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:


**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

1 Introduction
  1.1 Related Work .................................................. 3
  1.2 Our Approach .................................................. 4
    1.2.1 Our method in detail ...................................... 5
    1.2.2 Main Contributions ......................................... 6

2 Preliminaries
  2.1 Procedures ....................................................... 7
  2.2 Symbolic Execution ............................................. 10
  2.3 Equivalence ..................................................... 11

3 Our Contribution
  3.1 Modular Symbolic Execution ................................... 13
  3.2 Symbolic Execution vs. Modular Symbolic Execution .............. 14
    3.2.1 Symbolic Execution \( \subseteq \) Modular Symbolic Execution .... 19
    3.2.2 Symbolic Execution \( \supseteq \) Modular Symbolic Execution .... 21
    3.2.3 Deeper Call Graphs ........................................... 23
  3.3 Difference Summary ............................................ 27
  3.4 Computing Difference Summaries ................................ 29
    3.4.1 Call Graph Traversal ....................................... 29
    3.4.2 Computing the Difference Summaries for a Pair of Procedures . 30
  3.5 Abstraction and Refinement .................................... 32
    3.5.1 Abstraction ................................................... 32
    3.5.2 Refinement ..................................................... 34
  3.6 Comparison to Related Work ................................... 36

4 Experimental Results
  4.1 Benchmarks and Results ........................................ 39
  4.2 Analysis .......................................................... 41
5 Conclusion and Future Work

Hebrew Abstract
List of Figures

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. .......................................................... 4

2.1 Examples of procedure versions ............................................. 9

3.1 Procedure versions in need of refinement ............................... 34

4.1 LoopMult and LoopUnrch benchmarks ................................. 41
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.
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 $\text{sum}_{p_i}$ ($i \in \{1, 2\}$) for each procedure $p_i$. The summary is obtained using symbolic execution. It includes path summarizations $(R_\pi, T_\pi)$ for a subset of the finite paths $\pi$ of $p_i$, where $R_\pi$ is the reachability condition for $\pi$ to be traversed and $T_\pi$ is the state transformation describing transformation from initial states to final states when $\pi$ is executed.

Next, we compute a (partial) difference summary $(C(p_1, p_2), U(p_1, p_2))$ 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 $U(p_1, p_2) \equiv \text{true}$ we can conclude that no differences are propagated from $p_1, p_2$. 

5
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 $\Pi$ 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 $\text{main}_P$.

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 $V_v^p$.
- The *hidden variables* of a procedure $p$ are the local variables used by the procedure, denoted $V_h^p$.
- The *variables* of a procedure $p$ are both its visible and hidden variables, denoted $V_p (V_p = V_v^p \cup V_h^p)$.
- A *state* $\sigma_p$ 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: \( \vec{x} = \vec{e} \), where \( \vec{x} = x_1, \ldots, x_n \) is a list of variable in \( V_p \) and \( \vec{e} = e_1, \ldots, e_n \) 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 \( \pi \) 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 \( \sigma_p^v \) if the path traversed in \( p \) from \( \sigma_p^v \) is finite and maximal. A program terminates on a visible state \( \sigma_{main}^v \) 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.
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 $\pi$ be a finite path in procedure $p$.

- The **Reachability Condition** of $\pi$, denoted $R_{\pi}(V_p^v)$, is a condition on the visible states at the beginning of $\pi$, which guarantees that the control will traverse $\pi$.

- The **State Transformation** of $\pi$, denoted $T_{\pi}(V_p^v)$, describes the final state of $\pi$, obtained if control traverses $\pi$ starting with some valuation $\sigma_p^v$ of $V_p^v$.

$$T_{\pi}(V_p^v)$$ is 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 $\pi$. 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,

$$R_{\alpha}(x) = x < 0 \quad R_{\beta}(x) = ((-x < 0) \land x \geq 2) \equiv x \geq 2$$

$$T_{\alpha}(x) = (-1) \quad T_{\beta}(x) = (x)$$

A path $\pi$ is called **feasible** if $R_{\pi}$ is satisfiable, meaning that there exists an input that traverses the path $\pi$. Note that, in $p1$ from Figure 2.1, 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 $R_\pi(V_p^\pi)$ and state transformation $T_\pi(V_p^\pi)$. The computation is performed in stages, where for every $1 \leq i \leq n + 1$, $R_{\pi i}^i(V_p)$ and $T_{\pi i}^i(V_p)$ are the path condition and state transformation for path $l_1, \ldots, l_{i-1}$, respectively. Initialization:

- For every $x \in V_p$, $T_1^1(V_p)[x] = x$.
- $R_\pi^1(V_p) = \text{true}$.

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

- Assignment $\bar{x} = \bar{e}$: $R_{\pi i}^{i+1}(V_p) := R_{\pi i}^i(V_p)$, $\forall x_i \in var_s(x)$. $T_{\pi i}^{i+1}(V_p)[x_i] := e_i[V_p \leftarrow T_{\pi i}^i(V_p)]$ and $\forall y \not\in var_s(x)$, $T_{\pi i}^{i+1}(V_p)[y] := T_{\pi i}^i(V_p)[y]$

- 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 $B(V_p)$: $T_{\pi i}^{i+1}(V_p) := T_{\pi i}^i(V_p)$, and

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

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

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

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, $V_p^{\pi/h}$ will not appear in $R_{\pi}^{n+1}(V_p)$ and $T_{\pi}^{n+1}(V_p) \downarrow_{V_p^\pi}$.
Definition 2.2.1. Path Summary
Given a finite maximal path $\pi$ in $p$, a Path Summary (also known as a partition-effect pair in [25]) is the pair $(R_\pi(V_v^p), T_\pi(V_v^p))$.

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

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

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 $p_1$ from Figure 2.1. Any subset of the set $\{(x < 0, -1), (x \geq 0 \land x \geq 2, x), (x \geq 0 \land x < 2, 3)\}$ is a summary for $p_1$. For the summary

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

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

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 $V_v^{p_1}$ and $V_v^{p_2}$, 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 $\sigma_p^v$ be a visible state for $p_1$ and $p_2$.

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

- $p_1$ and $p_2$ are **mutually terminate** for $\sigma_p^v$ if and only if the following holds: $p_1$ terminates on $\sigma_p^v$ if and only if $p_2$ terminates on $\sigma_p^v$.

- $p_1$ and $p_2$ are **fully equivalent** for $\sigma_p^v$ if and only if $p_1$ and $p_2$ are partially equivalent for $\sigma_p^v$ and mutually terminate for $\sigma_p^v$. 
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:

\[
\text{sum}_g = \{(r^1, t^1), \ldots, (r^n, t^n)\}.
\]

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 \( T^i_{\pi}(V_p) \):

\[
R^{i+1}_{\pi} = R^i_{\pi} \land (\bigvee_{(r, t) \in \text{sum}_g} r(T^i_{\pi}[Y])) \quad (3.1)
\]

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

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

where:

- \( \text{ITE}(b, e_1, e_2) \) 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 \( j \)th element (for \( y_j \)) of the path transformation \( t^k \).

- \( UK \) represents the value that is given if no path condition from \( \text{sum}_g \) is satisfied. That is, \( UK \) is returned when an unexplored path is traversed. Note, however, that since we added \( (\bigvee_{(r, t) \in \text{sum}_g} r(T^i_{\pi}[Y])) \) to the path condition \( R^i_{\pi} \), a path that satisfies \( R^{i+1}_{\pi} \) will never return \( UK \). Thus, \( UK \) is just a place holder.

\[^{1}\text{We use } r(T^i_{\pi}[Y]) \text{ to indicate that every } v_k \in V_g^* \text{ is replaced by the expression } T^i_{\pi}[y_k].\]
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 \( \pi \) in \( p \) includes a call to procedure \( g \). Then \( \pi \) 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 \( \pi^{\text{in}} \) in the in-lined version of \( p \) there is a corresponding path \( \pi \) in \( p \) such that:
  - \( R_{\pi^{\text{in}}} \rightarrow R_\pi \)
  - \( R_{\pi^{\text{in}}} \rightarrow T_{\pi^{\text{in}}} = T_\pi \)

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

### 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 \( \text{sum}_{p_1}, \ldots, \text{sum}_{p_k} \), 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 \( l_i^{pre} : V_{p_i} := Y_i \).
  3. Add assignment node \( l_i^{post} : Y_i := V_{p_i} \).
  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 \((l_i^{pre}, l_i^{entry})\), where \( l_i^{entry} \) is the entry node of \( CFG_{p_i} \).
  4. Add edge \((l_i^{exit}, l_i^{post})\), 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 \( R_{\pi}, T_{\pi} \) computed with symbolic execution for some \( \pi \) of length \( n \) in \( CFG_{q}^{in} \), \((V_q^h)^{in} \) will not appear in \( R_{\pi}^{n+1}((V_q^h)^{in}) \) and \( T_{\pi}^{n+1}((V_q^h)^{in}) \downarrow_{(V_q^v)^{in}} \).

**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 \( l_m^{in} = l_i^{post} \)

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 \( l_j^{in} = l_i^{pre} \), there exists \( r > j \) such that:

- \( l_r^{in} = l_i^{post} \), 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 \( \pi \) be a legal path in \( CFG_{q}^{in} \),
• Its \textbf{p$_i$-projected path}, denoted $\pi \downarrow_{p_i}$, is the interval of nodes between $i^{\text{start}}_i$ and the first $i^{\text{exit}}_i$ following it, if such $i^{\text{start}}_i$ exists, or empty otherwise.\footnote{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.}

• Its \textbf{q-projected path}, denoted $\pi \downarrow_q$, is the path obtained from $\pi$ by the following operations:
  
  - Every node $i^{\text{pre}}_i$ is replaced by $i$, the original calling node to $p_i$.
  - Every node not in $CFG_q$ is removed (including nodes from called procedures, and $i^{\text{post}}_i$ nodes).

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

\textbf{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.

\textbf{Definition 3.2.6.} \textbf{Covered path}

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

\textbf{Lemma 3.2.7.} Let $\pi^1 = l^1_1, \ldots, l^1_n$ and $\pi^2 = l^2_1, \ldots, l^2_m$ be two paths in $CFG_p$ with no procedure calls, such that there exists an edge $(l^1_i, l^2_i)$. Then the path $\pi^1 \cdot \pi^2 = l^1_1, \ldots, l^1_n, l^2_1, \ldots, l^2_m$ is a path in $CFG_p$ and:

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

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

\textbf{Proof.} We prove the lemma by induction on $m$, the length of $\pi^2$:

\textbf{Base:} If $m = 0$ then $\pi^2$ is empty, $(\pi^1 \cdot \pi^2) = \pi^1$, and:

1. $R_{\pi^1 \cdot \pi^2} = R_{\pi^1} = R_{\pi^1} \land true = R_{\pi^1} \land R_{\pi^2}(T_{\pi^1})$ since $R_{\pi^2} = true$.

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

\textbf{Step:} Assume correctness for $\pi' = l^2_1, \ldots, l^2_{m-1}$, and consider the last node, $l^2_m$.

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

  1. $R_{\pi^1 \cdot \pi^2} = R_{(\pi^1 \cdot \pi')} \cdot l^2_m = R_{(\pi^1 \cdot \pi') \cdot l^2_m} = (a) \ R_{(\pi^1 \cdot \pi') \cdot l^2_m} = R_{\pi^1 \cdot \pi'} = (b) \ R_{\pi^1} \land R_{\pi'}(T_{\pi^1}) = R_{\pi^1} \land R_{\pi^2}(T_{\pi^1}) = (a) \ R_{\pi^1} \land R_{\pi^2}(T_{\pi^1}) = R_{\pi^1} \land R_{\pi^2}(T_{\pi^1})$
(a) Definition of $R$ for assignment.
(b) Induction hypothesis for $\pi'$.

2. $\forall x_l \in vars(\bar{x})$. $T_{\pi',\pi}^{1,2}[x_l] = T_{(\pi',\pi)^2}[x_l] = T_{(\pi',\pi)^2}[x_l] = (a)\quad R_{\pi',\pi}^{1,2}[x_l] = R_{(\pi',\pi)^2}[x_l] = a
\begin{align*}
&= c_l(T_{(\pi',\pi)^2}[V_p]) = c_l(T_{\pi',\pi}^{1,2}[V_p]) = (b) \\
&= c_l(T_{\pi',\pi}^{1,2}[V_p]) = c_l(T_{\pi',\pi}^{1,2}[V_p]) = (a) \\
&= T_{\pi',\pi}^{m+1}[x_l] = T_{\pi',\pi}^{m+1}[x_l]
\end{align*}

$\forall y \in V_p \setminus vars(\bar{x})$. $T_{\pi',\pi}^{1,2}[y] = T_{(\pi',\pi)^2}[y] = T_{(\pi',\pi)^2}[y] = (a)\quad T_{\pi',\pi}^{m+1}[y] = T_{\pi',\pi}^{m+1}[y] = (b)\quad T_{\pi',\pi}^{m+1}[y] = T_{\pi',\pi}^{m+1}[y]$

where:
(a) Definition of $T$ for test.
(b) Induction hypothesis for $\pi'$.

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

1. $R_{\pi',\pi}^{1,2} = R_{(\pi',\pi)^2}^{1,2} = R_{(\pi',\pi)^2}^{1,2} = (a)\quad R_{(\pi',\pi)^2}^{1,2} = R_{(\pi',\pi)^2}^{1,2}$
\begin{align*}
&= R_{\pi',\pi}^{1,2} \land \bar{B}(T_{\pi',\pi}^{1,2}[V_p]) = \bar{R}_{\pi',\pi}^{1,2} \land \bar{B}(T_{\pi',\pi}^{1,2}[V_p]) = (b) \\
&= R_{\pi',\pi}^{1,2} \land \bar{B}(T_{\pi',\pi}^{1,2}[V_p]) = \bar{R}_{\pi',\pi}^{1,2} \land \bar{B}(T_{\pi',\pi}^{1,2}[V_p]) = (a) \\
&= R_{\pi',\pi}^{1,2} \land \bar{B}(T_{\pi',\pi}^{1,2}[V_p]) = R_{\pi',\pi}^{1,2} \land \bar{B}(T_{\pi',\pi}^{1,2}[V_p]) = (b) \\
&= R_{\pi',\pi}^{1,2} \land \bar{B}(T_{\pi',\pi}^{1,2}[V_p]) = R_{\pi',\pi}^{1,2} \land \bar{B}(T_{\pi',\pi}^{1,2}[V_p]) = (a) \\
&= R_{\pi',\pi}^{1,2} \land \bar{B}(T_{\pi',\pi}^{1,2}[V_p]) = R_{\pi',\pi}^{1,2} \land \bar{B}(T_{\pi',\pi}^{1,2}[V_p]) = (b) \\
\end{align*}

2. $T_{\pi',\pi}^{1,2} = T_{(\pi',\pi)^2}^{1,2} = T_{(\pi',\pi)^2}^{1,2} = (a)\quad T_{(\pi',\pi)^2}^{1,2} = T_{(\pi',\pi)^2}^{1,2}$
\begin{align*}
&= T_{\pi',\pi}^{1,2} = T_{\pi',\pi}^{1,2} = (a)\quad T_{\pi',\pi}^{1,2} = T_{\pi',\pi}^{1,2}
\end{align*}

where:
(a) Definition of $T$ for test.
(b) Induction hypothesis for $\pi'$.

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 $\pi$ be a legal path in $CFG_{\pi}$, we assume that $R_{\pi}, T_{\pi}$ were computed by standard symbolic execution, and that $R_{\pi,\pi',\pi}^{i}, T_{\pi,\pi'}^{i}$ where computed by modular symbolic execution.

**Corollary 3.1.** Let $\pi = l_1^{in}, \ldots, l_m^{in}$ be a legal path in $CFG_{\pi}$, if $l_l^{in} = l_i^{pre}$ and $l_l^{in} = l_i^{post}$, where $Y_i = \{y_1, \ldots, y_r\}$ and $V_p^{m} = \{v_1, \ldots, v_r\}$ then:

1. $R_{\pi}^{n+1} = R_{\pi}^{i} \land R_{\pi,\pi'}^{i}(T_{\pi}^{i}[Y_i])$

2. For every $y_l \in Y_i$, $T_{\pi}^{n+1}[y_l] = T_{\pi,\pi'}^{i}(T_{\pi}^{i}[Y_i])[y_l]$
3. For every \( x \in V_q \setminus Y_i \), \( T_{\pi}^{n+1}[x] = T_2^d[x] \)

Proof. We get the corollary from the lemma, if we mark \( \pi' = l^{n}_1, \ldots, l^{n}_{j-1} \):

1. \( R_{\pi}^{n+1} = R_{\pi} = R_{\pi', l^{prev}_{i}, \pi \downarrow_{pi_i}, l^{post}_{i}} = (a) \ R_{\pi'} \land R_{\pi \downarrow_{pi_i}, l^{post}_{i}}(T_{\pi'}) = (b) \)
   
   \[
   = R_{\pi'} \land R_{l^{prev}_{i}}(T_{\pi'}) \land R_{\pi \downarrow_{pi_i}, l^{post}_{i}}(T_{l^{prev}_{i}}(T_{\pi'})) = (c) \\
   = R_{\pi'} \land R_{\pi \downarrow_{pi_i}, l^{post}_{i}}(T_{l^{prev}_{i}}(T_{\pi'})) = (d) \\
   = R_{\pi'} \land R_{\pi \downarrow_{pi_i}(T_{l^{prev}_{i}}(T_{\pi'}))} \land R_{l^{post}_{i}}(T_{\pi \downarrow_{pi_i}}(T_{l^{prev}_{i}}(T_{\pi'}))) = (e) \\
   = R_{\pi'} \land R_{\pi \downarrow_{pi_i}(T_{\pi'}(V_{pi}))} = (f) \ R_{\pi'} \land R_{\pi \downarrow_{pi_i}}(T_{\pi'}(V_{pi})) = (g) \\
   = R_{\pi'} \land R_{\pi \downarrow_{pi_i}}(T_{\pi'}[Y_i]) = R_{\pi} \land R_{\pi \downarrow_{pi_i}}(T_{2}^d[Y_i])
   \]

where:

(a) Lemma 3.2.7 for \( \pi^1 = \pi' \), \( \pi^2 = l^{prev}_{i} \cdot \pi \downarrow_{pi_i} \cdot l^{post}_{i} \).
(b) Lemma 3.2.7 for \( \pi^1 = l^{prev}_{i} \), \( \pi^2 = \pi \downarrow_{pi_i} \cdot l^{post}_{i} \).
(c) \( R_{l^{prev}_{i}} = true \).
(d) Lemma 3.2.7 for \( \pi^1 = \pi \downarrow_{pi_i} \), \( \pi^2 = l^{post}_{i} \).
(e) \( R_{l^{post}_{i}} = true \).
(f) \( R_{\pi \downarrow_{pi_i}} \) is defined over \( V_{pi}^v \).
(g) \( l^{prev}_{i} : V_{pi}^v = Y_i \).

2. Let \( y_i \in Y_i \), then:

\[
T_{\pi}^{n+1}[y_i] = T_{\pi', l^{prev}_{i}, \pi \downarrow_{pi_i}, l^{post}_{i}}[y_i] = (a) \ T_{l^{post}_{i}}(T_{\pi', l^{prev}_{i}, \pi \downarrow_{pi_i}})[y_i] = (b) \\
= T_{\pi', l^{prev}_{i}, \pi \downarrow_{pi_i}}[v_i] = (c) \ T_{\pi \downarrow_{pi_i}}(T_{\pi', l^{prev}_{i}})[v_i] = (d) \ T_{\pi \downarrow_{pi_i}}(T_{l^{prev}_{i}}(T_{\pi'}))[v_i] = (e) \\
= T_{\pi \downarrow_{pi_i}}(T_{l^{prev}_{i}}(T_{\pi'})(V_{pi}))[v_i] = (f) \ T_{\pi \downarrow_{pi_i}}(T_{\pi'}[Y_i])[v_i] = T_{\pi \downarrow_{pi_i}}(T_{2}^d[Y_i])[v_i]
\]

where:

(a) Lemma 3.2.7 for \( \pi^1 = \pi' \cdot l^{prev}_{i} \cdot \pi \downarrow_{pi_i} \) and \( \pi^2 = l^{post}_{i} \).
(b) \( l^{post}_{i} : Y_i = V_{pi}^v \) and therefore \( T_{l^{post}_{i}}[f][y_i] = f[v_i] \).
(c) Lemma 3.2.7 for \( \pi^1 = \pi' \cdot l^{prev}_{i} \) and \( \pi^2 = \pi \downarrow_{pi_i} \).
(d) Lemma 1 for \( \pi^1 = \pi' \) and \( \pi^2 = l^{prev}_{i} \).
(e) \( T_{\pi \downarrow_{pi_i}} \) is defined over \( V_{pi}^v \).
(f) \( l^{prev}_{i} : V_{pi}^v = Y_i \).

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

\[
T_{\pi}^{n+1}[x] = T_{\pi', l^{prev}_{i}, \pi \downarrow_{pi_i}, l^{post}_{i}}[x] = (a) \ T_{l^{post}_{i}}(T_{\pi', l^{prev}_{i}, \pi \downarrow_{pi_i}})[x] = (b) \\
= T_{\pi', l^{prev}_{i}, \pi \downarrow_{pi_i}}[x] = (c) \ T_{\pi \downarrow_{pi_i}}(T_{\pi', l^{prev}_{i}})[x] = (d) \ T_{\pi \downarrow_{pi_i}}[x] = (e) \\
= T_{l^{prev}_{i}}(T_{\pi'}[x]) = (f) \ T_{\pi'}[x] = T_2^d[x]
\]

18
where:

(a) Lemma 3.2.7 for $\pi^1 = \pi' \cdot \pi^{\text{pre}} \cdot \pi \downarrow_{p_i}$ and $\pi^2 = \pi^\text{post}_i$.
(b) $\pi^\text{post}_i : Y_i = V^v_{p_i}$ and since $x \notin Y_i$, $T^{\text{post}_i} f [x] = f [x]$.
(c) Lemma 3.2.7 for $\pi^1 = \pi' \cdot \pi^{\text{pre}} \cdot \pi \downarrow_{p_i}$.
(d) $x \in V^q_i$ and therefore according to our assumption that there are no common variable names between functions, $x \notin V^v_{p_i}$. $\pi \downarrow_{p_i}$ is a path in $CFG_{p_i}$ and therefore does not change $x$.
(e) Lemma 3.2.7 for $\pi^1 = \pi'$ and $\pi^2 = \pi^{\text{pre}}$.
(f) $\pi^{\text{pre}} : V^v_{p_i} = Y_i$ and since $x \notin V^v_{p_i}$, $T^{\text{pre}} f [x] = f [x]$.

3.2.1 Symbolic Execution $\subseteq$ Modular Symbolic Execution

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

Theorem 3.2. Let $\pi$ be a legal, calling-covered path in $CFG^m_q$, its $q$-projected path $\pi \downarrow_q$ satisfies:

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

Proof. We prove the theorem by induction on the length of legal paths $\pi$ ($\pi = l^m_1, \ldots, l^m_m$). We denote the length of $\pi \downarrow_q$ by $n$.

Base: If $m = 0$ then $\pi, \pi \downarrow_q$ are empty and:

1. $R_\pi = R^1_\pi = \text{true} \rightarrow R_{\pi \downarrow_q} = R^1_{\pi \downarrow_q} = \text{true}$
2. $\forall x \in V^q_i. (T^1_\pi [x] = T^1_{\pi \downarrow_q} [x] = x = T^1_{\pi \downarrow_q} [x] = T_{\pi \downarrow_q} [x])$, and therefore
   $R_\pi \rightarrow T_\pi \downarrow_{V^v_q} = T_{\pi \downarrow_q}$

Step: Assume correctness for all legal paths of length strictly smaller than $m$. We consider the last node, $l^m_m$.

- If node $l^m_m$ is an assignment node $\bar{x} = \bar{e}$, then by definition $\pi' = l^m_1, \ldots, l^m_{m-1}$ is legal, $\pi \downarrow_q = (\pi' \downarrow_q, l^m_m)$, and:

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

19
2. \( R_\pi = R_{\pi}^{m+1} = (a) \ R_\pi^m = R_{\pi'} \rightarrow (b) \ (T_\pi \downarrow V_q = T_{\pi,\downarrow_q}) \rightarrow \)
\( (T_\pi \downarrow V_q = T_{\pi,\downarrow_q}) \rightarrow (c) \ (T_\pi \downarrow V_q = T_{\pi,\downarrow_q}^{n+1}) \),
where:
(a) Definition of \( R \) for assignment.
(b) Induction hypothesis for the legal path \( \pi' \).
(c) \( \forall x \in \text{vars} (\bar{x}). T_{\pi,\downarrow_q}^m [x] := e_l(T_{\pi,\downarrow_q}^m [V_q]) = e_l(T_{\pi,\downarrow_q}^m [V_q]) = T_{\pi,\downarrow_q}^{n+1} [x] \)
\( \forall y \in V_q \ \text{vars} (\bar{x}), T_{\pi,\downarrow_q}^{n+1} [y] := T_{\pi,\downarrow_q}^{n+1} [y] = T_{\pi,\downarrow_q}^{n+1} [y] \).

- 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. \( R_\pi = R_{\pi}^{m+1} = (a) \ R_\pi^m \wedge \tilde{B}(T_{\pi,\downarrow_q}^m) = R_{\pi'} \wedge \tilde{B}(T_{\pi'} \rightarrow (b) \)
  \( (R_{\pi',\downarrow_q} \wedge \tilde{B}(T_{\pi',\downarrow_q})) = (R_{\pi',\downarrow_q} \wedge \tilde{B}(T_{\pi,\downarrow_q})) = (a) \ R_{\pi,\downarrow_q}^{m+1} = R_{\pi',\downarrow_q} \),
where \( \tilde{B} \) is either \( B \) or \( \neg B \), according to the edge marking on \( \pi \), and:
(a) Definition of \( R \) for test.
(b) Induction hypothesis for the legal path \( \pi' \).

2. \( R_\pi = R_{\pi}^{m+1} \rightarrow (a) \ R_\pi^m = R_{\pi'} \rightarrow (b) \ (T_\pi \downarrow V_q = T_{\pi,\downarrow_q}) \rightarrow \)
\( (T_\pi \downarrow V_q = T_{\pi,\downarrow_q}) \rightarrow (c) \ (T_\pi \downarrow V_q = T_{\pi,\downarrow_q}^{n+1}) \),
where:
(a) Definition of \( R \) for test.
(b) Induction hypothesis for the legal path \( \pi' \).
(c) \( T_{\pi,\downarrow_q}^{n+1} = T_{\pi,\downarrow_q}^m \) and \( T_{\pi,\downarrow_q}^{n+1} = T_{\pi,\downarrow_q}^m \) (definition of \( T \) for test).

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

2. \( R_\pi = R_{\pi}^{m+1} \rightarrow (a) \ (R_\pi^j \wedge R_{\pi,\downarrow_q}(T_{\pi,\downarrow_q}^j [Y_i])) = (R_{\pi'} \wedge R_{\pi,\downarrow_q}(T_{\pi,'\downarrow_q} [Y_i])) \rightarrow (b) \)
\( (T_{\pi'} \downarrow V_q = T_{\pi,'\downarrow_q}) \wedge R_{\pi,\downarrow_q}(T_{\pi,'\downarrow_q} [Y_i]) \)
\( (T_{\pi'} \downarrow V_q = T_{\pi,'\downarrow_q}) \wedge R_{\pi,\downarrow_q}(T_{\pi,'\downarrow_q} [Y_i]) \)
\( (T_{\pi'} \downarrow V_q = T_{\pi,'\downarrow_q}) \wedge R_{\pi,\downarrow_q}(T_{\pi,'\downarrow_q} [Y_i]) \)
20
and therefore:

$$\forall x \in V_q \setminus Y_i, \ T_{\pi_q}^{n+1}[x] = (c) \ T_{\pi_q}^{n}[x] = T_{\pi}^{j}[x] = (a) \ T_{\pi}^{m+1}[x]$$

$$\forall y_l \in Y_i, \ T_{\pi_q}^{n+1}[y_l] = ITE(r^{1} (T_{\pi_q}^{n}[Y_i]), r^{1} (T_{\pi_q}^{n}[Y_i])[y_l], \ldots) = (d)$$

$$T_{\pi_q}^{n+1}(T_{\pi_q}^{n}[Y_i])[y_l] = T_{\pi_q}^{n+1}(T_{\pi_q}^{n}[Y_i])[y_l] = (a) \ T_{\pi}^{m+1}[y_l]$$

where:

(a) Corollary 3.1.
(b) Induction hypothesis for the legal path $\pi'$.
(c) Definition of $T$ for a procedure call in the modular version.
(d) We assumed $\pi$ is calling-covered, and therefore $(R_{\pi_q}, T_{\pi_q}) \in sum_{\pi}$.

Also, $R_{\pi_q}$ is implied by $R_{\pi}$, and all reachability conditions in the summary are disjoint.

\[\square\]

### 3.2.2 Symbolic Execution $\supseteq$ Modular Symbolic Execution

For each path $\pi$ 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 $\pi$ has corresponding in-lined behaviors.

Let $\pi$ be a finite path in $CFG_q$, we assume that $R_{\pi}, T_{\pi}$ were computed by modular symbolic execution.

**Theorem 3.3.** Let $\pi$ be a finite path in $CFG_q$, and $\sigma_q^v$ 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 $\sigma_q^v$ in the in-lined program are in their procedures’ summaries. Then there exists a path $\pi^{in}$ in $CFG_{q}^{in}$ that satisfies:

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

**Proof.** Given a path $\pi = l_1, \ldots, l_n$ we define $\pi^{in}$ inductively, while maintaining that $\pi^{in}$ satisfies all three conditions.

**Base:** $n = 0$, meaning $\pi$ is empty, then $\pi^{in}$ is empty as well and we get:

1. $\pi^{in} \downarrow_q = \pi$ by definition.
2. $\sigma_q^v \models true$ and therefore $\sigma_q^v \models R_{\pi^{in}}$ since $R_{\pi^{in}} = R_{\pi}^{1} = true$.  

21
3. \( \forall x \in V_q. T_{\pi^{in}}(\sigma^v_q)[x] = T_{\pi^{in}}^{1}(\sigma^v_q)[x] = x = T_{\pi}^{1}(\sigma^v_q)[x] = T_{\pi}(\sigma^v_q)[x] \).

**Step:** Let \( \pi = l_1, \ldots, l_n \) be a path in \( CFG_q \). We assume that for \( \pi' = l_1, \ldots, l_{n-1}, \\pi'^{in} = l_1^{in}, \ldots, l_{n-1}^{in} \) 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. \( \sigma^v_q \models R_{\pi} = R_{\pi^{in}}^{n+1} = (a) R_{\pi}^{n+1} = R_{\pi'} \) and therefore by induction hypothesis \( \sigma^v_q \models R_{\pi^{in}} = R_{\pi^{in}}^{m+1} = (a) R_{\pi^{in}}^{m+1} = R_{\pi^{in}} \) where:
     (a) By the definition of \( R \) for assignment.

- If \( \pi' \) 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. \( \sigma^v_q \models R_{\pi} = R_{\pi^{in}}^{n+1} = (a) R_{\pi}^{n+1} \land \tilde{B}(T_{\pi'}^{n+1}) = R_{\pi'} \land \tilde{B}(T_{\pi'}) \) and therefore by induction hypothesis \( \sigma^v_q \models R_{\pi^{in}} \land \tilde{B}(T_{\pi^{in}}) = R_{\pi^{in}}^{m+1} \land \tilde{B}(T_{\pi^{in}}) = (a) R_{\pi^{in}}^{m+1} = R_{\pi^{in}} \) where:
      (a) By the definition of \( R \) for test.
  3. \( T_{\pi}(\sigma^v_q) = T_{\pi^{in}}^{n+1}(\sigma^v_q) = (a) T_{\pi}^{n+1}(\sigma^v_q) = T_{\pi'}(\sigma^v_q) = (b) T_{\pi^{in}}^{m+1}(\sigma^v_q) \downarrow_{V_q} = (a) T_{\pi^{in}}^{m+1}(\sigma^v_q) \downarrow_{V_q} = T_{\pi^{in}}(\sigma^v_q) \downarrow_{V_q} \)
      where:
      (a) By the definition of \( T \) for test.
  (b) Induction hypothesis for \( \pi' \).

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

  1. \( \pi^{in} \downarrow_q = (\pi'^{in} \downarrow_q, l_i) = (\pi', l_n) = \pi \) by definition.
  2. \( \sigma^v_q \models R_{\pi} = R_{\pi^{in}}^{n+1} = (a) R_{\pi}^{n+1} \lor \sum_{(r,t) \in \sum_{p_i}} r(T_{\pi}[Y_i]) = R_{\pi'} \lor \sum_{(r,t) \in \sum_{p_i}} r(T_{\pi'}[Y_i]), \)
      therefore by induction hypothesis and \( (*) \)
      \( \sigma^v_q \models R_{\pi^{in}} \land R_{\pi_i}(T_{\pi^{in}}[Y_i]) = R_{\pi^{in}}^{m} \land R_{\pi_i}(T_{\pi^{in}}[Y_i]) = (b) R_{\pi^{in}}^{m+k+1} = R_{\pi^{in}}, \)
      where:
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.

### 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 \( p_1, \ldots, p_k \) from locations \( l_1, \ldots, l_k \) with inputs \( Y_1, \ldots, Y_k \), 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 \( p_1, \ldots, p_k \) from nodes \( l_1, \ldots, l_k \), respectively. We obtain the in-lined version \( CFG_{q}^{\text{in}} \) 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 \( l_i^{\text{pre}} : V_{p_i}^{\text{pre}} := Y_i \).
  3. Add assignment node \( l_i^{\text{post}} : Y_i := V_{p_i}^{\text{pre}} \).
  4. Add all the nodes from \( CFG_{p_i}^{\text{in}} \).

- **Changes in edges:**
  1. Remove edge \((l_i, l_j)\), add edge \((l_i, l_i^{\text{pre}})\).
2. Remove edge \((l_i, l_i)\), add edge \((l_i^{\text{post}}, l_i)\).

3. Add edge \((l_i^{\text{pre}}, l_i^{\text{entry}})\), where \(l_i^{\text{entry}}\) is the entry node of \(CFG_{p_i}\).

4. Add edge \((l_i^{\text{exit}}, l_i^{\text{post}})\), where \(l_i^{\text{exit}}\) is the exit node of \(CFG_{p_i}\).

5. Add all edges from \(CFG_{p_i}\).

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 \(\pi\) be a legal path in \(CFG_q\),

- Its **modular \(p_i\)-projected path**, denoted \(\pi \downarrow^m_{p_i}\), is the sequence of nodes from \(CFG_{p_i}\) that appear in \(\pi\), with sub-calls replaced by the original calling site.

- Its **in-lined \(p_i\)-projected path**, denoted \(\pi \downarrow^\text{lin}_{p_i}\), is the interval of nodes between \(l_i^{\text{start}}\) and the first \(l_i^{\text{exit}}\) following it, if such \(l_i^{\text{start}}\) exists, or empty otherwise.

**Observation 3.2.10.** A modular \(p_i\)-projected path \(\pi \downarrow^m_{p_i}\) is a path in \(CFG_{p_i}\).

**Observation 3.2.11.** An in-lined \(p_i\)-projected path \(\pi \downarrow^\text{lin}_{p_i}\) is a path in \(CFG_{p_i}\).

**Corollary 3.4.** \((\pi \downarrow^\text{lin}_{p_i}) \downarrow_{p_i} = (\pi \downarrow^m_{p_i})\)

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

**Definition 3.2.12. Covered Path**

We say that a path \(\pi\) in \(CFG_q\) is **calling-covered** if for every \(p \in Q\), \(\pi \downarrow^m_p\) is covered by \(\text{sum}_p\).

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 \(\pi\) be a legal, calling-covered path in \(CFG_q\), 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 \(CFG_q^m\). The base case where \(c = 1\) is Theorem 3.2. For the step we assume the depth of \(CFG_q^m\) 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 \(\pi\) (\(\pi = l_1^m, \ldots, l_j^m\)). 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, \(l_j^m\).

- If node \(l_j^m\) is an assignment node or a test node, then it’s the same proof as in the proof of Theorem 3.2.

- If node \(l_j^m\) is a node \(l_i^\text{post}\) then there exists \(j < n\) such that \(l_j = l_i^\text{pre}\), by definition \(\pi' = l_i^m, \ldots, l_j^m\) 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. \(R_\pi = R_\pi^{n+1} = (a) \left( R_\pi^n \land R_{\pi \downarrow_q}^{n+1}(T_\pi^n[Y_i]) \right) = (R_\pi^n \land R_{\pi \downarrow_q}^{n+1}(T_\pi^n[Y_i])) \rightarrow (b) \left( R_{\pi' \downarrow_q}^{n+1} \land R_{\pi \downarrow_q}^{n+1}(T_{\pi' \downarrow_q}[Y_i]) \right) \rightarrow (c) \left( R_{\pi' \downarrow_q}^{n+1} \land R_{\pi \downarrow_q}^{n+1}(T_{\pi' \downarrow_q}[Y_i]) \right) = \left( R_{\pi' \downarrow_q}^{n+1} \land \left( \bigvee_{(r,t) \in \text{sum}_{p_i}} r(T_{\pi' \downarrow_q}[Y_i]) \right) \right) = (c) R_{\pi \downarrow_q}^{n+1}\)

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 \(\pi'\).

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

  (d) We assumed \(\pi\) is calling-covered, and therefore \((R_{\pi^{\downarrow q}}^{n+1}, T_{\pi^{\downarrow q}}^{n+1}) \in \text{sum}_{p_i}\).

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

2. \(R_\pi = R_\pi^{n+1} \rightarrow (a) \left( R_\pi^n \land R_{\pi \downarrow_q}^{n+1}(T_\pi^n[Y_i]) \right) = (R_\pi^n \land R_{\pi \downarrow_q}^{n+1}(T_\pi^n[Y_i])) \rightarrow (b) \left( (T_\pi^n \downarrow_q = T_{\pi' \downarrow_q}^{n+1}[Y_i]) \land R_{\pi \downarrow_q}^{n+1}(T_{\pi' \downarrow_q}[Y_i]) \right) \rightarrow (c) \left( (T_\pi^n \downarrow_q = T_{\pi' \downarrow_q}^{n+1}[Y_i]) \land R_{\pi \downarrow_q}^{n+1}(T_{\pi' \downarrow_q}[Y_i]) \right) = \left( (T_\pi^n \downarrow_q = T_{\pi' \downarrow_q}^{n+1}) \land R_{\pi \downarrow_q}^{n+1}(T_{\pi' \downarrow_q}[Y_i]) \right)

and therefore:

\[
\forall x \in V_q \setminus Y_i, \ T_{\pi \downarrow_q}^{n+1}[x] = (d) T_{\pi \downarrow_q}^n[x] = T_\pi^n[x] = (a) T_{\pi \downarrow_q}^{n+1}[x] \\
\forall y_i \in Y_i, \ T_{\pi \downarrow_q}^{n+1}[y_i] = \text{ITE}(r(T_{\pi \downarrow_q}^n[Y_i]), l^j(T_{\pi \downarrow_q}^n[Y_i])[v_i], \ldots) = (e) T_{\pi \downarrow_q}^n(T_{\pi \downarrow_q}^m[Y_i])[v_i] = T_{\pi \downarrow_q}^m(T_{\pi \downarrow_q}^m[Y_i])[v_i] = (f) T_{\pi \downarrow_q}^{n+1}[y_i] = (a) T_{\pi \downarrow_q}^{n+1}[y_i]
\]

where:
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 $\pi'$.

(c) External Induction hypothesis for $\pi_{\pi_i}^\text{in}$, 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 $\pi$ is calling-covered, and therefore $(R_{\pi_{\pi_i}^\text{in}}, T_{\pi_{\pi_i}^\text{in}}) \in \sum_{p_i}$. Also, $R_{\pi_{\pi_i}^\text{in}}$ is implied by $R_{\pi}$, and all reachability conditions in the summary are disjoint.

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

Symbolic Execution $\supseteq$ Modular Symbolic Execution

**Theorem 3.6.** Let $\pi$ be a finite path in $CFG_q$, and $\sigma_q^v$ 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 $\sigma_q^v$ in the in-lined program are in their procedures’ summaries. Then there exists a path $\pi_{\text{in}}$ in $CFG_{in_q}$ that satisfies:

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

**Proof.** We prove by induction on $c$, the depth of the call graph $CFG_{in_q}$. The base case where $c = 1$ is Theorem 3.3. For the step we assume the depth of $CFG_{in_q}$ is $c + 1$ and we assume correctness for all $CFG$s of lower depth (bounded by $c$).

To prove for depth $c + 1$, we use an internal induction on the length of $\pi$. Given a path $\pi = l_1, \ldots, l_n$ we define $\pi_{\text{in}}$ inductively, while maintaining that $\pi_{\text{in}}$ 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 $\pi' = l_1, \ldots, l_{n-1}$, $\pi_{\text{in}} = l_1^{\text{in}}, \ldots, l_{n-1}^{\text{in}}$ 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 $\pi_{\text{in}} = (\pi_{\text{in}}', l_1^{\text{pre}}, l_1^{\text{in}}, \ldots, l_k^{\text{in}}, l_k^{\text{post}})$, where $\pi_i = l_1^{\text{in}}, \ldots, l_k^{\text{in}}$ is a path in $CFG_{in_{p_i}}$ that is traversed from $T_{\pi'}(\sigma_q^v)[Y_i]$, meaning:

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

26
1. $\pi^{in}_q = (\pi^{in}_{q \downarrow}, l_i) = (\pi', l_n) = \pi$ by definition.

2. $\sigma^u_q = R^{u+1}_\pi$ \iff \begin{align*}
&= R^{\pi+1}_\pi \overset{(a)}{=} R^{\pi+1}_\pi \\lor \\sum_{p_i} r(T_{\pi^{in}_i}[Y_i]) = R^{\pi+1}_\pi \\lor \\sum_{p_i} r(T_{\pi^{in}_i}[Y_i]),
&\text{where by the internal induction hypothesis and (*)}
\end{align*}

$\sigma^u_q \Rightarrow R^{\pi+1}_\pi \\land R^{\pi+1}_\pi(T_{\pi^{in}_i}[Y_i]) = R^{\pi+1}_\pi \\land R^{\pi+1}_\pi(T_{\pi^{in}_i}[Y_i]) \overset{(b)}{=} R^{\pi+1}_\pi \\land R^{\pi+1}_\pi = R^{\pi+1}_\pi,$

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 $\pi_i$), then we can apply the lemma and its corollary.

3. $\forall x \in V_q \setminus Y_i. T^{\pi}(\sigma^u_q)[x] = T^{\pi+1}(\sigma^u_q)[x] \overset{(a)}{=} T^{\pi+1}(\sigma^u_q)[x] =$

$\overset{(d)}{=} T^{\pi+1}(\sigma^u_q)[x] = (b) T^{\pi+1}(\sigma^u_q)[x] = T^{\pi+1}(\sigma^u_q)[x] = (d) T^{\pi+1}(\sigma^u_q)[x] =$

$\forall y \in Y_i. T^{\pi}(\sigma^u_q)[y] = T^{\pi+1}(\sigma^u_q)[y] \overset{(a)}{=} (c)$

$\overset{(d)}{=} T^{\pi+1}(\sigma^u_q)[y] = (b) T^{\pi+1}(\sigma^u_q)[y] = (d) T^{\pi+1}(\sigma^u_q)[y] =$

$\overset{(e)}{=} T^{\pi+1}(\sigma^u_q)[y] = T^{\pi+1}(\sigma^u_q)[y]$  \hspace{1cm} \hspace{1cm} \hspace{1cm}

where:

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

(b) Induction hypothesis for $\pi'$.

(c) $\sim$, and $\pi_i$ must be covered by $\text{sum}_{p_i}$ since $\sigma^v_i \Rightarrow R^{\pi+1}_\pi$ and therefore $\sigma^v_q \Rightarrow \sum_{p_i} r(T_{\pi^{in}_i}[Y_i])$ and all reachability conditions in the summary are disjoint.

(d) Corollary 3.1.

(e) External induction hypothesis, since the depth of $CFG^{in}_{p_i}$ 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

\[
\Delta_{\text{Full}}{p_1, p_2} = (\text{ch}_{p_1, p_2}, \text{unch}_{p_1, p_2}, \text{termin}_{\text{ch}}_{p_1, p_2})
\]

where,

- \( \text{ch}_{p_1, p_2} \) is the set of visible states for which both procedures terminate with different final states.
- \( \text{unch}_{p_1, p_2} \) is the set of visible states for which both procedures either terminate with the same final states, or both do not terminate.
- \( \text{termin}_{\text{ch}}_{p_1, p_2} \) is the set of visible states for which exactly one procedure terminates.

Note that \( \text{ch}_{p_1, p_2} \cup \text{unch}_{p_1, p_2} \cup \text{termin}_{\text{ch}}_{p_1, p_2} \) covers the entire visible state space. The three sets are related to the state equivalence notions of Definition 2.3.1 as follows.

- \( \text{ch}_{p_1, p_2} \) is the set of the visible states that violate partial equivalence. It only captures differences between terminating paths.
- \( \text{termin}_{\text{ch}}_{p_1, p_2} \) is the set of visible states that violate mutual termination.
- \( \text{unch}_{p_1, p_2} \) 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:

\[
\text{ch}_{p_1, p_2} = \{ \{ x \mapsto 4 \} \}
\]
\[
\text{unch}_{p_1, p_2} = \{ \{ x \mapsto c \} \mid c \neq 2 \land c \neq 4 \}
\]
\[
\text{termin}_{\text{ch}}_{p_1, p_2} = \{ \{ x \mapsto 2 \} \}
\]

For input 2 the old version \( p_1 \) does not change \( x \), while the new version \( p_2 \) reaches an infinite loop, and therefore 2 is in \( \text{termin}_{\text{ch}}_{p_1, p_2} \). 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 \( \text{unch}_{p_1, p_2} \). For input 4, \( p_1 \) does not change \( x \), while \( p_2 \) changes \( x \) to 3, and therefore 4 is in \( \text{ch}_{p_1, p_2} \).

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

**Definition 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 $\neg U(p_1, p_2)$, respectively.

The algorithm presented in the next section is based on the notion of path difference, presented below. Recall that for a given path $\pi$, its path summary is the pair $(R_\pi, T_\pi)$ (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 $\pi_1$ and $\pi_2$ be finite paths in $CFG_{p_1}$ and $CFG_{p_2}$, respectively. Then the **Path Difference** of $\pi_1$ and $\pi_2$ is a triplet $(d, T_{\pi_1}, T_{\pi_2})$, where $d$ is defined as follows:

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

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 $\sigma$ that satisfies $d$, path $\pi_1$ is traversed from $\sigma$ in $CFG_{p_1}$ and path $\pi_2$ is traversed from $\sigma$ in $CFG_{p_2}$. Moreover, when starting from $\sigma$, the final state of $\pi_1$ will be different from the final state of $\pi_2$ (at least for one of the variables in $V^v_p$). If $d$ is satisfiable we say that $\pi_1$ and $\pi_2$ 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

---

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 \(\Delta_{p_1,p_2}\). 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. $match = \text{ComputeProcedureMatching}(P_1, P_2)$
2. $\text{FoundDiff}[(p_1, p_2)] = \emptyset$, for each $(p_1, p_2) \in match$
3. $workSet := \emptyset$
4. $newWorkSet := \{(p_1, p_2) \in match : p_1$ different syntactically from $p_2\}$
5. while $newWorkSet \neq workSet$
   6. $workSet := newWorkSet$
   7. $newWorkSet := workSet \cup \{(q_1, q_2) \in match : \exists (p_1, p_2) \in workSet \text{ s.t. } q_1 \text{ calls } p_1 \text{ or } q_2 \text{ calls } p_2\}$
8. end while
9. for each $(p_1, p_2) \in workSet$
   10. $\text{Diff}[(p_1, p_2)] := (false, false)$
   11. $\text{Sum}[p_1] := \emptyset$, $\text{Sum}[p_2] := \emptyset$
12. end for
13. for each $(p_1, p_2) \in match \setminus workSet$
   14. $\text{Diff}[(p_1, p_2)] := (false, true)$
   15. $\text{Sum}[p_1] := \emptyset$, $\text{Sum}[p_2] := \emptyset$
16. end for
17. while $workSet \neq \emptyset$
   18. $(p_1, p_2) := \text{ChooseNext}(workSet)$ \hspace{1cm} \text{\texttt{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 $g_2$, $\text{Diff}[(g_1, g_2)] = (*, true)$ then
      20. $\text{newDiff} := (false, true)$
   21. else
   22. $\text{Sum}[p_1] := \text{ModularSymbolicExecution}(p_1, \text{Sum})$
   23. $\text{Sum}[p_2] := \text{ModularSymbolicExecution}(p_2, \text{Sum})$
   24. $(\text{newDiff, newFoundDiff}) := \text{ConstProcDiffSum}((\text{Sum}[p_1], \text{Sum}[p_2], \text{Diff}[(p_1, p_2)])$)
   25. $\text{FoundDiff}[(p_1, p_2)] := \text{FoundDiff}[(p_1, p_2)] \cup \text{newFoundDiff}$
26. end if
27. if $\text{Diff}[(p_1, p_2)] \neq \text{newDiff}$ then
   28. $\text{Diff}[(p_1, p_2)] := \text{newDiff}$
   29. $workSet := workSet \cup \{(q_1, q_2) \in match : q_1 \text{ calls } p_1 \text{ or } q_2 \text{ calls } p_2\}$
30. end if
31. end while
32. return $(\text{Diff, FoundDiff})$
Algorithm 3.2 ConstProcDiffSum(sum_p1, sum_p2, oldDiff)

Input: Procedure summaries sum_p1, sum_p2 of procedures p_1, p_2, respectively, and oldDiff, previously computed ∆_{p_1,p_2}

Output: updated ∆_{p_1,p_2}, found_diff_{p_1,p_2}

1: (C(p_1,p_2), U(p_1,p_2)) := oldDiff
2: found_diff_{p_1,p_2} = ∅
3: for each (r_1,t_1) in sum_p1 do
4:     for each (r_2,t_2) in sum_p2 do
5:         diffCond := r_1 ∧ r_2 ∧ t_1 ≠ t_2
6:         if diffCond is SAT then
7:             found_diff_{p_1,p_2} := found_diff_{p_1,p_2} ∪ \{(diffCond, t_1, t_2)\}
8:         end if
9:     end for
10:     eqCond := r_1 ∧ r_2 ∧ t_1 = t_2
11:     if eqCond is SAT then
12:         U(p_1,p_2) := U(p_1,p_2) ∪ eqCond
13:     end if
14: end for
15: return ((C(p_1,p_2), U(p_1,p_2)), found_diff_{p_1,p_2})

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-
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:

  \[
  R_{i+1}^p = R_i^p \pi
  \]

- Second, to deal with the lack of knowledge of the output of \( g \), we introduce a set of uninterpreted functions \( UF_g = \{ UF_j^g | 1 \leq j \leq |V^g| \} \)\(^5\). The uninterpreted function \( UF_j^g(T^i[y]) \) replaces \( UK \) in \( T_{i+1}^p[y] \) (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 \( T_{i+1}^p[y] \) with:

\[
ITE(U(g_1, g_2)(T^i[x]), UF_{g_1, g_2}(T^i[x]), UF_{g_1}(T^i[x])).
\]

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

\[
ITE(U(g_1, g_2)(T^i[x]), UF_{g_1, g_2}(T^i[x]), UF_{g_2}(T^i[x])).
\]

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 \( T^i[x] \), 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 \( p_1, p_2 \) 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 \( \Delta_{p_1, p_2} = (false, x < 2 \lor x > 4) \). When symbolic execution of a procedure \( g \) reaches a procedure call \( p_1(a) \), 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.
void f1 (int & x) {
  if (x == 5) {
    abs (x);
  } else if (x == 0) {
    x = 0;
  } else return;
}

void f2 (int & x) {
  if (x == 5) {
    abs (x);
  } else if (x == 0) {
    x = 1;
  } else return;
}

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

Figure 3.1: Procedure versions in need of refinement

calling procedure g, we will perform:

\[
R_{\pi'}^{i+1} = R_{\pi}^i
\]

\[
\forall y_j \neq a. T_{\pi'}^{i+1}[y_j] = T_{\pi}^i[y_j]
\]

\[
T_{\pi'}^i[a] = ITE(T_{\pi}^i[a] < 0, -1, ITE(T_{\pi}^i[a] \geq 2, T_{\pi}^i[a],
ITE(T_{\pi}^i[a] < 2 \lor T_{\pi}^i[a] > 4, UF_{p_1,p_2}(T_{\pi}^i[a]), UF_{p_1}(T_{\pi}^i[a]))).
\]

3.5.2 Refinement

Using the described abstraction, the computed \(R_{\pi}, T_{\pi}\) may contain symbols of uninterpreted functions, and therefore so could \(\text{diffCond} = r_1 \land r_2 \land t_1 \neq t_2\) and \(\text{eqCond} = r_1 \land r_2 \land t_1 = t_2\) (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 \(\text{diffCond}\) to \(C(p_1, p_2)\) or \(\text{eqCond}\) 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 \(\text{abs}(5)\) cannot be zero. Therefore, we need to analyse \(\text{abs}\), even though it was not changed or affected.

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

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

$$\varphi \land \neg \left( \bigvee_{(r,t) \in \text{sum}_{g_k}} r(T^{i_{k-1}}_{\pi}[Y_k]) \right) \land V^v_g = T^{i_{k-1}}_{\pi}[Y_k].$$

where:

- $\varphi$ - restricts the paths traversed in $g_k$ to ones feasible under the call from $\pi'$.
- $\neg \left( \bigvee_{(r,t) \in \text{sum}_{g_k}} r(T^{i_{k-1}}_{\pi}[Y_k]) \right)$ - restricts the paths traversed in $g_k$ to ones not previously explored.
- $V^v_g = T^{i_{k-1}}_{\pi}[Y_k]$ - 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 $\pi$, add $(R_\pi, T_\pi)$ to $\text{sum}_{g_k}$, replace the old $\text{sum}_{g_k}$ with the new $\text{sum}_{g_k}$ in $\varphi$ 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 $\pi$ 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 $\text{abs}_1=\text{abs}_2=\text{abs}$ are empty, but it is known that both versions are identical (unmodified syntactically). We get (using symbolic execution and Algorithm 3.2) the $\text{diffCond}$ for $p_1$ and $p_2$:

$$\text{diffCond} = \left[ x = 5 \land \left( \text{ITE}(\text{true}, \text{UF}_{\text{abs}_1}, \text{abs}_2(x), \text{UF}_{\text{abs}_1}(x)) = 0 \right) \land \right. \left. \left[ x = 5 \land \left( \text{ITE}(\text{true}, \text{UF}_{\text{abs}_1}, \text{abs}_2(x), \text{UF}_{\text{abs}_2}(x)) = 0 \right) \land 0 \neq 1 \right] \equiv \left[ x = 5 \land \text{UF}_{\text{abs}_1}(x) = 0 \right] \equiv x = 5 \land x = 0 \right.$$  

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

$$\left[ x = 5 \land \left( \text{ITE}(x \geq 1, x, \text{ITE}(\text{true}, \text{UF}_{\text{abs}_1}, \text{abs}_2(x), \text{UF}_{\text{abs}_1}(x)) = 0) \land \right. \left. \left[ x = 5 \land \left( \text{ITE}(x \geq 1, x, \text{ITE}(\text{true}, \text{UF}_{\text{abs}_1}, \text{abs}_2(x), \text{UF}_{\text{abs}_2}(x)) = 0) \land 0 \neq 1 \right] \equiv \left[ x = 5 \land \left( \text{ITE}(x \geq 1, x, \text{UF}_{\text{abs}_1}, \text{abs}_2(x)) = 0 \right) \right) \equiv x = 5 \land x = 0 \right.$$  

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

35
Once a difference summary is computed, we can choose whether to refine the difference by exploring more paths in the individual procedures; or, if \textit{diffCond} or \textit{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 (\textsc{ModDiffRef}), and always delaying the refinement (\textsc{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 \textsc{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 \textsc{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 \textsc{SymDiff}. The algorithm presented in [16] is able to deal with loops and recursion; however, the algorithm is not fully implemented in \textsc{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 \textsc{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. \textsc{SymDiff} also has a mode that infers common invariants, as described 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.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>ModDiff</th>
<th>ModDiffRef</th>
<th>RVT</th>
<th>SymDiff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Const</td>
<td>0.545s</td>
<td>0.541s</td>
<td>4.06s</td>
<td>14.562s</td>
</tr>
<tr>
<td>Add</td>
<td>0.213s</td>
<td>0.2s</td>
<td>3.85s</td>
<td>14.549s</td>
</tr>
<tr>
<td>Sub</td>
<td>0.258s</td>
<td>0.308s</td>
<td>5.01s</td>
<td></td>
</tr>
<tr>
<td>Comp</td>
<td>0.841s</td>
<td>0.539s</td>
<td>5.19s</td>
<td></td>
</tr>
<tr>
<td>LoopSub</td>
<td>0.847s</td>
<td>1.179s</td>
<td></td>
<td>F</td>
</tr>
<tr>
<td>UnchLoop</td>
<td>F</td>
<td>2.838s</td>
<td></td>
<td>F</td>
</tr>
<tr>
<td>LoopMult2</td>
<td>1.666s</td>
<td>1.689s</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>LoopMult5</td>
<td>F</td>
<td>3.88s</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>LoopMult10</td>
<td>F</td>
<td>9.543s</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>LoopMult15</td>
<td>F</td>
<td>21.55s</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>LoopMult20</td>
<td>F</td>
<td>49.031s</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>LoopUnrch2</td>
<td>0.9s</td>
<td>0.941s</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>LoopUnrch5</td>
<td>1.131s</td>
<td>1.126s</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>LoopUnrch10</td>
<td>1.147s</td>
<td>1.168s</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>LoopUnrch15</td>
<td>1.132s</td>
<td>1.191s</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>LoopUnrch20</td>
<td>1.157s</td>
<td>1.215s</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

(a) Semantically equivalent

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>ModDiff</th>
<th>ModDiffRef</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoopSub</td>
<td>1.187s</td>
<td>2.426s</td>
</tr>
<tr>
<td>UnchLoop</td>
<td>F</td>
<td>8.053s</td>
</tr>
<tr>
<td>LoopMult2</td>
<td>3.01s</td>
<td>3.451s</td>
</tr>
<tr>
<td>LoopMult5</td>
<td>F</td>
<td>5.914s</td>
</tr>
<tr>
<td>LoopMult10</td>
<td>F</td>
<td>10.614s</td>
</tr>
<tr>
<td>LoopMult15</td>
<td>F</td>
<td>14.024s</td>
</tr>
<tr>
<td>LoopMult20</td>
<td>F</td>
<td>25.795s</td>
</tr>
<tr>
<td>LoopUnrch2</td>
<td>2.157s</td>
<td>2.338s</td>
</tr>
<tr>
<td>LoopUnrch5</td>
<td>2.609s</td>
<td>3.216s</td>
</tr>
<tr>
<td>LoopUnrch10</td>
<td>2.658s</td>
<td>3.481s</td>
</tr>
<tr>
<td>LoopUnrch15</td>
<td>2.835s</td>
<td>3.446s</td>
</tr>
<tr>
<td>LoopUnrch20</td>
<td>3.185s</td>
<td>3.342s</td>
</tr>
</tbody>
</table>

(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).
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 $5 \leq a < 7 \land b = 5$), 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


The central contributions of this work are:

- The algorithm is unique in that it generates upper and lower bounds of semantic differences.

- The algorithm uses abstraction and instrumentation to enable better balance between efficiency and accuracy.

- The authors develop the idea of symbolic modularization.

- They present an algorithm for modular reasoning of different code versions.

- They formulate a method to define and analyze the behavior of the algorithm.

- They validate the performance of the algorithm using a variety of test cases.

- They conclude that the algorithm is effective and scalable.
התקציר

הנחיות כבונת בלוקים, תיאור וtesy התשובה מתכונתית על ידי בטוץ שניים בין זרח. ילי יכלן
ל琮י את הנ绾ים בחכמה של שבית ראוזה של תכנית, יי מסיים מבורך בדית לופית מקו.
הנחיות נכתנה, תקן של החשוב מסכום לשגרה הונשא (ובגדה בוברב) בקטיות של הטלדליים..sensor
לиласפ נכות לשגרה החשווה. יעקב, ומכידים 작업 מ越し דיבור לעפיות לתחום. באומן
ככל שמעתה או ליאת בידור, לקם הטלה מספר. עם זה, ומכידם אגניצים לכל הת אהוב
מצב נורות הבלתיים עגבי המופיים המבוקשים מרהיבות את הגראות והיוסים, ומכבעים
ה.datatablesים עבורי המגיניים הסופיים של התוכנה. התכנית העיקרית היא את הנבנאים יי מדומיין ממסק
לתרבו המבוקשים עזר תכנית באומן. התוכנית והשעיה העקרו שה(Canvas פונה התכנית, יכ
ות מקובב לברד בן שaviest המסטת היי מסקנס (אף עזר לו) אמצע על התוכנית
בכל זה הפך להсталות יי דמידיקות לתוך. ולא נובע שהאנטיות ליבו מהגרות התוכנית
אות הלאונ ההתגזרת המייעץ vortex קוריביצי של שטיוטת לכל את המ Derneği המנסחת
ות_cls ידיע. אלא במכות שאונאנסים יסתיום, יא תקנות פיתחוניות בדחל, יי.
הלספאס את תכל על כלכל התאצות הינם.

הנתון התוכנית של להתקף לשוח הקלטים של כל כל שגרת מאמצי (אני מחלק שימוקית
הממנים את הנקט של שגרת מתה אחר,ה) מבית עובד שגרה. קולטמ בורה, התכנית של השגרה
שונה, קולטמ בורה הקטל של השגרה (אם עוזרות) זיוא ששתית אינ עזרת, קולטמ בורה
לא יידוע группа יי הבול.

אمواد הלאונהתים יכלו עבידי לע כל שתי התוכניות, אבל אז פועל הצבע על העלה הרביה יי כביר
 nettד לסכוםือ יי התוכניות, יי תקן הביא לתוך קבוצת שגרה הנקט בזבוב בין
טוטל הלספאס של הנקט והנקט קוריביצי את מועטי המרכזים ביאור מוסק שגי התוכנית, טוטלüm עבידי
ה牵挂ת. לבוש שלון או מתחים קר ארויים בתוכנים המשג運用ים בוטים. הגראות שגל הזה
מודולרי של התוכנית ליום של שגרה. שגרה התוכנית מתוכן אינ מועתקות, 일본 או אחר
משתמשים בסכום שלבח.

לעמל הייעוץ, אני מיציגים שכר החיבור באסיגרתוע (בעזרות פונקציות לא מפורשות) שינק
עלרל לפי הדירוג. אני מאריסים כיז תקן של החשוב במקיקות אל מפורשות שימוקית, כי לובא
את הiedad על ספרות מדינית, הוא נושאי יא כולם מידיים על התוכנית העברית של השגרה.
הלאונהתים של לעמל מעילת לעיל, הלח מתחים הסכום והérique בין הגראות, עד ללקודוס
הנחיות לשגרה. מאחר שלודיות בין דאצ הסכום ביאור מוסק, אני מבצעים עזר (מעילת-
לпромה) מפשית מתכונת הקראיה הנכונה, כי לייצר שיכום שית הצעה לע רכז השגרה הקראיה.
המחקר מבצע בחנויות של פרופסור אורנה גרומברג, בפקולטה להנדסת מחשבים. חלק מה呼吁אות ב بيانה זה תורגמו כמונות פנים המחבר ושוחזרו ליומן המחקר בכל במחילה מחומש.

מחק מחנו של המחבל:


תודה

ברצון להודות לארנה גרומברגشر住房和מחבל המחטימוח, מקור ושם והברה במחילה המחטימוח. בנושף על מהדידה תל אביב קורוינגר, עופר שפריכמן, עופר גוטמן וכל האנשים שעזרו לי לארך הדר.

אני מודה לברנחי להמלצות דרכי קל. מליום וברנחי במחק המחבר ولمعباد נסיבות סביר על התמימה הכפיפה הקדימה בחשילות.
ניתוח מודולרי מונחה-דרישה של הבדלים סמנטיים בין גרסאות של תוכנה

תפקיד על מחקר

לשם مليוי חלקי של הדרישת ל컵לת התואר
מניחר למידות ב_cmosע המתחב

אנה טרוסטנרקי

הוות ל텐ט הსכני - מכון טכנולוגי לישראל
תמונת 5777 חופה יולי 2017
ניתוח מודולרי מונה-דרישה של המבזק

صومטיים בינ-גרסאות של תוכנה

אנה טורסנטסקי