conjunction with the new part, before satisfiability is checked. If one considers the constraints within a path condition to be represented as a constraint graph (V, E) where the vertices V are the symbolic variables and the edges E indicate whether two variables are part of the same constraint, then slicing can be defined as a graph reachability problem:

1. Build a constraint graph for `knownPC` $\wedge$ `newPC`
2. Find all variables R reachable from variables in `newPC`
3. Return the conjunction of all the constraints containing variables R.

Slicing has two benefits. It firstly reduces the size of path conditions to be checked for satisfiability, but in this work this benefit is not our main aim. We are more interested in the second benefit: it allows one to get more matching of path conditions both within the same program but also across other programs. Complete path conditions are unlikely to ever match a complete path condition of a different program and never one from the same program. However, once one only looks at a subset of the path condition, as produced by slicing, matching becomes a real possibility and in Section 4 we will show that it actually happens quite often.

3.2 Canonization

We use a standard approach to constructing the normal form for linear integer arithmetic, but we believe one must augment this step by adding heuristic steps before and after the normal form calculation to maximize the chance of finding matches.

Pre-Heuristic Currently we have one heuristic in the pre-stage, namely, variable reordering based on the lexicographic order of the variable names. This reordering allows one to match constraints that in one constraint might be $x > y$ and in another $y < x$, since they will both become $x > y$. This reordering is quite weak at the moment, since we don't attempt to reorder expressions, such as $x + y > z$ versus $z < x+y$; we only reorder constraints with single variables on both sides of the operator. Belying its simplicity, this heuristic makes a substantial difference to the amount of reuse achieved in our examples. Our system allows one to easily attach custom heuristics to control canonization.

Normal Form Finding a normal form for linear integer arithmetic path conditions is well studied and in this step we follow the standard approach of converting each constraint to the form

$$ax + by + cz + \ldots + k \; op \; 0 \quad \text{where } op \in \{=, \neq, \leq\}.$$

We handle $=$ and $\neq$ constraints by themselves, but all other inequalities are transformed into constraints over only $\leq$. The first step is to transform each constraint such that all variables (and constants) appear only on the left hand-side of the equation, with a 0 on the right-hand side; this step is enough to convert $=$ and $\neq$ to their final state. However for $\leq$ we need potentially two more steps (1) to multiply each side by -1 to transform $>$ and $\geq$ to respectively $<$ and $\leq$, and (2) for $<$ we must add 1 on the left-hand side to transform it to a $\leq$ inequality. Finally we join all the separately transformed canonical constraints by adding conjunctions to recreate a path condition. Note we always place the constant after all variables on the left-hand side and all simple arithmetic operations over constants are performed throughout to ensure just one constant in each constraint.

For example, the path condition

$$x + y < z \wedge x = z \wedge x + 10 > y$$

is transformed as follows. $x + y < z$ becomes $x + y - z < 0$ then $x + y - z + 1 \leq 0$, whereas $x = z$ becomes $x - z = 0$, and lastly $x + 10 > y$ is converted to $x - y + 10 > 0$, then to $-x + y - 10 < 0$ and finally to $-x + y - 9 \leq 0$. Thus the final output of this phase is

$$x + y - z + 1 \leq 0 \wedge x - z = 0 \wedge -x + y - 9 \leq 0.$$

Post-Heuristic There are currently two heuristics in this phase. The first one orders the constraints from the previous phase (normal form calculation) in a lexicographic order, returning a new (ordered) path condition. Then follows a renaming phase, that is based simply on the left-to-right order that the variables appear in the path condition. For example, if we take the path condition at the end of the example above, namely $x + y - z + 1 \leq 0 \wedge x - z = 0 \wedge -x + y - 9 \leq 0$, the reordering produces the transformed path condition $-x+y-9 \leq 0 \wedge x+y-z+1 \leq 0 \wedge x-z = 0$ (since the string "$-x + y - 9 \leq 0$" comes lexicographically before "$x + y - z + 1 \leq 0$", etc.). Renaming then replaces x with v_0 (since it is the first variable from the left), y with v_1 and z with v_2 resulting in the final path condition

$$-v_0 + v_1 - 9 \leq 0 \wedge v_0 + v_1 - v_2 + 1 \leq 0 \wedge v_0 - v_2 = 0.$$

3.3 Storage

The requirement on the storage component is that it must be fast, but also easy to integrate into the rest of the system. A system that meets both requirements is the Redis [23] key-value store. It is fast, since it is actually an in-memory database, but it does persist its contents to disk after a configurable time (or number of updates). We use the Jedis

Java interface code [13] to allow communication directly to Redis from our Java implementation. Redis runs on most POSIX systems, but we used it only on Ubuntu Linux and Mac OS X. However there is no requirement that the Redis server runs on the same system as our framework, and it can be accessed by connecting to it over the internet. Since it is in-memory, the only penalty is the network latency and throughput. Although our current implementation only supports Redis, any database can be used. However, in-memory databases should always be preferred, otherwise the storage/retrieval latency will reduce the advantage of reuse.

As can be seen from Figure 1 we simply require `get` and `put` operations on the store. These are the core operations supported by Redis, with `get(key)` and `put(key,value)` calls. Redis is optimized to handle strings as keys, and as such we use the string representation for a path condition as the key. For the `value` field we also use strings: "true" and "false" for satisfiability, and the string representation of integers for model counting.

The system supports a list of Redis stores to be configured at startup which then allows an analysis to search them in order to find a solution. When more than one Redis store is configured, all `put` operations are made immediately to the local store, but asynchronously also to all the other stores in the list (this allows the main system to continue, while results are written to potentially slow stores). In addition when a remote lookup returns a result, it is then also *pushed* asynchronously to all the other stores in the list. This allows results to propagate around the network of stores. In turn, this allows reuse across locations, since results that were pushed to a store can now be reused by another analysis that uses the same store. We believe this is a viable community analysis model, to the best of our knowledge the first of its kind where people doing different analyses now have the opportunity to benefit from each others' results.

3.4 The Green Solver Interface

At the core of Green is the `Solver` class. It allows clients to register the six components used to answer queries: a decision procedure, a constraint solver (often the same as the decision procedure), a model counter (for counting the number of solutions), a canonizer, a slicer and a store. A seventh component is required to translate path conditions to instances of our `Instance` class. Users are free to supply their own extensions of these components, but the goal is to centralize not only the query results, but also as much as possible of the hard work. There is therefore a `DefaultCanonizer` and `DefaultSlicer`, a `RedisStore`, support for CVC3 [1] and Choco [5] as both decision procedures and constraint solvers, and for LattE [10, 16] as a model counter. If a feature is not needed, the component can be omitted.

Changing Symbolic PathFinder (SPF) [21] to use Green instead of its traditional decision procedure infrastructure, took 16 hours of work. Similarly, to integrate it into a Dynamic Symbolic Execution system took 1.5 hours. In both cases the integration was done by people other than the main developers of the respective tools. We believe this illustrates that the integration is quite manageable and for tool developers it might be even easier.

4. EVALUATION

Below we evaluate the performance of our framework for four different scenarios: (1) across runs of the same program or very similar programs, (2) across different programs, (3) across different tools, (4) with a focus on the contribution of slicing and canonization.

All experiments were conducted on a Apple iMac with a 3.4 GHz Intel processor with 4 cores and 8 Gb of memory. It runs the Mac OS X 10.7.3 operating system and the Darwin 11.3.0 kernel.

The experiments that follow are based on six programs:

- `TriTyp` implements DeMillo and Offutt's solution [9, Figure 6] of Myers's triangle classification problem [17];
- `Euclid` implements Euclid's algorithm for the greatest common divisor using only addition and subtraction;
- `TCAS` is a Java version of the classic anti-Collision avoidance system available from the SIR repository [26];
- `BinomialHeap` comes from [28];
- `BinTree` implements a binary search tree with element insertion, deletion, and search from [28]; and
- `TreeMap` uses a red-black tree [6] to implement a Java `Map`-like interface, also from [28].

We use four performance metrics: (a) For symbolic execution, we measure the running time t_- when our framework is not used, and the running time t_+ when our framework is used, and we compute the *time ratio* $T_S = t_+/t_-$. (b) We also count the number of SAT queries n_- that are given to the decision procedure when our framework is not used, and the number of queries n_+ when our framework *is* in use. We then calculate the *reuse ratio* $R_S = (n_- - n_+)/n_-$. (c) For model counting (#SAT), we again measure the running times u_-/u_+ when our framework is used/not used and calculate the time ratio $T_\# = u_+/u_-$. (d) We count the number of #SAT queries m_-/m_+ when our framework is used/not used, and calculate the *reuse ratio* $R_\# = (m_- - m_+)/m_-$.

The time ratios give an indication of how much our framework speeds up the analysis. For instance, if $T_S = 0.1$, the analysis is ten times faster. Lower values are better. The reuse ratios indicate what proportion of queries are "avoided". If $R_S = 0.75$, three quarters of all queries are reused and we only need to compute 25% of the original set of queries. Higher numbers are better. When R_S or $R_\#$ is 1.0, all queries are found in the store and no invocations of the decision procedure or model counter are needed.

Unless otherwise stated, all the experiments reported below were run using Green interfaced with Symbolic PathFinder (SPF).

4.1 Across Runs

We consider three degrees of changing a program: small increments (using mutations), scaling the behavior of the program, and a larger regression that modifies the behavior but not the intent of the program.

4.1.1 Small Increments

Program changes are often small. We use first-order mutations [14] to illustrate the savings achieved when the results from previous analyses of earlier variations of a single program are reused. Jumble [15] was used to produce a number of mutations either by changing a comparison operator ($=$, $\neq$, $<$, $\leq$, $>$, $\geq$) or by changing an addition operator inside an integer expression ($+$, $-$). Note that all possible mutations

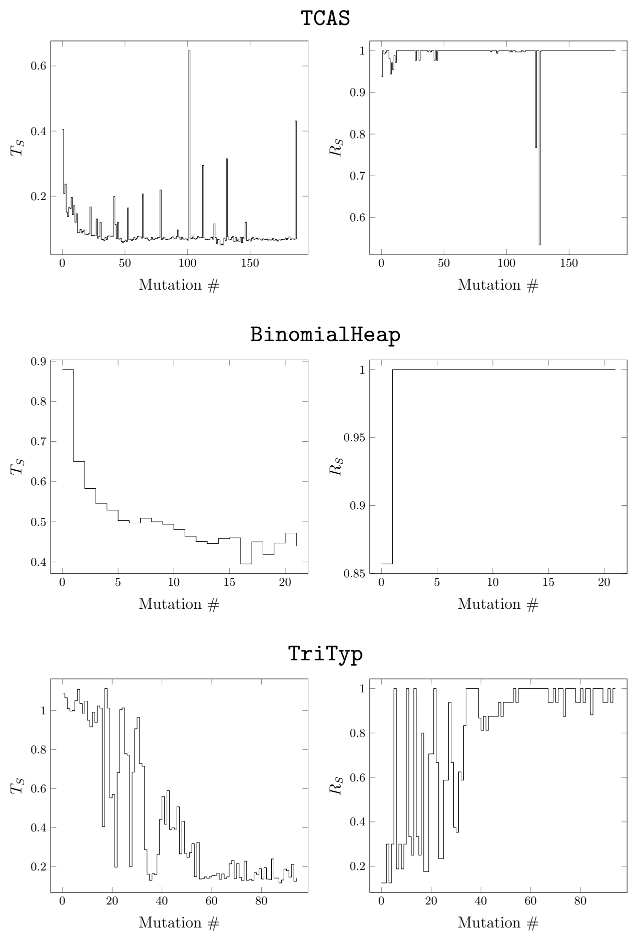

Figure 3: Reuse during mutation analysis

— not only a subset — are investigated. Figure 3 shows two charts for each of three input programs: the left-hand chart shows T_S, and the right-hand chart shows R_S. We start with an empty store and then analyze each mutation in turn, all the while, populating the store. We do not show $T_\#$ or $R_\#$; their graphs follow roughly the same pattern, except that the improvements are even more dramatic.

All three cases exhibit similar behavior: as more and more mutations are analyzed, the time consumption falls (to a minimum of 0.117 in the case of `TriTyp`) while the number of SAT queries reused increases to practically 1.000. For some programs (as exemplified by `TriTyp`) the early values of T_S are greater than 1: this happens because initially there is little reuse and the extra overhead cannot be recouped by the savings made. Other cases cause outliers because the mutated code leads to an exception and the running time of the mutant is so small that the natural variance in execution times leads to a high value for T_S.

4.1.2 Scaling

A second aspect of programs that result in very similar variants is *scaling*. For example, each of the container classes (`BinomialHeap`, `BinTree`, `TreeMap`) is parameterized with n, the number of insert/remove operations. When $n = 2$, SPF analyses four possible sets of operations: (1) $insert(x)$, $insert(y)$, (2) $insert(x)$, $delete(y)$, (3) $delete(x)$, $insert(y)$, and (4) $delete(x)$, $delete(y)$. The x and y values are symbolic. Table 1 shows how our approach deals with scaling. Each container was analyzed for $n = 2 \ldots 5$, starting with an empty cache. For $n = 2$, the time ratio for symbolic execution is quite high because there are fewer SAT queries, and the time consumption with and without the store is dominated by the overhead of the analysis. For larger n, T_S improves and the best results in this section is the five-fold speed increase for `BinTree` when $n = 5$. For #SAT, the $n = 2$ values are lower (= better) because the #SAT queries are more expensive, even though the number of SAT and #SAT queries are the same. For `BinomialHeap` and $n = 5$ there is a ten-fold increase in execution speed, and the value of $T_\#$ for `BinTree` and `TreeMap` may be even higher, but those experiments exceeded a pre-set time limit.

Table 2: Large regression

Symbolic execution			
	T_S	R_S	Self-reuse
`BinTreeBug`	0.322	0.806	0.806
`BinTree`	0.128	1.000	0.000
Model counting			
	$T_\#$	$R_\#$	Self-reuse
`BinTreeBug`	0.146	0.744	0.744
`BinTree`	0.147	1.000	0.000

4.1.3 Larger Regression

The `remove` operation of `BinTree`, the binary search tree implementation, has two variants: the original code contains a number of subtle but serious bugs that make it impossible to remove the root element of the tree, and that incorrectly deletes subtrees under very special conditions. The `remove` operation has been replaced by a correct implementation (and we use it for the other experiments in this section), but it is instructive to look at how our approach fares when dealing with such a large regression where a substantial piece of code has changed, but the general purpose of the program remains the same. The correct version of `remove` contains 80 LOC compared to 44 LOC in the buggy implementation, and the structure of these methods differ significantly.

Table 2 shows the results of analysis of the incorrect `BinTreeBug` with an initially empty store, followed by the correct version `BinTree`. As the analysis proceeds, the store is populated with query results. The first time a query is encountered, its results are calculated and stored. When the same result is found again, it is already in the store and does not need to be recalculated. We call this *self-reuse*. As the first line of Table 2 shows, this is the only kind of reuse available during the analysis of `BinTreeBug` in this scenario. On the other hand, the subsequent analysis of `BinTree` performs no self-reuse, because all of its queries have already been calculated and stored by the `BinTreeBug` analysis.

4.2 Across Programs

Can we achieve a similarly high level of reuse even when the programs are not the same? A natural starting point is to consider programs that while not the same, share some common functionality. The container programs `Binomial-`

Table 1: Performance on scaling examples

Symbolic execution

	T_S				R_S			
	$n=2$	$n=3$	$n=4$	$n=5$	$n=2$	$n=3$	$n=4$	$n=5$
`BinomialHeap`	0.961	0.720	0.443	0.259	0.643	0.827	0.865	0.896
`BinTree`	0.970	0.607	0.317	0.206	0.615	0.763	0.835	0.850
`TreeMap`	0.963	0.706	0.402	0.316	0.571	0.723	0.768	0.770

Model counting

	$T_\#$				$R_\#$			
	$n=2$	$n=3$	$n=4$	$n=5$	$n=2$	$n=3$	$n=4$	$n=5$
`BinomialHeap`	0.654	0.327	0.209	0.105	0.467	0.677	0.751	0.802
`BinTree`	0.573	0.259	0.143	–	0.500	0.675	0.782	–
`TreeMap`	0.782	0.588	0.438	–	0.250	0.235	0.258	–

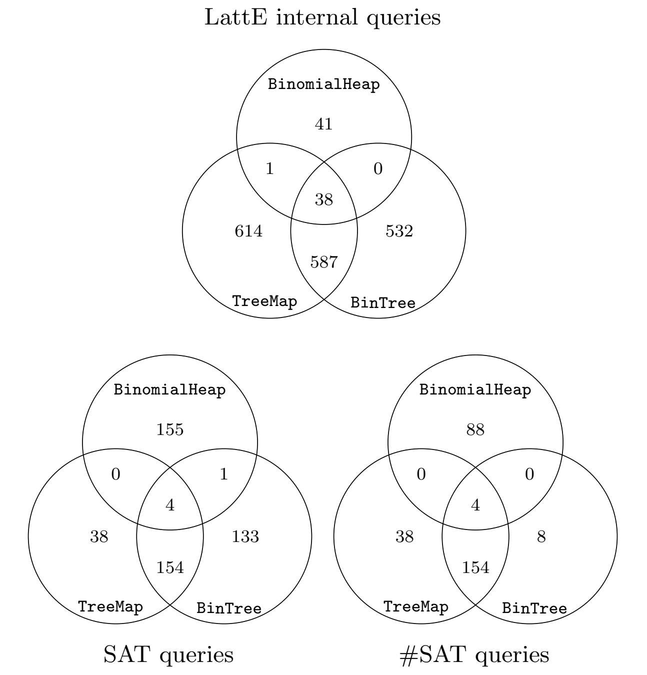

Figure 4: Shared keys for `BinomialHeap`, `BinTree`, and `TreeMap`

`Heap`, `BinTree`, and `TreeMap` are good examples of such programs since they record a collection of integer values and provide operations to add, remove, and check for containment of values – among other operations. Can the results from one of these programs be reused by the others?

The three Venn diagrams in Figure 4 show that this is indeed the case. The lower left diagram shows that 80.6% $(0+4+154=158$ of $158+38=196)$ of the SAT queries of `TreeMap` are shared by `BinomialHeap` and `BinTree`. For `BinTree` the figure is 54.5% while for `BinomialHeap` it is only 3.1%. This share is a similarly measly 4.3% for the number of #SAT queries as shown in the right lower diagram. However, the model counting process breaks each #SAT query into a number of subqueries and their results are also stored. This subresults reuse is shown in the upper diagram: `TreeMap` shares 50.5% of its subqueries with the others, `BinTree` shares 54.0% subqueries, and `BinomialHeap` shares 48.8% subqueries.

Of course one should keep in mind that even relatively small changes to the programs can reduce the potential for reuse dramatically. Nevertheless, we believe that the reuse data shown here raises some interesting questions about the commonality of constraints encountered across programs and we plan to conduct a large and broad study to better understand reuse opportunities.

4.2.1 Matrix of Programs

Taking our six programs (and some of the variants of the containers) in pairs, we evaluated the reuse that occurs when, starting with an empty store, the first program is analyzed and then the second. The values of R_S for this exercise are shown in Table 3. The first-run programs are shown in the leftmost column, and the second-run programs in the top row. At one extreme, there are many pairs with no shared constraints. At the other end of the spectrum are the related programs (such as the pair $(\texttt{BinTree}_4, \texttt{TreeMap}_3)$) with a high level of reuse, as we expected and demonstrated above. Encouragingly, there are many pairs in the middle ground that are not related but nevertheless have many common constraints.

4.2.2 Trends Over Sequences

Even unrelated programs may still share some, even many, constraints. There are 720 possible orderings of our six programs. From this we selected a random sample of 100 and, for each ordering, we analyze the six programs one after another starting with an empty store. Naturally, the first program in such a sequence cannot reuse any previous work (since there is none). We measure the reuse that occurs when the second, third, fourth, and so on, program is run. The minimum, maximum, median, and lower and upper quartile of the R_S values are presented as box plots in Figure 5. First note that the distribution is quite broad, stretching from 0.100 to close to 0.950. It is encouraging that all of the quartiles move towards 1.000, indicating that reuse is taking place, and that the store is being populated with computed queries.

4.3 Across Tools

To explore the generality of the Green framework, we performed a case study that involved refactoring an existing analysis to use Green. We then compared the performance

Table 3: 2×2 matrix of across-program reuse

	`BinTree`$_3$	`BinTree`$_4$	`TreeMap`$_3$	`TreeMap`$_4$	`BinoHeap`$_3$	`BinoHeap`$_4$	`Euclid`	`TriTyp`	`TCAS`
`BinTree`$_3$	–	0.357	1.000	0.342	0.351	0.311	0.000	0.000	0.012
`BinTree`$_4$	1.000	–	1.000	0.880	0.351	0.311	0.000	0.000	0.012
`TreeMap`$_3$	0.747	0.232	–	0.342	0.179	0.115	0.000	0.000	0.012
`TreeMap`$_4$	0.747	0.712	1.000	–	0.179	0.115	0.000	0.000	0.012
`BinoHeap`$_3$	0.200	0.160	0.234	0.154	–	0.743	0.000	0.188	0.012
`BinoHeap`$_4$	0.200	0.160	0.234	0.154	1.000	–	0.000	0.188	0.012
`Euclid`	0.000	0.000	0.000	0.000	0.000	0.000	–	0.000	0.133
`TriTyp`	0.000	0.000	0.000	0.000	0.179	0.115	0.000	–	0.000
`TCAS`	0.116	0.076	0.234	0.154	0.179	0.115	0.065	0.000	–

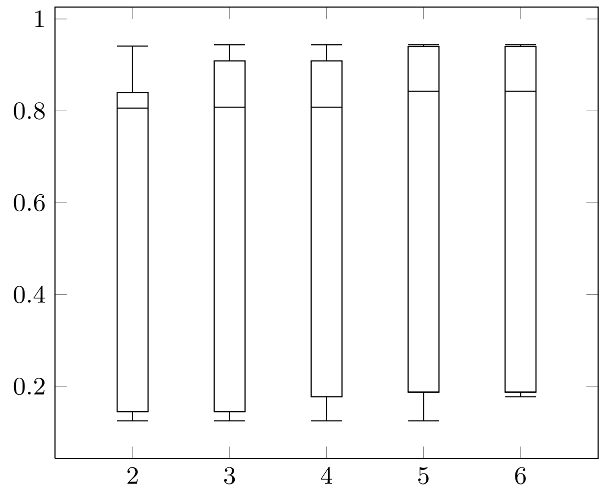

Figure 5: Sequences

of the solver-based implementation with reuse and running in combination with traditional symbolic execution.

4.4 Refactoring a DSE

We selected an analysis that has been implemented in the third author's research group, but with which none of the authors are involved. The analysis performs an offline form of dynamic symbolic execution (DSE). A program is analyzed using the SOOT analysis framework and instrumentation is inserted to record traces which encode symbolic constraints that correspond to the branch outcomes along a program execution. An offline component processes the trace encoding a path condition and converts it into a set of constraints which are then solved. More specifically, for path condition $c_1 \wedge c_2 \wedge \ldots c_k$ the following constraints are generated and solved: $(\neg c_1)$, $(c_1 \wedge \neg c_2)$, ..., $(c_1 \wedge c_2 \wedge \ldots c_{k-1} \wedge \neg c_k)$.

The offline DSE analysis component consists of 2400 lines of Java source code. The Choco solver is used to solve constraints. The core component of the analysis is the *trace interpreter*; it consists of 499 lines of Java code. The interpreter reads in the recorded trace file, constructs a representation of the constraints, and uses the Choco API to produce a Choco-specific representation of the constraints. Each of those constraints is then solved by calling a pair of methods in the Choco API. It took us approximately 5 hours to understand the analysis and approximately 1.5 hours to refactor the 41 lines needed to convert the Choco-specific processing to use Green's APIs.

Table 4: Multiple analysis performance

Analysis	Order	Time (sec)	DP Time (sec)	DP calls	DP reuse
DSE original	1st	9.360	5.690	416	0
DSE solver	1st	1.620	0.067	35	381
SymExec	2nd	1.000	0.070	30	648
SymExec	1st	1.020	0.095	44	634
DSE solver	2nd	1.510	0.043	6	410

We did not talk to the developers of the DSE analysis until after we had completed the refactoring. We discussed our changes with them and showed them the output of the system tests we ran on the refactored analysis. They confirmed that the changes looked correct and the results were as expected.

We note that in the refactored analysis we can shift between Choco and other solvers by modifying a single constructor parameter. In the results discussed below we used CVC3 as the decision procedure.

4.5 Performance of Solver-based DSE

We explored the performance of the original and refactored analysis using the Java version of the `TCAS` program. The DSE analysis requires a test suite to generate traces and we used a pre-existing branch adequate test suite consisting of 16 tests.

Table 4 shows the performance of several variants of the analysis in isolation and in combination with a solver-based symbolic execution. *DSE original* is a version of the DSE analysis that neither slices constraints nor reuses solver results; we used CVC3 as the solver to enable the performance resulting from reuse to be apparent. *DSE solver* enables slicing, canonization, and reuse. The second and third rows were run in order after clearing the results store; similarly for the fourth and fifth rows.

Enabling slicing and reuse reduced the solver time significantly, by over 98%, and the overall runtime by more than 82%. This is due to the fact that more than 91% of the constraints had solutions that were reused. Running DSE before symbolic execution resulted in a modest improvement in the performance of symbolic execution, since an additional 14 out of 30 constraints had solutions computed by DSE that could be reused. Switching the order of the analysis had a more significant effect on DSE, which saved the computation of 29 out of 35 constraints. We expected that DSE

Table 5: **Effect of slicing, canonization, and the store**

	–store		*+store*	
	–canon	*+canon*	*–canon*	*+canon*
–slice	95506	94739	96448	50467
+slice	27129	27369	20410	5603

	–store		*+store*	
	–canon	*+canon*	*–canon*	*+canon*
+slice	0.284	0.289	0.212	0.111

	–store		*+store*	
	–slice	*+slice*	*–slice*	*+slice*
+canon	0.992	1.009	0.523	0.275

	–slice		*+slice*	
	–canon	*+canon*	*–canon*	*+canon*
+store	1.010	0.752	0.533	0.205

run after symbolic execution would have perfect reuse, but when analyzing the 6 constraints that could not be reused we discovered that they are all related to DSE dropping a single conjunct when processing a particular branch (we have reported the problem to the developers).

4.6 Slicing and Canonization

Table 5 presents a more detailed analysis of the contribution of slicing, canonization, and storage. The `BinomialHeap` program was run eight times to investigate all the possible combinations of these three components. In each run, the parameter value $n = 5$ (explained in Section 4.1.2) is used. The eight values in the top two data rows of the table give the time, in milliseconds, spent in the decision procedure. (Recall that we are using CVC3.) The leftmost column and the two rows in the heading specify the components that were in use: $–x/+x$ means that component x is inactive/active.

The three lower parts of the table summarize the impact of each of the three components. For instance, when canonization is on, but the store is not, switching slicing on reduces the time from 95506 to 27129 milliseconds, and the impact factor is 27129/95506 = 0.284. The analysis of the program produced 6636 path conditions of which 863 were unique. Altogether, there were 92414 (non-distinct) conjuncts which slicing reduced by 45.1% to 50720 conjuncts.

The first thing to notice is that without slicing and canonization, the storage takes more time, since there is no results reuse, but there is some overhead to use the store. However with both slicing and canonization enabled, the improvement from using a store is almost an order of magnitude. Storing actually also benefits from just the canonization by almost a factor of 2. However slicing by itself improves reuse almost 5-fold.

If we look at the impact of each optimization individually, it is clear that slicing has the biggest impact on the performance of the decision procedure. This is a worthwhile result by itself, since it is quite easy to implement slicing in a symbolic execution framework and clearly the benefit is substantial. Canonization didn't help the decision procedure at all. In theory this step could produce constraints in a form that benefits the decision procedure, but we intended it to increase reuse (i.e., when storage is enabled). Since we are only considering reuse within the same analysis run here (self-reuse), storage must be used with one of the other optimizations.

The exact ratios reported here are clearly a function of the particular decision procedure and how it copes with the nature of the constraints that a particular problems produces, as well as the structure and complexity of the path conditions. We have selected a representative (but not the best possible) program and parameter size.

5. RELATED WORK

Researchers have been exploring the application of constraint solving to program analysis for over three decades, building on the work on Nelson and Oppen [18]. During this time tremendous advances have been made both in constraint solving and in program analyses that have made their combination effective in reasoning about large complex software systems, e.g., see [2].

For the most part, these advances have been developed *within* a specific system and this has limited their broad adoption by the research community. We discuss several of these systems below and highlight the novel techniques that they introduced. Green incorporates those techniques, adds additional capability in the form of canonization and disk-based caching, and, importantly, presents this suite of capabilities in a form that is both analysis- and solver-independent.

5.1 Slicing and Caching

When presented with a complex constraint to solve it is natural to factor it into independent subproblems. This strategy has long been applied in research on constraint solving and it is present today in modern SMT solvers – which purify formulas into theory-specific sub-formulas for solution. This kind of decomposition can enable customized algorithms to be applied to the independent subformulas, as in SMT, or can simply decrease the time to compute a solution, because there are fewer clauses or variables. Several researchers have observed an additional benefit of problem decomposition – decomposed constraints recur across the space of possible solutions.

An example of this strategy applied within a solver is the integration of *component caching* into an extension of a DPLL-based SAT solver, zChaff [31], to perfom model counting [24]. Model counting, as discussed earlier, is a particularly expensive form of constraint solving that requires the exhaustive consideration of the set of satisfiable solutions. The insight in this work is that by identifying independent fragments of a formula that recur across the solution space, one can count the solutions for a fragment a single time, cache that count, then retrieve it when the fragment is encountered later in the search. This amounts to achieving the kind of reuse we provide, but *within a single analysis run only*. It is important to note that when attempting to generalize this approach to count models over richer theories, such as LIA, there are additional opportunities for reuse when one canonizes constraints as we do.

In our own recent work, we see these benefits play out when applying model counting to a variant of symbolic execution that calculates path probabilities [10]. In fact, the inspiration for Green was our experience developing this probabilistic symbolic execution. While we did not store model counts across program runs, the benefits of canonization and reuse were very apparent. Since the development of Green we have refactored our own probabilistic symbolic execution system[3] to use Green rather than implement its capabilities as part of the process of invoking a specific external tool.

An example of constraint solver reuse applied outside of the solver within an analysis tool is the EXE symbolic execution system [4]. The key insight here is that the structure of a symbolic execution is such that when a program has independent branches, the path condition will be comprised of independent constraints. Decomposing those constraints offers the opportunity for reducing the number of calls to a constraint solver. This is a key feature of EXE, and its successor KLEE, that allow it to scale to large programs. Our work builds on this insight and takes it a step further. Whereas EXE can decompose independent sub-problems it can only determine sub-problem equivalence through syntactic equivalence. Our framework exposes a wide-range of additional equivalences and expresses them as syntactic equivalence on the canonized constraints. We believe that state-of-the-art tools like EXE and KLEE could further benefit from canonization.

5.2 Canonizing Constraints

Canonization of logical formulas and its use in supporting decision procedures has been studied, e.g., [7]. An important application of these ideas is its support for identifying equivalences within a given formula [8]. The key insight here is that if one can define a canonical form, then formulas converted to that form expose semantic equivalence as syntactic equivalence. This is precisely the same insight we leverage, except that we exploit syntactic equivalence to improve reuse rather than judge two formulas equivalent.

There are many settings in which the general strategy of equivalence-based reduction has been developed and applied to improve the performance of program analyses. For example, the ideas of state and thread symmetries have been used to perform property-preserving state-space reductions in software model checking [12]

5.3 Regression Symbolic Execution

In recent years, there have been a number of efforts focused on program equivalence checking, or differencing, that exploit deep program analyses of the type provided by symbolic execution. In such analyses, one attempts to co-analyze two program versions which are, often, very similar.

This first work that we are aware of to consider this problem is that of Siegel et al. [25]. While not strictly about regression analysis, this analysis constructed a driver that placed a sequential version of a program first, followed by a parallel version of the same program. The path conditions accumulated along the analysis of the first effectively *focus* the analysis along the second program. This approach uses a custom-built constraint solver that implements value numbering – a classic compiler optimization technique that computes a canonical form for expressions and is used to detect equivalences. Like other constraint solvers, this technique can ignore redundant constraints. They do not perform anything like slicing and we believe that their technique could be improved by doing so.

Our own work [19] used a form of overapproximating symbolic execution to *skip* portions of the program that are provably identical across the versions. In followup work, Person et al. developed a variant of this approach that targets the SPF symbolic execution system at representative paths through the regions of the programs that have been potentially impacted by a change [20]. While these approaches clearly exploit the similarity between programs, after all they are explicitly *regression analyses*, they do not exploit the fact that the constraint solver calls performed on the two versions are likely to be very similar. This could easily be added to their analyses using Green.

In light of the recent work on regression symbolic execution, the developers of EXE have targeted their techniques at the problem [22]. As with the other work along these lines, their technique achieves a kind of reuse across the analysis of two program versions by integrating them into a single analysis run. While we don't focus on regression analysis, it is clear from the results of our evaluation that Green permits a similar kind of reuse and not just for pairs of program versions, but across entire version histories.

Finally, we have recently become aware of an alternative approach to symbolic execution that *memoizes* portions of the symbolic execution tree [29]. This approach records a much richer body of information – the tree of symbolic constraints computed by the analysis – which can be exploited in subsequent analyses. When applied to regression analysis, this would offer the potential to completely skip exploration of portions of the program behaviors, whereas a symbolic execution using our solver would only skip the solver calls in those portions. When symbolic execution does need to be performed, we believe that our solver can offer reuse opportunities to speed up their analysis. Moreover, Green is more broadly applicable across different analyses, because it is not tied directly to symbolic execution.

6. CONCLUSION AND FUTURE WORK

We have presented Green– a framework that we believe offers benefits for a wide-range of program analyses that use constraint solvers. We have shown evidence that constraint analysis results can be reused both within the run and across runs and that this can significantly reduce analysis time. Perhaps the most surprising finding is that one can reuse constraint solutions from analyses of completely different programs. We also showed the intriguing possibility of reusing results between different tools. In our case it was on the same program, but of course it could also be across programs.

Our future work involves extending the range of theories and solvers supported by Green, developing strategies for efficiently sharing constraint solutions in a network of stores shared by the community, and exploring the extent to which different programs share common constraints.

Acknowledgments

Matthew Dwyer's work was supported in part by a Research Award from the United State's Fulbright Scholar Program to visit Stellenbosch University, South Africa from August through December of 2011. The authors would like to thank Elena Sherman for sharing her DSE application with them and for helping to assess our refactoring of her code.

[3]`http://probsym.googlecode.com`

7. REFERENCES

[1] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[2] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2), Feb. 2010.

[3] C. Cadar, D. Dunbar, and D. Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[4] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2):10:1–10:38, Dec. 2008.

[5] Choco. Choco Solver. `http://www.emn.fr/z-info/choco-solver/`.

[6] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms.* McGraw-Hill Higher Education, 2001.

[7] L. de Moura, S. Owre, H. RueÃ§, J. Rushby, and N. Shankar. The ICS decision procedures for embedded deduction. In *Automated Reasoning*, volume 3097, pages 218–222. Springer, 2004.

[8] L. de Moura, H. Rueß, and N. Shankar. Justifying equality. *Electron. Notes Theor. Comput. Sci.*, 125(3):69–85, July 2005.

[9] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Trans. Software Eng.*, 17(9):900–910, 1991.

[10] J. Geldenhuys, W. Visser, and M. B. Dwyer. Probabilistic symbolic execution. In *Proceedings of the 21st International Symposium on Software Testing and Analysis*, 2012. To appear.

[11] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, Mar. 2012.

[12] R. Iosif. Symmetry reductions for model checking of concurrent dynamic software. *International Journal on Software Tools for Technology Transfer (STTT)*, 6:302–319, 2004.

[13] Jedis. Redis Java Interface. `http://code.google.com/p/jedis/`.

[14] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 2008(5):1–32, 2011.

[15] Jumble. Jumble mutation testing tool. `http://jumble.sourceforge.net/`.

[16] LattE. LattE Integrale, UC Davis, Mathematics. `http://www.math.ucdavis.edu/~latte`.

[17] G. J. Myers. *Art of Software Testing.* John Wiley & Sons, Inc., 1979.

[18] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, Oct. 1979.

[19] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 226–237, 2008.

[20] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 504–515, 2011.

[21] C. S. Păsăreanu and N. Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 179–180, New York, NY, USA, 2010. ACM.

[22] D. Ramos and D. Engler. Practical, low-effort equivalence verification of real code. In *Computer Aided Verification*, pages 669–685. 2011.

[23] Redis. Redis NoSQL Database. `http://redis.io`.

[24] T. Sang, F. Bacchus, P. Beame, H. A. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing*, 2004.

[25] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 157–168, 2006.

[26] SIR. SIR Repository. `http://sir.unl.edu`.

[27] N. Tillmann and J. De Halleux. Pex: white box test generation for .net. In *Proceedings of the 2nd international conference on Tests and proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[28] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for Java containers using state matching. In L. L. Pollock and M. Pezzè, editors, *ISSTA*, pages 37–48. ACM, 2006.

[29] G. Yang, C. Păsăreanu, and S. Khurshid. Memoized symbolic execution. In *Proceedings of the 21st International Symposium on Software Testing and Analysis*, 2012. to appear.

[30] Yikes. Yices SMT Solver. `http://yices.csl.sri.com/`.

[31] zChaff. SAT Research Group, Princeton University. `http://www.princeton.edu/~chaff/zchaff.html`.