Reusing Constraint Proofs in Program Analysis

Andrea Aquino[§], Francesco A. Bianchi[§], Meixian Chen[§], Giovanni Denaro[♯], Mauro Pezzè[§♯]

[§] Università della Svizzera italiana (USI)
Via Giuseppe Buffi, 13
Lugano, Switzerland 6900
{aquina, biancfr, chenm, pezzem}@usi.ch

[♯] University of Milano-Bicocca
Viale Sarca, 336
Milano, Italy 20126
denaro@disco.unimib.it

ABSTRACT

Symbolic analysis techniques have largely improved over the years, and are now approaching an industrial maturity level. One of the main limitations to the scalability of symbolic analysis is the impact of constraint solving that is still a relevant bottleneck for the applicability of symbolic techniques, despite the dramatic improvements of the last decades.

In this paper we discuss a novel approach to deal with the constraint solving bottleneck. Starting from the observation that constraints may recur during the analysis of the same as well as different programs, we investigate the advantages of complementing constraint solving with searching for the satisfiability proof of a constraint in a repository of constraint proofs. We extend recent proposals with powerful simplifications and an original canonical form of the constraints that reduce syntactically different albeit equivalent constraints to the same form, and thus facilitate the search for equivalent constraints in large repositories. The experimental results we attained indicate that the proposed approach improves over both similar solutions and state of the art constraint solvers.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging — *Symbolic execution*

Keywords

Constraint solving for symbolic program analysis, constraint canonicalization, proof reuse

1. INTRODUCTION

Symbolic techniques are becoming increasingly popular for program analysis, and are now used for verifying industrially relevant software systems like device drivers and embedded software components [23, 13]. The dramatic improvement of both symbolic analysis approaches [8] and constraint solving techniques [14] has largely improved the scalability and the applicability of symbolic analysis, but has not yet eliminated the bottleneck of constraint solving [21, 2].

Modern constraint solvers are efficient and can deal with large constraints when applied within the boundaries of the considered theory, but become less and less efficient with formulas of increasing size, and remain bounded within the limits of the given theory.

Unfortunately, when symbolically analyzing complex software systems, execution paths become extremely large, and are characterized by enormous path conditions containing heterogeneous constraints. When analyzing realistic complex systems, it is common to deal with millions of formulas, many of which contain tens of thousands of clauses.

Recent work has observed that constraints recur quite often within the analysis of a single program, as well as across different programs [25]. This preliminary observation is confirmed by our study, whose results are reported later in this paper, and suggests the possibility of reusing constraint satisfiability proofs by incrementally storing and reusing them when constraints occur again during the analysis of either the same or a different program.

The reuse of satisfiability proofs presents several challenges: (i) eliminating redundancy and identifying sets of clauses whose satisfiability can be proved independently from the other clauses in the formula, (ii) generalizing formulas to reduce the impact of program specific details, for example by abstracting from concrete aspects like the specific variable names or the order of the clauses and the terms in the formula, (iii) defining a mechanism to efficiently store and retrieve satisfiability proofs.

Some of these challenges have been addressed in Klee [11] that proposes both constraint slicing to reduce the size of the constraints by identifying sub-constraints whose satisfiability can be proved independently, and a technique to identify sub and super formulas based on set inclusion, and Green [25] that proposes a syntactic normalization of constraints for saving and retrieving satisfiability proofs from a simple in-memory repository.

In this paper, we present the *REusing-Constraint-proofs-in-program-AnaLysis* (*Recal*) approach, a comprehensive approach to efficiently store and retrieve satisfiability proofs. *Recal* (i) includes the constraint slicing approach proposed in Klee to reduce the size of the constraints to be analyzed, (ii) proposes a new constraint normalization algorithm that goes beyond the simple syntactic normalization introduced in Green, and that includes intra-clause constraint simplification, inter-clause term pruning, and a canonical form for abstracting program specific details and efficiently saving and

ISSTA'15 , July 12–17, 2015, Baltimore, MD, USA
Copyright 2015 ACM 978-1-4503-3620-8/15/07 ...$15.00.
DOI: http://dx.doi.org/10.1145/2771783.2771802 .

retrieving satisfiability proofs, (iii) defines a way to compare constraints for inclusion that extends the mechanisms proposed in Klee by exploring the logical implications between clauses, to deduce the satisfiability of a constraint based on the satisfiability of super or sub constraints.

The main contribution of this paper are (i) the definition of a canonical form for constraints, and the design of an algorithm to reduce constraints to their canonical form to efficiently index satisfiability proofs and speed up proof search, (ii) the introduction of a technique for inter-program search by containment that allows the satisfiability of complex constraints to be inferred from the satisfiability of super or sub constraints, (iii) the integration of constraint simplification, normalization and search within a coherent approach that works for intra and inter program analysis, (iv) the presentation of a set of experimental results about the frequency of occurrence of constraints during program analysis and the effectiveness of the approach proposed in this paper, referring to a large benchmark of real and synthetic constraints.

This paper is organized as follows. Section 2 introduces the overall approach and the logical architecture of the proposed solution. Section 3 discusses the canonical form that represents a core element of the approach and a core contribution of the paper. Section 4 presents the experimental results that compare *Recal* with the state of the art solutions. Section 5 discusses the related work, highlighting the original contributions of this paper. Section 6 concludes summarising the contributions of this paper, and outlining our current research plans.

2. THE CONSTRAINT REUSE APPROACH

In this section we present our approach that stores and retrieves satisfiability proofs of solved constraints to infer the satisfiability of new constraints.

In this work we focus on constraints produced during symbolic execution that consist in formulas in conjunctive form, and we restrict our attention to quantifier-free linear integer arithmetic that represents a large set of formulas occurring during symbolic execution and is addressed by the most popular constraint solvers. Thus we focus on constraints expressed in terms of propositional formulas that are conjunctions of linear (in)equalities over integers, a proper subset of the quantifier-free linear integer arithmetic (QF_LIA) logic.

We search for satisfiability proofs of a given formula in four phases: (i) *Slicing* This phase slices a formula into sub-formulas that are mutually independent, that is, sub-formulas containing different variables, (ii) *Normalization* This phase simplifies a formula by transforming its clauses into a simpler equivalent set of clauses, (iii) *Canonicalization* This phase produces the canonical form, which encodes a formula as a matrix that uniquely identifies equivalent formulas, (iv) *Search* This phase retrieves constraint satisfiability proofs from the repository.

As shown in Figure 1, the phases are composed of steps that are discussed below.

2.1 Slicing

This phase implements the formula slicing algorithm proposed in Klee by Cadar et al. [11] that splits a formula into a set of independent sub-formulas. Two formulas are independent if and only if they do not share any variable. The satisfiability of a complex formula can be determined from the satisfiability of its sub-formulas: if all the independent sub-formulas are satisfiable, the original formula is satisfiable too, otherwise the original formula is unsatisfiable. Slicing formulas produces smaller formulas that are easier to handle and more likely to be equivalent to previously solved formulas.

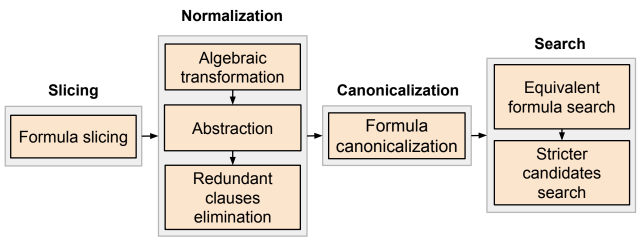

Figure 1: Approach overview

Original constraint:

$C_0 \equiv a + 3b < 2 \wedge c < 0 \wedge x - 1 < 6 - 2x \wedge 2a + c \neq 1 \wedge x \neq 4$

Constraint after slicing: $C_0 \equiv C_1 \wedge C_2$

$C_1 \equiv a + 3b < 2 \wedge c < 0 \wedge 2a + c \neq 1$
$C_2 \equiv x - 1 < 6 - 2x \wedge x \neq 4$

Figure 2: Example of slicing

The formula slicing algorithm (i) initializes an undirected graph with a node for each clause in the formula, (ii) connects the nodes that correspond to clauses that share at least a variable, (iii) extracts the sub-formulas that correspond to the connected components of the graph. Figure 2 presents a simple example of formula slicing.

The sub-formulas produced in the Slicing step are treated independently in the next phases and are aggregated at the end to infer the satisfiability value of the original formula.

2.2 Normalization

The normalization phase transforms a formula into a simpler equivalent one by applying algebraic transformations, abstracting variable names and eliminating redundant clauses, as illustrated in Figure 1.

The *algebraic transformation* step simplifies a formula by applying three simplifications to its clauses: (i) summing all the terms that refer to the same variable, and the constant terms, (ii) replacing the comparison operator with either $\leq$ or $\neq$ according to the rules described in Figure 3, (iii) dividing all the coefficients by their greatest common divisor.

Original (in)equality	Simplified (in)equality
$terms \leq k$	$terms \leq k$
$terms \neq k$	$terms \neq k$
$terms < k$	$terms \leq k - 1$
$terms \geq k$	$-1 * (terms) \leq -k$
$terms > k$	$-1 * (terms) \leq -k - 1$
$terms = k$	$terms \leq k \wedge -1 * (terms) \leq -k$

Figure 3: Comparison simplifications

The *abstraction* step abstracts a formula as a matrix. Given a formula with n clauses and m variables, it associates the clauses with indexes from 1 to n and the variables with indexes from 1 to m, and transforms the formula into a matrix M of size $n \times (m + 2)$ such that:

Input formula	$x - 1 < 6 - 2x \wedge x \neq 4$
Summing up terms	$3x < 7 \wedge x \neq 4$
Transforming comparison operators	$3x \leq 6 \wedge x \neq 4$
Dividing clauses by their gcd	$x \leq 2 \wedge x \neq 4$
Abstraction	$\begin{vmatrix} 1 & 2 & \leq \\ 1 & 4 & \neq \end{vmatrix}$
Redundant clauses elimination	$\begin{vmatrix} 1 & 2 & \leq \end{vmatrix}$

Figure 4: Example of normalization

- $M[i, j] = c$ iff the i-th clause of the formula contains term $c * v_j$ and $1 \leq j \leq m$,
- $M[i, m + 1]$ = constant term of the i-th clause,
- $M[i, m + 2]$ = comparison operator of the i-th clause (either $\leq$ or $\neq$).

We call the $(m+1)$th column of the resulting matrix the *constant terms column*, the $(m+2)$th column the *comparisons column* and the rest of the matrix the *terms sub-matrix.*

The *redundant clauses elimination* step removes redundant clauses from a formula. A clause is redundant if it is implied by other clauses in the formula and thus can be removed without affecting the satisfiability of the formula. *Recal* eliminates the clauses which are redundant according to the following definition: given a clause c_1 in a formula, c_1 is redundant if the formula contains a clause c_2 characterized by the same variable coefficients of c_1, and with either (i) the same comparison operator and the same constant term of c_1, or (ii) the comparison operator $\leq$ and the constant term that is less than the constant term of c_1. In a nutshell, a clause is redundant if it is equal to (or identifies a larger set of values than) another clause in the formula.

Figure 4 illustrates the normalization phase through a simple example, spelling out the three elements of the algebraic transformation step.

2.3 Canonicalization

The formula canonicalization phase produces the canonical form of a formula. The canonical form consists in a matrix representation that is unique for equivalent formulas. Two formulas are equivalent if they can be transformed into each other by reordering clauses and renaming variables. We discuss the algorithm that produces the canonical form by permuting rows and columns of the matrix representation of the formula in detail in the next section.

2.4 Search

The canonical form is the core element for an efficient search of constraint proofs. When searching for a proof, we first search for equivalent formulas, and then for stricter candidates that are formulas whose satisfiability implies the satisfiability of the target one.

To search for equivalent formulas, we exploit the canonical form to build an efficient search index. The canonical form reduces the complex problem of comparing formulas for equivalence to the simpler problem of comparing their canonical forms for equality, thus we use the canonical form to index the repository of satisfiability proofs.

Technically, we hash the canonical form of the target formula, and we perform a quick lookup into the index that stores pairs of the form <*canonical_form_hash*, *satisfiability_proofs_reference*> retrieving all the entries in the repository that contain proofs for the target formula.

If the search does not return any entry, *Recal* searches for a stricter candidate. In general, a formula F_1 is stricter than a formula F_2 if the satisfiability of F_1 implies the satisfiability of F_2, and thus we can reuse the solutions of F_1 to produce valid solutions for F_2 or we can deduce that F_1 is unsatisfiable if F_2 is unsatisfiable.

It is easy to prove that if a satisfiable formula F is stricter than a formula G, then G is satisfiable, and if an unsatisfiable formula F is weaker than a formula G, then G is unsatisfiable. The proof refers to the well known property that the logical implication maps to a subset relation in the solution space. Thus the satisfiability value of a formula can be deduced from the satisfiability proof of any stricter satisfiable formula or any weaker unsatisfiable formula.

Recal automatically identifies a specific kind of stricter relation that can be deduced from clause comparison as follows: a formula F is stricter than a formula G iff for each clause $c_G \in G$ there exists a clause $c_F \in F$ that implies c_G. Notice that F can contain additional clauses not corresponding to clauses in G. We also say that the formula G is weaker than F. For example, the formula $x \leq 2 \wedge x + y \leq -1 \wedge y \leq 0$ is stricter than the formula $a \leq 3 \wedge a + b \neq 0$. In this example the correspondence between the first two clauses of the formulas ($x \leq 2 \Rightarrow a \leq 3$ and $x + y \leq -1 \Rightarrow a + b \neq 0$) is straightforward once defined the correspondence between x, y and a, b ($x = a$ and $y = b$).

To identify a formula which is stricter or weaker than a given formula it is necessary to compare the latter with all the formulas in the repository, and this has an unacceptable cost.

Starting from the observation that the redundancy relation we introduced above is a special case of logical implication among clauses, we developed a technique that enables a fast search for weaker and stricter formulas exploiting such relation.

Drawing inspiration from the Google Search term-to-pages inverted index [3], we created an inverted index that associates the clauses of the formulas in the repository with all the formulas that contain such clauses. The inverted index stores pairs of the form $\langle$*clause_entry*, $\{\langle$*formula_reference*, *comparison_operator*, *free_coefficient*$\rangle, \ldots\}\rangle$, one for each clause that appears in a formula in the repository. The clause entries depend on the coefficients of the variables as computed by the *canonicalization* algorithm. Thus, clauses with the same sets of variable coefficients map to the same clause entry in the inverted index. The inverted index associates each clause entry with the set of formulas that contain a clause c with the same variable coefficients of the clause entry, and keeps also track of the comparison operator and the constant term of the clause c.

The inverted index allows logarithmic time access to the set of formulas that contain a given list of clauses, thus dramatically reducing the complexity of searching for stricter formulas.

Given a formula F that contains the clauses $c_1, \ldots, c_n$, we search for stricter candidates as follows. For each clause c_i, we access the inverted index to identify the set $S_i = \{G^i_1, G^i_2, ...\}$ of formulas that contain a clause that logically implies c_i. The comparison operators and the constant terms stored in the inverted index allow us to evaluate the logical implication between the clauses according to the definition of clause redundancy that we have given above. We compute the intersection of the sets S_i to identify the formulas that contain a corresponding clause for any clause in F and, if the intersection set is not empty, we select a satisfiable formula G out of this set. Since both F and G are in canonical form, we know that for each clause in F there is a corresponding stricter clause in G that refers to the same variables. Thus G is a satisfiable formula that is stricter than F.

To search for weaker candidates, we identify the set of formulas with clauses that can be logically implied by any c_i in F using the inverted index. This is the set of formulas that can be weaker than F, and we refer to these formulas to identify an unsatisfiable formula that is weaker than F, thus proving it unsatisfiable.

In this way, we reduce the problem of comparing a target formula for stricter-/weaker-ness with all formulas in the repository to comparing the comparison operator and the constant term of the target formula with the ones of a small subsets of clauses identified with the inverted index.

3. CANONICAL FORM

The core of our approach is an efficient method to determine whether two linear constraints in conjunctive form represent the same constraint, regardless of both the order of the clauses and the terms they contain, and the name of their variables. If it is the case, we can reuse the solutions or the proof of unsatisfiability of one of them to solve the other. For example, the two constraints

$$C_1 : 3x + y \leq 0 \land x + 2y \leq 0$$
$$C_2 : 2a + b \leq 0 \land a + 3b \leq 0$$

can be reduced to the same constraint by swapping the two clauses of the second constraint, renaming the variables x and b as v_1 and the variables y and a as v_2, and writing the terms that refer to variable v_1 before the ones that refer to variable v_2 in each clause, thus obtaining:

$$C_1 \equiv C_2 \equiv \; 3v_1 + v_2 \leq 0 \land v_1 + 2v_2 \leq 0.$$

This rewriting of the two constraints indicates a straightforward mapping between the solutions of C_1 and C_2.

In this section, we formalize this notion of equivalence as an equivalence relation among the conjunctive linear constraints that can be rewritten as the same constraint by suitably permuting clauses and variables, and define a canonical form of the constraints in the equivalence classes of this relation. The canonical form allows us to compare two constraints for equivalence by comparing their canonical forms for equality. Thus, the problem of determining whether a constraint C is equivalent to some constraint out of a set of constraints $\{C_1, C_2, ..., C_n\}$ can be reduced to the problem of determining whether the canonical form of C is the same as the canonical form of some constraint in the set.

We define the canonical form of a conjunctive linear constraint C as the constraint $\hat{C}$ that corresponds to the fixed point of a deterministic algorithm that iteratively permutes the clauses and the variables of C. The algorithm, called *canonicalization*, always converges to a fixed point and guarantees that the constraint at the fixed point is unique for all input constraints that belong to the same equivalence class.

Definition 3.1. *Let $LC_\land$ be the set of all conjunctive linear constraints.*
Let $permute(C) \in 2^{LC_\land}$ denote the set of constraints that can be obtained by permuting the clauses and the variables of a constraint $C \in LC_\land$.
Let $\equiv$ be an equivalence relation over $LC_\land$ such that $C_1 \equiv C_2 \iff C_1 \in permute(C_2)$, that is, C_1 and C_2 are equivalent if it is possible to permute properly the clauses and rename the variables of C_2 to obtain C_1. It is easy to verify that this relation is reflexive, symmetric and transitive.
Then, $canonicalization : LC_\land \to LC_\land$ is a deterministic and always terminating algorithm such that:

$$\forall C \in LC_\land,\; canonicalization(C) \in permute(C)$$

$$\forall C_1, C_2 \in LC_\land, C_1 \equiv C_2 \iff canonicalization(C_1) = canonicalization(C_2).$$

Algorithm 1 specifies *canonicalization*. The algorithm starts producing the matrix representation of the input formula (line 1). Given a formula with n clauses and m variables, the algorithm numbers progressively the variables that appear in the formula, builds an $n \times m$ matrix, and sets the value of the (i,j)th element of the matrix to the coefficient of the j-th variable in the i-th clause. If a variable does not appear in a clause, the algorithm sets the corresponding coefficient to 0. The algorithm augments the matrix with two columns $m+1$ and $m+2$ containing the constant terms and the comparison operators of the formula, respectively. Each row of the resulting matrix represents a clause in the original formula. The top part of Figure 5 shows the matrices obtained from some sample constraints (Matrix at the beginning of *canonicalization*).

Algorithm 1 *canonicalization* (C)

```
Require:
    $C \in LC_\land$, a conjunctive linear constraint
Ensure:
    Returns a matrix representing the canonical form of $C$
1.  $M \leftarrow constraintAsMatrix(C)$
2.  /* Phase 1 begins */
3.  $M \leftarrow orderRowsByComparisonOp(M)$
4.  $M \leftarrow orderRowsByConstantTerm(M)$
5.  if $converged(M)$ then
6.      return M
7.  /* Phase 2 begins */
8.  $M \leftarrow orderRowsByGreatestValues(M)$
9.  $M \leftarrow orderColsByGreatestValues(M)$
10. if $converged(M)$ then
11.     return M
12. /* Phase 3 begins */
13. repeat
14.     $M' \leftarrow M$
15.     $subM \leftarrow extractItemsWithStableCols(M)$
16.     $M \leftarrow orderRowsLexicographicallyBySubM(M, subM)$
17.     $subM \leftarrow extractItemsWithStableRows(M)$
18.     $M \leftarrow orderColsLexicographicallyBySubM(M, subM)$
19. until not $changed(M, M')$ or $converged(M)$
20. if $converged(M)$ then
21.     return M
22. /* Phase 4 begins */
23. $subM \leftarrow extractRowsAndColsWithUnstableOrder(M)$
24. $permutations \leftarrow enumerateAllValidPermutations(subM)$
25. $p \leftarrow lexicographicMax(permutations)$
26. $M \leftarrow applyPermutation(M, p)$
27. return $M$
```

The algorithm processes the matrix representation of the input constraint in four phases (starting at lines 2, 7, 12 and 22, respectively). As illustrated in Figure 5, each phase sorts the rows and the columns of the matrix according to some

- Input constraints:

$$C_1 \begin{array}{l} \text{3x+y} \leq 0 \\ \wedge\ \text{y} \neq 0 \\ \wedge\ \text{y - 1} \leq 0 \\ \wedge\ \text{x+2y} \leq 0 \end{array} \quad C_2 \begin{array}{l} \text{2a+b} \leq 0 \\ \wedge\ \text{a} \neq 0 \\ \wedge\ \text{a+3b} \leq 0 \\ \wedge\ \text{a - 1} \leq 0 \end{array} \quad C_3 \begin{array}{l} \text{2i+j} \leq 0 \\ \wedge\ \text{i + 2j} \leq 0 \\ \wedge\ \text{i} \neq 0 \\ \wedge\ \text{i+3j} \leq 0 \\ \wedge\ \text{i - 1} \leq 0 \end{array}$$

- Matrix at the beginning of *canonicalization*:

$$C_1 \left| \begin{array}{cccc} 3 & 1 & 0 & \leq \\ 0 & 1 & 0 & \neq \\ 0 & 1 & -1 & \leq \\ 1 & 2 & 0 & \leq \end{array} \right| C_2 \left| \begin{array}{cccc} 2 & 1 & 0 & \leq \\ 1 & 0 & 0 & \neq \\ 1 & 3 & 0 & \leq \\ 1 & 0 & -1 & \leq \end{array} \right| C_3 \left| \begin{array}{cccc} 2 & 1 & 0 & \leq \\ 1 & 2 & 0 & \leq \\ 1 & 0 & 0 & \neq \\ 1 & 3 & 0 & \leq \\ 1 & 0 & -1 & \leq \end{array} \right|$$

- Matrix after phase 1 of *canonicalization*:

$$C_1 \left| \begin{array}{cccc} 3 & 1 & \mathbf{0} & \leq \\ 1 & 2 & \mathbf{0} & \leq \\ \mathbf{0} & \mathbf{1} & \underline{\mathbf{-1}} & \leq \\ \mathbf{0} & \mathbf{1} & \underline{\mathbf{0}} & \neq \end{array} \right| C_2 \left| \begin{array}{cccc} 2 & 1 & \mathbf{0} & \leq \\ 1 & 3 & \mathbf{0} & \leq \\ \mathbf{1} & \mathbf{0} & \underline{\mathbf{-1}} & \leq \\ \mathbf{1} & \mathbf{0} & \underline{\mathbf{0}} & \neq \end{array} \right| C_3 \left| \begin{array}{cccc} 2 & 1 & \mathbf{0} & \leq \\ 1 & 2 & \mathbf{0} & \leq \\ 1 & 3 & \mathbf{0} & \leq \\ \mathbf{1} & \mathbf{0} & \underline{\mathbf{-1}} & \leq \\ \mathbf{1} & \mathbf{0} & \underline{\mathbf{0}} & \neq \end{array} \right|$$

- Matrix after phase 2 of *canonicalization*:

$$C_1 \left| \begin{array}{cccc} \underline{\mathbf{3}} & \underline{\mathbf{1}} & \underline{\mathbf{0}} & \leq \\ \underline{\mathbf{1}} & \underline{\mathbf{2}} & \underline{\mathbf{0}} & \leq \\ \underline{\mathbf{0}} & \underline{\mathbf{1}} & \underline{\mathbf{-1}} & \leq \\ \underline{\mathbf{0}} & \underline{\mathbf{1}} & \underline{\mathbf{0}} & \neq \end{array} \right| C_2 \left| \begin{array}{cccc} \underline{\mathbf{3}} & \underline{\mathbf{1}} & \underline{\mathbf{0}} & \leq \\ \underline{\mathbf{1}} & \underline{\mathbf{2}} & \underline{\mathbf{0}} & \leq \\ \underline{\mathbf{0}} & \underline{\mathbf{1}} & \underline{\mathbf{-1}} & \leq \\ \underline{\mathbf{0}} & \underline{\mathbf{1}} & \underline{\mathbf{0}} & \neq \end{array} \right| C_3 \left| \begin{array}{cccc} \underline{\mathbf{3}} & \underline{\mathbf{1}} & \underline{\mathbf{0}} & \leq \\ \mathbf{1} & \mathbf{2} & \underline{\mathbf{0}} & \leq \\ \mathbf{2} & \mathbf{1} & \underline{\mathbf{0}} & \leq \\ \underline{\mathbf{0}} & \underline{\mathbf{1}} & \underline{\mathbf{-1}} & \leq \\ \underline{\mathbf{0}} & \underline{\mathbf{1}} & \underline{\mathbf{0}} & \neq \end{array} \right|$$

- Matrix after phase 3 of *canonicalization*:

$$C_3 \left| \begin{array}{cccc} \underline{\mathbf{3}} & \underline{\mathbf{1}} & \underline{\mathbf{0}} & \leq \\ \underline{\mathbf{2}} & \underline{\mathbf{1}} & \underline{\mathbf{0}} & \leq \\ \underline{\mathbf{1}} & \underline{\mathbf{2}} & \underline{\mathbf{0}} & \leq \\ \underline{\mathbf{0}} & \underline{\mathbf{1}} & \underline{\mathbf{-1}} & \leq \\ \underline{\mathbf{0}} & \underline{\mathbf{1}} & \underline{\mathbf{0}} & \neq \end{array} \right|$$

Figure 5: Examples of *canonicalization*

order relation. Each phase preserves the order established by the previous phases and strengthens the order between the rows and the columns that were not assigned a relative order in the previous phases. The algorithm converges as soon as all rows and columns are relatively ordered (lines 6, 11, 21 or 27). We first present the four phases, and then prove that the algorithm converges to a unique form for all constraints in the same equivalence class.

In the first phase (lines 3—4), the algorithm sorts the rows of the matrix in decreasing order considering only their comparison operators (line 3) and constant terms (line 4). In the examples we provide we assume that the comparison $\leq$ is always greater than $\neq$. In Figure 5, we represent in bold the items that converge to a stable row or column position after each phase, and underline the items that converge to a stable position on both dimensions. For example, after phase 1, the order of the last two rows of the matrix C_1 is fixed and will not change until the end of the algorithm, while the order of the first two rows may change.

In the second phase (lines 8—9), the algorithm sorts the rows and the columns of the matrix considering only the vectors of coefficients of each row and column, thus leaving untouched the last two columns. The order is determined by comparing the coefficients of the two vectors sorted in decreasing order. For example, comparing the first and second rows of matrix C_2 after the first phase, we determine that $1, 3$ is greater than $2, 1$ and then the algorithm swaps the first two rows. The readers should notice that the comparison is limited to the terms sub-matrix. It then compares the first and second column of the so transformed matrix, obtaining that $3, 1, 0, 0$ is greater than $1, 2, 1, 1$ and thus swaps the first two columns obtaining the matrix in Figure 5 after the second phase.

$$C_1 \left| \begin{array}{ccccc} 0 & 0 & 1 & 0 & \leq \\ 0 & 1 & 0 & 0 & \leq \\ 1 & 0 & 0 & 0 & \leq \end{array} \right| C_2 \left| \begin{array}{ccccc} 0 & 0 & 1 & 0 & \leq \\ 1 & 0 & 0 & 0 & \leq \\ 0 & 1 & 0 & 0 & \leq \end{array} \right| C_3 \left| \begin{array}{ccccc} 1 & 0 & 0 & 0 & \leq \\ 0 & 1 & 0 & 0 & \leq \\ 0 & 0 & 1 & 0 & \leq \end{array} \right|$$

Figure 6: Example of matrices corresponding to equivalent constraints but different after the third step of *canonicalization*

This phase does not enforce any order for rows and columns that contain identical sets of coefficients, though possibly in different positions in the respective vectors. In Figure 5 only the second and third rows of C_3 contain the same elements, though in different positions. In the example, the first two steps of the algorithm produce the canonical forms for C_1 and C_2 thus proving the equivalence of the two constraints.

In the third phase (lines 13—19) the algorithm orders the rows and columns not yet considered according to the lexicographic order of the items which are stable with respect to their column or row, respectively (lines 15, 16 and 17, 18). In Figure 5, this step sorts the second and third rows of matrix C_3. After this step, the algorithm has produced the canonical form for C_3.

As shown in Figure 6, matrices with few values repeated many times may still be different after the third step of the algorithm, despite corresponding to equivalent constraints.

The fourth phase of the algorithm applies to these rare cases. In the fourth phase (lines 23—26), the algorithm sorts the rows and the columns that have not been ordered yet by exhaustively enumerating all possible permutations of rows and columns. It extracts the sub-matrix formed by the still unordered rows and columns (line 23), enumerates the matrices that correspond to permutations of rows and columns that do not violate the relative order established in the previous phases (line 24), identifies the maximum permutation according to the lexicographic order of all matrices flattened as sequences of numbers (line 25), and applies this permutation to the original matrix (line 26). Since the possible permutations are always a finite set, this phase always terminates. In the example of Figure 6 the fourth phase of the algorithm transforms matrices C_1 and C_2 into C_3, which is their canonical form.

While the first three phases are computationally inexpensive, the fourth phase of *canonicalization* is not, and can thus affect the overall performance of the algorithm. The fourth phase applies only to matrices containing sub-matrices with the same comparison operator for all rows, equal constant terms, equal sets of variable coefficients, and exactly equal sequences of the coefficients of those variables for which the first three phases succeeded to establish a stable relative order. Intuitively, this is a rare case, as confirmed by the experiments discussed in Section 4.

The following theorem guarantees that *canonicalization* converges to the canonical form stated in Definition 3.1.

THEOREM 3.2. *Let $LC_\wedge$ be the set of all conjunctive linear constraints, and $\equiv$ the equivalence relation defined in Definition 3.1, then:*

$\forall C_1, C_2 \in LC_\wedge, C_1 \equiv C_2 \iff$ canonicalization$(C_1) =$ canonicalization(C_2).

The formal proof of the theorem is not trivial. Here we discuss the validity of the theorem informally. [1]

[1]The interested readers can find the complete proof at `http://star.inf.usi.ch/recal/proof`.

The right-to-left implication is trivially true because *canonicalization* computes a permutation of the clauses and the variables of the input constraint. Thus, if two constraints can be made equal permuting their clauses and variables, they satisfy the definition of the equivalence relation $\equiv$.

The left-to-right implication requires that *canonicalization* converges to the same matrix when executed on equivalent constraints. We develop the proof as follows.

First, since C_1 and C_2 are equivalent, there exists a bijective correspondence between the equivalent clauses and variables of the two constraints. Let $\sim$correspondence be this bijection.

Next, we prove that the four phases of *canonicalization* yield the same decisions on the relative order between any $\sim$corresponding pairs of clauses and variables of C_1 and C_2. In the first phase, this is true because each clause of C_1 has the same comparison operator and the same constant term as its $\sim$corresponding clause of C_2.

In the second phase, it is true because $\sim$corresponding clauses and variables share the same sets of coefficients, and they result in the same relative order or unknown order in both C_1 and C_2.

All iterations of the third phase take order decisions based on the clauses and the variables with stable positions after the first two phases and the previous iterations. Since the first two phases guarantee the same relative order of any $\sim$corresponding clauses and variables in both C_1 and C_2, then the stable positions of C_1 $\sim$correspond to the stable positions of C_2 at the beginning of this phase, and thus the first iteration results in the same order decisions for any $\sim$corresponding clauses and variables of the constraints. By induction, this can be proved true for any iteration.

Finally, the fourth phase considers the clauses and the variables with yet unstable positions that, because of the properties of the previous phases, are in $\sim$correspondence between C_1 and C_2. Thus, the enumeration of all possible permutations produces exactly the same set of permuted matrices for both constraints leading to the same final order. $\square$

In general, the problem of determining the equivalence of constraints is as hard as the *Graph Isomorphism* problem, for which no polynomial-time algorithm is known yet [19]. The complexity of *canonicalization* is polynomial up to the third phase, which is usually enough to determine the equivalence between constraints in many practical cases. In the experiments reported in Section 4 the algorithm *canonicalization* converges within step three on 93% of the formulas that we analyzed.

4. EMPIRICAL STUDY

We experimentally evaluated the *Recal* approach with a prototype implementation that we used to estimate the benefits of reusing constraint proofs in program analysis, and in particular the effectiveness of our approach for this purpose. The benefits of reusing constraint proofs depend on both the amount of constraint proofs that can be successfully reused, and the performance gain of reusing rather then recomputing constraint proofs multiple times.

In this section, we discuss a set of experiments that quantify the effectiveness of our approach by using a benchmark of (hundreds of thousands of) constraints produced when analysing 21 subject programs with the Crest [10] and the JBSE [7] symbolic executors. We measure the amount of constraint proofs that can be reused both through the constraints that derive from the analysis of a given program and across the sets of constraints related to different programs. We characterize these two measurements as intra- and inter-program constraint proof reuse, respectively. We compare the results of *Recal* with the ones obtained with the Green approach of Visser et al. [25], to quantify the improvement of *Recal* on the state of the art.

We use a second benchmark of constraints that we synthesized randomly, to further unveil the potential of the canonicalization algorithm described in Section 3 that represents a major novel contribution of *Recal*. We measure the performance of reusing constraints with *Recal* with respect to solving constraints with two popular SMT solvers, namely Z3 and MathSat [14, 12]. We extend the performance measurements to a benchmark of constraints maintained in the SMT community [2].

Below, we present the relevant details of our prototype, outline the research questions that drive our experiments, and describe the design and the results of our experiments in detail.

4.1 Prototype

We experimentally validated *Recal* by means of $Recal_{py}$, a prototype implementation written in Python.

$Recal_{py}$ implements only the first three phases of the *canonicalization* algorithm described in Section 3, thus it runs in polynomial time at the cost of some inaccuracy in identifying as equivalent the constraints that need phase 4 to converge to canonical form. The *clauses-to-formulas inverted index* maps hashes of clauses (computed using the SHA-1 hashing algorithm [20]) to sets of formulas, to efficiently merge the selected sets.

4.2 Research Questions and Hypotheses

The empirical evaluation of our approach is driven by the following research questions:

RQ1: Can Recal effectively identify reusable constraint proofs?

We measure the effectiveness of *Recal* as the portion of constraints that can be solved by reusing the proof of some other previously solved constraints.

We measure both intra- and inter-program constraint proof reuse, referring to the constraints produced during the analysis of the same and different programs, respectively. In detail, inter-program reuse measures the effectiveness of *Recal* with respect to the constraints for which we cannot reuse any intra-program proof, but for which there exists a reusable proof produced during the analysis of a different program.

We measure the effectiveness of *Recal* both in absolute terms and with respect to Green, to quantify the improvement over the state of the art.

We also measure the contribution of the *canonicalization* algorithm by comparing the effectiveness of *Recal* with and without it, being *canonicalization* a key contribution of *Recal*.

We refine the original research question in the following experimental hypotheses:

H_1 *Recal* achieves high intra-program constraint reuse.

H_2 *Recal* achieves higher intra-program constraint reuse than Green.

H_3 *Recal* achieves high inter-program constraint reuse.

H_4 *Recal* achieves higher inter-program constraint reuse than Green.

H_5 *Recal* with *canonicalization* achieves higher constraint reuse than *Recal* without *canonicalization*.

RQ2: Can Recal identify reusable constraint proofs more efficiently than solving the constraint with a SMT solver?

We compare the time that *Recal* spends to identify the reusable proofs, with the time that state of the art SMT solvers spend to solve the same constraints. In the experiments we used Z3 and MathSat as sample solvers [14, 12].

We derive the additional experimental hypothesis:

H_6 The time spent to find a constraint proof is less than the time spent to prove the constraint.

4.3 Design of the Experiments

We evaluate the above experimental hypotheses through three different datasets of constraints: the *program*, *random* and *SMT datasets*.

The *program dataset* contains constraints generated by analyzing a set of 21 third party subject programs with symbolic execution. We analyzed both C and Java programs using the symbolic executors Crest [10] and JBSE [7], respectively. We stored both the constraints that were sent to the constraint solver during the analysis, and the corresponding proofs. Figure 7 lists the subject programs (column *Program*), their size in lines of code (column *LOC*), their source programming language (column *Language*) and the number of constraints that we collected and stored in our dataset during the analysis of each program (column *Constraints*). The programs are taken from different repositories and are used as case studies in many papers.[2]

Program	LOC	Language	#Constraints
old-tax	78	Java	27
new-tax	78	Java	35
afs	75	Java	48
dijkstra	142	Java	85
doubly-linked-list	806	Java	114
swapwords	30	Java	173
kbfiltr	599	C	188
wbs	297	Java	191
ball	15	Java	202
reverseword	32	Java	303
block	79	Java	336
division	87	Java	1,257
multiplication	50	Java	1,263
collision	21	Java	1,741
avl	519	Java	2,985
tcas	200	Java	9,780
treemap	806	Java	13,556
cdaudio	2171	C	55,329
floppy	1137	C	100,006
grep	10068	C	100,126
diskperf	1104	C	103,505

Figure 7: Subject programs

The *random dataset* consists of constraints automatically generated with random coefficients and random comparison operators according to a uniform probability distribution.

The *SMT dataset* includes a subset of constraints that belong to the collections of linear integer constraints proposed as a benchmark in the constraint solving research community.[3] The dataset includes the 289 constraints from the original benchmark that (i) are in conjunctive form and (ii) take half a second or more to be solved with Z3, and thus challenge the performance of Z3.

In our experiments, we measure the intra- and inter-program reuse of *Recal* and Green on each subject program referring to the program dataset as follows. Given a reuse approach (either *Recal* or Green) and a program P, we measure the intra-program reuse starting with an empty repository of constraints, scanning the list of constraints of P in the dataset, and using the approach to search for a reusable proof for each of these constraints incrementally. Whenever we succeed to find a reusable proof for a given constraint, we increment the hit counter, and whenever the search fails, we add the constraint to the repository. After scanning all the constraints of P, the intra-program reuse of the approach on P is the hit rate, computed as the ratio between the hit counter and the total number of constraints of P, that is the percentage of hits.

Similarly, we measure the inter-program reuse of an approach on P by scanning all the constraints of P referring to a repository populated with all the constraints in the program dataset excluding the ones of P. In the experiments reported in this section we computed the inter-program reuse by considering only the constraints of P that are not intra-program redundant.

We use the random dataset to investigate the validity of hypothesis H_5. We populate the constraint repository by randomly generating 4 samples of up to 100,000 constraints each, of size 1x1 (1 clause and 1 variable), 2x2, 3x3 and 4x4. We prune the duplicates, and then use *Recal* to search for another randomly generated set of up to 100,000 constraints for each size. We measure the constraint reuse as the ratio of the hit counter, i.e., the number of constraints found in the repository, and the amount of searched constraints. We repeat the experiment by using *Recal* with and without *canonicalization* to compare the measurements.

We use both the program and the SMT datasets to investigate the validity of hypothesis H_6. We populate the constraint repository with all the constraints of the datasets, search for all constraints with *Recal*, and measure the time to find each proof. We then compare the time to find each constraint with the time required to solve the constraint with Z3 and MathSat.

All the experiments were executed on a Macbook Pro laptop equipped with 2.3 GHz Intel Core i7 processor and 16 GB of RAM.

4.4 Results

In this section, we discuss the results of the experiments presented in the former subsections about intra- and inter-program reuse (H_1 – H_4), *canonicalization* (H_5) and performance aspects (H_6).

4.4.1 Research Hypothesis H_1 – H_2: Intra-program Reuse

Figures 8 and 9 report the results of the intra-program reuse experiments. The table in Figure 8 indicates the intra-program reuse obtained when searching for reusable proofs with Green (column *Green*), *Recal* without the stricter candidate search feature (column *Recal* $^-$) and with the stricter candidate search feature (columns *Recal* $^+$) for the programs

[2]The interested readers can find the references to the subject programs at `http://star.inf.usi.ch/recal/refs`.

[3]`http://www.cs.nyu.edu/~barrett/smtlib/`

Program	#Formulas	Reuse rate (%) with Green	Recal $^-$	Recal $^+$
old-tax	27	0	11	37
new-tax	35	0	11	54
afs	48	85	90	90
dijkstra	85	1	47	47
doubly-linked-list	114	2	33	54
swapwords	173	0	50	99
kbfiltr	188	89	90	95
wbs	191	94	95	96
ball	202	79	82	87
reverseword	303	97	99	99
block	336	3	2	7
division	1,257	0	14	99
multiplication	1,263	0	99	99
collision	1,741	88	97	97
avl	2,985	54	74	88
tcas	9,780	92	99	99
treemap	13,556	95	98	99
cdaudio	55,329	99	>99	>99
floppy	100,006	>99	>99	>99
grep	100,126	>99	>99	>99
diskperf	103,505	>99	>99	>99
TOTAL	391,250			

Figure 8: Intra-program reuse with Green and *Recal* with (+) and without (−) stricter candidate search

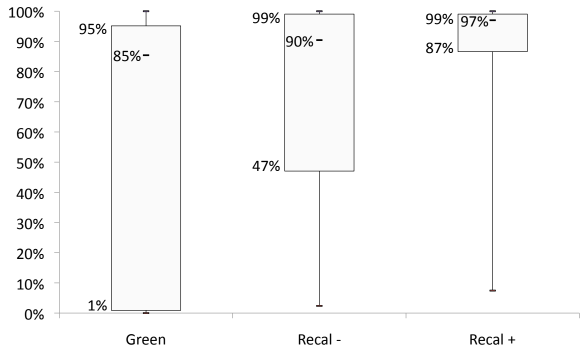

Figure 9: Intra-program reuse rates distribution

in the program dataset. The intra-program reuse is the percentage of constraints with reusable proofs, as defined in the former subsection. The data reported in the figure indicate that all approaches obtain high intra-program reuse for some programs in the dataset albeit with different reuse rates.

Figure 9 plots the distribution of the reuse rates through the programs for the three approaches to compare the reuse rates of the approaches. The plots indicate that *Recal* obtains reuse rates both consistently high, especially when executed with the stricter candidate search feature, and higher than Green. The median of the distribution, that is, the lowest reuse rate measured across the top half of the programs taken in decreasing order of reuse rate, is high for all approaches: 85%, 90% and 97% with Green, *Recal* $^-$ and *Recal* $^+$, respectively. However, the reuse rate across three-fourths of the programs (third quartile) grows as 1%, 47% and 87% with Green, *Recal* $^-$ and *Recal* $^+$, respectively. The data support the validity of the hypotheses H_1 and H_2.

4.4.2 Research Hypothesis H_3 – H_4: Inter-program Reuse

Figures 10 and 11 report the results of the inter-program reuse experiments. The table in Figure 10 indicates the amount of constraints whose proofs are not found with intra-program search (columns *Not-found intra-prog Green* and *Not-found intra-prog Recal*) and the amount of these constraints whose proofs are found with inter-program search (columns *Found inter-prog Green* and *Found inter-prog Recal*). It is evident from the totals reported in the last row that the total amount of proofs that are not found with the intra-program search of *Recal* is much less (about 20%) than the proofs not found with Green.

Despite searching within a much smaller set of constraints left after the intra-program search, *Recal* finds more reusable proofs than Green even with the inter-program search. Figure 11 visualizes the difference between the inter-program searches performed with *Recal* and Green, plotting the distribution of the reuse rates computed as ratios between the number of *Found inter-prog* and the number of *Not-found intra-prog* constraints.

The data suggest the validity of the hypothesis H_3 showing significant yet not astonishing inter-program reusability. The inter-program reuse rate is more than 35% for half of the programs (the median in Figure 11). However, the statistical significance of the results is hindered by the statistical power of the dataset that suffers from the limited set of constraints that are not found after the intra-program search. In fact, more than 99% of the constraints in the dataset are already hit during the intra-program experiment.

The data support the hypothesis that *Recal* improves on the state of the art (H_4), since the first and third quartiles of the distribution of the reuse rates range from 14 to 59% with *Recal*, and only from 0 to 14% with Green.

	Formulas (#)			
	Not-found intra-prog		Found inter-prog	
Program	Green	Recal	Green	Recal
old-tax	27	17	0	10
new-tax	35	16	0	5
afs	7	5	1	2
dijkstra	45	45	4	5
doubly-linked-list	112	52	14	18
swapwords	173	2	0	0
kbfiltr	20	9	8	8
wbs	10	7	7	4
ball	42	27	2	3
reverseword	3	3	0	1
block	328	311	3	4
division	1,257	12	0	11
multiplication	1,263	8	0	8
collision	138	52	4	10
avl	1,321	349	42	48
tcas	176	91	26	35
treemap	536	151	64	55
cdaudio	296	82	17	18
floppy	11	11	4	11
grep	98	17	0	1
diskperf	26	13	14	10
Total	5,924	1,279	210	266

Figure 10: Inter-program reuse: Green and *Recal*

4.4.3 Research Hypothesis H_5: Canonicalization

The *canonicalization* improves the ability of recognizing equivalent formulas, permitting to ignore the order of their clauses and variables. In our experiments, we observed that this kind of formulas occurs rarely during intra-program search. This phenomenon is not surprising, since we derived the formulas used in the experiments with symbolic execution that covered relatively small portions of the program paths. We expect that equivalent formulas with different order of clauses and variables may occur much more often during inter-

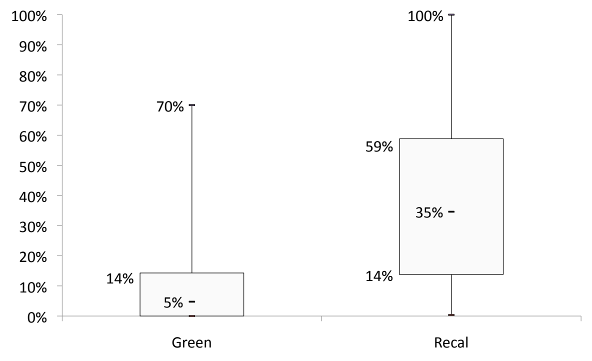

Figure 11: Inter-program reuse rates distribution

program search, because of the higher diversity of constraints in different programs. As shown in Figure 10 the set of formulas found during inter-program search is quite small, and thus not sufficient for designing good experiments.

To thoroughly measure the impact of *canonicalization*, we experimented with the random dataset introduced in Section 4.3 that better represents the diversity of formulas that can be found during inter-program search.

Size	#Space	#C	Can-	Can+
1x1	4	4	100%	100%
2x2	144	100	100%	100%
3x3	21,952	13,960	96%	100%
4x4	12,960,000	72,231	4%	89%

Legenda:

Size	constraint size, that is clauses (#) x variables (#)
#Space	number of possible constraints of the given size
#C	number of constraints stored in the caching phase
Can-	found constraints (%) when using syntactic search
Can+	found constraints (%) when using canonicalization

Figure 12: Reuse rates of *canonicalization*

Figure 12 reports the reuse rates measured with and without *canonicalization* (columns *Can+* and *Can-*). Column *#Space* indicates the number of constraints of a given size that can be generated, and column *#C* indicates the number of constraints in the search repository. As discussed in Subsection 4.3 we populate the search repository by randomly selecting 100.000 constraints out of the possible ones, and pruning the duplicates. Columns *Can+* and *Can-* indicate the reuse rates when searching for a new set of 100,000 random constraints in the repository with and without *canonicalization*, respectively.

When dealing with constraints of small size (1x1, 2x2), the reference population of constraints is small, and the search repository includes all possible constraints, thus trivially resulting in 100% reuse rates. The probability of randomly selecting the same constraint twice decreases with the size of the constraints. Thus, when experimenting with large populations of constraints, the probability of a search hit due to the presence of the searched constraint in the repository quickly drops. For 3x3 constraints the effect is marginal, but for 4x4 constraints the impact is dramatic. The data reported in the table show that *Recal* without *canonicalization* suffers enormously from this effect, while with *canonicalization* the hit rate is impressive.

These results support the validity of the hypothesis H_5 according to which *canonicalization* is very effective in improving the search results of *Recal*, and can thus improve the scalability of the approach.

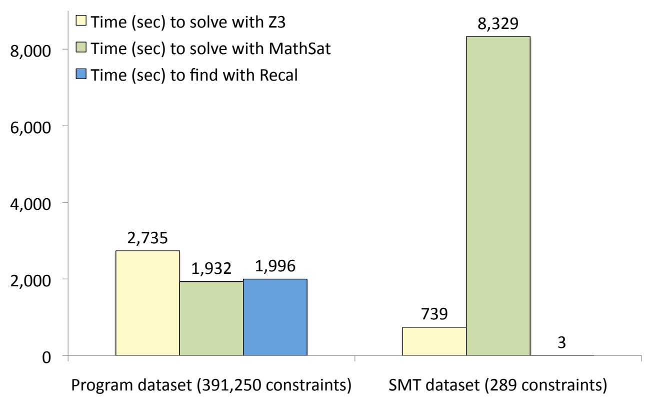

Figure 13: Performance: Z3, MathSat, *Recal*

4.4.4 Research Hypothesis H_6: Performance

The long term goal of *Recal* is to improve the scalability of symbolic analysis by reducing the impact of constraint solving. Figure 13 compares the performance of finding reusable constraint proofs with *Recal* with the performance of solving the same constraints with Z3 and MathSat, two of the most popular constraint solvers. *Recal*, Z3 and MathSat present comparable performances when dealing with the constraints in the program dataset, while the performance figures change dramatically when dealing with the SMT dataset. This confirms the hypothesis H_6: *Recal* may present performance similar to the ones of state of the art solvers with respect to many simple constraints, in particular the ones in the program dataset, but can dramatically improve for complex constraints, like the ones in the SMT dataset. In this latter case, we see an amazing drop in the execution time that falls from more than 2 hours for MathSat and more than 10 minutes for Z3 to 3 seconds for *Recal*.

These results confirm the well known fact that constraint solvers can be very efficient for many constraints, but extremely slow in some cases. As observed by Palikareva and Cadar solving the few expensive constraints takes most of the overall analysis time [21]. The consistently good performance figures indicate that *Recal* can reduce the impact of solving the hard-to-prove constraints, and thus largely improve the scalability of symbolic program analysis.

4.5 Threats to Validity

This section acknowledges the threats that may limit the validity of our experimental results, and briefly discusses the countermeasures that we adopted to mitigate such threats.

The validity of our experiments depends on the correctness of $Recal_{py}$. To mitigate this threat, we carefully tested the $Recal_{py}$ prototype and manually confirmed a sample of the reusable proofs identified in the experiments.

The specificity of the considered subject programs and program analysis tools can impact the construct validity of the experiments. We have selected the subject programs from popular online software repositories, including repositories used for other scientific testing and analysis experiments. In this paper, we have considered constraints produced with symbolic execution that represents a widely studied analysis technique that integrates with SMT solvers. The results may depend on some characteristics of these kinds of constraints and may not generalize to other kinds of constraints. Our current research agenda plans to extend the results to other analysis techniques, including software model checking and pre-post condition checking.

The performance of *Recal* may depend on the size of the constraint repository. *Recal* uses search indexes to speed up the search of equivalent and stricter constraints. In our experiments we refer to relatively small repositories, and the results may not apply to very large repositories, like the ones that we envisage if this approach will be used by many user communities. Indeed, answering this question represents the main goal of our research agenda.

With reference to the external validity, the result of a single experiment cannot be directly generalized. Nonetheless, the consistency of the reuse data measured across the subject programs is a promising indication that our results may generalize. Our future plans include extending the experiments to other types of program analysis and beyond the experimental samples of this paper.

5. RELATED WORK

In this section we survey previous work on mitigating the performance issues of constraint solving in program analysis.

A first class of approaches aims at optimizing the solving effectiveness, either by improving the effectiveness of the constraint solvers, or by complementing the solvers with external optimizations. Several research efforts address the engineering of increasingly powerful constraint solvers that also cope with increasing sets of theories [14, 17, 1, 27, 6, 24, 12]. In Section 4, we compared our approach with Z3 and MathSat, two of the most popular constraint solvers [14, 12].

Despite the dramatic improvements attained in the last decade, constraint solvers remain a major bottleneck in symbolic analysis, and thus researchers investigate external optimizations that can improve the efficiency and the effectiveness of constraint solving in this context. Braione et al. and Erete and Orso propose approaches that optimize the constraints before calling the constraint solvers by exploiting domain-specific and contextual information [16, 9]. Dinges et al. combine linear constraint solving and heuristic search to improve the ability of dynamic symbolic execution to solve non-linear path-conditions [15]. Palikareva and Cadar integrate multiple solvers to exploit the different strengths of distinct solvers on different constraints [21].

Another research direction is to reduce the number of queries to constraint solvers by caching and reusing results from previous analysis sessions. The existing approaches can be classified either as (i) approaches that share analysis results across the analysis of incremental versions of the same program to avoid repeating the same analysis multiple times [22, 26, 5], or (ii) approaches that cache and reuse available proofs for constraints that recur multiple times during the analysis of the same or different programs [11, 4, 25, 18]. We discuss in further detail this latter class of approaches that are closely related to our technique.

Cadar et al. report the implementation of a component to cache and reuse constraint proofs within the symbolic executor Klee [11]. Visser et al. engineer the software library Green to provide third-party program analyzers with functionality to reuse constraint proofs stored in an in-memory database [25]. The *Recal* approach shares with Klee and Green the use of constraint slicing to identify sub-constraints the satisfiability of which can be proved independently. *Recal* borrows from Green the normalization step, though the redundant clause elimination step is an original contribution of *Recal*. *Recal* shares with Klee the ability of recognizing constraints that include subsets of the clauses of other constraints, though *Recal* and Klee propose different technical solutions, i.e., the inverted index described in Section 2.4 and UBTree based data structures, respectively. Jia et al. address the same problem with a prefix tree data structure [18].

The definition of *canonicalization* to recognize equivalent constraints independently from the order of appearance of the clauses and the variables in the constraints is an original contribution of *Recal*, as well as the combination of all above techniques in a comprehensive approach. Our experiments on constraint proof reuse refer to a much larger corpus of programs than the ones reported in the previous work [11, 25]. Our results confirm that constraint proof reuse is a viable solution to improve the performance of program analysis, and that *Recal* significantly overcomes Green on the amounts of identified constraint proofs.

6. CONCLUSION

In this paper we presented a novel approach to reduce the impact of constraint solving on the scalability of symbolic analysis. As observed by Palikareva and Cadar, modern constraint solvers are very efficient for many constraints, but may be extremely inefficient for few others, whose solving time takes most of the overall analysis time [21].

Inspired by the preliminary work of Visser et al. [25], we presented an approach that searches within a repository of constraint proofs, thus reducing the need of solving constraints during symbolic analysis. Our work extends the preliminary idea of Visser et al. by both proposing stronger simplifications of the formulas and introducing a canonical form of the constraints. As a result, our approach can reduce equivalent albeit syntactically different formulas to the same form, and thus boost the chances of recognizing equivalent constraints and reuse their proofs. The canonical form allows us to define an efficient search index, and a novel search strategy to identify both equivalent and stricter formulas, that is, formulas whose satisfiability result logically implies the proof of the searched one.

The experiments reported in the paper provide empirical data that validate the new idea, and show that our approach improves over the preliminary work of Visser et al., indicating that *Recal* is a valid candidate to complement state of the art constraint solvers.

In this paper we present and validate the approach referring to linear constraints, which can be solved with strong theories and efficient solvers. We are currently extending our technique to deal with non-linear constraints that can benefit even more from this type of approach, given the substantial lack of practical theories and efficient tools to compute proofs of non-linear problems.

7. ACKNOWLEDGMENTS

This work is partially supported by the Swiss SNF project CloSE (200021_149997/1) and by the Italian PRIN project IDEAS (2012E47TM2_006).

8. REFERENCES

[1] A. Armando, M. P. Bonacina, S. Ranise, and S. Schulz. New results on rewrite-based satisfiability procedures. *ACM Transactions on Computational Logic*, 2009.

[2] C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB), 2010.

[3] L. A. Barroso, J. Dean, and U. Holzle. Web search for a planet: The Google cluster architecture. *Micro, Ieee*, 2003.

[4] J. Belt, Robby, and X. Deng. Sireum/Topi LDP: A lightweight semi-decision procedure for optimizing symbolic execution-based analyses. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the Symposium on The Foundations of Software Engineering*, 2009.

[5] D. Beyer, S. Löwe, E. Novikov, A. Stahlbauer, and P. Wendler. Precision reuse for efficient regression verification. In *Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, 2013.

[6] C. Borralleras, S. Lucas, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. SAT modulo linear arithmetic for solving polynomial constraints. *Journal of Automated Reasoning*, 2012.

[7] P. Braione, G. Denaro, B. Křena, and M. Pezzè. Verifying LTL properties of bytecode with symbolic execution. In *Proceedings of the 3rd Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, 2008.

[8] P. Braione, G. Denaro, and M. Pezzè. Enhancing symbolic execution with built-in term rewriting and constrained lazy initialization. In *Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, 2013.

[9] P. Braione, G. Denaro, and M. Pezzè. Enhancing symbolic execution with built-in term rewriting and constrained lazy initialization. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, 2013.

[10] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 23rd International Conference on Automated Software Engineering*, 2008.

[11] 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 Conference on Operating Systems Design and Implementation*, 2008.

[12] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT5 SMT Solver. In N. Piterman and S. Smolka, editors, *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2013.

[13] D. Davidson, B. Moench, T. Ristenpart, and S. Jha. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *Proceedings of the 22nd Conference on Security*, 2013.

[14] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[15] P. Dinges and G. Agha. Solving complex path conditions through heuristic search on induced polytopes. In *Proceedings of the 22nd Symposium on the Foundations of Software Engineering*, 2014.

[16] I. Erete and A. Orso. Optimizing constraint solving to better support symbolic execution. In *Proceedings of the 4th International Conference on Software Testing, Verification and Validation Workshops*, 2011.

[17] S. Jha, R. Limaye, and S. A. Seshia. Beaver: Engineering an efficient smt solver for bit-vector arithmetic. In *Proceedings of the 21st International Conference on Computer Aided Verification*, 2009.

[18] X. Jia, C. Ghezzi, and S. Ying. Enhancing reuse of constraint solutions to improve symbolic execution. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2015.

[19] J. Köbler, U. Schöning, and J. Torán. *The Graph Isomorphism Problem: Its Structural Complexity.* Birkhauser Verlag, 1993.

[20] NIST. Fips publication 180-1: Secure hash standard. Technical report, National Institute of Standards and Technology, 1995.

[21] H. Palikareva and C. Cadar. Multi-solver support in symbolic execution. In *Proceedings of the 25th International Conference on Computer Aided Verification*, 2013.

[22] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32nd Conference on Programming Language Design and Implementation*, 2011.

[23] N. Tillmann and J. de Halleux. Pex — white box test generation for .net. In *Proceedings of the 2nd International Conference on Tests and Proofs*, 2008.

[24] T. Van Khanh and M. Ogawa. Smt for polynomial constraints on real numbers. *Electronic Notes Theoretical Computer Science*, 2012.

[25] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: Reducing, reusing and recycling constraints in program analysis. In *Proceedings of the 20th Symposium on the Foundations of Software Engineering*, 2012.

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

[27] H. Zankl and A. Middeldorp. Satisfiability of non-linear (ir)rational arithmetic. In *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. Springer, 2010.