Modular Demand-Driven Analysis of Semantic Difference for Program Versions

Anna Trostanetski

Modular Demand-Driven Analysis of Semantic Difference for Program Versions

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Anna Trostanetski

Submitted to the Senate
of the Technion — Israel Institute of Technology
Tamuz 5777 Haifa July 2017

This research was carried out under the supervision of Prof. Orna Grumberg, in the Faculty of Computer Science.

Some of the results in this dissertation have been published in a paper by the author and collaborators in a conference (SAS 2017) during the course of this masters, the most up-to-date version of which is:

> Anna Trostanetski, Orna Grumberg, and Daniel Kroening. Modular demand-driven analysis of semantic difference for program versions. In *International Static Analysis Symposium*. Springer, NY, USA, 2017.

Acknowledgements

I would like to thank Orna Grumberg, who was an amazing advisor, a mentor, an inspiration and a friend to me during the course of my studies.
I would also like to thank Daniel Kroening, Ofer Strichman, Ofer Guthmann, and all others that helped me along the way.

The generous financial help Of The Technion, and Randy L. and Melvin R. Berlin Fellowship in the Cyber Security Research are hereby gratefully acknowledged.

Contents

List of Figures

Abstract

Programs are often built in stages, a new (patched) program version is built on top of an old one. If we could understand the semantic difference between two consecutive program versions, it would be very beneficial for the fast development process of correct programs. We can use the correctness of the old (and checked) version to infer the correctness of the new version. Code reviews, security vulnerability checks, and new feature verification would become easier if the reviewer were to understand the semantic differences between both versions. In general this problem is undecidable, yet we devise an algorithm for computing over– and under–approximations of the semantic (input–output) differences between program versions. We aim at providing precise enough abstractions for real code, and allowing guidance by the user to reach good results that match their needs. Since this information is used during the development process, it may be sufficient (and possibly preferable) to give results for intermediate procedures, instead of the entire program. We provide a mechanism for guiding the analysis towards interesting procedures, and the precision of the approximation is constantly improved by our anytime algorithm.

While our algorithm can work for very different versions of code, it will work better on syntactically similar versions. 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 are *abstracted* using uninterpreted functions, and may be *refined* on demand. We show how we can use common uninterpreted functions to use our knowledge of equivalence when no precise summery is available. Our algorithm works bottom-up from the locations of the syntactic changes, towards the main procedure. When the precision of the abstractions used is not sufficient, we run (top–down) refinement to create new summaries that are sufficiently precise. The refinement is guided by the context of the call we analyse.

We define *modular symbolic execution* and prove its connection to standard symbolic execution. We use modular symbolic execution to analyse each path in each procedure at most once, without re-analysing paths in called procedures.

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

Chapter 1

Introduction

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 [30].
- Revealing security vulnerabilities that were eliminated by the new version [11]. This information can be used to produce zero-day attacks.
- More generally, identifying and characterizing changes in the program's functionality [24].

1.1 Related Work

Semantic difference has been widely studied, and several techniques have been suggested.

Abstract interpretation is applied to characterize differences or prove equivalence in [23, 24].

In [14, 15] different notions of equivalence are defined, proof rules for showing the equivalence between recursive procedures are given. These ideas are extended to less similar procedures in [29].

Symbolic execution is used to find differences between programs in [5, 25, 26], and syntactic similarity is used to direct symbolic execution to the "interesting" paths. In [6], both versions are run symbolically together, one "shadowing" the other. This allows using dynamic values to guide the execution towards changed behavior.

Symbolic execution is also used in [28], where the differences found are not over–approximating or under–approximating the real ones; yet is effective for finding new bugs using the differences between memory access of individual procedures between program versions.

Figure 1.1: 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.

1.2 Our Approach

In this dissertation we present a modular and demand-driven algorithm for finding semantic difference between two closely-related, syntactically similar imperative programs.

We assume that the programs are sequential, deterministic, and we do not handle pointers and aliasing.

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 1.1). 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.

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.

1.2.1 Our method 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, 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_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 describing transformation from 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 state. 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, 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 $C(p_1, p_2)$ 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.

1.2.2 Main Contributions

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.
- We develop a new notion of modular symbolic execution.

Chapter 2

Preliminaries

2.1 Procedures

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

Definition 2.1.1. Call Graph
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 2.1.2. Control Flow Graph (CFG)
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: $\bar{x} = \bar{e}$, where $\bar{x} = x_1, \ldots, x_n$ is a list of variable in V_p and $\bar{e} = e_1, \ldots, e_n$ a list of expression over V_p. All expressions e_i are computed before being assigned to the variables x_i simultaneously. 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 2.1.3. Path
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.

[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.

```
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   }
```

Figure 2.1: Examples of procedure versions

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 2.1.4. Reachability Condition, State Transformation
Let π be a finite path in procedure p.

- 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 $1 \leq i \leq |V_p^v|$, $f_i = f'_i$.

Example 2.1.5. Consider procedure p1 in Figure 2.1. 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 Figure 2.1, the path $(2, 6, 8, 9)$ is not feasible.

2.2 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 $\bar{x} = \bar{e}$: $R_\pi^{i+1}(V_p) := R_\pi^i(V_p)$, $\forall x_l \in vars(x).\ T_\pi^{i+1}(V_p)[x_l] := e_l[V_p \leftarrow T_\pi^i(V_p)]$ and $\forall y \notin vars(x),\ T_\pi^{i+1}(V_p)[y] := T_\pi^i(V_p)[y]$
- 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

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

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

$$R_\pi(V_p^v) = R_\pi^{n+1}(V_p)$$
$$T_\pi(V_p^v) = T_\pi^{n+1}(V_p) \downarrow_{V_p^v}$$ [3]

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

[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 $T_\pi^{n+1}(V_p) \downarrow_{V_p^v}$.

Definition 2.2.1. Path Summary
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 2.2.2. Procedure Summary
A **Procedure Summary** (also known as a symbolic summary in [25]), for a procedure p, is a set of path summaries

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

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 2.2.3. Uncovered part of a Procedure Summary
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 2.2.4. We return to $p1$ from Figure 2.1. 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 $x \geq 0 \wedge x < 2$.

2.3 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 $V_{p_1}^v$ 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 2.3.1. State-Equivalences
Let σ_p^v be a visible state for p_1 and p_2.

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

Chapter 3

Our Contribution

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

Modular symbolic execution is defined as symbolic execution for assignment and test instructions (see Section 2.2). For procedure call instruction $g(Y)$ (where $Y \subseteq V_p$) it is defined as follows. For given $R^i_\pi(V_p)$ and $T^i_\pi(V_p)$:

$$R^{i+1}_\pi = R^i_\pi \wedge (\bigvee_{(r,t)\in sum_g} r(T^i_\pi[Y])) \tag{3.1}$$

$$\forall x \notin Y.\ T^{i+1}_\pi[x] = T^i_\pi[x] \tag{3.2}$$

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

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.

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

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 Section 3.5.1 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.

We prove in Section 3.2 the connection between modular symbolic execution and standard 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:

- 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:
  - $R_{\pi} \leftrightarrow \bigvee_{i=1}^{n} R_{\pi_i^{in}}$
  - $\forall i \in [n].\ R_{\pi_i^{in}} \rightarrow T_{\pi_i^{in}} = T_{\pi}$

3.2 Symbolic Execution vs. Modular Symbolic Execution

We formally define and prove the relationship between standard symbolic execution, defined on the program obtained by in-lining procedures, and modular symbolic execution, defined on the original program. For simplicity we assume here that we have a single procedure q that calls procedures $p_1, \ldots, p_k$ from locations $l_1, \ldots, l_k$ with inputs $Y_1, \ldots, Y_k$, respectively. First we assume procedures $p_1, \ldots, p_k$ contain no procedure calls. We deal with further sub-calls in Subsection 3.2.3. We further assume we are given the summaries $sum_{p_1}, \ldots, sum_{p_k}$, and that different procedures do not have common variable names.

We start by defining an in-lined CFG to which the standard symbolic execution will be applied.

Definition 3.2.1. INLINED CFG
Let q be a procedure, represented by CFG_q, that calls procedures $p_1, \ldots, p_k$ from

nodes $l_1, \ldots, l_k$, respectively. We obtain the in-lined version CFG_q^{in} from CFG_q, by performing the following changes for every $i \in [k]$:

- Changes in nodes:
  1. Remove node l_i $(l_i : p_i(Y_i))$.
  2. Add assignment node $l_i^{\text{pre}} : V_{p_i}^v := Y_i$.
  3. Add assignment node $l_i^{\text{post}} : Y_i := V_{p_i}^v$.
  4. Add all the nodes from CFG_{p_i}.
- Changes in edges:
  1. Remove edge (l, l_i), add edge (l, l_i^{pre}).
  2. Remove edge (l_i, l), add edge (l_i^{post}, l).
  3. Add edge $(l_i^{\text{pre}}, l_i^{\text{entry}})$, where l_i^{entry} is the entry node of CFG_{p_i}.
  4. Add edge $(l_i^{\text{exit}}, l_i^{\text{post}})$, where l_i^{exit} is the exit node of CFG_{p_i}.
  5. Add all edges from CFG_{p_i}.

The hidden variables of CFG_q^{in} are $(V_q^h)^{in} \triangleq V_q^h \cup \bigcup_{i=1}^k V_{p_i}$ (disjoint sets according to our assumption). The visible variables of CFG_q^{in} are the visible variables of q, $(V_q^v)^{in} \triangleq V_q^v$. Note that indeed hidden variables are not used before they are assigned in CFG_q^{in}, since we assign each visible variable of p_i at node l_i^{pre}. Therefore again we conclude that, when R_π, T_π computed with symbolic execution for some π of length n in CFG_q^{in}, $(V_q^h)^{in}$ will not appear in $R_\pi^{n+1}((V_q)^{in})$ and $T_\pi^{n+1}((V_q)^{in}) \downarrow_{(V_q^v)^{in}}$.

Definition 3.2.2. Legal Path
A finite path $\pi = l_1^{in}, \ldots, l_m^{in}$ in CFG_q^{in} is called **legal** if:

- $l_m^{in} \in CFG_q$, or
- There exists $i \in [k]$ such that $l_m^{in} = l_i^{\text{post}}$

A legal path can not end inside a called procedure. Thus, by the definition of CFG_q^{in}, for every legal path $\pi = l_1^{in}, \ldots, l_m^{in}$, and for every node $l_j^{in} = l_i^{\text{pre}}$, there exists $r > j$ such that:

- $l_r^{in} = l_i^{\text{post}}$, and
- for every $j < s < r$, l_s^{in} is a node from CFG_{p_i}.

We can now decompose each legal path to its original paths.

Definition 3.2.3. Projected Path
Let π be a legal path in CFG_q^{in},

- Its **p_i-projected path**, denoted $\pi \downarrow_{p_i}$, is the interval of nodes between l_i^{start} and the first l_i^{exit} following it, if such l_i^{start} exists, or empty otherwise.[2]
- Its **q-projected path**, denoted $\pi \downarrow_q$, is the path obtained from π by the following operations:
  - Every node l_i^{pre} is replaced by l_i, the original calling node to p_i.
  - Every node not in CFG_q is removed (including nodes from called procedures, and l_i^{post} nodes).

Observation 3.2.4. A p_i-projected path $\pi \downarrow_{p_i}$ is a path in CFG_{p_i}.

Observation 3.2.5. A q-projected path $\pi \downarrow_q$ is a path in CFG_q.

Next to prove our claims we need to make sure that paths are covered by the procedure summaries that are used to replace their procedures.

Definition 3.2.6. Covered path

- We say that a path π in CFG_p is **covered by** sum_p if $(R_\pi, T_\pi) \in sum_p$.
- We say that a path π in CFG_q^{in} is **calling-covered** if for every $i \in [k]$, $\pi \downarrow_{p_i}$ is covered by sum_{p_i}

Lemma 3.2.7. *Let $\pi^1 = l_1^1, \ldots, l_n^1$ and $\pi^2 = l_1^2, \ldots, l_m^2$ be two paths in CFG_p with no procedure calls, such that there exists an edge (l_n^1, l_1^2). Then the path $\pi^1 \cdot \pi^2 = l_1^1, \ldots, l_n^1, l_1^2, \ldots, l_m^2$ is a path in CFG_p and:*

1. $R_{\pi^1 \cdot \pi^2} = R_{\pi^1} \wedge R_{\pi^2}(T_{\pi^1})$
2. $T_{\pi^1 \cdot \pi^2} = T_{\pi^2}(T_{\pi^1})$

Proof. We prove the lemma by induction on m, the length of π^2:
Base: If $m = 0$ then π^2 is empty, $(\pi^1 \cdot \pi^2) = \pi^1$, and:

1. $R_{\pi^1 \cdot \pi^2} = R_{\pi^1} = R_{\pi^1} \wedge \mathit{true} = R_{\pi^1} \wedge R_{\pi^2}(T_{\pi^1})$ since $R_{\pi^2} = \mathit{true}$.
2. $T_{\pi^1 \cdot \pi^2} = T_{\pi^1} = T_{\pi^2}(T_{\pi^1})$, since T_{π^2} is the identity function.

Step: Assume correctness for $\pi' = l_1^2, \ldots, l_{m-1}^2$, and consider the last node, l_m^2.

- If node l_m^2 is an assignment node $\bar{x} = \bar{e}$, then:

  1. $$\begin{aligned} R_{\pi^1 \cdot \pi^2} &= R_{(\pi^1 \cdot \pi'), l_m^2} = R^{n+m+1}_{(\pi^1 \cdot \pi'), l_m^2} =^{(a)} R^{n+m}_{(\pi^1 \cdot \pi'), l_m^2} = R_{\pi^1 \cdot \pi'} =^{(b)} \\ &= R_{\pi^1} \wedge R_{\pi'}(T_{\pi^1}) = R_{\pi^1} \wedge R^m_{\pi^2}(T_{\pi^1}) =^{(a)} R_{\pi^1} \wedge R^{m+1}_{\pi^2}(T_{\pi^1}) = \\ &= R_{\pi^1} \wedge R_{\pi^2}(T_{\pi^1}) \end{aligned}$$

     where:

[2] For simplicity we assume that on every path each procedure appears at most once, which is not necessarily true in the presence of loops. We can easily deal with it by indexing called intervals by occurrence as well as procedure.

(a) Definition of R for assignment.

(b) Induction hypothesis for π'.

2. $\forall x_l \in vars(\bar{x}).\ T_{\pi^1\cdot\pi^2}[x_l] = T_{(\pi^1\cdot\pi'),l_m^2}[x_l] = T^{n+m+1}_{(\pi^1\cdot\pi'),l_m^2}[x_l] =^{(a)}$
$= e_l(T^{n+m}_{(\pi^1\cdot\pi'),l_m^2}[V_p]) = e_l(T_{\pi^1\cdot\pi'}[V_p]) =^{(b)}$
$= e_l(T_{\pi'}(T_{\pi^1})[V_p]) = e_l(T^m_{\pi^2}(T_{\pi^1})[V_p]) =^{(a)}$
$= T^{m+1}_{\pi^2}(T_{\pi^1})[x_l] = T_{\pi^2}(T_{\pi^1})[x_l]$
$\forall y \in V_p \setminus vars(\bar{x}).\ T_{\pi^1\cdot\pi^2}[y] = T_{(\pi^1\cdot\pi'),l_m^2}[y] = T^{n+m+1}_{(\pi^1\cdot\pi'),l_m^2}[y] =^{(a)}$
$= T^{n+m}_{(\pi^1\cdot\pi'),l_m^2}[y] = T_{\pi^1\cdot\pi'}[y] =^{(b)} T_{\pi'}(T_{\pi^1})[y] =$
$= T^m_{\pi^2}(T_{\pi^1})[y] =^{(a)} T^{m+1}_{\pi^2}(T_{\pi^1})[y] = T_{\pi^2}(T_{\pi^1})[y]$

where:

(a) Definition of T for assignment.

(b) Induction hypothesis for π'.

- If node l_m^2 is a test node $B(V_p)$, then:

1. $R_{\pi^1\cdot\pi^2} = R_{(\pi^1\cdot\pi'),l_m^2} = R^{n+m+1}_{(\pi^1\cdot\pi'),l_m^2} =^{(a)} R^{n+m}_{(\pi^1\cdot\pi'),l_m^2} \wedge \tilde{B}(T^{n+m}_{(\pi^1\cdot\pi'),l_m^2}[V_p]) =$
$= R_{\pi^1\cdot\pi'} \wedge \tilde{B}(T_{\pi^1\cdot\pi'}[V_p]) =^{(b)} R_{\pi^1} \wedge R_{\pi'}(T_{\pi^1}) \wedge \tilde{B}(T_{\pi'}(T_{\pi^1})[V_p]) =$
$= R_{\pi^1} \wedge R^m_{\pi^2}(T_{\pi^1}) \wedge \tilde{B}(T^m_{\pi^2}(T_{\pi^1})[V_p]) =^{(a)} R_{\pi^1} \wedge R^{m+1}_{\pi^2}(T_{\pi^1}) =$
$= R_{\pi^1} \wedge R_{\pi^2}(T_{\pi^1})$

where $\tilde{B}$ is either B or $\neg B$, according to the edge marking on π^2, and:

(a) Definition of R for test.

(b) Induction hypothesis for π'.

2. $T_{\pi^1\cdot\pi^2} = T_{(\pi^1\cdot\pi'),l_m^2} = T^{n+m+1}_{(\pi^1\cdot\pi'),l_m^2} =^{(a)} T^{n+m}_{(\pi^1\cdot\pi'),l_m^2} = T_{\pi^1\cdot\pi'} =^{(b)}$
$= T_{\pi'}(T_{\pi^1}) = T^m_{\pi^2}(T_{\pi^1}) =^{(a)} T^{m+1}_{\pi^2}(T_{\pi^1}) = T_{\pi^2}(T_{\pi^1})$

where:

(a) Definition of T for test.

(b) Induction hypothesis for π'. □

The theorems below, showing the connection between symbolic execution and modular symbolic execution, rely on the corollary below that summarizes the effect of in-lining and symbolically executing a path.

Let π be a legal path in CFG_q^{in}, we assume that R_π, T_π were computed by standard symbolic execution, and that $R_{\pi\downarrow_q}, T_{\pi\downarrow_q}$ where computed by modular symbolic execution.

Corollary 3.1. *Let $\pi = l_1^{in}, \ldots, l_m^{in}$ be a legal path in CFG_q^{in}, if $l_j^{in} = l_i^{pre}$ and $l_n^{in} = l_i^{post}$, where $Y_i = \{y_1, \ldots, y_r\}$ and $V_{p_i}^v = \{v_1, \ldots, v_r\}$ then:*

1. $R_\pi^{n+1} = R_\pi^j \wedge R_{\pi\downarrow_{p_i}}(T_\pi^j[Y_i])$
2. *For every $y_l \in Y_i$, $T_\pi^{n+1}[y_l] = T_{\pi\downarrow_{p_i}}(T_\pi^j[Y_i])[v_l]$*

3. For every $x \in V_q \setminus Y_i$, $T_\pi^{n+1}[x] = T_\pi^j[x]$

Proof. We get the corollary from the lemma, if we mark $\pi' = l_1^{in}, \ldots, l_{j-1}^{in}$:

1. $$\begin{aligned} R_\pi^{n+1} = R_\pi = R_{\pi' \cdot l_i^{\text{pre}} \cdot \pi\downarrow_{p_i} \cdot l_i^{\text{post}}} &=^{(a)} R_{\pi'} \wedge R_{l_i^{\text{pre}} \cdot \pi\downarrow_{p_i} \cdot l_i^{\text{post}}}(T_{\pi'}) =^{(b)} \\ &= R_{\pi'} \wedge R_{l_i^{\text{pre}}}(T_{\pi'}) \wedge R_{\pi\downarrow_{p_i} \cdot l_i^{\text{post}}}(T_{l_i^{\text{pre}}}(T_{\pi'})) =^{(c)} \\ &= R_{\pi'} \wedge R_{\pi\downarrow_{p_i} \cdot l_i^{\text{post}}}(T_{l_i^{\text{pre}}}(T_{\pi'})) =^{(d)} \\ &= R_{\pi'} \wedge R_{\pi\downarrow_{p_i}}(T_{l_i^{\text{pre}}}(T_{\pi'})) \wedge R_{l_i^{\text{post}}}(T_{\pi\downarrow_{p_i}}(T_{l_i^{\text{pre}}}(T_{\pi'}))) =^{(e)} \\ &= R_{\pi'} \wedge R_{\pi\downarrow_{p_i}}(T_{l_i^{\text{pre}}}(T_{\pi'})) =^{(f)} R_{\pi'} \wedge R_{\pi\downarrow_{p_i}}(T_{l_i^{\text{pre}}}(T_{\pi'})[V_{p_i}^v]) =^{(g)} \\ &= R_{\pi'} \wedge R_{\pi\downarrow_{p_i}}(T_{\pi'}[Y_i]) = R_\pi^j \wedge R_{\pi\downarrow_{p_i}}(T_\pi^j[Y_i]) \end{aligned}$$

   where:

   (a) Lemma 3.2.7 for $\pi^1 = \pi'$, $\pi^2 = l_i^{\text{pre}} \cdot \pi \downarrow_{p_i} \cdot l_i^{\text{post}}$.

   (b) Lemma 3.2.7 for $\pi^1 = l_i^{\text{pre}}$, $\pi^2 = \pi \downarrow_{p_i} \cdot l_i^{\text{post}}$.

   (c) $R_{l_i^{\text{pre}}} = \mathit{true}$.

   (d) Lemma 3.2.7 for $\pi^1 = \pi \downarrow_{p_i}$, $\pi^2 = l_i^{\text{post}}$.

   (e) $R_{l_i^{\text{post}}} = \mathit{true}$.

   (f) $R_{\pi\downarrow_{p_i}}$ is defined over $V_{p_i}^v$.

   (g) $l_i^{\text{pre}} :\ V_{p_i}^v = Y_i$.

2. Let $y_l \in Y_i$, then:

$$\begin{aligned} T_\pi^{n+1}[y_l] = T_{\pi' \cdot l_i^{\text{pre}} \cdot \pi\downarrow_{p_i} \cdot l_i^{\text{post}}}[y_l] &=^{(a)} T_{l_i^{\text{post}}}(T_{\pi' \cdot l_i^{\text{pre}} \cdot \pi\downarrow_{p_i}})[y_l] =^{(b)} \\ &= T_{\pi' \cdot l_i^{\text{pre}} \cdot \pi\downarrow_{p_i}}[v_l] =^{(c)} T_{\pi\downarrow_{p_i}}(T_{\pi' \cdot l_i^{\text{pre}}})[v_l] =^{(d)} T_{\pi\downarrow_{p_i}}(T_{l_i^{\text{pre}}}(T_{\pi'}))[v_l] =^{(e)} \\ &= T_{\pi\downarrow_{p_i}}(T_{l_i^{\text{pre}}}(T_{\pi'})[V_{p_i}^v])[v_l] =^{(f)} T_{\pi\downarrow_{p_i}}(T_{\pi'}[Y_i])[v_l] = T_{\pi\downarrow_{p_i}}(T_\pi^j[Y_i])[v_l] \end{aligned}$$

   where:

   (a) Lemma 3.2.7 for $\pi^1 = \pi' \cdot l_i^{\text{pre}} \cdot \pi \downarrow_{p_i}$ and $\pi^2 = l_i^{\text{post}}$.

   (b) $l_i^{\text{post}} :\ Y_i = V_{p_i}^v$ and therefore $T_{l_i^{\text{post}}}(f)[y_l] = f[v_l]$.

   (c) Lemma 3.2.7 for $\pi^1 = \pi' \cdot l_i^{\text{pre}}$ and $\pi^2 = \pi \downarrow_{p_i}$.

   (d) Lemma 1 for $\pi^1 = \pi'$ and $\pi^2 = l_i^{\text{pre}}$.

   (e) $T_{\pi\downarrow_{p_i}}$ is defined over $V_{p_i}^v$.

   (f) $l_i^{\text{pre}} :\ V_{p_i}^v = Y_i$.

3. Let $x \in V_q \setminus Y_i$, then:

$$\begin{aligned} T_\pi^{n+1}[x] = T_{\pi' \cdot l_i^{\text{pre}} \cdot \pi\downarrow_{p_i} \cdot l_i^{\text{post}}}[x] &=^{(a)} T_{l_i^{\text{post}}}(T_{\pi' \cdot l_i^{\text{pre}} \cdot \pi\downarrow_{p_i}})[x] =^{(b)} \\ &= T_{\pi' \cdot l_i^{\text{pre}} \cdot \pi\downarrow_{p_i}}[x] =^{(c)} T_{\pi\downarrow_{p_i}}(T_{\pi' \cdot l_i^{\text{pre}}})[x] =^{(d)} T_{\pi' \cdot l_i^{\text{pre}}}[x] =^{(e)} \\ &= T_{l_i^{\text{pre}}}(T_{\pi'})[x] =^{(f)} T_{\pi'}[x] = T_\pi^j[x] \end{aligned}$$

where:

(a) Lemma 3.2.7 for $\pi^1 = \pi' \cdot l_i^{\text{pre}} \cdot \pi \downarrow_{p_i}$ and $\pi^2 = l_i^{\text{post}}$.

(b) l_i^{post} : $Y_i = V_{p_i}^v$ and since $x \notin Y_i$ $T_{l_i^{\text{post}}}(f)[x] = f[x]$.

(c) Lemma 3.2.7 for $\pi^1 = \pi' \cdot l_i^{\text{pre}}$ and $\pi^2 = \pi \downarrow_{p_i}$.

(d) $x \in V_q$ and therefore according to our assumption that there are no common variable names between functions, $x \notin V_{p_i}$. $\pi \downarrow_{p_i}$ is a path in CFG_{p_i} and therefore does not change x.

(e) Lemma 3.2.7 for $\pi^1 = \pi'$ and $\pi^2 = l_i^{\text{pre}}$.

(f) l_i^{pre} : $V_{p_i}^v = Y_i$ and since $x \notin V_{p_i}^v$ $T_{l_i^{\text{pre}}}(f)[x] = f[x]$. □

3.2.1 Symbolic Execution ⊆ Modular Symbolic Execution

To show the relation between standard and modular symbolic execution we show first that every in-lined path π analysed using standard symbolic execution has a corresponding path (its projection), that when analysed with modular symbolic execution, contains the behaviors from π.

Theorem 3.2. *Let π be a legal, calling-covered path in CFG_q^{in}, its q-projected path $\pi \downarrow_q$ satisfies:*

1. $R_\pi(V_q^v) \rightarrow R_{\pi\downarrow_q}(V_q^v)$
2. $R_\pi(V_q^v) \rightarrow T_\pi(V_q^v) \downarrow_{V_q} = T_{\pi\downarrow_q}(V_q^v)$

Proof. We prove the theorem by induction on the length of legal paths π ($\pi = l_1^{in}, \ldots, l_m^{in}$). We denote the length of $\pi \downarrow_q$ by n.
Base: If $m = 0$ then $\pi, \pi \downarrow_q$ are empty and:

1. $R_\pi = R_\pi^1 = true \rightarrow R_{\pi\downarrow_q} = R_{\pi\downarrow_q}^1 = true$
2. $\forall x \in V_q$. $(T_\pi[x] = T_\pi^1[x] = x = T_{\pi\downarrow_q}^1[x] = T_{\pi\downarrow_q}[x])$, and therefore $R_\pi \rightarrow T_\pi \downarrow_{V_q} = T_{\pi\downarrow_q}$

Step: Assume correctness for all legal paths of length strictly smaller than m. We consider the last node, l_m^{in}:

- If node l_m^{in} is an assignment node $\bar{x} = \bar{e}$, then by definition $\pi' = l_1^{in}, \ldots l_{m-1}^{in}$ is legal, $\pi \downarrow_q = (\pi' \downarrow_q, l_m^{in})$, and:

  1. $R_\pi = R_\pi^{m+1} =^{(a)} R_\pi^m = R_{\pi'} \rightarrow^{(b)} R_{\pi'\downarrow_q} = R_{\pi\downarrow_q}^n =^{(a)} R_{\pi\downarrow_q}^{n+1} = R_{\pi\downarrow_q}$, where:
     (a) Definition of R for assignment.
     (b) Induction hypothesis for the legal path π'.

2. $R_\pi = R_\pi^{m+1} =^{(a)} R_\pi^m = R_{\pi'} \rightarrow^{(b)} (T_{\pi'} \downarrow_{V_q} = T_{\pi'\downarrow_q}) \rightarrow$
$(T_\pi^m \downarrow_{V_q} = T_{\pi\downarrow_q}^n) \rightarrow^{(c)} (T_\pi^{m+1} \downarrow_{V_q} = T_{\pi\downarrow_q}^{n+1})$,
where:
   (a) Definition of R for assignment.
   (b) Induction hypothesis for the legal path π'.
   (c) $\forall x_l \in vars(\bar{x})$. $T_\pi^{m+1}[x_l] := e_l(T_\pi^m[V_q]) = e_l(T_{\pi\downarrow_q}^n[V_q]) = T_{\pi\downarrow_q}^{n+1}[x_j]$
   $\forall y \in V_q \setminus vars(\bar{x})$, $T_\pi^{m+1}[y] := T_\pi^m[y] = T_{\pi\downarrow_q}^n[y] = T_{\pi\downarrow_q}^{n+1}[y]$.

- If node l_m^{in} is a test node $B(V_q)$, then by definition $\pi' = l_1^{in}, \ldots l_{m-1}^{in}$ is legal, $\pi \downarrow_q = (\pi' \downarrow_q, l_m^{in})$, and:
   1. $R_\pi = R_\pi^{m+1} =^{(a)} R_\pi^m \wedge \tilde{B}(T_\pi^m) = R_{\pi'} \wedge \tilde{B}(T_{\pi'}) \rightarrow^{(b)}$
   $(R_{\pi'\downarrow_q} \wedge \tilde{B}(T_{\pi'\downarrow_q})) = (R_{\pi\downarrow_q}^n \wedge \tilde{B}(T_{\pi\downarrow_q}^n)) =^{(a)} R_{\pi\downarrow_q}^{n+1} = R_{\pi\downarrow_q}$,
   where $\tilde{B}$ is either B or $\neg B$, according to the edge marking on π, and:
      (a) Definition of R for test.
      (b) Induction hypothesis for the legal path π'.
   2. $R_\pi = R_\pi^{m+1} \rightarrow^{(a)} R_\pi^m = R_{\pi'} \rightarrow^{(b)} (T_{\pi'} \downarrow_{V_q} = T_{\pi'\downarrow_q}) \rightarrow$
   $(T_\pi^m \downarrow_{V_q} = T_{\pi\downarrow_q}^n) \rightarrow^{(c)} (T_\pi^{m+1} \downarrow_{V_q} = T_{\pi\downarrow_q}^{n+1})$,
   where:
      (a) Definition of R for test.
      (b) Induction hypothesis for the legal path π'.
      (c) $T_\pi^{m+1} = T_\pi^m$ and $T_{\pi\downarrow_q}^{n+1} = T_{\pi\downarrow_q}^n$ (definition of T for test).

- If node l_m^{in} is a node l_i^{post} then there exists $j < n$ such that $l_j = l_i^{\text{pre}}$, by definition $\pi' = l_1^{in}, \ldots l_{j-1}^{in}$ is legal, $\pi \downarrow_q = (\pi' \downarrow_q, l_i)$, where l_i is the original call node to p_i from CFG_q, and:
   1. $R_\pi = R_\pi^{m+1} =^{(a)} (R_\pi^j \wedge R_{\pi\downarrow_{p_i}}(T_\pi^j[Y_i])) = (R_{\pi'} \wedge R_{\pi\downarrow_{p_i}}(T_{\pi'}[Y_i])) \rightarrow^{(b)}$
   $(R_{\pi'\downarrow_q} \wedge R_{\pi\downarrow_{p_i}}(T_{\pi'\downarrow_q}[Y_i])) = (R_{\pi\downarrow_q}^n \wedge R_{\pi\downarrow_{p_i}}(T_{\pi\downarrow_q}^n[Y_i])) \rightarrow^{(c)}$
   $(R_{\pi\downarrow_q}^n \wedge (\bigvee_{(r,t)\in sum_{p_i}} r(T_{\pi\downarrow_q}^n[Y_i]))) =^{(d)} R_{\pi\downarrow_q}^{n+1}$
   where:
      (a) Corollary 3.1.
      (b) Induction hypothesis for the legal path π'.
      (c) We assumed π is calling-covered, and therefore $(R_{\pi\downarrow_{p_i}}, T_{\pi\downarrow_{p_i}}) \in sum_{p_i}$.
      (d) Definition of R for a procedure call in the modular version.
   2. $R_\pi = R_\pi^{m+1} \rightarrow^{(a)} (R_\pi^j \wedge R_{\pi\downarrow_{p_i}}(T_\pi^j[Y_i])) = (R_{\pi'} \wedge R_{\pi\downarrow_{p_i}}(T_{\pi'}[Y_i])) \rightarrow^{(b)}$
   $\left((T_{\pi'} \downarrow_{V_q} = T_{\pi'\downarrow_q}) \wedge R_{\pi\downarrow_{p_i}}\left(T_{\pi'\downarrow_q}[Y_i]\right)\right) =$
   $\left(\left(T_\pi^j \downarrow_{V_q} = T_{\pi\downarrow_q}^n\right) \wedge R_{\pi\downarrow_{p_i}}\left(T_{\pi\downarrow_q}^n[Y_i]\right)\right)$

and therefore:

$$\forall x \in V_q \setminus Y_i.\ T^{n+1}_{\pi\downarrow_q}[x] =^{(c)} T^{n}_{\pi\downarrow_q}[x] = T^{j}_{\pi}[x] =^{(a)} T^{m+1}_{\pi}[x]$$
$$\forall y_l \in Y_i.\ T^{n+1}_{\pi\downarrow_q}[y_l] = ITE(r^1(T^{n}_{\pi\downarrow_q}[Y_i]), t^1(T^{n}_{\pi\downarrow_q}[Y])[v_l], \dots) =^{(d)}$$
$$T_{\pi\downarrow_{p_i}}(T^{n}_{\pi\downarrow_q}[Y_i])[v_l] = T_{\pi\downarrow_{p_i}}(T^{j}_{\pi}[Y_i])[v_l] =^{(a)} T^{m+1}_{\pi}[y_l]$$

where:

(a) Corollary 3.1.

(b) Induction hypothesis for the legal path π'.

(c) Definition of T for a procedure call in the modular version.

(d) We assumed π is calling-covered, and therefore $(R_{\pi\downarrow_{p_i}}, T_{\pi\downarrow_{p_i}}) \in sum_{p_i}$. Also, $R_{\pi\downarrow_{p_i}}$ is implied by R_π, and all reachability conditions in the summary are disjoint. □

3.2.2 Symbolic Execution ⊇ Modular Symbolic Execution

For each path π analysed with modular symbolic execution there exists a set of corresponding in-lined paths that show the same behavior. Therefore for this direction we say that given a path and an input, there exists an in-lined (single) corresponding path that behaves the same as the modularly analysed path for that input. Since we show this for any input we get that the entire behavior of π has corresponding in-lined behaviors.

Let π be a finite path in CFG_q, we assume that R_π, T_π were computed by modular symbolic execution.

Theorem 3.3. *Let π be a finite path in CFG_q, and σ^v_q a visible state, such that $\sigma^v_q \models R_\pi(V^v_q)$ and for all $i \in k$ the p_i-projected paths traversed from σ^v_q in the in-lined program are in their procedures' summaries. Then there exists a path π^{in} in CFG^{in}_q that satisfies:*

1. $\pi^{in} \downarrow_q = \pi$
2. $\sigma^v_q \models R_{\pi^{in}}(V^v_q)$
3. $T_{\pi^{in}}(\sigma^v_q) \downarrow_{V_q} = T_\pi(\sigma^v_q)$

Proof. Given a path $\pi = l_1, \dots, l_n$ we define π^{in} inductively, while maintaining that π^{in} satisfies all three conditions.
Base: $n = 0$, meaning π is empty, then π^{in} is empty as well and we get:

1. $\pi^{in} \downarrow_q = \pi$ by definition.
2. $\sigma^v_q \models true$ and therefore $\sigma^v_q \models R_{\pi^{in}}$ since $R_{\pi^{in}} = R^1_{\pi^{in}} = true$.

3. $\forall x \in V_q.\ T_{\pi^{in}}(\sigma_q^v)[x] = T^1_{\pi^{in}}(\sigma_q^v)[x] = x = T^1_{\pi}(\sigma_q^v)[x] = T_{\pi}(\sigma_q^v)[x]$.

Step: Let $\pi = l_1, \ldots, l_n$ be a path in CFG_q. We assume that for $\pi' = l_1, \ldots, l_{n-1}$, $\pi'^{in} = l_1^{in}, \ldots, l_{m-1}^{in}$ is defined and maintains the conditions.

- If l_n is an assignment node $\bar{x} = \bar{e}$, then we define $\pi^{in} = (\pi'^{in}, l_n)$ and:

  1. $\pi^{in} \downarrow_q = (\pi'^{in} \downarrow_q, l_n) = (\pi', l_n) = \pi$ by definition.
  2. $\sigma_q^v \models R_\pi = R_\pi^{n+1} =^{(a)} R_\pi^n = R_{\pi'}$ and therefore by induction hypothesis $\sigma_q^v \models R_{\pi'^{in}} = R^m_{\pi^{in}} =^{(a)} R^{m+1}_{\pi^{in}} = R_{\pi^{in}}$ where:
     (a) By the definition of R for assignment.
  3. $\forall y \in V_q \setminus vars(\bar{x}).\ T_\pi(\sigma_q^v)[y] = T_\pi^{n+1}(\sigma_q^v)[y] =^{(a)} T_\pi^n(\sigma_q^v)[y] =$
     $= T_{\pi'}(\sigma_q^v)[y] =^{(b)} T_{\pi'^{in}}(\sigma_q^v)[y] = T^m_{\pi^{in}}(\sigma_q^v)[y] =^{(a)}$
     $= T^{m+1}_{\pi^{in}}(\sigma_q^v)[y] = T_{\pi^{in}}(\sigma_q^v)[y]$
     $\forall x_l \in vars(\bar{x}).\ T_\pi(\sigma_q^v)[x_l] = T_\pi^{n+1}(\sigma_q^v)[x_l] =^{(a)} e_l(T_\pi^n[V_q])(\sigma_q^v) =$
     $= e_l(T_{\pi'}[V_q])(\sigma_q^v) =^{(b)} e_l(T_{\pi'^{in}}[V_q])(\sigma_q^v) = e_l(T^m_{\pi^{in}}[V_q])(\sigma_q^v) =^{(a)}$
     $= T^{m+1}_{\pi^{in}}(\sigma_q^v)[x_l] = T_{\pi^{in}}(\sigma_q^v)[x_l]$
     where:
     (a) By the definition of T for assignment.
     (b) Induction hypothesis for π'.

- If l_n is a test node $B(V_q)$, then we define $\pi^{in} = (\pi'^{in}, l_n)$ and:

  1. $\pi^{in} \downarrow_q = (\pi'^{in} \downarrow_q, l_n) = (\pi', l_n) = \pi$ by definition.
  2. $\sigma_q^v \models R_\pi = R_\pi^{n+1} =^{(a)} R_\pi^n \wedge \tilde{B}(T_\pi^n) = R_{\pi'} \wedge \tilde{B}(T_{\pi'})$, and therefore by induction hypothesis $\sigma_q^v \models R_{\pi'^{in}} \wedge \tilde{B}(T_{\pi'^{in}}) = R^m_{\pi^{in}} \wedge \tilde{B}(T^m_{\pi^{in}}) =^{(a)} R^{m+1}_{\pi^{in}} = R_{\pi^{in}}$ where:
     (a) By the definition of R for test.
  3. $T_\pi(\sigma_q^v) = T_\pi^{n+1}(\sigma_q^v) =^{(a)} T_\pi^n(\sigma_q^v) = T_{\pi'}(\sigma_q^v) =^{(b)}$
     $= T_{\pi'^{in}}(\sigma_q^v) \downarrow_{V_q} = T^m_{\pi^{in}}(\sigma_q^v) \downarrow_{V_q} =^{(a)} T^{m+1}_{\pi^{in}}(\sigma_q^v) \downarrow_{V_q} = T_{\pi^{in}}(\sigma_q^v) \downarrow_{V_q}$
     where:
     (a) By the definition of T for test.
     (b) Induction hypothesis for π'.

- If l_n is a call node to $p_i(Y_i)$, then we define $\pi^{in} = (\pi'^{in}, l_i^{\text{pre}}, l_1^i, \ldots, l_k^i, l_i^{\text{post}})$, where $\pi_i = l_1^i, \ldots, l_k^i$ is a path in CFG_{p_i} that is traversed from $T_{\pi'}(\sigma_q^v)[Y_i]$. meaning:

$$(*) \qquad \sigma_q^v \models R_{\pi_i}(T_{\pi'}[Y_i]).$$

  1. $\pi^{in} \downarrow_q = (\pi'^{in} \downarrow_q, l_i) = (\pi', l_n) = \pi$ by definition.
  2. $\sigma_q^v \models R_\pi = R_\pi^{n+1} =^{(a)} R_\pi^n \wedge \bigvee_{(r,t) \in sum_{p_i}} r(T_\pi^n[Y_i]) = R_{\pi'} \wedge \bigvee_{(r,t) \in sum_{p_i}} r(T_{\pi'}[Y_i])$, therefore by induction hypothesis and $(*)$
     $\sigma_q^v \models R_{\pi'^{in}} \wedge R_{\pi_i}(T_{\pi'^{in}}[Y_i]) = R^m_{\pi^{in}} \wedge R_{\pi_i}(T^m_{\pi^{in}}[Y_i]) =^{(b)} R^{m+k+1}_{\pi^{in}} = R_{\pi^{in}}$,
     where:

(a) Definition of R for a procedure call in the modular version.

(b) Corollary 3.1.

3. $\forall x \in V_q \setminus Y_i.\ T_\pi(\sigma_q^v)[x] = T_\pi^{n+1}(\sigma_q^v)[x] =^{(a)} T_\pi^n(\sigma_q^v)[x] =$
$= T_{\pi'}(\sigma_q^v)[x] =^{(b)} T_{\pi'^{in}}(\sigma_q^v)[x] = T_{\pi^{in}}^m(\sigma_q^v)[x] =^{(d)}$
$= T_{\pi^{in}}^{m+k+1}(\sigma_q^v)[x] = T_{\pi^{in}}(\sigma_q^v)[x]$

$\forall y_l \in Y_i.\ T_\pi(\sigma_q^v)[y_l] = T_\pi^{n+1}(\sigma_q^v)[y_l] =^{(a)}$
$= ITE(r^1(T_\pi^n(\sigma_q^v)[Y_i]), t^1(T_\pi^n(\sigma_q^v)[Y_i])[v_l], \dots) =^{(c)}$
$= T_{\pi_i}(T_\pi^n[Y_i])(\sigma_q^v)[v_l] = T_{\pi_i}(T_{\pi'}[Y_i])(\sigma_q^v)[v_l] =^{(b)}$
$= T_{\pi_i}(T_{\pi'^{in}}[Y_i])(\sigma_q^v)[v_l] = T_{\pi_i}(T_{\pi^{in}}^m[Y_i])(\sigma_q^v)[v_l] =^{(d)}$
$= T_{\pi^{in}}^{m+k+1}(\sigma_q^v)[y_l] = T_{\pi^{in}}(\sigma_q^v)[y_l]$

where:

(a) Definition of T for a procedure call in the modular version.

(b) Induction hypothesis for π'.

(c) $(*)$, and π_i must be covered by sum_{p_i} since $\sigma_q^v \models R_\pi$ and therefore $\sigma_q^v \models \bigvee_{(r,t)\in sum_{p_i}} r(T_{\pi'}[Y_i])$ and all reachability conditions in the summary are disjoint.

(d) Corollary 3.1. □

3.2.3 Deeper Call Graphs

We proved so far our claims only for call graphs of depth 1. To extend to deeper call graphs we first need to define some new definitions.

We assume that q calls procedures $p_1, \dots, p_k$ from locations $l_1, \dots, l_k$ with inputs $Y_1, \dots, Y_k$, respectively. And the set of all procedures transitively called from q is Q.

Definition 3.2.8. Inlined CFG
Let q be a procedure, represented by CFG_q, that calls procedures $p_1, \dots, p_k$ from nodes $l_1, \dots, l_k$, respectively. We obtain the in-lined version CFG_q^{in} from CFG_q, by performing the following changes for every $i \in [k]$:

- Changes in nodes:
  1. Remove node l_i $(l_i : p_i(Y_i))$.
  2. Add assignment node $l_i^{\text{pre}} : V_{p_i}^v := Y_i$.
  3. Add assignment node $l_i^{\text{post}} : Y_i := V_{p_i}^v$.
  4. Add all the nodes from $CFG_{p_i}^{in}$.
- Changes in edges:
  1. Remove edge (l, l_i), add edge (l, l_i^{pre}).

2. Remove edge (l_i, l), add edge (l_i^{post}, l).
3. Add edge $(l_i^{\text{pre}}, l_i^{\text{entry}})$, where l_i^{entry} is the entry node of $CFG_{p_i}^{in}$.
4. Add edge $(l_i^{\text{exit}}, l_i^{\text{post}})$, where l_i^{exit} is the exit node of $CFG_{p_i}^{in}$.
5. Add all edges from $CFG_{p_i}^{in}$.

The depth of an in-lined call graph is the call depth of the deepest call[3] from q.

The definitions of legal paths and q-projected path remain the same. We now need two versions of p_i-projected paths:

Definition 3.2.9. p_i-Projected Path
Let π be a legal path in CFG_q^{in},

- Its **modular p_i-projected path**, denoted $\pi \downarrow_{p_i}^{m}$ is the sequence of nodes from CFG_{p_i} that appear in π, with sub-calls replaced by the original calling site.
- Its **in–lined p_i-projected path**, denoted $\pi \downarrow_{p_i}^{in}$ is the interval of nodes between l_i^{start} and the first l_i^{exit} following it, if such l_i^{start} exists, or empty otherwise.

Observation 3.2.10. A modular p_i-projected path $\pi \downarrow_{p_i}^{m}$ is a path in CFG_{p_i}.

Observation 3.2.11. An in–lined p_i-projected path $\pi \downarrow_{p_i}^{in}$ is a path in $CFG_{p_i}^{in}$.

Corollary 3.4. $(\pi \downarrow_{p_i}^{in}) \downarrow_{p_i} = (\pi \downarrow_{p_i}^{m})$

As before we need to clarify when our summaries have enough information.

Definition 3.2.12. Covered Path
We say that a path π in CFG_q^{in} is **calling-covered** if for every $p \in Q$, $\pi \downarrow_p^m$ is covered by sum_p.

To cope with further sub-calls, we apply the same theorems by induction on the depth of the call graph.

The proofs we have for depth 1 will be used as base cases.

Symbolic Execution $\subseteq$ Modular Symbolic Execution

Theorem 3.5. *Let π be a legal, calling-covered path in CFG_q^{in}, its q-projected path $\pi \downarrow_q$ satisfies:*

1. $R_\pi(V_q^v) \rightarrow R_{\pi\downarrow_q}(V_q^v)$
2. $R_\pi(V_q^v) \rightarrow T_\pi(V_q^v) \downarrow_{V_q} = T_{\pi\downarrow_q}(V_q^v)$

[3]Recursion can be unwound up to the needed depth, and since we analyse paths to a certain depth, this suits our needs.

Proof. We prove by induction on c, the depth of the call graph CFG_q^{in}. The base case where $c = 1$ is Theorem 3.2. For the step we assume the depth of CFG_q^{in} is $c+1$ and we assume correctness for all CFGs of lower depth (bounded by c).
To prove for depth $c+1$ we use an internal induction on the length of legal paths π ($\pi = l_1^{in}, \ldots, l_m^{in}$). We denote the length of $\pi \downarrow_q$ by n.
Base: If $m = 0$, the same proof as in the base case in the proof of Theorem 3.2.
Step: Assume correctness for all legal paths of length strictly smaller than m. We consider the last node, l_m^{in}:

- If node l_m^{in} is an assignment node or a test node, then it's the same proof as in the proof of Theorem 3.2
- If node l_m^{in} is a node l_i^{post} then there exists $j < n$ such that $l_j = l_i^{\text{pre}}$, by definition $\pi' = l_1^{in}, \ldots l_{j-1}^{in}$ is legal, $\pi \downarrow_q = (\pi' \downarrow_q, l_i)$, where l_i is the original call node to p_i from CFG_q, and:

  1. $R_\pi = R_\pi^{m+1} =^{(a)} (R_\pi^j \wedge R_{\pi\downarrow_{p_i}^{in}}(T_\pi^j[Y_i])) = (R_{\pi'} \wedge R_{\pi\downarrow_{p_i}^{in}}(T_{\pi'}[Y_i])) \rightarrow^{(b)}$
     $(R_{\pi'\downarrow_q} \wedge R_{\pi\downarrow_{p_i}^{in}}(T_{\pi'\downarrow_q}[Y_i])) \rightarrow^{(c)} (R_{\pi'\downarrow_q} \wedge R_{\pi\downarrow_{p_i}^{m}}(T_{\pi'\downarrow_q}[Y_i])) =$
     $(R_{\pi\downarrow_q}^n \wedge R_{\pi\downarrow_{p_i}^{m}}(T_{\pi\downarrow_q}^n[Y_i])) \rightarrow^{(d)} (R_{\pi\downarrow_q}^n \wedge (\bigvee_{(r,t)\in sum_{p_i}} r(T_{\pi\downarrow_q}^n[Y_i]))) =^{(e)} R_{\pi\downarrow_q}^{n+1}$

     where:

     (a) Corollary 3.1, since if all the sub-calls are in-lined, then we can apply the lemma and its corollary.
     (b) Internal Induction hypothesis for the legal path π'.
     (c) External Induction hypothesis for $\pi \downarrow_{p_i}^{in}$, since the depth of CFG_{p_i} is bounded by c.
     (d) We assumed π is calling-covered, and therefore $(R_{\pi\downarrow_{p_i}^m}, T_{\pi\downarrow_{p_i}}^m) \in sum_{p_i}$.
     (e) Definition of R for a procedure call in the modular version.

  2. $R_\pi = R_\pi^{m+1} \rightarrow^{(a)} (R_\pi^j \wedge R_{\pi\downarrow_{p_i}^{in}}(T_\pi^j[Y_i])) = (R_{\pi'} \wedge R_{\pi\downarrow_{p_i}^{in}}(T_{\pi'}[Y_i])) \rightarrow^{(b)}$
     $\left(\left(T_{\pi'} \downarrow_{V_q} = T_{\pi'\downarrow_q}\right) \wedge R_{\pi\downarrow_{p_i}^{in}}\left(T_{\pi'\downarrow_q}[Y_i]\right)\right) \rightarrow^{(c)}$
     $\left(\left(T_{\pi'} \downarrow_{V_q} = T_{\pi'\downarrow_q}\right) \wedge R_{\pi\downarrow_{p_i}^{m}}\left(T_{\pi'\downarrow_q}[Y_i]\right)\right) =$
     $\left(\left(T_\pi^j \downarrow_{V_q} = T_{\pi\downarrow_q}^n\right) \wedge R_{\pi\downarrow_{p_i}^{m}}\left(T_{\pi\downarrow_q}^n[Y_i]\right)\right)$

     and therefore:

     $$\forall x \in V_q \setminus Y_i.\ T_{\pi\downarrow_q}^{n+1}[x] =^{(d)} T_{\pi\downarrow_q}^n[x] = T_\pi^j[x] =^{(a)} T_\pi^{m+1}[x]$$
     $$\forall y_l \in Y_i.\ T_{\pi\downarrow_q}^{n+1}[y_l] = ITE(r^1(T_{\pi\downarrow_q}^n[Y_i]), t^1(T_{\pi\downarrow_q}^n[Y])[v_l], \ldots) =^{(e)}$$
     $$T_{\pi\downarrow_{p_i}^m}(T_{\pi\downarrow_q}^n[Y_i])[v_l] = T_{\pi\downarrow_{p_i}^m}(T_\pi^j[Y_i])[v_l] =^{(f)}$$
     $$T_{\pi\downarrow_{p_i}^{in}}(T_\pi^j[Y_i])[v_l] =^{(a)} T_\pi^{m+1}[y_l]$$

     where:

(a) Corollary 3.1, since if all the sub-calls are in-lined, then we can apply the lemma and its corollary.

(b) Internal induction hypothesis for the legal path π'.

(c) External Induction hypothesis for $\pi \downarrow_{p_i}^{in}$, since the depth of CFG_{p_i} is bounded by c.

(d) Definition of T for a procedure call in the modular version.

(e) We assumed π is calling-covered, and therefore $(R_{\pi\downarrow_{p_i}^m}, T_{\pi\downarrow_{p_i}^m}) \in sum_{p_i}$. Also, $R_{\pi\downarrow_{p_i}^m}$ is implied by R_π, and all reachability conditions in the summary are disjoint.

(f) External Induction hypothesis for $\pi \downarrow_{p_i}^{in}$, since the depth of CFG_{p_i} is bounded by c. Also, $R_{\pi\downarrow_{p_i}^{in}}$ is implied by R_π. □

Symbolic Execution ⊇ Modular Symbolic Execution

Theorem 3.6. *Let π be a finite path in CFG_q, and σ_q^v a visible state, such that $\sigma_q^v \models R_\pi(V_q^v)$ and for all $i \in k$ the p_i-projected paths traversed from σ_q^v in the in-lined program are in their procedures' summaries. Then there exists a path π^{in} in CFG_q^{in} that satisfies:*

1. $\pi^{in} \downarrow_q = \pi$
2. $\sigma_q^v \models R_{\pi^{in}}(V_q^v)$
3. $T_{\pi^{in}}(\sigma_q^v) \downarrow_{V_q} = T_\pi(\sigma_q^v)$

Proof. We prove by induction on c, the depth of the call graph CFG_q^{in}. The base case where $c = 1$ is Theorem 3.3. For the step we assume the depth of CFG_q^{in} is $c+1$ and we assume correctness for all CFGs of lower depth (bounded by c).
To prove for depth $c+1$, we use an internal induction on the length of π. Given a path $\pi = l_1, \ldots, l_n$ we define π^{in} inductively, while maintaining that π^{in} satisfies all three conditions.
Base: $n = 0$, the same construction and proof as in the base case in the proof of Theorem 3.3.
Step: Let $\pi = l_1, \ldots, l_n$ be a path in CFG_q. We assume that for $\pi' = l_1, \ldots, l_{n-1}$, $\pi'^{in} = l_1^{in}, \ldots, l_{m-1}^{in}$ is defined and maintains the conditions.

- If l_n is an assignment node or a test node, then it's the same construction and proof as in the base case in the proof of Theorem 3.3.
- If l_n is a call node to $p_i(Y_i)$, then we define $\pi^{in} = (\pi'^{in}, l_i^{\text{pre}}, l_1^i, \ldots, l_k^i, l_i^{\text{post}})$, where $\pi_i = l_1^i, \ldots, l_k^i$ is a path in $CFG_{p_i}^{in}$ that is traversed from $T_{\pi'}(\sigma_q^v)[Y_i]$. meaning:

$$(*) \qquad \sigma_q^v \models R_{\pi_i}(T_{\pi'}[Y_i]).$$

1. $\pi^{in} \downarrow_q = (\pi'^{in} \downarrow_q, l_i) = (\pi', l_n) = \pi$ by definition.
2. $\sigma_q^v \models R_\pi = R_\pi^{n+1} =^{(a)} R_\pi^n \wedge \bigvee_{(r,t)\in sum_{p_i}} r(T_\pi^n[Y_i]) = R_{\pi'} \wedge \bigvee_{(r,t)\in sum_{p_i}} r(T_{\pi'}[Y_i])$, therefore by the internal induction hypothesis and $(*)$
   $\sigma_q^v \models R_{\pi'^{in}} \wedge R_{\pi_i}(T_{\pi'^{in}}[Y_i]) = R_{\pi^{in}}^m \wedge R_{\pi_i}(T_{\pi^{in}}^m[Y_i]) =^{(b)} R_{\pi^{in}}^{m+k+1} = R_{\pi^{in}}$, where:
   (a) Definition of R for a procedure call in the modular version.
   (b) Corollary 3.1, since if all the sub-calls are in-lined (as in π_i), then we can apply the lemma and its corollary.
3. $\forall x \in V_q \setminus Y_i.\ T_\pi(\sigma_q^v)[x] = T_\pi^{n+1}(\sigma_q^v)[x] =^{(a)} T_\pi^n(\sigma_q^v)[x] =$
   $= T_{\pi'}(\sigma_q^v)[x] =^{(b)} T_{\pi'^{in}}(\sigma_q^v)[x] = T_{\pi^{in}}^m(\sigma_q^v)[x] =^{(d)}$
   $= T_{\pi^{in}}^{m+k+1}(\sigma_q^v)[x] = T_{\pi^{in}}(\sigma_q^v)[x]$
   $\forall y_l \in Y_i.\ T_\pi(\sigma_q^v)[y_l] = T_\pi^{n+1}(\sigma_q^v)[y_l] =^{(a)}$
   $= ITE(r^1(T_\pi^n(\sigma_q^v)[Y_i]), t^1(T_\pi^n(\sigma_q^v)[Y_i])[v_l], \dots) =^{(c)}$
   $= T_{\pi_i \downarrow_{p_i}}(T_\pi^n[Y_i])(\sigma_q^v)[v_l] =^{(e)} T_{\pi_i}(T_\pi^n[Y_i])(\sigma_q^v)[v_l] =$
   $= T_{\pi_i}(T_{\pi'}[Y_i])(\sigma_q^v)[v_l] =^{(b)} T_{\pi_i}(T_{\pi'^{in}}[Y_i])(\sigma_q^v)[v_l] =$
   $= T_{\pi_i}(T_{\pi^{in}}^m[Y_i])(\sigma_q^v)[v_l] =^{(d)}$
   $= T_{\pi^{in}}^{m+k+1}(\sigma_q^v)[y_l] = T_{\pi^{in}}(\sigma_q^v)[y_l]$
   where:
   (a) Definition of T for a procedure call in the modular version.
   (b) Induction hypothesis for π'.
   (c) $(*)$, and π_i must be covered by sum_{p_i} since $\sigma_q^v \models R_\pi$ and therefore $\sigma_q^v \models \bigvee_{(r,t)\in sum_{p_i}} r(T_{\pi'}[Y_i])$ and all reachability conditions in the summary are disjoint.
   (d) Corollary 3.1.
   (e) External induction hypothesis, since the depth of $CFG_{p_i}^{in}$ is bounded by c. □

3.3 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 3.3.1. 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 2.3.1 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 3.3.2. Consider the procedures in Figure 2.1. 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 underapproximate 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 3.3.3. 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 $\neg U(p_1,p_2)$[4], 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 2.2.1).

Definition 3.3.4. 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:

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

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^v_p). If d is satisfiable we say that π_1 and π_2 ***show difference***.

3.4 Computing Difference Summaries

3.4.1 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, 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

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

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 3.1, 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 $(false, false)$ (line 10). 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 (line 19) – 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 3.4.2) 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.

3.4.2 Computing the Difference Summaries for a Pair of Procedures

Algorithm ConstProcDiffSum (presented in Algorithm 3.2) 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 the set

Algorithm 3.1 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(P_1, P_2)
2: FoundDiff[(p_1, p_2)] = $\emptyset$, for each $(p_1, p_2) \in match$
3: $workSet := \emptyset$
4: $newWorkSet$:= $\{(p_1, p_2) \in match : p_1$ different syntactically from $p_2\}$
5: **while** $newWorkSet \neq workSet$ **do**
6: $workSet := newWorkSet$
7: $newWorkSet := workSet \cup \{(q_1, q_2) \in match : \exists (p_1, p_2) \in workSet$ s.t. q_1 calls p_1 or q_2 calls $p_2\}$
8: **end while**
9: **for** each $(p_1, p_2) \in workSet$ **do**
10: Diff[(p_1, p_2)] := (*false*, *false*)
11: Sum[p_1]:=$\emptyset$, Sum[p_2]:=$\emptyset$
12: **end for**
13: **for** each $(p_1, p_2) \in match \backslash workSet$ **do**
14: Diff[(p_1, p_2)] := (*false*, *true*)
15: Sum[p_1]:=$\emptyset$, Sum[p_2]:=$\emptyset$
16: **end for**
17: **while** $workSet \neq \emptyset$ **do**
18: (p_1, p_2) := CHOOSENEXT($workSet$) ▷ heuristic order
19: **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, Diff[(g_1, g_2)]=(*,*true*) **then**
20: newDiff := (*false*,*true*)
21: **else**
22: Sum[p_1] := MODULARSYMBOLICEXECUTION(p_1,Sum)
23: Sum[p_2] := MODULARSYMBOLICEXECUTION(p_2,Sum)
24: (newDiff,newFoundDiff) :=CONSTPROCDIFFSUM(Sum[p_1],Sum[p_2],Diff[(p_1, p_2)])
25: FoundDiff[(p_1, p_2)]:=FoundDiff[(p_1, p_2)] $\cup$ newFoundDiff
26: **end if**
27: **if** Diff[(p_1, p_2)] $\neq$ newDiff **then**
28: Diff[(p_1, p_2)] := newDiff
29: $workSet := workSet \cup \{(q_1, q_2) \in match : q_1$ calls p_1 or q_2 calls $p_2\}$
30: **end if**
31: **end while**
32: **return** (Diff, FoundDiff)

Algorithm 3.2 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})
```

$found_diff_{p_1,p_2}$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.

3.5 Abstraction and Refinement

3.5.1 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 the unexplored behaviors of the called procedures by means of uninter-

preted 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 Equation (3.1) 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 $UF_g = \{ UF_g^j \mid 1 \leq j \leq |V_g^v| \}$ [5]. The uninterpreted function $UF_g^j(T_\pi^i[Y])$ replaces UK in $T_\pi^{i+1}[y_j]$ (Equation (3.2)), 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 Equation (3.2) 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 3.5.1. Consider again procedures $p1$, $p2$ in Figure 2.1. 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 $\Delta_{p_1,p_2} = (\mathit{false}, x < 2 \vee x > 4)$. When symbolic execution of a procedure g reaches a procedure call $p1(a)$, where a is a variable of the

[5] 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.

```
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 }
```

Figure 3.1: Procedure versions in need of refinement

calling procedure g, we will perform:

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

3.5.2 Refinement

Using the described abstraction, the computed R_π, T_π may contain symbols of uninterpreted functions, and therefore so could $\mathit{diffCond} = r_1 \wedge r_2 \wedge t_1 \neq t_2$ and $\mathit{eqCond} = r_1 \wedge r_2 \wedge t_1 = t_2$ (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.

Example 3.5.2. To conclude that the procedures in Figure 3.1 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 $U(p_1, p_2)$ ($\varphi \in \{\mathit{diffCond}, \mathit{eqCond}\}$) 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

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

where:

- φ - restricts the paths traversed in g_k to ones feasible under the call from π'.
- $\neg \left(\bigvee_{(r,t) \in sum_{g_k}} r\left(T_{\pi'}^{i_k-1}[Y_k]\right) \right)$ - restricts the paths traversed in g_k to ones not previously explored.
- $V_g^v = T_{\pi'}^{i_k-1}[Y_k]$ - links between the inputs to g_k to the visible variables of g_k, which are the ones that will appear during the traversal.

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 3.5.3. Consider again the procedures in Figure 3.1. Assume that the current summaries of abs_1=abs_2=abs are empty, but it is known that both versions are identical (unmodified syntactically). We get (using symbolic execution and Algorithm 3.2) 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 Chapter 4 we describe the results on our benchmarks in two extreme modes: running refinement always immediately when needed (ModDiffRef), and always delaying the refinement (ModDiff).

3.6 Comparison to 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 Chapter 4. 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.

Chapter 4

Experimental Results

We implemented the algorithm presented in section 3.4 with the abstractions from Section 3.5 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.

4.1 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 4.1a), 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.

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

Table 4.1: 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).

```
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

Figure 4.1: LoopMult and LoopUnrch benchmarks

4.2 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 Figure 4.1c, finds 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 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 Figure 4.1a, 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 Figure 4.1b), 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 4.1b). 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 4.1b). Since both SymDiff and RVT are only capable of proving equivalences, not disproving them, we did not compare to those tools.

Chapter 5

Conclusion and Future Work

In this dissertation 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, that can be refined on demand. Our experimental results demonstrate the advantage of our approach due to these features.

Some ideas for future work are:

- Incorporate the ideas shown here with some of the ideas from other works, such as [14] or [5, 26].
- Extend the implementation to support pointers and memory allocation.

Bibliography

[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] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, pages 367–381. Springer, 2008.

[5] J. D. Backes, S. Person, N. Rungta, and O. Tkachuk. Regression verification using impact summaries. In *Model Checking Software (SPIN)*, volume 7976 of *LNCS*, pages 99–116. Springer, 2013.

[6] C. Cadar and H. Palikareva. Shadow symbolic execution for better testing of evolving software. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 432–435. ACM, 2014.

[7] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[8] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, pages 168–176. Springer, 2004.

[9] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, pages 570–574. Springer, 2005.

[10] N. Francez. *Program verification*. Addison-Wesley Longman, 1992.

[11] D. Gao, M. K. Reiter, and D. Song. BinHunt: Automatically finding semantic differences in binary programs. In *Information and Communications Security*, LNCS, pages 238–255. Springer, 2008.

[12] P. Godefroid. Compositional dynamic test generation. In *ACM SigPlan Notices*, volume 42, pages 47–54. ACM, 2007.

[13] B. Godlin and O. Strichman. Inference rules for proving the equivalence of recursive procedures. *Acta Informatica*, 45(6):403–439, 2008.

[14] B. Godlin and O. Strichman. Regression verification. In *Proceedings of the 46th Annual Design Automation Conference*, pages 466–471. ACM, 2009.

[15] B. Godlin and O. Strichman. Regression verification: proving the equivalence of similar programs. *Software Testing, Verification and Reliability*, 23(3):241–258, 2013.

[16] M. Kawaguchi, S. K. Lahiri, and H. Rebelo. Conditional equivalence. *Tech. Rep. MSR-TR-2010–119*, 2010.

[17] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[18] D. Kroening and O. Strichman. Equality logic and uninterpreted functions. In *Decision Procedures*, pages 59–80. Springer, 2008.

[19] D. Kroening and G. Weissenbacher. Interpolation-based software verification with WOLVERINE. In *Computer Aided Verification*, LNCS, pages 573–578. Springer, 2011.

[20] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. SYMDIFF: A language-agnostic semantic diff tool for imperative programs. In *Computer Aided Verification*, LNCS, pages 712–717. Springer, 2012.

[21] S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 345–355. ACM, 2013.

[22] K. L. McMillan. Lazy abstraction with interpolants. In *Computer Aided Verification*, LNCS, pages 123–136. Springer, 2006.

[23] N. Partush and E. Yahav. Abstract semantic differencing for numerical programs. In *Static Analysis*, LNCS, pages 238–258. Springer, 2013.

[24] N. Partush and E. Yahav. Abstract semantic differencing via speculative correlation. In *ACM SIGPLAN Notices*, volume 49, pages 811–828. ACM, 2014.

[25] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Pasareanu. Differential symbolic execution. In *Foundations of Software Engineering*, pages 226–237. ACM, 2008.

[26] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *ACM SIGPLAN Notices*, volume 46, pages 504–515. ACM, 2011.

[27] D. A. Ramos and D. Engler. Under-constrained symbolic execution: correctness checking for real code. In *24th USENIX Security Symposium*, pages 49–64, 2015.

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

[29] O. Strichman and M. Veitsman. Regression verification for unbalanced recursive functions. In *FM 2016: Formal Methods: 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings 21*, pages 645–658. Springer, 2016.

[30] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Software Reliability Engineering, 1997. Proceedings., The Eighth International Symposium on*, pages 264–274. IEEE, 1997.

אנו מגדירים הרצה סימבולית מודולרית, ומוכיחים את הקשרים בינה לבין הרצה סימבולית סטנדרטית. אנו משתמשים בהרצה סימבולית מודולרית כדי לנתח כל מסלול בשגרה, תוך הימנעות מחזרה על מסלולים משגרות שנקראות מספר פעמים.

מימשנו את האלגוריתם שלנו, והשווינו אותו לכלים מובילים. התוצאות מראות כי הכלי שלנו מהיר יותר, ומצליח להוכיח שקילות גם כאשר הכלים האחרים נכשלים בכך.

תרומות מרכזיות

התרומות המרכזיות של עבודה זו הן:

- אנו מציגים אלגוריתם מודולרי ומונחה-דרישה לחישוב הבדלים סמנטיים בין גרסאות תוכנה.
- האלגוריתם שלנו ייחודי בכך שהוא מייצר קירוב עליון וקירוב תחתון להבדלים הסמנטיים.
- אנו מציגים כלים לאבסטרקציה ועידון המאפשרים לנו לשלוט באיזון שבין היעילות והדיוק של התוצאה.
- אנו מפתחים את הרעיון של הרצה סימבולית מודולרית.

תקציר

תכניות נבנות בשלבים, כאשר גרסה חדשה מתקבלת ע"י ביצוע שינויים בגרסה קודמת. אילו יכולנו להבין את ההבדלים בהתנהגות של שתי גרסאות של תוכנה, היינו מסייעים רבות לפיתוח מהיר של תוכנה נכונה. ניתן להשתמש בנכונות של הגרסא הישנה (שנבדקה בעבר) ובמידע על הבדלים כדי להסיק נכונות של הגרסא החדשה. ניתן גם לבדוק האם השינוי אכן תואם לציפיה של המפתח. באופן כללי משימה זו אינה כריעה, כלומר אינה ניתנת לחישוב. עם זאת, אנו מציגים אלגוריתם כללי לחישוב מקורב (אומדן) של השינויים הסמנטיים (מבחינת יחסי קלט־פלט) בין שתי תוכניות. אנו מאפיינים מצבים התחלתיים עבורם המצבים הסופיים המתקבלים מהרצת שתי הגרסאות הם שונים, וגם מצבים התחלתיים עבורם המצבים הסופיים זהים. המטרה העיקרית היא שהאומדנים יהיו מדוייקים מספיק לצרכי המשתמש עבור תכניות אמיתיות. היות והשימוש העיקרי הוא בזמן פיתוח תכנה, יתכן כי חיושב מקורב של ההבדל בין שגרות מסויימות יהיה מספיק (ואף עדיף למפתח) מאשר על התוכנית כולה. יתכן גם שתוצאות ביניים שכאלו יהיו מדוייקות יותר. אנו נותנים אפשרות למשתמש להנחות את האלגוריתם לשגרות מעניינות, ומייצרים קירובים של השינויים לכל אחת מהשגרות המשתפרים תוך כדי ריצה. לא מובטח שהאלגוריתם יסתיים, אך תוצאותיו משתפרות במהלך הריצה, וניתן להפסיק אותו בכל עת ולקבל תוצאות ביניים.

תוצאות הניתוח שלנו הן חלוקה של טווח הקלטים של כל זוג שגרות מתאימות (אנו מניחים שקיימת התאמה בין זוגות של שגרות משתי הגרסאות) לשלוש קבוצות: קלטים עבורם הפלט של השגרות שונה, קלטים עבורם הקלט של השגרות (אם עוצרות) זהה או ששתיהן אינן עוצרות, וקלטים עבורם לא ידוע האם יש הבדל.

אמנם האלגוריתם יכול לעבוד על כל שתי תוכניות, אבל הוא פועל בצורה יעילה הרבה יותר כאשר ההבדלים הסינטקטיים בין התוכניות הם יחסית מעטים. שינויים סינטקטיים בין גרסאות עוקבות של תוכנה הם לעיתים קרובות אכן מעטים ומרוכזים באזור מסויים של התכנית, לעיתים עמוק בעץ הקריאות. בגישה שלנו אנו מנתחים רק אזורים בתכנית המושפעים מן השינוי. הניתוח שלנו הוא גם מודולרי, כזה המנתח כל זוג שגרות בנפרד. שגרות הנקראות מתוכן אינן מועתקות, ובמקום זאת אנו משתמשים בסיכום שלהן.

למען היעילות, אנו מייצרים סיכומי שגרות אבסטרקטיים (בעזרת פונקציות לא מפורשות) שניתן לעדן לפי דרישה. אנו מראים כיצד ניתן להשתמש בפונקציות לא מפורשות משותפות כדי לבטא את הידע שלנו על שקילות סמנטית, גם כאשר אין לנו מידע על ההתנהגות המדוייקת של השגרות. האלגוריתם שלנו פועל מלמטה-למעלה, החל מהשינויים הסינטקטיים בין הגרסאות, ועד לנקודות הכניסה של התכניות. כאשר הדיוק של אחד הסיכומים אינו מספיק, אנו מבצעים עידון (מלמעלה-למטה) המשתמש בהקשר הקריאה הנוכחי, כדי לייצר סיכום חדש העונה על צרכי השגרה הקוראת.

המחקר בוצע בהנחייתה של פרופסור ארנה גרומברג, בפקולטה למדעי המחשב.

חלק מן התוצאות בחיבור זה פורסמו כמאמר מאת המחבר ושותפיו למחקר בכנס במהלך תקופת מחקר המגיסטר של המחבר:

Anna Trostanetski, Orna Grumberg, and Daniel Kroening. Modular demand-driven analysis of semantic difference for program versions. In *International Static Analysis Symposium*. Springer, NY, USA, 2017.

תודות

ברצוני להודות לארנה גרומברג שהייתה מנחה מדהימה, מקור השראה וחברה במהלך לימודיי.
בנוסף אני מודה לדניאל קרונינג, עופר שטריכמן, עופר גוטמן וכל האנשים שעזרו לי לאורך הדרך.

אני מודה לטכניון ולמלגת רנדי ל. ומלווין ר. ברלין בתכנית למחקר ולימוד אבטחת סייבר על התמיכה הכספית הנדיבה בהשתלמותי.

ניתוח מודולרי מונחה-דרישה של הבדלים סמנטיים בין גרסאות של תוכנה

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים במדעי המחשב

אנה טרוסטנצקי

הוגש לסנט הטכניון – מכון טכנולוגי לישראל
תמוז 5777 חיפה יולי 2017

ניתוח מודולרי מונחה-דרישה של הבדלים סמנטיים בין גרסאות של תוכנה

אנה טרוסטנצקי