

# Modular Demand-Driven Analysis of Semantic Difference for Program Versions<sup>★</sup>

Anna Trostanetski<sup>1</sup>, Orna Grumberg<sup>1</sup> and Daniel Kroening<sup>2</sup>

<sup>1</sup> Technion – Israel Institute of Technology  
{annat,orna}@cs.technion.ac.il,

<sup>2</sup> 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 are different. 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 procedure call graph. Our approach analyses only those parts of the programs that are affected by the changes. Moreover, the analysis is *modular*, applied to a single pair of procedures at a time. Called procedures are not inlined. Rather, their previously computed summaries and *difference summary* are used.

For efficiency, procedure summaries and difference summaries can be *abstracted* and may be *refined* on-demand.

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

## 1 Introduction

In this work we present a modular and demand-driven algorithm for finding 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 correct, a “golden model” for the new, less-tested version [26].

---

<sup>★</sup> Supported by the ERC project 280053 (CPROVER) and the H2020 FET OPEN 712689 SC<sup>2</sup>.

- Revealing security vulnerabilities that were eliminated by the new version [9]. This information can be used to produce zero-day attacks.
- More generally, identifying and characterizing changes in the program’s functionality [21].

Semantic difference has been widely studied, and several techniques have been suggested [9, 12–14, 18, 20–23]. In our work we aim at enhancing 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, changes between versions are small and limited to procedures deep inside the call graph (see Figure 1). In such cases, it would be helpful to know how these changes affect the program as a whole, without analysing the whole 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 program as a whole. Our work is therefore particularly beneficial when applied to programs that are syntactically similar. Even though it is applicable to programs which are very different from each other, our technique would yield less savings in those cases.

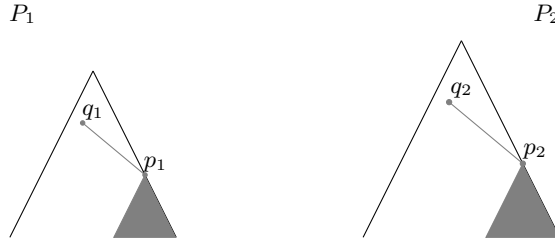


Figure 1: Call graphs of two program versions  $P_1, P_2$ , where their syntactic differences are local to the procedures  $p_1, p_2$ , and the bodies of procedures  $q_1, q_2$  are identical.

Our 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 difference 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.

As mentioned before, the goal of this work is to analyse the difference between two program versions which are relatively similar. Our main concern is to avoid unnecessary analysis, thus achieving scalability. 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 finding 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 could prove equivalence or find differences, while the others failed to do so.

### 1.1 Our Approach

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  $P_2$ , which are syntactically different.

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

Next, we compute a (partial) *difference summary*  $(u\_ch_{p_1, p_2}, u\_unch_{p_1, p_2})$  for  $p_1, p_2$ , where  $u\_ch_{p_1, p_2}$  is a set of initial states for which  $p_1$  and  $p_2$  terminate with different final states.  $u\_unch_{p_1, p_2}$  is a set of initial states for which  $p_1$  and  $p_2$  terminate with identical final state. Both sets are under-approximations. However, the complement of  $u\_unch_{p_1, p_2}$ , denoted  $\neg u\_unch_{p_1, p_2}$ , 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  $u\_unch_{p_1, p_2} \equiv true$  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, in building their procedure summaries and difference summary, we use the already computed summaries of  $p_1, p_2$ .

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  $u\_ch_{p_1, p_2}$  and  $u\_unch_{p_1, p_2}$  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 tradeoff between the amount of computation and the obtained precision will be manageable.

## 2 Preliminaries

We start by defining some basic notions 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**  $\sigma_p$  is a valuation of the procedure's variables,  $\sigma_p = \{v \mapsto c \mid v \in V_p, c \in D_v\}$ , 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 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.

We modify the definitions of the notions of equivalence from [11] 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  $V_{p_1}^v$  and  $V_{p_2}^v$ , respectively. Since their sets of visible variables might be different, we take the union  $V_{p_1}^v \cup V_{p_2}^v$  as their set of visible variables  $V_p^v$ . Any valuation of this set can be viewed as a visible state of both procedures.

**Definition 1. State-Equivalences**

Let  $\sigma_p^v$  be a visible state for  $p_1$  and  $p_2$ .

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

Next we define a Control Flow Graph (CFG), also known as a flow program in [8], for procedure  $p$ .

**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 vertices 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  $V_p$ .
- **Procedure call:**  $g(Y)$ , where  $Y \subseteq V_p$  and the values of variables in  $Y$  are assigned to the visible variables of procedure  $g$ . 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 by  $T$ , and the other by  $F$ .

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

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  $\sigma_p^v$  if the path (defined in the appendix) traversed in  $p$  from  $\sigma_p^v$  is finite and maximal. A program terminates on a visible state  $\sigma_p^v$  if its main procedure terminates.

The following semantic characteristics are associated with finite paths, similarly to the definitions for flow programs in [8]. 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 3.** Let  $\pi$  be a finite path in procedure  $p$ .

- The **Reachability Condition** of  $\pi$ , denoted  $R_\pi(V_p^v)$ , is a condition on the visible states at the beginning of  $\pi$ , which guarantees that the control will traverse  $\pi$ .
- The **State Transformation** of  $\pi$ , denoted  $T_\pi(V_p^v)$ , describes the final state of  $\pi$ , obtained if control traverses  $\pi$  starting with some valuation  $\sigma_p^v$  of  $V_p^v$ .  $T_\pi(V_p^v)$  is expressed in terms 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  $V$  at the beginning of  $\pi$ . Let  $T_\pi(V_p^v) = (f_1, \dots, f_{|V_p^v|})$  and  $T_{\pi'}(V_p^v) = (f'_1, \dots, f'_{|V_p^v|})$  be two state transformations. Then,  $T_\pi(V_p^v) = T_{\pi'}(V_p^v)$  if and only if, for every  $1 \leq i \leq |V_p^v|$ ,  $f_i = f'_i$ .

<pre> void p1(int&amp; x) { 1:  if (x &lt; 0) { 2:      x = -1; 3:      return; 4:  } 5:  if (x &gt;= 2) 6:      return; 7:  while (x == 2) 8:      x = 2; 9:  x = 3; } </pre>	<pre> void p2(int&amp; x) { 1:  if (x &lt; 0) { 2:      x = -1; 3:      return; 4:  } 5:  if (x &gt; 4) 6:      return; 7:  while (x == 2) 8:      x = 2; 9:  x = 3; } </pre>
--	---

Figure 2: Examples of procedure versions

*Example 1.* Consider procedure p1 in Figure 2. Its only visible variable is  $x$ , used as both input and output. Consider the paths that correspond to the following line numbers:  $\alpha = (1, 2, 3)$  and  $\beta = (1, 5, 6)$ . Then,

$$\begin{aligned}
R_\alpha(x) &= x < 0 & R_\beta(x) &= ((\neg(x < 0)) \wedge x \geq 2) \equiv x \geq 2 \\
T_\alpha(x) &= -1 & T_\beta(x) &= x
\end{aligned}$$

A path  $\pi$  is called **feasible** if  $R_\pi$  is satisfiable, meaning that there exists an input that traverses the path  $\pi$ . Note that, in p1 from Figure 2, the path  $(1, 5, 7, 8, 7, 9)$  is not feasible.

In our work, a program is represented by its *call graph* (defined in the appendix), and each procedure  $p$  is represented by its control flow graph  $CFG_p$ .

## 2.1 Symbolic Execution

*Symbolic execution* [5, 15] (path-based) is a form of alternative execution of a procedure 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  $\pi = l_1, \dots, l_n$ , we use symbolic execution to compute the reachability condition  $R_\pi(V_p^v)$  and state transformation  $T_\pi(V_p^v)$ . The computation takes place 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  $l_1, \dots, l_{i-1}$ , respectively. Initialization:

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

Assume  $R_\pi^i(V_p)$ ,  $T_\pi^i(V_p)$  are already defined.  $R_\pi^{i+1}(V_p)$ ,  $T_\pi^{i+1}(V_p)$  are 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.
- 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  $\pi$  we get:

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

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

**Definition 4.** Given a finite maximal path  $\pi$  in  $p$ , a **Path Summary** (also known as a partition-effect pair in [22]) is the pair  $(R_\pi(V_p^v), T_\pi(V_p^v))$ .

**Definition 5.** A **Procedure Summary** (also known as a symbolic summary in [22]), for a procedure  $p$ , is a set of path summaries

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

<sup>1</sup> Since we assume that all inputs are given through visible variables, and therefore no hidden variable is used before it is initialized,  $V_p^h$  won't appear in  $R_\pi^{n+1}(V_p)$  and  $= T_\pi^{n+1}(V_p) \downarrow_{V_p^v}^1$ .

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 ( $\bigvee_{(r,t) \in sum_p} r \neq TRUE$ ).

**Definition 6.** Given a procedure summary  $sum_p$ , its **Uncovered Part** 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 path is unsatisfiable ( $\bigvee_{(r,t) \in sum_p} r \equiv TRUE$ ) is called a **full** summary. Note that a full summary could only exist for procedures that halt on every input.

*Example 2.* We return to  $p_1$  from Figure 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  $x \geq 0 \wedge x < 2$ .

### 3 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 7.** 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 1 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 Figure 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  $p_1$  does not change  $x$ , while the new version  $p_2$  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,  $p_1$  does not change  $x$ , while  $p_2$  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 8.** Given two procedures  $p_1, p_2$ , their **Difference Summary**

$$\Delta_{p_1, p_2} = (u\_ch_{p_1, p_2}, u\_unch_{p_1, p_2}),$$

consists of two sets of states where,

- $u\_ch_{p_1, p_2} \subseteq ch_{p_1, p_2}$ .
- $u\_unch_{p_1, p_2} \subseteq unch_{p_1, p_2}$ .

A difference summary gives us both an under-approximation and an over-approximation of the difference between procedures, given by  $u\_ch_{p_1, p_2}$  and  $\neg u\_unch_{p_1, p_2}$ <sup>2</sup>, respectively.

The algorithm presented in the next section is based on the notion of path difference, presented below. Recall that for a given path  $\pi$ , its path summary is  $(R_\pi, T_\pi)$  (see definition 4).

**Definition 9.** Let  $p_1$  and  $p_2$  be two procedures with the same visible variables  $V_{p_1}^v = V_{p_2}^v = V_p^v$ , and let  $\pi_1$  and  $\pi_2$  be finite paths in  $CFG_{p_1}$  and  $CFG_{p_2}$ , respectively. Then the **Path Difference** of  $\pi_1$  and  $\pi_2$  is a triplet  $(d, T_{\pi_1}, T_{\pi_2})$ , where  $d$  is defined as follows:

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

We call  $r$  the *condition* of the path difference. Note that  $r$  implies the reachability conditions of both paths, meaning that for any visible state  $\sigma$  that satisfies  $r$ , path  $\pi_1$  is traversed from  $\sigma$  in  $CFG_{p_1}$  and path  $\pi_2$  is traversed from  $\sigma$  in  $CFG_{p_2}$ . Moreover, when starting from  $\sigma$ , the final state of  $\pi_1$  will be different from the final state of  $\pi_2$  (at least for one of the variables in  $V_p^v$ ). If  $r$  is satisfiable we say that  $\pi_1$  and  $\pi_2$  **show difference**.

### 3.1 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:  $sum_g = \{(r^1, t^1), \dots, (r^n, t^n)\}$ .

*Modular symbolic execution* is defined as symbolic execution for assignment and test instructions (see Section 2.1). For procedure call instruction  $g(Y)$  (where  $Y \subseteq V_p$ ) it is defined as follows. For given  $R_\pi^i(V_p), T_\pi^i(V_p)$ :

$$R_\pi^{i+1} = R_\pi^i \wedge \left( \bigvee_{(r, t) \in sum_g} r(T_\pi^i[Y]) \right) \quad (1)$$

$$\forall y_j \notin Y. T_\pi^{i+1}[y_j] = T_\pi^i[y_j] \quad (2)$$

$$\forall y_j \in Y. T_\pi^{i+1}[y_j] = ITE(r^1(T_\pi^i[Y]), t_j^1(T_\pi^i[Y]), ITE(r^2(T_\pi^i[Y]), t_j^2(T_\pi^i[Y]), \\ ITE(\dots, ITE(r^n(T_\pi^i[Y]), t_j^n(T_\pi^i[Y]), UK) \dots))),$$

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.

<sup>2</sup> We use  $\neg$  for set complement with respect to the state space.

- $t_j^k$  refers to the  $j$ th 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 is,  $UK$  is returned when an unexplored path is traversed. Note, however, that since we added  $(V_{(r,t) \in sum_g} r(T_\pi^i(V_p)[Y]))$  to the path condition  $T_\pi^{i+1}(V_p)$ , a path that satisfies  $R_\pi^{i+1}(V_p)$  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 that traverse paths in  $g$  that have already been analyzed. For other paths, the reachability condition will be unsatisfiable. In Section 5.1 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.

## 4 Computing Difference Summaries

### 4.1 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. 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.

We assume that the call graphs of the entire programs are available, as well as the CFGs for every procedure in both programs. We further assume that we can obtain a matching between procedures in the different programs, based on their names. Added and removed procedures will be matched to the empty procedure.

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 compatible with recursive programs as well.

For each pair of matched procedures, the algorithm computes a Difference summary  $\text{Diff}[(p_1, p_2)]$ , which is a pair of  $u\_ch_{p_1, p_2}$  and  $u\_unch_{p_1, p_2}$ .

The algorithm computes a set *workSet*, which includes all pairs of procedures for which  $\text{Diff}$  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  $\text{Diff}$  for the

procedures in *workSet* by *(false, false)* (line 10). We can also safely conclude that all other procedures are not affected (line 13).

Next we analyse all pairs of procedures in *workSet* (lines 15–29), 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 (line 17) – 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 Section 3.1) 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 Section 4.2) for computing a difference summary for  $p_1$  and  $p_2$ .

Each time a difference summary changes (line 25), we need to re-analyse all callers to check how this newly learned information propagates (line 27).

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.

## 4.2 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  $\Delta_{p_1, p_2}$ . In addition, it returns 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  $u\_ch_{p_1, p_2}$  (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  $u\_unch_{p_1, p_2}$ . 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.

**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:  $match = \text{COMPUTEPROCEDUREMATCHING}(P_1, P_2)$ 
2:  $\text{FoundDiff}[(p_1, p_2)] = \emptyset$ , for each  $(p_1, p_2) \in match$ 
3:  $workSet := \emptyset$ 
4:  $newWorkSet := \{(p_1, p_2) \in match : p_1 \text{ different syntactically from } p_2\}$ 
5: while  $newWorkSet \neq workSet$  do
6:    $workSet := newWorkSet$ 
7:    $newWorkSet := workSet \cup \{(q_1, q_2) \in match : \exists (p_1, p_2) \in workSet \text{ s.t. } q_1 \text{ calls } p_1 \text{ or } q_2 \text{ calls } p_2\}$ 
8: end while
9: for each  $(p_1, p_2) \in workSet$  do
10:    $\text{Diff}[(p_1, p_2)] := (false, false)$ 
11: end for
12: for each  $(p_1, p_2) \in match \setminus workSet$  do
13:    $\text{Diff}[(p_1, p_2)] := (false, true)$ 
14: end for
15: while  $workSet \neq \emptyset$  do
16:    $(p_1, p_2) := \text{CHOOSENEXT}(workSet)$  ▷ heuristic order
17:   if  $p_1, p_2$  are syntactically identical and for all  $(g_1, g_2) \in match$  s.t.  $p_1$  calls  $g_1$  or  $p_2$  calls  $g_2$ ,  $\text{Diff}[(g_1, g_2)] = (*, true)$  then
18:      $newDiff := (false, true)$ 
19:   else
20:      $\text{Sum}[p_1] := \text{MODULARSYMBOLICEXECUTION}(p_1, \text{Sum})$ 
21:      $\text{Sum}[p_2] := \text{MODULARSYMBOLICEXECUTION}(p_2, \text{Sum})$ 
22:      $(newDiff, newFoundDiff) := \text{CONSTPROCDIFFSUM}(\text{Sum}[p_1], \text{Sum}[p_2], \text{Diff}[(p_1, p_2)])$ 
23:      $\text{FoundDiff}[(p_1, p_2)] := \text{FoundDiff}[(p_1, p_2)] \cup newFoundDiff$ 
24:   end if
25:   if  $\text{Diff}[(p_1, p_2)] \neq newDiff$  then
26:      $\text{Diff}[(p_1, p_2)] := newDiff$ 
27:      $workSet := workSet \cup \{(q_1, q_2) \in match : q_1 \text{ calls } p_1 \text{ or } q_2 \text{ calls } p_2\}$ 
28:   end if
29: end while
30: return ( $\text{Diff}$ ,  $\text{FoundDiff}$ )
```

---

**Algorithm 2** CONSTPROCDIFFSUM( $sum_{p_1}, sum_{p_2}, \text{oldDiff}$ )

---

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

```

1:  $(u\_ch_{p_1, p_2}, u\_unch_{p_1, p_2}) := \text{oldDiff}$ 
2:  $found\_diff_{p_1, p_2} = \emptyset$ 
3: for each  $(r_1, t_1)$  in  $sum_{p_1}$  do
4:   for each  $(r_2, t_2)$  in  $sum_{p_2}$  do
5:      $diffCond := r_1 \wedge r_2 \wedge t_1 \neq t_2$ 
6:     if  $diffCond$  is SAT then
7:        $u\_ch_{p_1, p_2} := u\_ch_{p_1, p_2} \vee diffCond$ 
8:        $found\_diff_{p_1, p_2} := found\_diff_{p_1, p_2} \cup \{(diffCond, t_1, t_2)\}$ 
9:     end if
10:     $eqCond := r_1 \wedge r_2 \wedge t_1 = t_2$ 
11:    if  $eqCond$  is SAT then
12:       $u\_unch_{p_1, p_2} := u\_unch_{p_1, p_2} \vee eqCond$ 
13:    end if
14:  end for
15: end for
16: return  $((u\_ch_{p_1, p_2}, u\_unch_{p_1, p_2}), found\_diff_{p_1, p_2})$ 
```

---

## 5 Abstraction and Refinement

### 5.1 Abstraction

In Section 3.1 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 some unexplored behaviors of the called procedures by means of uninterpreted functions [16]. 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 equation (1) to  $R_{\pi}^{i+1} = R_{\pi}^i$ .
- Second, to deal with the uncertainty of the output of  $g$ , we introduce a set of uninterpreted functions  $UF_g = \{UF_g^j : 1 \leq j \leq |V_g^v|\}$ <sup>3</sup>. The uninterpreted function  $UF_g^j(T_{\pi}^i[Y])$  replaces  $UK$  in  $T_{\pi}^{i+1}[y_j]$  (equation (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  $u\_unch_{g_1, g_2}$  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 Equation (2) of the modular symbolic execution for procedure call  $g_1(Y)$ , where we replace  $UK$  in  $S_{i+1}[y_j]$  with

$$ITE(u\_unch_{g_1, g_2}(S_i[Y]), UF_{g_1, g_2}^j(S_i[Y]), UF_{g_1}^j(S_i[Y])).$$

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

$$ITE(u\_unch_{g_1, g_2}(S_i[Y]), UF_{g_1, g_2}^j(S_i[Y]), UF_{g_2}^j(S_i[Y])).$$

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  $S_i[Y]$ , 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  $g_1, g_2$ .

<sup>3</sup> An obvious optimization will be 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.

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

$$\begin{aligned} \text{sum}_{p_1}(x) &= \{(x < 0, -1), (x \geq 2, x)\} \\ \text{sum}_{p_2}(x) &= \{(x < 0, -1), (x > 4, x)\} \end{aligned}$$

and their difference summary be  $\Delta_{p_1, p_2} = (\text{false}, x < 2 \vee x > 4)$ . 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:

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

## 5.2 Refinement

Using the described abstraction, the computed  $R_\pi, T_\pi$  may contain symbols of uninterpreted functions, and therefore so could  $\text{diffCond} = r_1 \wedge r_2 \wedge t_1 \neq t_2$  and  $\text{eqCond} = r_1 \wedge r_2 \wedge t_1 = t_2$  (lines 5, 10 in Algorithm CONSTPROCDIFFSUM). As a result  $u\_ch_{p_1, p_2}$  and  $u\_unch_{p_1, p_2}$  may include conditions that are *spurious*, that is, conditions 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  $\text{diffCond}$  to  $u\_ch_{p_1, p_2}$  or  $\text{eqCond}$  to  $u\_unch_{p_1, p_2}$ , we need to check whether it is *spurious*. To reveal 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.

```

int p1(int x) {
    if (x == 5)
        if (abs(x) == 0)
            return 0;
    return x;
}

int p2(int x) {
    if (x == 5)
        if (abs(x) == 0)
            return 1;
    return x;
}

int abs(int x) {
    if (x >= 1)
        return x;
    else
        return -x;
}

```

Figure 3: Procedure versions in need of refinement

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

We use the technique introduced in [3]: Let  $\varphi$  be a formula we wish to add to either  $u\_ch_{p_1,p_2}$  or  $u\_unch_{p_1,p_2}$  ( $\varphi \in \{\text{diffCond}, \text{eqCond}\}$ ), such that  $\varphi$  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  $i_k$  on the single path from  $p_k$ , described by  $\varphi$ . For every  $k \in \{1, 2\}$  do:

1. Apply an SMT solver to find a satisfying assignment  $\sigma$  to

$$\varphi \wedge \neg \left( \bigvee_{(r,t) \in \text{sum}_{g_k}} r(T_{\pi}^i i_k - 1[Y_k]) \right).$$

That is, check whether there exists an input state that traverses an unexplored path in  $g_k$  when running the path in  $p_k$ .

2. Use  $\sigma$  to find the values of inputs to  $g_k$ . Run  $g_k$  on those (concrete) inputs. Let  $\pi$  be the path (sequence of labels) traversed when running  $g_k$ .
3. Apply symbolic execution on  $\pi$  and compute its path summary  $(R_{\pi}, T_{\pi})$ .
4. Add  $(R_{\pi}, T_{\pi})$  to  $\text{sum}_{g_k}$ ; replace it in  $\varphi$  and check for satisfiability again.

As a result, we either find a real difference or similarity, or eliminate all the spurious path difference summaries that involve the explored path 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 Figure 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  $p_2$ :

$$\begin{aligned} \text{diffCond} &= [x = 5 \wedge \text{ITE}(\text{true}, UF_{\text{abs}_1, \text{abs}_2}(x), UF_{\text{abs}_1}(x)) = 0 \wedge \\ &\quad x = 5 \wedge \text{ITE}(\text{true}, UF_{\text{abs}_1, \text{abs}_2}(x), UF_{\text{abs}_2}(x)) = 0 \wedge 0 \neq 1] \\ &\equiv [x = 5 \wedge UF_{\text{abs}_1, \text{abs}_2}(x) = 0] \end{aligned}$$

Next we get the assignment  $x = 5$  from the SMT solver, and perform symbolic execution, updating the summary for  $\text{abs}$ :  $(x \geq 1, x)$ . Now *diffCond* is:

$$\begin{aligned} &[x = 5 \wedge \text{ITE}(x \geq 1, x, \text{ITE}(\text{true}, UF_{\text{abs}_1, \text{abs}_2}(x), UF_{\text{abs}_1}(x))) = 0 \wedge \\ &\quad x = 5 \wedge \text{ITE}(x \geq 1, x, \text{ITE}(\text{true}, UF_{\text{abs}_1, \text{abs}_2}(x), UF_{\text{abs}_2}(x))) = 0 \wedge 0 \neq 1] \\ &\equiv [x = 5 \wedge \text{ITE}(x \geq 1, x, UF_{\text{abs}_1, \text{abs}_2}(x)) = 0] \end{aligned}$$

which is now unsatisfiable. We thus managed to eliminate a spurious difference without computing the full summary of  $\text{abs}$ .

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 section 7 we describe the results on our benchmarks in two extreme modes: running refinement always immediately when needed (MDDIFFREF), and always delaying the refinement (MDDIFF).



## 6 Related Work

A formal definition of equivalence between programs is given in [11]. We extend these definitions to obtain a finer-grained characterization of the differences.

We extend the path-wise symbolic summaries and deltas given in [22], and show how to use them in modular symbolic execution, while abstracting unknown parts.

The SYMDIFF [18] tool and the Regression Verification Tool (RVT) [12] 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. There was an attempt to characterize differences with SYMDIFF [14], a work that has similar ideas to ours. Yet it only applies to programs with no loops or recursions, while our tool is capable of dealing soundly with loops, and as our experiments show, often is able to produce full summaries for programs with unbounded loops. Moreover, [14] lacks summaries of equivalent procedures that are important as shown in Figure 3. 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 Section 7. Both tools are, however, capable of proving equivalence between programs (using e.g. invariants, proof rules) we cannot handle. Our techniques can be seen as an orthogonal improvement.

Under-constrained symbolic execution, meaning symbolic execution of a procedure that is not the entry point of the program is presented in [24,25], where it is used to improve scalability while using the old version as a golden model. The algorithm presented in [24,25] 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 [24], 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.

Our approach is similar to the compositional symbolic execution presented in [3,10], that is applied to single programs. However, the analysis in [3,10] 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.

TODO: Add [4]

## 7 Experimental Results

We implemented the algorithm presented in section 4 with the abstractions from Section 5 on top of the CProver framework, which also forms the foundation of the verification tools CBMC [6], SATABS [7], IMPACT [19] and WOLVERINE [17]. 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 used two versions of our tool – without refinement (MD-DIFF for Modular Demand-driven Difference), and with refinement (MD-DIFFREF), both with unwinding limit of 5.

**SymDiff and RVT:** We compared our results to SYMDIFF and RVT. For SYMDIFF, we used the *smack* [2] 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.

### 7.1 Benchmarks and Analysis of the Results

Benchmark	MDDIFF	MDDIFFREF	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

Benchmark	MDDIFF	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

(a) Semantically equivalent.

(b) Semantically different

Table 1: Experimental results. Numbers are time in sec, F indicates a failure to prove equivalence in (a), and that the main procedure summary was not full in (b).

We analysed 28 C language benchmarks, 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. 16 benchmarks were semantically equivalent (presented in Table 1a), while some benchmarks had semantically different procedures in them.

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 its 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.

We now explain in detail the benefit of our method on specific benchmarks. The *LoopUnrch* benchmarks illustrate the advantages of summaries. Our tool analysed *foo1* and *foo2* from Figure 4c, found a condition under which those procedures are different (for example inputs -1,1), and a condition under which they are equivalent ( $a \geq 0$ ). In all the versions of this benchmark *foo1* and *foo2* were called with positive (increasing) values of *a* (and *b*), and as a result we were able to prove equivalence efficiently in all versions, both with and without refinement.

The *LoopMult* benchmarks illustrate the advantages of refinement. Our tool analysed *foo1* and *foo2* from Figure 4a, found a condition under which those procedures are different (for example inputs 1,-1), and a condition under which they are equivalent. We also summarised 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 Figure 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 Figure 4b). Thus, MD-DIFF is not able to prove equivalence (the summary of *foo1/2* does not cover the needed paths), while MD-DIFFREF analyses the additional needed paths (where  $5 \leq a < 7 \wedge b = 5$ ), and is able to prove equivalence. As the number of the *LoopMult* benchmark increases, the length of the paths needed and their number increases, and the analysis takes more time, accordingly, but only necessary paths are explored.

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

## 8 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.

<pre> int foo1(int a, int b) {     int c=0;     for (int i=1;i&lt;=b;++i)         c+=a;     return c; }  int foo2(int a, int b) {     int c=0;     for (int i=1;i&lt;=a;++i)         c+=b;     return c; } </pre>	<pre> int main(int x, char*argv[]) {     //LoopMult2     return foo(2,2); }  int main(int x, char*argv[]) {     //LoopMult5     if (x&gt;=5 &amp;&amp; x&lt;7)         return foo(x,5);     return 0; } </pre>	<pre> int foo1(int a, int b) {     int c=0;     if (a&lt;0) {         for (int i=1;i&lt;=b;++i)             c+=a;     }     return c; }  int foo2(int a, int b) {     int c=0;     if (a&lt;0) {         for (int i=1;i&lt;=a;++i)             c+=b;     }     return c; } </pre>
<p>(a) procedures foo1 and foo2 in LoopMult benchmarks</p>	<p>(b) main functions of LoopMult2 and LoopMult5</p>	<p>(c) procedures foo1 and foo2 in LoopUnrch benchmarks</p>

Figure 4: LoopMult and LoopUnrch benchmarks

## References

1. MD-Diff benchmarks; <https://github.com/AnnaTrost/MD-Diff>
2. SMACK Software Verifier And Verification Toolchain; <https://github.com/smackers/smack>
3. Anand, S., Godefroid, P., Tillmann, N.: Demand-driven compositional symbolic execution. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 367–381. Springer (2008)
4. Backes, J.D., Person, S., Rungta, N., Tkachuk, O.: Regression verification using impact summaries. In: Model Checking Software (SPIN). LNCS, vol. 7976, pp. 99–116. Springer (2013)
5. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Communications of the ACM 56(2), 82–90 (2013)
6. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 168–176. Springer (2004)
7. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 570–574. Springer (2005)
8. Francez, N.: Program verification. Addison-Wesley Longman (1992)
9. Gao, D., Reiter, M.K., Song, D.: Binhunt: Automatically finding semantic differences in binary programs. In: Information and Communications Security, pp. 238–255. Springer (2008)
10. Godefroid, P.: Compositional dynamic test generation. In: ACM SigPlan Notices. vol. 42, pp. 47–54. ACM (2007)
11. Godlin, B., Strichman, O.: Inference rules for proving the equivalence of recursive procedures. Acta Informatica 45(6), 403–439 (2008)
12. Godlin, B., Strichman, O.: Regression verification. In: Proceedings of the 46th Annual Design Automation Conference. pp. 466–471. ACM (2009)

13. Godlin, B., Strichman, O.: Regression verification: proving the equivalence of similar programs. *Software Testing, Verification and Reliability* 23(3), 241–258 (2013)
14. Kawaguchi, M., Lahiri, S.K., Rebelo, H.: Conditional equivalence. Tech. Rep. MSR-TR-2010-119 (2010)
15. King, J.C.: Symbolic execution and program testing. *Communications of the ACM* 19(7), 385–394 (1976)
16. Kroening, D., Strichman, O.: Equality logic and uninterpreted functions. In: *Decision Procedures*, pp. 59–80. Springer (2008)
17. Kroening, D., Weissenbacher, G.: Interpolation-based software verification with WOLVERINE. In: *International Conference on Computer Aided Verification*. pp. 573–578. Springer (2011)
18. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: Symdiff: A language-agnostic semantic diff tool for imperative programs. In: *International Conference on Computer Aided Verification*. pp. 712–717. Springer (2012)
19. McMillan, K.L.: Lazy abstraction with interpolants. In: *International Conference on Computer Aided Verification*. pp. 123–136. Springer (2006)
20. Partush, N., Yahav, E.: Abstract semantic differencing for numerical programs. In: *Static Analysis*, pp. 238–258. Springer (2013)
21. Partush, N., Yahav, E.: Abstract semantic differencing via speculative correlation. In: *ACM SIGPLAN Notices*. vol. 49, pp. 811–828. ACM (2014)
22. Person, S., Dwyer, M.B., Elbaum, S., Pasareanu, C.S.: Differential symbolic execution. In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. pp. 226–237. ACM (2008)
23. Person, S., Yang, G., Rungta, N., Khurshid, S.: Directed incremental symbolic execution. In: *ACM SIGPLAN Notices*. vol. 46, pp. 504–515. ACM (2011)
24. Ramos, D.A., Engler, D.: Under-constrained symbolic execution: correctness checking for real code. In: *24th USENIX Security Symposium (USENIX Security 15)*. pp. 49–64 (2015)
25. Ramos, D.A., Engler, D.R.: Practical, low-effort equivalence verification of real code. In: *International Conference on Computer Aided Verification*. pp. 669–685. Springer (2011)
26. Wong, W.E., Horgan, J.R., London, S., Agrawal, H.: A study of effective regression testing in practice. In: *Software Reliability Engineering, 1997. Proceedings., The Eighth International Symposium on*. pp. 264–274. IEEE (1997)

## Appendix

### 0.1 Basic Definitions

#### Definition 1. CALL GRAPH

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

$p_1$  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  $\text{main}_P$ .

#### Definition 2. PATH

Given a  $\text{CFG}_p$  of procedure  $p$ , a path  $\pi = l_1, l_2, \dots$  is a sequence of nodes (finite or infinite) in the graph  $\text{CFG}_p$ , such that:

1. For all  $i$  there exists an edge from  $l_i$  to  $l_{i+1}$  in  $\text{CFG}_p$ .
2.  $l_1$  is the entry node of  $p$ .

Path  $\pi$  is **maximal** if it is either infinite or it is finite and ends in the exit node of  $p$ .