Modular Demand-Driven Analysis of Semantic Difference for Program Versions

Anna Trostanetski[1(✉)], Orna Grumberg[1(✉)], and Daniel Kroening[2(✉)]

[1] Technion – Israel Institute of Technology, Haifa, Israel
{annat,orna}@cs.technion.ac.il
[2] University of Oxford, Oxford, UK
kroening@cs.ox.ac.uk

Abstract. In this work we present a *modular* and *demand-driven* analysis of the semantic difference between program versions. Our analysis characterizes initial states for which final states in the program versions differ. It also characterizes states for which the final states are identical. Such characterizations are useful for regression verification, for revealing security vulnerabilities and for identifying changes in the program's functionality.

Syntactic changes in program versions are often small and local and may apply to procedures that are deep in the call graph. Our approach analyses only those parts of the programs that are affected by the changes. Moreover, the analysis is *modular*, processing a single pair of procedures at a time. Called procedures are not inlined. Rather, their previously computed summaries and *difference summaries* are used. For efficiency, procedure summaries and difference summaries can be *abstracted* and may be *refined* on demand.

We have compared our method to well established tools and observed speedups of one order of magnitude and more. Furthermore, in many cases our tool proves equivalence or finds differences while the others fail to do so.

1 Introduction

In this work we present a modular and demand-driven algorithm for computing the semantic difference between two closely-related, syntactically similar imperative programs. The need to identify semantic difference often arises when a new (patched) program version is built on top of an old one. The difference between the versions can be used for:

- Regression testing, which checks whether the new version introduces security bugs or errors. The old version is considered to be a correct, "golden model" for the new, less-tested version [30].
- Revealing security vulnerabilities that were eliminated by the new version [11]. This information can be used to produce attacks against the old version.

Supported by the ERC project 280053 (CPROVER), the H2020 FET OPEN 712689 SC^2 and the Prof. A. Pazy Research Foundation.

F. Ranzato (Ed.): SAS 2017, LNCS 10422, pp. 405–427, 2017.
DOI: 10.1007/978-3-319-66706-5_20

– More generally, identifying and characterizing changes in the program's functionality [24].

Semantic difference has been widely studied, and a wide range of techniques have been suggested [11,14–16,20,23–26]. We aim at enhancing the scalability and precision of existing techniques by exploiting the modular structure of programs and avoiding unnecessary analysis.

We consider two program versions, consisting of (matched) procedure calls, arranged in call graphs. Some of the matched procedures are known to be syntactically different while the others are identical. Often, the changes between versions are small and limited to procedures deep inside the call graph (Fig. 1). In such cases, it would be helpful to know how these changes affect the program as a whole, without analysing the full program. To achieve this, we first compute a *difference summary* between syntactically different procedures p_1, p_2 (*modified* procedures). Next, we analyse the procedures that call them, using the difference summary for p_1, p_2 computed before. No inlining of called procedures is applied. We also avoid analysing procedures that are not affected by the modified procedures. As a result, the required work may be significantly smaller than analysing the full program.

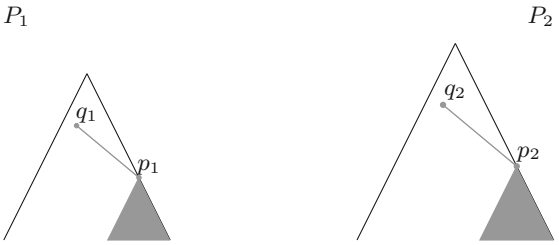

Fig. 1. Call graphs of two program versions P_1, P_2, where their syntactic differences are local to the procedures p_1, p_2, and the bodies of procedures q_1, q_2 are identical

Our work is therefore particularly beneficial when applied to programs that are syntactically similar. While applicable to programs that are very different from each other, our technique would yield less savings in those cases.

Our approach is guided by the following ideas. First, the analysis is *modular*. That is, it is applied to one pair of procedures at a time, thus it is confined to small parts of the program. Called procedures are not inlined. Rather, their previously computed summaries and difference summary are used. We note that any block of code can be treated as a procedure, not only those defined as procedures by the programmer. It is beneficial to choose the smallest possible blocks that were modified between versions, and identify them as "procedures".

Second, the analysis is restricted to only those pairs of procedures whose difference affects the behavior of the full programs.

Third, we provide both under- and over-approximations of the input-output differences between procedures, which can be strengthened on demand.

Finally, procedures need not be fully analysed. Unanalysed parts are *abstracted* and replaced with uninterpreted functions. The abstracted parts are *refined* upon demand if calling procedures need a more precise summary of the called procedures for their own summary.

Our analysis is not guaranteed to terminate. Yet it is an *anytime analysis*. That is, its partial results are meaningful. Furthermore, the longer it runs, the more precise its results are.

In our analysis we do not assume that loops are bounded. We are able to prove equivalence or provide an under- and over-approximation of the difference for unbounded behaviors of the programs. We are also able to handle recursive procedures.

We implemented our method and applied it to computing the semantic difference between program versions. We compared it to well established tools and observed speedups of one order of magnitude and more. Furthermore, in many cases our tool proves equivalence or finds differences while the others failed to do so.

Our Approach in Detail

We now describe our method in more detail. Our analysis starts by choosing a pair of matched procedures p_1 in program P_1 and p_2 in program P_2 that are syntactically different.

The basic block of our analysis is a (partial) *procedure summary* sum_{p_i} with $i \in \{1, 2\}$ for each procedure p_i. The summary is obtained using symbolic execution. It includes *path summarizations* (R_π, T_π) for a subset of the finite paths π of p_i, where R_π is the reachability condition for π to be traversed and T_π is the state transformation mapping initial states to final states when π is executed.

Next, we compute a (partial) *difference summary* $(C(p_1, p_2), U(p_1, p_2))$ for p_1, p_2, where $C(p_1, p_2)$ is a set of initial states for which p_1 and p_2 terminate with different final states. $U(p_1, p_2)$ is a set of initial states for which p_1 and p_2 terminate with identical final states. Both sets are under-approximations. However, the complement of $U(p_1, p_2)$, denoted $\neg U(p_1, p_2)$, also provides an over-approximation of the set of initial states for which the procedures are different.

Note that procedure summaries and difference summaries are both partial. This is because their computation in full is usually infeasible. More importantly, their full summaries are often unnecessary for computing the difference summary between programs P_1, P_2.

If $U(p_1, p_2) \equiv \mathit{true}$ we can conclude that no differences are propagated from p_1, p_2 to their callers. Their callers will not be further analysed then. Otherwise, we can proceed to analysing pairs of procedures q_1, q_2 that include calls to p_1, p_2, respectively. As mentioned before, for building their procedure summaries and difference summary, we use the already computed summaries of p_1, p_2. For the sake of modularity, we develop a new notion of *modular symbolic execution*. We formalize the definitions of symbolic execution and modular symbolic execution, and show the connections between the two.

The analysis terminates when we can fully identify the initial states of P_1, P_2 for which the programs agree/disagree on their final states. Alternatively, we can stop when a predefined threshold is reached. In this case the sets $C(p_1, p_2)$ and $U(p_1, p_2)$ of initial states are guaranteed to represent disagreement and agreement, respectively.

Side results of our analysis are the difference summaries computed for matched procedures in P_1, P_2, that can be reused if the procedures are called by other programs.

The main contributions of this work are:

- We present a modular and demand-driven algorithm for computing semantic difference between closely related programs.
- Our algorithm is unique in that it provides both under- and over-approximations of the differences between program versions.
- We introduce abstraction-refinement into the analysis process so that a trade-off between the amount of computation and the obtained precision will be manageable.
- We develop a new notion of modular symbolic execution.

2 Preliminaries

We start by defining some basic notions of programs and procedures.

Definition 1. *Let P be a program, containing the set of procedures $\Pi = \{p_1, \ldots, p_n\}$. The **call graph** for P is a directed graph with Π as nodes, and there exists an edge from p_i to p_j if and only if procedure p_i calls procedure p_j.*

The procedure p_1 is a special procedure in the program's call graph that acts as an entry point of the program; it is also referred to as the main procedure in the program P, denoted $main_P$.

Next we formalize the notions of variables and states of procedures.

- The ***visible variables*** of a procedure p are the variables that represent the arguments to the procedure and its return values, denoted V_p^v.
- The ***hidden variables*** of a procedure p are the local variables used by the procedure, denoted V_p^h.
- The ***variables*** of a procedure p are both its visible and hidden variables, denoted V_p ($V_p = V_p^v \cup V_p^h$).
- A ***state*** σ_p is a valuation of the procedure's variables, $\sigma_p = \{v \mapsto c | v \in V_p, c \in D_v\}$, where D_v is the (possibly infinite) domain of variable v.
- A ***visible state*** is the projection of a state to the visible variables.

Without loss of generality we assume that programs have no global variables, since those could be passed as arguments and return values along the entire program. We also assume, without loss of generality, that all program inputs are given to the main procedure at the beginning. The programs we analyze are deterministic, meaning that given a visible state of the main procedure at

the beginning of an execution (an ***initial state***), the execution of the program (finite or infinite) is fixed, and for a finite execution the visible state at the end of the execution is fixed (called ***final state***). The same applies to individual procedures as well.

In our work, a program is represented by its *call graph*, and each procedure p is represented by its control flow graph CFG_p (also known as a flow program in [10]), defined below.

Definition 2. *Let p be a procedure with variables V_p. The **Control Flow Graph (CFG)** for p is a directed graph CFG_p, in which the nodes represent instructions in p and the edges represent possible flow of control from one instruction to its successor(s) in the procedure code. Instructions include:*

- *Assignment: $x = e$, where x is a variable in V_p and e is an expression over V_p. An assignment node has one outgoing edge.*
- *Procedure call: $g(Y)$, where $Y \subseteq V_p$ and the values of variables in Y are assigned to the visible variables of procedure g.[1] The variables in Y are assigned with the values of the visible variables of g at the end of the execution of g. A call node has one outgoing edge, to the instruction in p following the return of procedure g.*
- *Test: $B(V_p)$, where $B(V_p)$ is a Boolean expression over V_p; a test node has two outgoing edges, one marked with T, and the other with F.*

A CFG contains one node with no incoming edges, called the entry node*, and one node with no outgoing edges, called the* exit node*.*

Definition 3. *Given CFG_p of procedure p, a **path** $\pi = l_1, l_2, \ldots$ is a sequence of nodes (finite or infinite) in the graph CFG_p, such that:*

1. *For all i there exists an edge from l_i to l_{i+1} in CFG_p.*
2. *l_1 is the entry node of p.*

*The path π is **maximal** if it is either infinite or it is finite and ends in the exit node of p.*

We assume that each procedure performs a transformation on the values of the visible variables, and has no additional side-effects. Procedure p ***terminates*** on a visible state σ_p^v if the path traversed in p from σ_p^v is finite and maximal. A program terminates on a visible state σ_{main}^v if its main procedure terminates.

The following semantic characteristics are associated with finite paths, similarly to the definitions for flow programs in [10]. The characteristics are given (for a path in a procedure p) in terms of quantifier-free First-Order Logic (FOL), defined over the set V_p^v of visible variables.

Definition 4. *Let π be a finite path in procedure p.*

[1] We assume that $Y = \{y_1, \ldots, y_n\}$ and $V_g^v = \{v_1, \ldots, v_n\}$, y_i is assigned to v_i at the entry node, and v_i is assigned to y_i at the exit node.

- *The **Reachability Condition** of π, denoted $R_\pi(V_p^v)$, is a condition on the visible states at the beginning of π, which guarantees that the control will traverse π.*
- *The **State Transformation** of π, denoted $T_\pi(V_p^v)$, describes the final state of π, obtained if control traverses π starting with some valuation σ_p^v of V_p^v.*

$T_\pi(V_p^v)$ is given by $|V_p^v|$ expressions over V_p^v, one for each variable x in V_p^v. The expression for x describes the effect of the path on x in terms of the values of V_p^v at the beginning of π. Let $T_\pi(V_p^v) = (f_1, \ldots, f_{|V_p^v|})$ and $T_{\pi'}(V_p^v) = (f'_1, \ldots, f'_{|V_p^v|})$ be two state transformations. Then, $T_\pi(V_p^v) = T_{\pi'}(V_p^v)$ if and only if, for every $1 \leq i \leq |V_p^v|$, $f_i = f'_i$.

```
1 void p1(int& x) {
2   if (x < 0) {
3     x = -1;
4     return;
5   }
6   if (x >= 2)
7     return;
8   while (x == 2)
9     x = 2;
10  x = 3;
11 }
```

```
1 void p2(int& x) {
2   if (x < 0) {
3     x = -1;
4     return;
5   }
6   if (x > 4)
7     return;
8   while (x == 2)
9     x = 2;
10  x = 3;
11 }
```

Fig. 2. Examples of procedure versions

Example 1. Consider procedure p1 in Fig. 2. Its only visible variable is x, used as both input and output. Consider the paths that correspond to the following line numbers: $\alpha = (2, 3, 4)$ and $\beta = (2, 6, 7)$. Then,

$$R_\alpha(x) = x < 0 \qquad R_\beta(x) = ((\neg(x < 0)) \wedge x \geq 2) \equiv x \geq 2$$
$$T_\alpha(x) = (-1) \qquad T_\beta(x) = (x)$$

A path π is called ***feasible*** if R_π is satisfiable, meaning that there exists an input that traverses the path π. Note that, in p1 from Fig. 2, the path $(2, 6, 8, 9)$ is not feasible.

2.1 Symbolic Execution

Symbolic execution [7,17] (path-based) is an alternative representation of a procedure execution that aims at systematically traversing the entire path space of a given procedure. All visible variables are assigned with symbolic values in place of concrete ones. Then every path is explored individually (in some heuristic order), checking for its feasibility using a constraint solver. During the

execution, a symbolic state T and symbolic path constraint R are maintained. The symbolic state maps procedure variables to symbolic expressions (and is naturally extended to map expressions over procedure variables), and the path constraint is a quantifier-free FOL formula over symbolic values.

Given a finite path $\pi = l_1, \ldots, l_n$, we use symbolic execution to compute the reachability condition $R_\pi(V_p^v)$ and state transformation $T_\pi(V_p^v)$. The computation is performed in stages, where for every $1 \leq i \leq n+1$, $R_\pi^i(V_p)$ and $T_\pi^i(V_p)$ are the path condition and state transformation for path $l_1, \ldots, l_{i-1}$, respectively. Initialization:

- For every $x \in V_p$, $T_\pi^1(V_p)[x] = x$.
- $R_\pi^1(V_p) = true$.

Assume $R_\pi^i(V_p)$ and $T_\pi^i(V_p)$ are already defined. $R_\pi^{i+1}(V_p)$ and $T_\pi^{i+1}(V_p)$ are then defined according to the instruction at node i:

- Assignment $x = e$: $R_\pi^{i+1}(V_p) := R_\pi^i(V_p)$, $T_\pi^{i+1}(V_p)[x] := e[V_p \leftarrow T_\pi^i(V_p)]$ and $\forall y \neq x$, $T_\pi^{i+1}(V_p)[y] := T_\pi^i(V_p)[y]$
- Procedure call $g(Y)$: The procedure g is in-lined with the necessary renaming and symbolic execution continues along a path in g, returning to p when (if) g terminates.[2]
- Test $B(V_p)$: $T_\pi^{i+1}(V_p) := T_\pi^i(V_p)$, and

$$R_\pi^{i+1}(V_p) := \begin{cases} R_\pi^i(V_p) \wedge B[V_p \leftarrow T_\pi^i(V_p)] & \text{if the edge } l_i \rightarrow l_{i+1} \text{ is marked T} \\ R_\pi^i(V_p) \wedge \neg B[V_p \leftarrow T_\pi^i(V_p)] & \text{otherwise} \end{cases}$$

As a result, when we reach the last node l_n of a finite path π we get[3]:

$$R_\pi(V_p^v) = R_\pi^{n+1}(V_p)$$
$$T_\pi(V_p^v) = T_\pi^{n+1}(V_p) \downarrow_{V_p^v}$$

As symbolic execution explores the program one path at a time, we start by summarizing single paths, and then extend to procedures.

Definition 5. *Given a finite maximal path π in p, a* ***Path Summary*** *(also known as a partition-effect pair in [25]) is the pair $(R_\pi(V_p^v), T_\pi(V_p^v))$.*

Definition 6. *A* ***Procedure Summary*** *(also known as a symbolic summary in [25]), for a procedure p, is a set of path summaries*

$$sum_p \subseteq \{(R_\pi(V_p^v), T_\pi(V_p^v)) | \pi \text{ is a finite maximal path in } CFG_p\}.$$

[2] Current values of Y are assigned to the visible variables of g, and assigned back at termination of g.

[3] Since we assume that all inputs are given through visible variables, and therefore no hidden variable is used before it is initialized, V_p^h will not appear in $R_\pi^{n+1}(V_p)$ and $T_\pi^{n+1}(V_p) \downarrow_{V_p^v}$.

Note that for a given CFG the reachability conditions of any pair of different maximal paths are disjoint, meaning that for every initial state at most one finite maximal path is traversed in the CFG. Thus, a procedure summary partitions the set of initial states into disjoint finite paths, and describes the effect of the procedure p on each path separately. This observation will be useful when procedure summaries are used to compute difference summaries between procedures.

Unfortunately, it is not always possible to cover all paths in symbolic execution due to the path explosion problem (even if all feasible paths are finite, their number may be very large or even infinite). Therefore we allow for a given summary sum_p not to cover all possible paths, meaning $\bigvee_{(r,t)\in sum_p} r$ may not be valid ($\bigvee_{(r,t)\in sum_p} r \not\equiv true$).

Definition 7. *Given a procedure summary sum_p, the **Uncovered Part** of sum_p is $\neg \bigvee_{(r,t)\in sum_p} r$.*

For all inputs that satisfy the uncovered part of the summary nothing is promised: the procedure p might not terminate on such inputs, or terminate with unknown outputs. A summary for which the uncovered part is unsatisfiable ($\bigvee_{(r,t)\in sum_p} r \equiv true$) is called a **full** summary. Note that a full summary only exists for procedures that halt on every input.

Example 2. We return to $p1$ from Fig. 2. Any subset of the set $\{(x < 0, -1), (x \geq 0 \wedge x \geq 2, x), (x \geq 0 \wedge x < 2, 3)\}$ is a summary for p_1. For the summary

$$sum_{p_1} = \{(x < 0, -1), (x \geq 0 \wedge x \geq 2, x)\},$$

the uncovered part is characterized by $x \geq 0 \wedge x < 2$.

2.2 Equivalence

We modify the notions of equivalence from [13] to characterize the set of visible states under which procedures are equivalent, even if they might not be equivalent for every initial state. Let p_1 and p_2 be two procedures with visible variables $V^v_{p_1}$ and $V^v_{p_2}$, respectively. Since their sets of visible variables might be different, we take the union $V^v_{p_1} \cup V^v_{p_2}$ as their set of visible variables V^v_p. Any valuation of this set can be viewed as a visible state of both procedures.

Definition 8. ***State-Equivalences***
Let σ^v_p be a visible state for p_1 and p_2.

- *p_1 and p_2 are* **partially equivalent** *for σ^v_p if and only if the following holds: If p_1 and p_2 both terminate on σ^v_p, then they terminate with the same final state.*
- *p_1 and p_2* **mutually terminate** *for σ^v_p if and only if the following holds: p_1 terminates on σ^v_p if and only if p_2 terminates on σ^v_p.*
- *p_1 and p_2 are* **fully equivalent** *for σ^v_p if and only if p_1 and p_2 are partially equivalent for σ^v_p and mutually terminate for σ^v_p.*

3 Modular Symbolic Execution

A major component of our analysis is the modular symbolic execution, which analyses one procedure at a time while avoiding inlining of called procedures. This prevents unnecessary execution of previously explored paths in called procedures. Assume procedure p calls procedure g. Also assume that a procedure summary for g is given by: $sum_g = \{(r^1, t^1), \ldots, (r^n, t^n)\}$.

Modular symbolic execution is defined as symbolic execution for assignment and test instructions (see Sect. 2.1). For procedure call instruction $g(Y)$ (where $Y \subseteq V_p$) it is defined as follows. For given $R^i_\pi(V_p)$ and $T^i_\pi(V_p)$[4]:

$$R^{i+1}_\pi = R^i_\pi \wedge \Big(\bigvee_{(r,t)\in sum_g} r(T^i_\pi[Y]) \Big) \tag{1}$$

$$\forall x \notin Y.\ T^{i+1}_\pi[x] = T^i_\pi[x] \tag{2}$$

$$\forall y_j \in Y.\ T^{i+1}_\pi[y_j] = ITE(r^1(T^i_\pi[Y]), t^1_j(T^i_\pi[Y]), ITE(r^2(T^i_\pi[Y]), t^2_j(T^i_\pi[Y]), ITE(\ldots, ITE(r^n(T^i_\pi[Y]), t^n_j(T^i_\pi[Y]), UK)\ldots)))$$

where:

- $ITE(b, e_1, e_2)$ is an expression that returns e_1 if the condition b holds and returns e_2, otherwise. It is similar to the conditional operator (?:) in some programming languages.
- t^k_j refers to the jth element (for y_j) of the path transformation t^k.
- UK represents the value that is given if no path condition from sum_g is satisfied. That it, UK is returned when an unexplored path is traversed. Note, however, that since we added $(\bigvee_{(r,t)\in sum_g} r(T^i_\pi[Y])$ to the path condition R^i_π, a path that satisfies R^{i+1}_π will never return UK. Thus, UK is just a place holder.

Modular symbolic execution, as defined here, restricts the analysis of procedure p to paths along which g is called with inputs traversing paths in g that have already been analyzed. For other paths, the reachability condition will be unsatisfiable. In Sect. 6.1 we define an abstraction, which replaces unexplored paths by uninterpreted functions. Thus, the analysis of p may include unexplored (abstracted) paths of g. If the analysis reveals that the unexplored paths are essential in order to determine difference or similarity on the level of p, then refinement is applied by symbolically analysing more of g's paths.

We prove in [29] the connection between modular symbolic execution and regular symbolic execution on the in-lined version of the program. Intuitively, as long as the paths taken in called procedures are covered by the summaries of the called procedures, the following holds: Assume that a path π in p includes a call to procedure g. Then π corresponds to a set of paths in the in-lined version, each of which executing a different path in g, more formally:

[4] We use $r(T^i_\pi[Y])$ to indicate that every $v_k \in V^v_g$ is replaced by the expression $T^i_\pi[y_k]$.

- For every path π^{in} in the in-lined version of p there is a corresponding path π in p such that:
  - $R_{\pi^{in}} \rightarrow R_{\pi}$
  - $R_{\pi^{in}} \rightarrow T_{\pi^{in}} = T_{\pi}$
- For every path π in p, there are paths $\pi_1^{in}, \ldots, \pi_n^{in}$ in the in-lined version of p such that:
  - $R_{\pi} \leftrightarrow \bigvee_{i=1}^{n} R_{\pi_i^{in}}$
  - $\forall i \in [n].\ R_{\pi_i^{in}} \rightarrow T_{\pi_i^{in}} = T_{\pi}$

4 Difference Summary

Throughout the rest of the paper, we refer to a syntactically different pair of procedures as ***modified***, and to a semantically different pair of procedures (not fully equivalent for every state) as ***affected***. Note that a modified procedure is not necessarily affected. Further, an affected procedure is not necessarily modified, but must call (transitively) a modified and affected procedure.

Our main goal is, given two program versions, to evaluate the difference and similarity between them. For that purpose we define the notion of difference summary, in an attempt to capture the semantic difference and similarity between the programs. A difference summary is defined for procedures and extends to programs, by computing the difference summary for the main procedures in the programs.

We start by defining the notion of full difference summary, which precisely captures the difference and similarity between the behaviors of two given procedures. In this section we give all definitions in terms of sets of states that might be infinite.

Definition 9. *A **Full Difference Summary** for two procedures p_1 and p_2 is a triplet*

$$\Delta Full_{p_1,p_2} = (ch_{p_1,p_2}, unch_{p_1,p_2}, termin\_ch_{p_1,p_2})$$

where,

- ch_{p_1,p_2} *is the set of visible states for which both procedures terminate with different final states.*
- $unch_{p_1,p_2}$ *is the set of visible states for which both procedures either terminate with the same final states, or both do not terminate.*
- $termin\_ch_{p_1,p_2}$ *is the set of visible states for which exactly one procedure terminates.*

Note that $ch_{p_1,p_2} \cup unch_{p_1,p_2} \cup termin\_ch_{p_1,p_2}$ covers the entire visible state space. The three sets are related to the state equivalence notions of Definition 8 as follows.

- ch_{p_1,p_2} is the set of the visible states that violate partial equivalence. It only captures differences between terminating paths.
- $termin\_ch_{p_1,p_2}$ is the set of visible states that violate mutual termination.

– $unch_{p_1,p_2}$ is the set of visible states for which the procedures are fully equivalent.

Example 3. Consider the procedures in Fig. 2. The full difference summary for this pair of procedures is:

$$\begin{aligned} ch_{p_1,p_2} &= \{\{x \mapsto 4\}\} \\ unch_{p_1,p_2} &= \{\{x \mapsto c\} \mid c \neq 2 \wedge c \neq 4\} \\ termin\_ch_{p_1,p_2} &= \{\{x \mapsto 2\}\} \end{aligned}$$

For input 2 the old version $p1$ does not change x, while the new version $p2$ reaches an infinite loop, and therefore 2 is in $termin\_ch_{p_1,p_2}$. For input 3, although the paths taken in the two versions are different, the final value of x is the same (3), and therefore 3 is in $unch_{p_1,p_2}$. For input 4, $p1$ does not change x, while $p2$ changes x to 3, and therefore 4 is in ch_{p_1,p_2}.

The full difference summary and any of its three components are generally incomputable, since they require halting information. We therefore suggest to under-approximate the desired sets. In the next section we present an algorithm that computes under-approximated sets and can also strengthen them. The strengthening extends the sets with additional states, thus bringing the computed summary "closer" to the full difference summary.

Definition 10. *Given two procedures p_1, p_2, their* ***Difference Summary***

$$\Delta_{p_1,p_2} = (C(p_1,p_2), U(p_1,p_2))$$

consists of two sets of states where

– $C(p_1,p_2) \subseteq ch_{p_1,p_2}$.
– $U(p_1,p_2) \subseteq unch_{p_1,p_2}$.

A difference summary gives us both an under-approximation and an over-approximation of the difference between procedures, given by $C(p_1,p_2)$ and $\neg U(p_1,p_2)$[5], respectively.

The algorithm presented in the next section is based on the notion of path difference, presented below. Recall that for a given path π, its path summary is the pair (R_π, T_π) (see Definition 5).

Definition 11. *Let p_1 and p_2 be two procedures with the same visible variables $V^v_{p_1} = V^v_{p_2} = V^v_p$, and let π_1 and π_2 be finite paths in CFG_{p_1} and CFG_{p_2}, respectively. Then the* ***Path Difference*** *of π_1 and π_2 is a triplet $(d, T_{\pi_1}, T_{\pi_2})$, where d is defined as follows:*

$$d(V^v_p) \leftrightarrow (R_{\pi_1}(V^v_p) \wedge R_{\pi_2}(V^v_p) \wedge \neg(T_{\pi_1}(V^v_p) = T_{\pi_2}(V^v_p))).$$

[5] We use $\neg$ for set complement with respect to the state space.

We call d the *condition* of the path difference. Note that d implies the reachability conditions of both paths, meaning that for any visible state σ that satisfies d, path π_1 is traversed from σ in CFG_{p_1} and path π_2 is traversed from σ in CFG_{p_2}. Moreover, when starting from σ, the final state of π_1 will be different from the final state of π_2 (at least for one of the variables in V_p^v). If d is satisfiable we say that π_1 and π_2 ***show difference***.

5 Computing Difference Summaries

5.1 Call Graph Traversal

Assume we are given two program versions, each consisting of one *main* procedure and many other procedures that call each other. Assume also a *matching* function, which associates procedures in one program with procedures in the other, based on names (added and removed procedures are matched to the empty procedure). Our objective is to efficiently compute difference summaries for matching procedures in the programs. We are particularly interested in the difference of their main procedures. This goal will be achieved gradually, where precision of the resulting summaries increases, as computation proceeds. In this section we replace the sets of states describing difference summaries by their characteristic functions, in the form of FOL formulas.

As mentioned before, any block of code can be treated as a procedure, not only those defined as procedures by the programmer.

Our main algorithm DiffSummarize, presented in Algorithm 1, provides an overview of our method. The algorithm does not assume that the call graph is cycle-free, and therefore is suitable for recursive programs as well.

For each pair of matched procedures, the algorithm computes a Difference summary Diff$[(p_1, p_2)]$, which is a pair of $C(p_1, p_2)$ and $U(p_1, p_2)$. Sum is a mapping from all procedures to their current summary.

The algorithm computes a set *workSet*, which includes all pairs of procedures for which Diff should be computed. The set *workSet* is initialized with all modified procedures, and all their callers (lines 3–8), as those are the only procedures suspected to be affected. We initially trivially under-approximate Diff for the procedures in *workSet* by $(false, false)$ (line10). We can also safely conclude that all other procedures are not affected (line 14).

Next we analyse all pairs of procedures in *workSet* (lines 17–31), where the order is chosen heuristically. Given procedures p_1 and p_2, if they are syntactically identical, and all called procedures have already been proven to be unaffected (line19) – we can conclude that p_1, p_2 are also unaffected. Otherwise, we compute sum_{p_1} and sum_{p_2} by running ModularSymbolicExecution (presented in Sect. 3) on the code of each procedure separately, up to a certain bound (chosen heuristically).

Since it is possible to visit a pair of procedures p_1, p_2 multiple times we keep the computed summaries in Sum$[p_1]$ and Sum$[p_2]$, and re-use them when re-analyzing the procedures to avoid recomputing path summaries of paths that

have already been visited. We then call algorithm ConstructProcDiffSum (explained in Sect. 5.2) for computing a difference summary for p_1 and p_2.

Each time a difference summary changes (line 27), we need to re-analyse all its callers to check how this newly learned information propagates (line 29).

Algorithm DiffSummarize is modular. It handles each pair of procedures separately, without ever considering the full program and without inlining called procedures.

As mentioned before, Algorithm DiffSummarize is not guaranteed to terminate. Yet it is an *anytime algorithm*. That is, its partial results are meaningful. Furthermore, the longer it runs, the more precise its results are.

Algorithm 1. DiffSummarize(P_1, P_2)

Input: Two program versions P_1, P_2
Output: Difference Summary and a set of Path Difference Summaries for each pair of matching procedures, including $main_{P_1}, main_{P_2}$

```
1: match = ComputeProcedureMatching(P1, P2)
2: FoundDiff[(p1, p2)] = ∅,  for each (p1, p2) ∈ match
3: workSet := ∅
4: newWorkSet:= {(p1, p2) ∈ match : p1 different syntactically from p2}
5: while newWorkSet ≠ workSet do
6:       workSet := newWorkSet
7:       newWorkSet := workSet ∪ {(q1, q2) ∈ match : ∃(p1, p2) ∈ workSet s.t. q1 calls p1 or q2
             calls p2}
8: end while
9: for each (p1, p2) ∈workSet do
10:      Diff[(p1, p2)] := (false, false)
11:      Sum[p1]:=∅, Sum[p2]:=∅
12: end for
13: for each (p1, p2) ∈ match\workSet do
14:      Diff[(p1, p2)] := (false, true)
15:      Sum[p1]:=∅, Sum[p2]:=∅
16: end for
17: while workSet≠ ∅ do
18:      (p1, p2) := chooseNext(workSet)                                  ▷ heuristic order
19:      if p1, p2 are syntactically identical and for all (g1, g2) ∈ match s.t. p1 calls g1 or p2 calls
             g2, Diff[(g1, g2)]=(*,true) then
20:          newDiff := (false,true)
21:      else
22:          Sum[p1] := ModularSymbolicExecution(p1,Sum)
23:          Sum[p2] := ModularSymbolicExecution(p2,Sum)
24:          (newDiff,newFoundDiff) :=ConstProcDiffSum(Sum[p1],Sum[p2],Diff[(p1, p2)])
25:          FoundDiff[(p1, p2)]:=FoundDiff[(p1, p2)] ∪ newFoundDiff
26:      end if
27:      if Diff[(p1, p2)] ≠ newDiff then
28:          Diff[(p1, p2)] := newDiff
29:          workSet := workSet ∪ {(q1, q2) ∈ match: q1 calls p1 or q2 calls p2}
30:      end if
31: end while
32: return (Diff, FoundDiff)
```

5.2 Computing the Difference Summaries for a Pair of Procedures

Algorithm ConstProcDiffSum (presented in Algorithm 2) accepts as input procedure summaries sum_{p_1}, sum_{p_2} and also the current difference summary of p_1, p_2. It returns an updated difference summary Δ_{p_1,p_2}. In addition, it returns

Algorithm 2. ConstProcDiffSum(sum_{p_1}, sum_{p_2},oldDiff)

Input: Procedure summaries sum_{p_1}, sum_{p_2} of procedures p_1, p_2, respectively, and oldDiff, previously computed Δ_{p_1,p_2}
Output: updated Δ_{p_1,p_2}, $found\_diff_{p_1,p_2}$

```
1: (C(p1, p2), U(p1, p2)) := oldDiff
2: found_diff_{p1,p2} = ∅
3: for each (r1, t1) in sum_{p1} do
4:     for each (r2, t2) in sum_{p2} do
5:         diffCond := r1 ∧ r2 ∧ t1 ≠ t2
6:         if diffCond is SAT then
7:             C(p1, p2) := C(p1, p2) ∨ diffCond
8:             found_diff_{p1,p2} := found_diff_{p1,p2} ∪ {(diffCond, t1, t2)}
9:         end if
10:        eqCond := r1 ∧ r2 ∧ t1 = t2
11:        if eqCond is SAT then
12:            U(p1, p2) := U(p1, p2) ∨ eqCond
13:        end if
14:    end for
15: end for
16: return ((C(p1, p2), U(p1, p2)), found_diff_{p1,p2})
```

the set $found\_diff_{p_1,p_2}$ of path differences, for every pair of paths in the two procedure summaries, which *shows difference*.

The construction of *diffCond* in line 5 ensures that (*diffCond*, t_1, t_2) is a valid path difference. We add *diffCond* to $C(p_1, p_2)$ (line 7), and (*diffCond*, t_1, t_2) to $found\_diff_{p_1,p_2}$ (line 8). Thus, we not only know under which conditions the procedures show difference, but also maintain the difference itself (by means of t_1 and t_2).

The construction of *eqCond* in line 10 ensures that for all states that satisfy it the final states of both procedures are identical, as required by the definition of $U(p_1, p_2)$. The satisfiability checks in lines 6, 11 are an optimization that ensures we do not complicate the computed formulas unnecessarily with unsatisfiable formulas.

We avoid recomputing previously computed path differences. For simplicity, we do not show it in the algorithm.

6 Abstraction and Refinement

6.1 Abstraction

In Sect. 3 we show how to define symbolic execution modularly. There, we restrict ourselves to procedure calls with previously analyzed inputs. However, full analysis of each procedure is usually not feasible and often not needed for difference analysis at the program level. In this section we show how partial analysis can be used better.

We abstract the unexplored behaviors of the called procedures by means of uninterpreted functions [18]. A demand-driven refinement is applied to the abstraction when greater precision is needed.

We modify the definition of *Modular symbolic execution* for procedure call instruction $g(Y)$ in the following manner:

- First, we now allow the symbolic execution of p to consider paths along which p calls g with inputs for which g traverses an unexplored path. To do so, we change the definition from Eq. (1) to $R_\pi^{i+1} = R_\pi^i$.
- Second, to deal with the lack of knowledge of the output of g, we introduce a set of uninterpreted functions $UF_g = \{UF_g^j | 1 \leq j \leq |V_g^v|\}$[6]. The uninterpreted function $UF_g^j(T_\pi^i[Y])$ replaces UK in $T_\pi^{i+1}[y_j]$ (Eq. (2)), where $y_j \in Y$ is the j-th parameter to g.

We can now improve the precision of $S_{i+1}[y_j]$ if we exploit not only the summaries of g_1 and g_2 but also their difference summaries. In particular, we can use the fact that $U(g_1, g_2)$ characterizes the inputs for which g_1 and g_2 behave the same. We thus introduce three sets of uninterpreted functions: $UF_{g_1}, UF_{g_2}, UF_{g_1,g_2}$.

We now revisit Eq. (2) of the modular symbolic execution for procedure call $g_1(Y)$, where we replace UK in $T_\pi^{i+1}[y_j]$ with

$$ITE(U(g_1, g_2)(T_\pi^i[Y]), UF_{g_1,g_2}^j(T_\pi^i[Y]), UF_{g_1}^j(T_\pi^i[Y])).$$

Similarly, for a procedure call $g_2(Y)$ we replace UK with

$$ITE(U(g_1, g_2)(T_\pi^i[Y]), UF_{g_1,g_2}^j(T_\pi^i[Y]), UF_{g_2}^j(T_\pi^i[Y])).$$

The set UF_{g_1,g_2} includes common uninterpreted functions, representing our knowledge of equivalence between g_1 and g_2 when called with inputs $T_\pi^i[Y]$, even though their behavior in this case is unknown. In some cases this could be enough to prove the equivalence of the calling procedures p_1, p_2. The sets UF_{g_1} and UF_{g_2} are separate uninterpreted functions, which give us no additional information on the differences or similarities of g_1, g_2.

Example 4. Consider again procedures $p1$, $p2$ in Fig. 2. Let their procedure summaries be

$$sum_{p_1}(x) = \{(x < 0, -1), (x \geq 2, x)\}$$
$$sum_{p_2}(x) = \{(x < 0, -1), (x > 4, x)\}$$

and their difference summary be $\Delta_{p_1,p_2} = (false, x < 2 \vee x > 4)$. When symbolic execution of a procedure g reaches a procedure call $p1(a)$, where a is a variable of the calling procedure g, we will perform:

$$\begin{aligned} R_\pi^{i+1} &= R_\pi^i \\ \forall y_j \neq a.\ T_\pi^{i+1}[y_j] &= T_\pi^i[y_j] \\ T_\pi^{i+1}[a] &= ITE(T_\pi^i[a] < 0, -1, ITE(T_\pi^i[a] \geq 2, T_\pi^i[a], \\ &\qquad ITE(T_\pi^i[a] < 2 \vee T_\pi^i[a] > 4, UF_{p1,p2}^x(T_\pi^i[a]), UF_{p1}^x(T_\pi^i[a])))). \end{aligned}$$

[6] An obvious optimization is to use the previous symbolic state for visible variables of p that are only used by g as inputs but are not changed in g. However, for simplicity of discussion we will not go into those details.

6.2 Refinement

Using the described abstraction, the computed R_π, T_π may contain symbols of uninterpreted functions, and therefore so could *diffCond*$= r_1 \wedge r_2 \wedge t_1 \neq t_2$ and *eqCond* $= r_1 \wedge r_2 \wedge t_1 = t_2$ (lines 5, 10 in Algorithm ConstProcDiffSum). As a result, $C(p_1, p_2)$ and $U(p_1, p_2)$ may include constraints that are *spurious*, that is, constraints that do not represent real differences or similarities between p_1 and p_2. This could occur due to the abstraction introduced by the uninterpreted functions. Thus, before adding *diffCond* to $C(p_1, p_2)$ or *eqCond* to $U(p_1, p_2)$, we need to check whether it is *spurious*. To address spuriousness, we may then need to apply *refinement* by further analysing unexplored parts of the procedures. This includes procedures that are known to be identical in both versions, since their behavior may affect the reachability or the final states, as demonstrated by the example below.

```
1 void f1(int& x) {
2   if (x == 5) {
3     abs(x);
4     if (x == 0) {
5       x = 0;
6       return;
7     }
8   }
9 }
```

```
1 void f2(int& x) {
2   if (x == 5) {
3     abs(x);
4     if (x == 0) {
5       x = 1;
6       return;
7     }
8   }
9 }
```

```
1 void abs(int& x) {
2   if (x >= 1)
3     return;
4   else
5     x = -x;
6 }
```

Fig. 3. Procedure versions in need of refinement

Example 5. To conclude that the procedures in Fig. 3 are equivalent, we need to know that *abs*(5) cannot be zero. Therefore, we need to analyse *abs*, even though it was not changed or affected.

We use the technique introduced in [4]: Let φ be a formula we wish to add to either $C(p_1, p_2)$ or $U(p_1, p_2)$ ($\varphi \in \{$*diffCond*, *eqCond*$\}$) such that φ includes symbols of uninterpreted functions. Before being added, it should be checked for spuriousness.

For every $k \in \{1, 2\}$, assume procedure p_k calls procedure $g_k(Y_k)$ at location l_{i_k} on the single path π' from p_k, described by φ. For every $k \in \{1, 2\}$ apply symbolic execution up to a certain limit on g_k with the pre-condition

$$\varphi \wedge \neg \left(\bigvee_{(r,t) \in sum_{g_k}} r\left(T_{\pi'}^{i_k - 1}[Y_k]\right) \right) \wedge V_g^v = T_{\pi'}^{i_k - 1}[Y_k].$$

When the reachability checks are performed with this precondition, only new paths reachable from this call in p_k are explored. For every such new path π,

add (R_π, T_π) to sum_{g_k}, replace the old sum_{g_k} with the new sum_{g_k} in φ and check for satisfiability again. As a result, we either find a real difference or similarity, or eliminate all the spurious path differences that involve the explored path π in g_k. The refinement suggested above can be extended in a straightforward manner to any number of function calls along a path.

Example 6. Consider again the procedures in Fig. 3. Assume that the current summaries of $\text{abs}_1 = \text{abs}_2 = \text{abs}$ are empty, but it is known that both versions are identical (unmodified syntactically). We get (using symbolic execution and Algorithm 2) the *diffCond* for p_1 and p_2:

$$\begin{aligned} \mathit{diffCond} = \Big[& x = 5 \wedge \Big(\mathit{ITE}\left(\mathit{true}, UF_{abs_1,abs_2}(x), UF_{abs_1}(x)\right) = 0 \Big) \\ & \wedge x = 5 \wedge \Big(\mathit{ITE}\left(\mathit{true}, UF_{abs_1,abs_2}(x), UF_{abs_2}(x)\right) = 0 \Big) \wedge 0 \neq 1 \Big] \\ \equiv \Big[& x = 5 \wedge UF_{abs_1,abs_2}(x) = 0 \Big] \end{aligned}$$

Next we use $x = 5$ as a pre-condition, and perform symbolic execution, updating the summary for abs: $(x \geq 1, x)$. Now *diffCond* is:

$$\begin{aligned} & \Big[x = 5 \wedge \Big(\mathit{ITE}\Big(x \geq 1, x, \mathit{ITE}(\mathit{true}, UF_{abs_1,abs_2}(x), UF_{abs_1}(x))\Big) = 0 \Big) \\ & \wedge x = 5 \wedge \Big(\mathit{ITE}\Big(x \geq 1, x, \mathit{ITE}\left(\mathit{true}, UF_{abs_1,abs_2}(x), UF_{abs_2}(x)\right)\Big) = 0 \Big) \wedge 0 \neq 1 \Big] \\ & \equiv \Big[x = 5 \wedge \Big(\mathit{ITE}\Big(x \geq 1, x, UF_{abs_1,abs_2}(x)\Big) = 0 \Big) \Big] \equiv x = 5 \wedge x = 0 \end{aligned}$$

which is now unsatisfiable. We thus managed to eliminate a spurious difference without computing the full summary of *abs*.

Once a difference summary is computed, we can choose whether to refine the difference by exploring more paths in the individual procedures; or, if *diffCond* or *eqCond* contains uninterpreted functions, to explore in a demand driven manner the procedures summarized by the uninterpreted functions; or continue the analysis in a calling procedure, where possibly the unknown parts of the current procedures will not be reachable. In Sect. 8 we describe the results on our benchmarks in two extreme modes: running refinement always immediately when needed (ModDiffRef), and always delaying the refinement (ModDiff).

7 Related Work

A formal definition of equivalence between programs is given in [13]. We extend these definitions to obtain a finer-grained characterization of the differences.

We extend the path-wise symbolic summaries and deltas given in [25], and show how to use them in modular symbolic execution, while abstracting unknown parts.

The SymDiff [20] tool and the Regression Verification Tool (RVT) [14] both check for partial equivalence between pairs of procedures in a program, while abstracting procedure calls (after transforming loops into recursive calls). Unlike our tool, both SymDiff and RVT are only capable of proving equivalences, not disproving them. In [16], a work that has similar ideas to ours, conditional equivalence is used to characterize differences with SymDiff. The algorithm presented in [16] is able to deal with loops and recursion; however, the algorithm is not fully implemented in SymDiff. Our tool is capable of dealing soundly with loops, and as our experiments show, is often able to produce full difference summaries for programs with unbounded loops. We also provide a finer-grained result, by characterizing the inputs for which there are (no) semantic differences.

Both SymDiff and RVT lack refinement, which often causes them to fail at proving equivalence, as shown by our experiments in Sect. 8. Both tools are, however, capable of proving equivalence between programs (using, among others, invariants and proof rules) that cannot be handled by our method. Our techniques can be seen as an orthogonal improvement. SymDiff also has a mode that infers common invariants, as descried in [21], but it failed to infer the required invariants for our examples.

Under-constrained symbolic execution, meaning symbolic execution of a procedure that is not the entry point of the program is presented in [27,28], where it is used to improve scalability while using the old version as a golden model. The algorithm presented in [27,28] does not provide any guarantees on its result, and it does not attempt to propagate found differences to inputs of the programs. By contrast, our algorithm does not stop after analysing only the syntactically modified procedures, but continues to their calling procedures. On the other hand, procedures that do not call modified procedures (transitively) are immediately marked as equivalent. Thus, we avoid unnecessary work. In [27], the new program version is checked, while assuming that the old version is correct. We do not use such assumptions, as we are interested in all differences: new bugs, bug fixes, and functional differences such as new features.

In [5,26] summaries and symbolic execution are also used to compute differences. The technique there leverages a light-weight static analysis to help guide symbolic execution only to potentially differing paths. In [6], symbolic execution is applied simultaneously on both versions, with the purpose of guiding symbolic execution to changed paths. Both techniques, however, lack modularity and abstractions. A possible direction for new research would be to integrate our approach with one of the two.

Our approach is similar to the compositional symbolic execution presented in [4,12], that is applied to single programs. However, the analysis in [4,12] is top-down while ours works bottom-up, starting from syntactically different procedures, proceeding to calling procedures only as long as they are affected by the difference of previously analyzed procedures. The analysis stops as soon as unaffected procedures are reached.

Our algorithm is unique in that it provides both an under- and over-approximations of the differences, while all the described methods have no guarantees or only provide one of the two.

8 Experimental Results

We implemented the algorithm presented in Sect. 5 with the abstractions from Sect. 6 on top of the CProver framework (version 787889a), which also forms the foundation of the verification tools CBMC [8], SatAbs [9], Impact [22] and Wolverine [19]. The implementation is available online [2]. Since we analyse programs at the level of an intermediate language (goto-language, the intermediate language used in the CProver framework), we can support any language that can be translated to this language (currently Java and C). We report results for two variants of our tool – without refinement (ModDiff for Modular Demand-driven Difference), and with refinement (ModDiffRef). The unwinding limit is set to 5 in both variants.

SymDiff and RVT: We compared our results to two well established tools, SymDiff and RVT. For SymDiff, we used the *smack* [3] tool to translate the C programs into the Boogie language, and then passed the generated Boogie files to the latest available online version of SymDiff.

8.1 Benchmarks and Results

We analysed 28 C benchmarks, where each benchmark includes a pair of syntactically similar versions. Our benchmarks are available online [1]. Our benchmarks were chosen to demonstrate some of the benefits of our technique, as explained below. A total of 16 benchmarks are semantically equivalent (Table 1), while some benchmarks contain semantically different procedures. When using refinement, our algorithm was able to prove all equivalences between programs but not between all procedures (although some were actually equivalent). RVT's refinement is limited to loop unrolling, and its summaries are limited as well. Thus, it cannot prove equivalence of ancestors of recursive procedures or loops that are semantically different. Also, if it fails to prove equivalence of semantically equivalent recursive procedures or loops, it cannot succeed in proving equivalence of their ancestors. As previously mentioned, RVT can sometimes prove equivalence when our tool cannot. The latest available version of SymDiff failed to prove most examples, possibly also for lack of refinement.

8.2 Analysis

We now explain in detail the benefit of our method on specific benchmarks. The *LoopUnrch* benchmarks illustrate the advantages of summaries. Our tool analyses foo1 and foo2 from Fig. 4c, finds a condition under which those procedures are different (for example inputs $-1, 1$), and a condition under which they are

Table 1. Experimental results. Numbers are time in seconds, F indicates a failure to prove equivalence in (a), and that the difference summary of main was not full (some differences were not found) in (b).

Benchmark	ModDiff	ModDiffRef	RVT	SymDiff
Const	0.545s	0.541s	4.06s	14.562s
Add	0.213s	0.2s	3.85s	14.549s
Sub	0.258s	0.308s	5.01s	F
Comp	0.841s	0.539s	5.19s	F
LoopSub	0.847s	1.179s	F	F
UnchLoop	F	2.838s	F	F
LoopMult2	1.666s	1.689s	F	F
LoopMult5	F	3.88s	F	F
LoopMult10	F	9.543s	F	F
LoopMult15	F	21.55s	F	F
LoopMult20	F	49.031s	F	F
LoopUnrch2	0.9s	0.941s	F	F
LoopUnrch5	1.131s	1.126s	F	F
LoopUnrch10	1.147s	1.168s	F	F
LoopUnrch15	1.132s	1.191s	F	F
LoopUnrch20	1.157s	1.215s	F	F

(a) Semantically equivalent

Benchmark	ModDiff	MDDiffRef
LoopSub	1.187s	2.426s
UnchLoop	F	8.053s
LoopMult2	3.01s	3.451s
LoopMult5	F	5.914s
LoopMult10	F	10.614s
LoopMult15	F	14.024s
LoopMult20	F	25.795s
LoopUnrch2	2.157s	2.338s
LoopUnrch5	2.609s	3.216s
LoopUnrch10	2.658s	3.481s
LoopUnrch15	2.835s	3.446s
LoopUnrch20	3.185s	3.342s

(b) Semantically different

equivalent ($a \geq 0$). In all versions of this benchmark, foo1 and foo2 are called with positive (increasing) values of a (and b), and hence the loop is never performed. We are able to prove equivalence efficiently in all versions, both with and without refinement.

The *LoopMult* benchmarks illustrate the advantages of refinement. Our tool analyses foo1 and foo2 from Fig. 4a, finds a condition under which those procedures are different (for example inputs $1, -1$), and a condition under which they are equivalent. We also summarise all behaviors that correspond to unwinding of the loop 5 times. This unwinding is sufficient when the procedures are calls with inputs $2, 2$ (benchmark *LoopMult2*, the first main from Fig. 4b), and therefore both MD-Diff and MD-DiffRef are able to prove equivalence quickly. This unwinding is, however, not sufficient for benchmark *LoopMult5* (the second main from Fig. 4b). Thus, MD-Diff is not able to prove equivalence (the summary of foo1/2 does not cover the necessary paths), while MD-DiffRef analyses the missing paths (where $5 \leq a < 7 \wedge b = 5$), and is able to prove equivalence. As the index of the *LoopMult* benchmark increases, the length of the required paths and their number increases, and the analysis takes more time, accordingly, but only necessary paths are explored.

The remaining 12 benchmarks are not equivalent, and our algorithm is able to find inputs for which they differ (presented in Table 1). Since both SymDiff and RVT are only capable of proving equivalences, not disproving them, we did not compare to those tools.

```
int foo1(int a, int b) {
  int c=0;
  for (int i=1; i<=b; ++i)
    c+=a;
  return c;
}

int foo2(int a, int b) {
  int c=0;
  for (int i=1; i<=a; ++i)
    c+=b;
  return c;
}
```

(a) procedures foo1 and foo2 in LoopMult benchmarks

```
int main(int x,
    char*argv[]) {
  //LoopMult2
  return foo(2,2);
}

int main(int x,
    char*argv[]) {
  //LoopMult5
  if (x>=5 && x<7)
    return foo(x,5);
  return 0;
}
```

(b) main functions of LoopMult2 and LoopMult5

```
int foo1(int a, int b) {
  int c=0;
  if (a<0) {
    for (int i=1;
        i<=b;++i)
      c+=a;
  }
  return c;
}

int foo2(int a, int b) {
  int c=0;
  if (a<0) {
    for (int i=1;
        i<=a;++i)
      c+=b;
  }
  return c;
}
```

(c) procedures foo1 and foo2 in LoopUnrch benchmarks

Fig. 4. LoopMult and LoopUnrch benchmarks

9 Conclusion

We developed a modular and demand-driven method for finding semantic differences and similarities between program versions. It is able to soundly analyse programs with loops and guide the analysis towards "interesting" paths. Our method is based on (partially abstracted) procedure summarizations, which can be refined on demand. Our experimental results demonstrate the advantage of our approach due to these features.

References

1. ModDiff benchmarks. https://github.com/AnnaTrost/ModDiff/tree/master/benchmarks
2. ModDiff tool. https://github.com/AnnaTrost/ModDiff
3. SMACK software verifier and verification toolchain. https://github.com/smackers/smack
4. Anand, S., Godefroid, P., Tillmann, N.: Demand-driven compositional symbolic execution. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 367–381. Springer, Heidelberg (2008). doi:10.1007/978-3-540-78800-3_28
5. Backes, J., Person, S., Rungta, N., Tkachuk, O.: Regression verification using impact summaries. In: Bartocci, E., Ramakrishnan, C.R. (eds.) SPIN 2013. LNCS, vol. 7976, pp. 99–116. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39176-7_7
6. Cadar, C., Palikareva, H.: Shadow symbolic execution for better testing of evolving software. In: Companion Proceedings of the 36th International Conference on Software Engineering, pp. 432–435. ACM (2014)
7. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**(2), 82–90 (2013)

8. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). doi:10.1007/978-3-540-24730-2_15
9. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005). doi:10.1007/978-3-540-31980-1_40
10. Francez, N.: Program Verification. Addison-Wesley Longman, Boston (1992)
11. Gao, D., Reiter, M.K., Song, D.: BinHunt: automatically finding semantic differences in binary programs. In: Chen, L., Ryan, M.D., Wang, G. (eds.) ICICS 2008. LNCS, vol. 5308, pp. 238–255. Springer, Heidelberg (2008). doi:10.1007/978-3-540-88625-9_16
12. Godefroid, P.: Compositional dynamic test generation. In: ACM SigPlan Notices, vol. 42, pp. 47–54. ACM (2007)
13. Godlin, B., Strichman, O.: Inference rules for proving the equivalence of recursive procedures. Acta Informatica **45**(6), 403–439 (2008)
14. Godlin, B., Strichman, O.: Regression verification. In: Proceedings of the 46th Annual Design Automation Conference, pp. 466–471. ACM (2009)
15. Godlin, B., Strichman, O.: Regression verification: proving the equivalence of similar programs. Softw. Test. Verif. Reliab. **23**(3), 241–258 (2013)
16. Kawaguchi, M., Lahiri, S.K., Rebelo, H.: Conditional equivalence. Technical report, MSR-TR-2010-119 (2010)
17. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
18. Kroening, D., Strichman, O.: Equality logic and uninterpreted functions. Decision Procedures. Texts in Theoretical Computer Science (An Eatcs Series), pp. 59–80. Springer, Heidelberg (2008). doi:10.1007/978-3-540-74105-3_3
19. Kroening, D., Weissenbacher, G.: Interpolation-based software verification with Wolverine. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 573–578. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22110-1_45
20. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: SYMDIFF: a language-agnostic semantic Diff tool for imperative programs. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 712–717. Springer, Heidelberg (2012). doi:10.1007/978-3-642-31424-7_54
21. Lahiri, S.K., McMillan, K.L., Sharma, R., Hawblitzel, C.: Differential assertion checking. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 345–355. ACM (2013)
22. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006). doi:10.1007/11817963_14
23. Partush, N., Yahav, E.: Abstract semantic differencing for numerical programs. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 238–258. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38856-9_14
24. Partush, N., Yahav, E.: Abstract semantic differencing via speculative correlation. In: ACM SIGPLAN Notices, vol. 49, pp. 811–828. ACM (2014)
25. Person, S., Dwyer, M.B., Elbaum, S., Pasareanu, C.S.: Differential symbolic execution. In: Foundations of Software Engineering, pp. 226–237. ACM (2008)
26. Person, S., Yang, G., Rungta, N., Khurshid, S.: Directed incremental symbolic execution. In: ACM SIGPLAN Notices, vol. 46, pp. 504–515. ACM (2011)
27. Ramos, D.A., Engler, D.: Under-constrained symbolic execution: correctness checking for real code. In: 24th USENIX Security Symposium, pp. 49–64 (2015)

28. Ramos, D.A., Engler, D.R.: Practical, low-effort equivalence verification of real code. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 669–685. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22110-1_55
29. Trostanetski, A., Grumberg, O., Kroening, D.: Modular demand-driven analysis of semantic difference for program versions. Technical report, CS-2017-02. http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-info.cgi/2017/CS/CS-2017-02
30. Wong, W.E., Horgan, J.R., London, S., Agrawal, H.: A study of effective regression testing in practice. In: The Eighth International Symposium on Software Reliability Engineering, Proceedings, pp. 264–274. IEEE (1997)