Abstract Semantic Differencing for Numerical Programs

Nimrod Partush
Technion
nimi@cs.technion.ac.il

Eran Yahav
Technion
yahave@cs.technion.ac.il

Abstract

We address the problem of computing semantic differences between a program and a patched version of the program. Our goal is to obtain a precise characterization of the difference between program versions, or establish their equivalence when no difference exists.

We focus on computing semantic differences in numerical programs where the values of variables have no a-priori bounds, and use abstract interpretation to compute an over-approximation of program differences. Computing differences and establishing equivalence under abstraction requires abstracting relationships between variables in the two programs. Towards that end, we first construct a *correlating program* in which these relationships can be tracked, and then use a *correlating abstract domain* to compute a sound approximation of these relationships. To better establish equivalence between correlated variables and precisely capture differences, our domain has to represent non-convex information. To balance precision and cost of this representation, our domain may over-approximate numerical information as long as equivalence between correlated variables is preserved.

We have implemented our approach in a tool called DIZY, built on the LLVM compiler infrastructure and the APRON numerical abstract domain library, and applied it to a number of challenging real-world examples, including programs from the GNU core utilities, Mozilla Firefox and the Linux Kernel. Our evaluation shows that DIZY often manages to establish equivalence, describes precise approximation of semantic differences when difference exists, and reports only a few false differences.

1. Introduction

Understanding the semantic difference between two versions of a program is invaluable in the process of software development. A developer making changes to a program is often interested in answering questions like: (i) did the patch add/remove the desired functionality? (ii) does the patch introduce other, *unexpected*, behaviors? (iii) which regression tests should be run? Answering these questions manually is difficult and time consuming.

Semantic differencing has received much attention in classical work (e.g., [9–11]) and has recently seen growing interest for various applications ranging from testing of concurrent programs [4], understanding software upgrades [14], to automatic generation of security exploits [2].

Existing Techniques Existing techniques mostly offer under-approximating solutions, the prominent of which is regression testing which provides limited assurance of behavior equivalence while consuming significant time and compute resources. Other approaches for computing semantics differences [20, 21] rely on symbolic execution techniques, may miss differences, and generally unable to prove equivalence. Previous work for equivalent checking [8] rely on unsound bounded model checking techniques to prove (input-output) equivalence of two closely related numerical programs, under certain conditions.

We present an approach based on abstract program interpretation [6] for a *sound*, succinct representation of changed program behaviors and proving equivalence. Our method focuses on abstracting relationships between variables, and therefore behaviors, in both versions allowing us to achieve a precise description of difference and prove equivalence while ignoring other program information which may encumber a traditional analysis but is less relevant in our setting

Problem Definition We define the problem of *semantics differencing* as follows: Given a pair of programs (P, P') which agree on the number and type of inputs, for every execution π of P that originate from an input i and a corresponding execution π' of P' that originates from the *same input* i our goal is:

- Check whether π and π' agree on output i.e., are output-equivalent.
- In case of difference in behavior, provide a description of difference.

We intentionally define the notion of input and output equivalence loosely at this point, and we discuss several realizations of these in later sections.

To answer the question of semantic differencing for infinite-state programs, we employ abstract interpretation. Though the notion of difference is well defined in the concrete case, defining and soundly computing it under abstraction is challenging as:

- Differencing requires correlation of *different program executions* meaning the abstraction must be able to capture input-equivalent executions, and distinguish ones that are not input-equivalent.
- Establishing equivalence of abstract output values does not entail equality between the concrete output value they represent.

To address these challenges, we introduce two new concepts: (i) a *correlating program*, a single program $P \bowtie P'$ that captures the behaviors of both P and P' in a way that facilitates abstract interpretation; (ii) a *correlating abstract domain*, tracking relationships between variables in P and variables in P' by tracking equivalences in $P \bowtie P'$.

Correlating Program Abstracting relationships allows us to maintain focus on difference while omitting (whenever necessary for scalability) parts of the behavior that does not entail difference. In order to monitor these relationships we created a *correlating program* which captures the behavior of both the original program and its patched version. Instead of designing a correlating semantics that is capable of co-executing two programs, we chose to automatically construct the correlating program such that we can benefit from the use of standard analysis frameworks for analyzing the resulting program. Another advantage of this new construct, is that you may apply other methods for equivalence checking directly

```
int sign(int x) {
  int sgn;
  if (x < 0)
    sgn = -1
  else
    sgn = 1
 return sgn
}
```

```
int sign'(int x) {
  int sgn';
  if (x' < 0)
    sgn' = -1
  else
    sgn' = 1
  if (x'==0)
    sgn' = 0
 return sgn'
}
```

Figure 1: Two simple implementations of the *sign* operation.

$x.x'$ constraints	sgn	sgn'
$x < 0$	$sgn \mapsto -1$	$sgn' \mapsto -1$
$x = 0$	$sgn \mapsto 1$	$sgn' \mapsto 0$
$x > 0$	$sgn \mapsto 1$	$sgn' \mapsto 1$

Figure 2: behavior of $sign$ and $sign'$.

on it [21] as the correlation allows for a finer-grained equivalence checking (between local variables and not only output).

Correlating Abstraction Our abstraction holds data of both sets of variables, joined together and is initialized to hold equality over all matched variables. This means we can reflect relationships without necessarily knowing the actual value of a variables (we can know that $x_{old} = x_{new}$ even though actual values are unknown). We ran our analysis over the correlating program while updated the domain to reflect program behavior. Since some updates may result in non-convex information (e.g. taking a condition of the form $x \neq 0$ into account), our domain has to represent non-convex information, at least temporarily. We address this by working with a powerset domain of a convex representation with partitioning according to equivalence criteria to avoid exponential blowup. Our domain may over-approximate numerical information as long as equivalence between correlated variables is preserved.

1.1 Main Contributions

The main contributions of this paper are as follows:

- we present a method for abstract interpretation of a pair of programs (P, P') for *sound* semantic equivalence and differencing by abstracting direct relationships between (P, P') variables in a partially disjunctive domain. We describe a partitioning technique for state reduction and scaling. We define a widening operator for abstracting unbound paths in our domain.
- we phrase a new technique for syntactically interleaving a pair of programs (P, P') for the creation of a *correlating program* $P \bowtie P'$ which contains the semantics of both programs. We propose an analysis over the program for characterizing program equivalence and difference, based on the aforementioned abstraction, given the properties of the correlating program which aligns (P, P') executions.
- We have implemented our approach in a tool based on the LLVM compiler infrastructure and the APRON numerical abstract domain library, and evaluated it using select patches from open-source software including GNU core utilities, Mozilla Firefox, and the Linux Kernel. Our evaluation shows that the tool often manages to establish equivalence, reports useful approximation of semantic differences when differences exists, and reports only a few false differences.

2. Overview

In this section, we provide an informal overview of our approach using a simple example.

Consider the two simple example programs of Fig. 1, inspired by an example from [22]. For these two programs, we would like to establish that the output of $sign$ and $sign'$ only differs in the case where $x = 0$ and that the difference is $sgn = 1 \neq sgn' = 0$. A precise characterization of behavior is show in Fig. 2.

Separate Analysis is Unsound As a first naive attempt to achieve such a description, one could try to analyze each version of the program separately and compare the (abstract) results. However, this is clearly unsound, as equivalence under abstraction does not entail concrete equivalence. For example, using a interval analysis [7] would yield that in both programs the value of `sgn` ranges in the same interval $[-1, 1]$, missing the fact that $sign$ never returns the value 0.

Establishing Equivalence under Abstraction To establish equivalence under abstraction, we need to abstract *relationships between the values of variables* in $sign$ and $sign'$. Specifically, we need to track the relationship between the values of `sgn` in both versions. Tracking relationships between variables of two program versions requires a joint representation in which these relationships can be tracked.

Correlating Program As a first step, our goal is to construct a program in which we can track relationships between variables of $sign$ and $sign'$. One naive option would be to construct a program $P; P'$ by sequentially composing the two program versions (as in [21]). However, establishing equivalence between variables in such program requires an analysis to precisely track paths across P so they can be later compared to the corresponding paths in P'. Fig. 3 informally illustrates the paths that have to be correlated through $sign; sign'$ to track the relationship between `sgn` and `sgn'`. To establish equivalence between `sgn` and `sgn'` an analysis must separately track paths in $sign$ that lead to different values of `sgn` until it reaches a corresponding path in $sign'$ which produces an equivalent value for `sgn'`.

Intuitively, establishing equivalence using the sequential composition $P; P'$ might require full path sensitivity, leading to an inherently non-scalable solution. Further, in the presence of loops and widening, applying widening separately to the loops of P and to those of P' may completely jeopardize any attempt to maintain relationships under abstraction.

To address these challenges, we construct a *correlating program* $P \bowtie P'$ where operations of P and P' are carefully interleaved to keep execution of both versions close to lock-step progress. Fig. 4 shows the correlating program for the programs of Fig. 1. The programs were transformed to a guarded command language form to allow for interleaving. Using the correlating program, we can directly track the relationship between `sgn` in $sign$ and its corresponding variable `sgn'` in $sign'$. Section 6 provides more details on the construction of a correlating program.

Correlating Abstract Domain To analyze a given correlating program, we introduce a *correlating abstract domain* that tracks relationships between corresponding variables in P and P' by tracking relationship in the correlating program $P \bowtie P'$. Unfortunately, any domain with convex constraints will still fail to capture the precise relationship between `sgn` and `sgn'`. For example, using the polyhedra abstract domain [7], the relationship between `sgn` and `sgn'` in the correlating program would be lost, leaving only the trivial $\langle 1 \geq sgn \geq -1, 1 \geq sgn' \geq -1 \rangle$ constraint. Although the result soundly reports a difference, we still know nothing of the difference between the programs.

An obvious, but prohibitively expensive, solution to the problem is to use disjunctive completion, moving to a powerset domain in which every abstract state is a set of convex objects (e.g., set of polyhedra). A state in such domain is a set of convex abstract

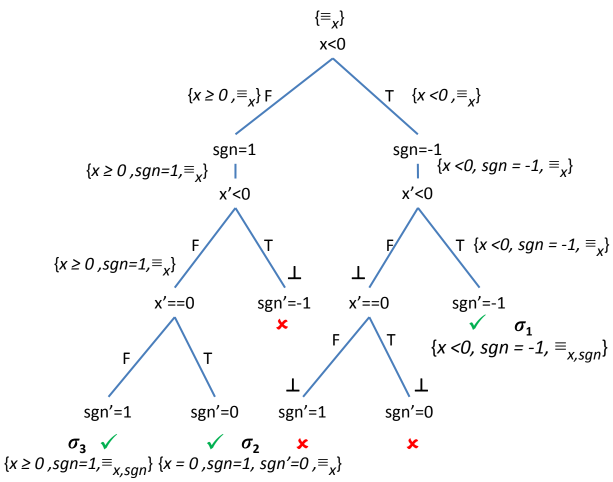

Figure 3: Joint $sign; sign'$ analysis

representations (e.g., polyhedra [7] or octagon [17]). For example, analyzing $sign \bowtie sign'$ using a powerset domain would yield:

$$\sigma^1_\times = \{x = x' < 0, sgn = sgn' \mapsto -1\}$$
$$\sigma^2_\times = \{x = x' = 0, sgn \mapsto 1, sgn' \mapsto -1\}$$
$$\sigma^3_\times = \{x = x' > 0, sgn = sgn' \mapsto 1\}$$

However, using such domain would significantly limit the applicability of the approach. The desirable solution is a partially disjunctive domain, in which only certain disjunctions are kept separate during the analysis, while others are merged. The challenge in our setting is in keeping the partition fine enough such that equivalence could be preserved, without reaching exponential blowup.

As the goal of work is to distinguish equivalent from differencing behaviors, using equivalence as criteria for merging paths is apt. The partitioning will abstract together paths that hold equivalence for the same set of variables, allowing for a maximum of $2^{|VC|}$ disjunctions in the abstract state, where VC is the set of correlated variables. So far we have implicitly defined VC as a correlation between P, P' input and outputs, but our approach is in fact parameterized by this matching, allowing for any P variable to be matched with any of P' which has the potential to provide a more precise result (in the cost of scaling) or alternatively provide a more coarse, scalable result by allowing less variables or only certain equivalence classes of 2^{VC}. A formal definition and discussion of VC is found in Section 4.

For example partitioning the result of Fig. 3 according to our criteria would abstract behaviors s_1 and s_3 together, as they hold equivalence for sgn. The merge would abstract away data regarding x and represent sgn as the $[-1, 1]$ interval, losing precision but gaining reduction in state size. This loss of precision is acceptable as it is complemented by the offending state s_2.

$$\sigma^1_\times = \{x = x', sgn = sgn' \mapsto [-1, 1]\}$$
$$\sigma^2_\times = \{x' = 0, sgn \mapsto 1, sgn' \mapsto -1\}$$

To gain a reduction of state size, we must perform partitioning dynamically during analysis. This cannot be achieved using a sequential composition $P; P'$. Looking at Fig. 3 we see that equivalence holds only at final states. Intuitively, this is because an operation in P having to "wait" for its equivalent operation to occur in P'. To overcome this, our correlating program $P \bowtie P'$ interleaves P and P' commands in an optimized manner, and informs

```
int sign(int x) {
  int x' = x;
  guard g1 = (x < 0);
  guard g1' = (x' < 0);
  int sgn;
  int sgn';
  if (g1) sgn = -1;
  if (g1') sgn' = -1;
  if (!g1) sgn = 1;
  if (!g1) sgn' = 1;
  guard g2' = (x' == 0);
  if (g2') sgn' = 0;
}
```

Figure 4: Correlating program $sign \bowtie sign'$.

```
int sum(int arr[], unsigned len) {
  int result = 0;
  for (unsigned i = 1; i < len; i+=2)
    result += arr[i];
 return result;
}
```
```
 int sum'(int arr[], unsigned len) {
   int result = 0;
   unsigned i = 0;
   while (i + 1 < len) {
     i++;
     result += arr[i];
     i++;
   }
   return result;
 }
```

Figure 5: Two equivalent versions of a simple looping program for partial array summation.

the analysis when programs have reached a point in which correlation can be established. The choice of *correlation points* (denoted CP) is done during the construction of the correlating program. We describe the specifics of creating $P \bowtie P'$ in Section 6 and only briefly note that the interleaving is chosen according to a syntactic diff process over a guarded command language version of the programs.

Windening Although we achieved a reduction in state size using partitioning, we have yet to account for programs with loops. Handling loops is where most previous approaches fall short [8, 15, 20, 21]. To overcome this, we define a widening operator for our domain, based on the convex sub-domain widening operator. The main challenge here, as our state is a set of convex objects, is finding an optimal pairwise matching between objects for a precise widened result. Ideally, we would like to pair objects that adhere to the same "looping path" meaning we would like to match a path π_i's abstraction with a path π_{i+1} that results from taking another step in the loop. This requires encoding path information along with the sub-state abstraction. This information is acquired by keeping guard values explicitly, as they appear in our correlating program, inside the state. As guard values (true or false) reflect branch outcomes, they can be used to match sub-states that advanced on the loop by matching their guard values.

We note that the correlating program is crucial to maintaining equivalence over loops. To demonstrate this we perform the simple exercise of checking equivalence of a small looping program with itself. Consider the array summation program in Fig. 5. Equivalence for these two small programs cannot be established soundly by approached based on under approximation. To emphasize the importance of the correlating program, we will first show the result

```
int sum(int arr[], unsigned len) {
  unsigned len' = len;
  int arr'[] = arr;
  int result = 0;
  int result' = 0;
  {
    unsigned i = 1;
    unsigned i' = 0;
l:  guard g = (i < len);
l': guard g' = (i' + 1 < len');
    if (g') i'++;
    if (g) result += arr[i];
    if (g') result' += arr'[i'];
    if (g') i'++;
    if (g) i+=2;
    if (g) goto l;
    if (g') goto l';
  }
}
```

Figure 6: $sum \bowtie sum'$

of an analysis of $sum; sum'$ which will be:

$$\sigma^1_\times = \{len = len' \leq 1, result = result' \mapsto 0\}$$
$$\sigma^2_\times = \{len = len' > 1\}$$

This loss of equivalence occurred due to the inability precisely track the relationship of $result$ and $result'$ over $sum; sum'$. As we widened the first loop to converge, all paths passing through that loop were merges together, losing the ability to be "matched" with the second loop waiting further down the road. Performing the same analysis on $sum \bowtie sum'$ instead as seen in Fig. 6, allows maintaining equivalence, as the loops are interleaved correctly to allow establishing $result = result'$ as a loop invariant, surviving the widening process to prove equivalence at the end as the result would be: $\sigma^1_\times = \{result = result'\}$. We note that we explicitly assume equivalence in array content for sum and sum'.

3. Preliminaries

We use the following standard concrete semantics definitions for a program:

A program location $loc \in Loc$, also referred to as label denoted lab, is a unique identifier for a certain location in a program corresponding to the value of the program counter at a certain point in the execution of the program. We also define two special labels for the start and exit locations of the program as $begin$ and fin respectively.

Given a set of variables Var, a set of possible values for these variables Val and the set of locations Loc, a *concrete program state* is a tuple $\sigma \triangleq \langle loc, values \rangle \in \Sigma$ mapping the set of program variables to their concrete value at a certain program location loc i.e. $values : Var \rightarrow Val$. The set of all possible states of a program P is denoted Σ_P.

We describe an imperative program P, as a tuple $(Val, Var, \rightarrow, \Sigma_0)$ where $\rightarrow: \Sigma_P \times \Sigma_P$ is a transition system which given a concrete program state returns the following state in the program and Σ_0 is a set of initial states of the program. Our formal semantics need not deal with errors states therefore we ignore crash states of the programs, as well as inter-procedural programs since our work deals with function calls by either ignoring them when equivalence was proven or by inlining them (we exclude recursion for now).

A program trace $\pi \in \Sigma^*_P$, is a sequence of states $\langle \sigma_0, \sigma_1, ... \rangle$ describing a single execution of the program. Each of the states corresponds to a certain location in the program where the trace originated from. Every program can be described by the set of all possible traces for its run $[\![P]\!] \subseteq \Sigma^*$. We refer to these semantics as concrete state semantics. We also define the following standard operations on traces:

- $label : \Sigma_P \rightarrow Lab$ maps a state to the program label at which it appears.
- $last : \Sigma^*_P \rightarrow \Sigma_P$ returns the last state in a trace.
- $pre : [\![P]\!] \rightarrow 2^{\Sigma^*_P}$ for a trace π is the set of all prefixes of π.
- $states : [\![P]\!] \rightarrow 2^{\Sigma_P}$ for a trace π is the set of actual states π is composed of.

We shortly describe a *product state* $\sigma_\times \in \Sigma_{P \times P'}$ as a pair of states $\langle \sigma, \sigma' \rangle$, a *product program* $P \times P'$ as a product of the transition systems of the underlying programs and *product trace* as a sequence of produce states.

4. Concrete Semantics

In this section, we define the notion of concrete difference between programs, based on a standard concrete semantics.

4.1 Concrete State Differencing

Comparing two programs P and P' under concrete semantics means comparing their *traces*. Towards that end, we first define the difference between two concrete states.

Intuitively, given two concrete states, the difference between them is the set of variables (and their values) where the two states map corresponding variables to different values. As variable names may differ between programs, we parameterize the definition by a mapping that establishes a correspondence between variables in P and those in P'. Concrete state differencing only compares values of corresponding variables.

Variable Correspondence A variable correspondence $VC \subseteq Var \times Var'$, is a partial mapping between two sets of program variables, those of P and those of P'. The variable correspondence mapping can be taken as input from the user however, our evaluation indicates that is often sufficient to use a standard name-based mapping for patched versions of programs: $VC_{EQ} \triangleq \{(v, v') | v \in Var \wedge v' \in Var' \wedge name(v) = name(v')\}$.

Concrete State Delta Given two concrete states $\sigma^\natural \in \Sigma^\natural_P, \sigma'^\natural \in \Sigma^\natural_{P'}$ and a variable correspondence mapping VC, we define the concrete state delta $\triangle_S(\sigma^\natural, \sigma'^\natural) : \Sigma^\natural_P \times \Sigma^\natural_{P'} \rightarrow 2^{Var \times Val}$:

DEFINITION 1. *Given tow concrete sates $\sigma^\natural \in \Sigma_P$, $\sigma'^\natural \in \Sigma_{P'}$, and a correspondence mapping VC, the* concrete state delta *is defined as:*

$$\triangle_S(\sigma^\natural, \sigma'^\natural) \triangleq \{(v, val) | (v, v') \in VC \wedge \sigma^\natural(v) = val \neq \sigma'^\natural(v')\}$$

Informally, $\triangle_S$ means the "part of the state $\sigma^\natural$ where corresponding variables do not agree on values (with respect to $\sigma'^\natural$)". Note that $\triangle_S$ is not symmetric. In fact, the direction in which $\triangle_S$ is used has meaning in the context of a program P and a patched version of it P. We define $\triangle^-_S = \triangle_S(\sigma^\natural, \sigma'^\natural)$ which means the values of the state that was "removed" in P' and $\triangle^+_S = \triangle_S(\sigma'^\natural, \sigma^\natural)$ which stands for the values "added" in P'. When there is no observable difference between the states we get that $\triangle^+_S(\sigma^\natural, \sigma'^\natural) = \triangle^-_S(\sigma^\natural, \sigma'^\natural) = \emptyset$, and say that the states are *equivalent* denoted $\sigma \equiv \sigma'$.

EXAMPLE 1. *Consider two concrete states $\sigma^\natural = (x \mapsto 1, y \mapsto 2, z \mapsto 3)$ and $\sigma'^\natural = (x' \mapsto 0, y' \mapsto 2, w' \mapsto 4)$ and using VC_{EQ} then $\triangle^-_S = (x \mapsto 1)$ since x and x' match and do not agree on value, y and y' agree (thus are not in delta) and z' is not in VC_{EQ}. Similarly, $\triangle^+_S = (x' \mapsto 0)$.*

```
void foo(unsigned x) {          void foo'(unsigned x) {
    unsigned i = 0;                 unsigned i = 0;
lab:guard g = (i >= x);         lab:guard g = (i >= 2*x);
    if (g) return;                  if (g) return;
    ...                             ...
    i++;                            i++;
    goto lab;                       goto lab;
}                               }
```

Figure 7: Two simple guarded versions of a loop procedure

We now use our notion of concrete state difference to define the difference between concrete program traces. Our goal is to compare traces that are input-equivalent and check whether they are output-equivalent. Therefore, we are interested in comparing traces that originate from equivalent input states. To differentiate traces, we need to compare the states along each trace, but which states should we compare? this is not a trivial question since traces can vary in length and order of states. We need a mapping for choosing the states to be differentiated within the two traces.

Trace Diff Points Given two traces $\pi \in [\![P]\!]$ and $\pi' \in [\![P']\!]$ that originate from equivalent input states, we define a trace index correspondence relation named *trace diff points* denoted DP_π as a matching of indexes specifying states where concrete state delta should be computed. Formally, $DP_\pi \subseteq \{(i,i')|0 \leq i \leq |\pi|, 0 \leq i' \leq |\pi'|\}$. The question of supplying this matching, in a way that results in meaningful delta, is not a trivial one, we delay this discussion until we define the trace delta.

Trace Delta Now that we have a way of matching states to be compared between two traces, we define the notion of trace difference:

DEFINITION 2. *Given two traces $\pi \in [\![P]\!]$ and $\pi' \in [\![P']\!]$ that originate from equivalent input states, and a trace index correspondence DP_π we define the trace delta $\triangle_T(\pi,\pi') : [\![P]\!] \times [\![P']\!] \to (\mathbb{N} \to 2^{Var \times Val})$ as state differentiations between all corresponding states in π and π':*

$$\triangle_T(\pi,\pi') = \{(i, \triangle_S(\sigma_i^\natural, \sigma_{i'}'^\natural)) \mid (i,i') \in DP_\pi, \triangle_S(\sigma_i^\natural, \sigma_{i'}'^\natural) \neq \emptyset\}$$

That is, for every $(i,i') \in DP_\pi$ such that $\triangle_S(\sigma_i^\natural, \sigma_{i'}'^\natural) \neq \emptyset$, $\triangle_T$ will contain the mapping $i \mapsto \triangle_S(\sigma_i^\natural, \sigma_{i'}'^\natural)$, thus the result will map certain states in π to their state delta with the corresponding state in π' (deemed interesting by DP_π). Since $\triangle_T(\pi,\pi')$ is based on state difference, we define $\triangle_T^+$ and $\triangle_T^-$ similarly to their underlying states difference operations.

One possible choice for DP_π is the endpoints of the two traces $\{(n,n')\}$ (assuming they are finite) meaning differentiating the final states of the executions or formally: $\triangle_n^- \triangleq \triangle_n(\pi,\pi') = \{n \mapsto \triangle_S(\sigma_n^\natural, \sigma_{n'}'^\natural)\}$ (we will also be interested in $\triangle_n^+$). The final state delta may not always be sufficient for truly describing the difference between traces as according to our definition of difference, we would be interested in checking difference at intermediate locations in the program (that emit output or check assertions). This can perhaps be achieved by instrumenting the semantics such that the state contains all "temporary" values for a variable along with the trace index (program location is not sufficient here as a trace can loop over a certain location). This solution encumbers the analysis with the addition of temporary variables and substantially complicates the selection of VC as it requires relating all of these temporary, indexed, variables. Such a correspondence may be extremely hard to produce. Also the number of variables here can range up to the length of the trace (which may be unbound). Finding a DP which allows correct differentiation is a daunting task as traces of separate (although similar) programs can vastly differ.

EXAMPLE 2. *Consider the two program versions shown in Fig. 7 and the following traces generated from the input $x = 2$: $\pi = \langle(x \mapsto 2, i \mapsto 0, g \mapsto 0), (x \mapsto 2, i \mapsto 1, g \mapsto 0), (x \mapsto 2, i \mapsto 2, g \mapsto 1)\rangle$ and $\pi' = \langle(x \mapsto 2, i \mapsto 0, g \mapsto 0), (x \mapsto 2, i \mapsto 1, g \mapsto 0), (x \mapsto 2, i \mapsto 2, g \mapsto 0), (x \mapsto 2, i \mapsto 3, g \mapsto 0), (x \mapsto 2, i \mapsto 4, g \mapsto 1)\rangle$, we see that even in this simple program, finding a correlation based on traces alone is hard. Instead, if one uses* program location *as a means of correlation, one can produce a result that describes how values of i range differently in the new version P. If we look at all the possible values for i at label lab and differentiate them (as a set) from the values in the patched version (in the same location), we get a meaningful result that i in the patched version can range from $x+1$ up to $2x$.*

4.2 Differencing at Program Labels

Here, we will formally describe the choosing of DP based on program locations.

Trace Delta using Program Labels Given two traces (π,π') and two program labels (l,l') we define a trace delta based on all states that are labeled l in P and l' in P'. First we define π_l as a subsequence of π where only states that are labeled l were chosen ($\pi'_{l'}$ is defined similarly). Next, we denote $\triangle_L(\pi_l,\pi'_{l'})$ as a means for comparing these sequences. As $\pi_l, \pi'_{l'}$ may vary in length and order, we cannot simply define it as applying $\triangle_S$ on each pair of states in $(\pi_l,\pi'_{l'})$ by order. In fact, $\triangle_L$ can be defined in different way to reflect different concepts of difference, for instance, it can be defined as the differentiating the last states of π_l and $\pi'_{l'}$ (assuming they are both finite) to reflect we are only interested in the final values in that location. We chose to define $\triangle_L$ as the difference between *the set of states* which appear in π_l against the set of those in $\pi'_{l'}$.

DEFINITION 3. *Given two traces π,π' and two program labels l,l' we define the trace delta based on labels as:*

$$\triangle_L(\pi_l,\pi'_{l'}) \triangleq \{\sigma^\natural \in states(\pi_l) | \neg\exists\sigma'^\natural \in states(\pi'_{l'}) \cdot \sigma^\natural \equiv \sigma'^\natural\}.$$

Meaning, all states that exist at label l in P but cannot be matched with any state existing at l' in P'.

EXAMPLE 3. *Consider Fig. 7, for π,π' that originate from $x = 2$ then $\triangle_L(\pi_{lab},\pi'_{lab'}) = \emptyset$ and $\triangle_L(\pi'_{lab'},\pi_{lab}) = \{(i' \mapsto 3), (i' \mapsto 4)\}$. We see that this notion of $\triangle$ indeed captures a useful description of difference.*

The problem of choosing DP is now reduced to the matching of labels as the trace indexing correspondence DP_π defined in Definition 4.1 is induced by the definition over labels. Since we need to differentiate sets of states belonging to a certain program label, we require a correspondence of *labels* and therefore we define the label diff points correspondence.

Label Diff Points Given two programs (P,P') and their sets of program labels (Lab, Lab'), we define a label correspondence relation named *label diff points* denoted $DP_{Lab} \subseteq Lab \times Lab'$ as a matching of labels between programs. From this point on any mention of the diff-points correspondence DP will refer to label diff-points DP_{Lab}. As discussed in Section 2, we allow a broad selection of differrncing points, including exit points, output locations and array accesses thus capturing differences beyond return value.

Now, we will move past the concrete semantics towards *abstract semantics*. This is required as it is unfeasible to describe difference based on traces. Before doing so, we must adjust our concrete semantics since a concrete semantics based on individual traces *will not allow us to correlate traces that originate from the same input*. This is the first formal indication of how a separate abstraction, that considers each of the programs by itself, cannot succeed.

4.3 Concrete Correlating Semantics

We define the correlating state and trace which bind the executions of both programs, P and P', together and define the notion of delta in this setting. Essentially, these will be states and traces of the product program $P \times P'$ but *only traces that originate from equivalent input states are considered*. This allows us to define the *correlating abstract semantics* which is key for successful differencing.

DEFINITION 4 (Correlating Concrete State). *A correlating concrete state $\sigma^{\natural}_{\times} : Var \cup Var' \rightarrow Val$ is a unified concrete state, mapping variables from both programs (P, P') to their values. The set of all possible correlating states is denoted $\Sigma^{\natural}_{P \times P'}$.*

DEFINITION 5 (Correlating Concrete Trace). *A correlating trace $\pi_{\times}$, is a sequence of correlating states $..., \sigma^{\natural}_{i_{\times}}, ...$ describing an execution of $P \times P'$. We restrict to traces that originate from equivalent input states i.e., $\sigma^{\natural}_0 \equiv \sigma'^{\natural}_0$. The $label_{\times}$, $last_{\times}$ and $pre_{\times}$ operations are defined similarly.*

We must remember however, that the number of traces to be compared is potentially unbounded which means that the delta we compute may be unbounded too. Therefore we must use an abstraction over the concrete semantics that will allow us to represent executions in a bounded way.

5. Abstract Correlating Semantics

In this section, we introduce our correlating abstract domain which allows bounded representation of product-program state while maintaining equivalence between correlated variables.

5.1 Abstract Correlating State

We represent variable information using standard relational abstract domains. As our analysis is path sensitive, we allow for a set of abstract sub-states, each adhering to a certain path in the product program. This abstraction is similar to the trace partitioning domain as described in [22].

Our power-set domain records precise state information but does not scale due to exponential blowup. As a first means of reducing state size, we define a special join operation that dynamically partitions the abstract state according to the set of equivalences maintained in each sub-state and joins all sub-states in the same partition together (using the sub-domain join operation). This join criteria allows separation of equivalence preserving paths thus achieving better precision. Second, to allow a feasible bound abstraction for programs with infinite number of paths, we define a widening operator which utilizes the sub-domain's widening but cleverly chooses which sub-states are to be widened, according to path information encoded in state. We start off by abstracting the correlating trace semantics in Sec. 4.3.

In the following, we assume an abstract relational domain $(D^{\sharp}, \sqsubseteq_D)$ equipped with operations $\sqcap_D$, $\sqcup_D$ and ∇_D, for representing sets of concrete states in $\Sigma_{P \times P'}$. We separate the set of program variables into original program variables denoted Var (which also include a special added variable for return value, if such exists) and the added guard variables denoted $Guard$ that are used for storing conditional values alone ($Guard$ also include a special added guard for return flag). We assume the abstract values in $D^{\sharp}$ are constraints over the variables and guards (we denote $D^{\sharp}_{Guard}$ for sub-domain abstraction of guards and $D^{\sharp}_{Var}$ for original variables), and do not go into further details regarding the particular abstract domain as it is a parameter of the analysis. We also assume that the sub-domain $D^{\sharp}$ allows for a sound over-approximation of the concrete semantics (given a sound interpretation of program operations). In our experiments, we use the polyhedra abstract domain [7] and the octagon abstract domain [17].

```
int f(int x) {            int f'(int x) {
    return x;                 return 2*x;
}                         }
```

Figure 8: Single path differentiation candidates

Correlating Abstract State A correlating abstract program state $\sigma^{\sharp} \in Lab_{\times} \rightarrow 2^{D^{\sharp}_{Guard} \times D^{\sharp}_{Var}}$, is a set of pairs $\langle ctx, data \rangle$ mapped to a product program label $l_{\times}$, where $ctx \in D^{\sharp}_{Guard}$ is the execution context i.e. an abstraction of guards values via the relational numerical domain and $data \in D^{\sharp}_{Var}$ is an abstraction of the variables. We separate abstractions over guard variables added by the transformation to GCL format from original program variables as there need not be any relationships between guard and regular variables.

5.2 Abstract Correlating Semantics

Tab. 1 describes the abstract transformers. The table shows the effect of each statement on a given abstract state $\sigma^{\sharp} = l_{\times} \mapsto S$. The abstract transformers are defined using the abstract transformers of the underlying abstract domain $D^{\sharp}$. We assume that any program P can be transformed such that it contains the operations described in alone (this is achieved by the GCL format). We also assume that for $[\![g := e]\!]^{\sharp}$ operations, e is a logical operation with binary value.

Next, we define the abstraction function $\alpha : 2^{\Sigma^*_{P \times P'}} \rightarrow 2^{D^{\sharp} \times D^{\sharp}}$ for a set of concrete traces $T \subseteq \Sigma^*_{P \times P'}$. As in our domain traces are abstracted together if they share the exact same path, we first define an operation $path : \Sigma^*_{P \times P'} \rightarrow Lab^*$ which returns a sequence of labels for a trace's states i.e. what is the path taken by that trace. We also allow applying $path$ on a set of traces to denote the set of paths resulting by applying the function of each of the traces. Finally we define the trace abstraction as following:

$$\alpha(T) \triangleq \{\sqcup_{path(\pi)=p} \beta(last(\pi)) | p \in path(T)\}$$

where $\beta(\sigma^{\natural}) = \langle \beta_{D^{\sharp}}(\sigma^{\natural}|_{Guard}), \beta_{D^{\sharp}}(\sigma^{\natural}|_{Var}) \rangle$ i.e. applying the the abstraction function of the abstract sub-domain $\beta_{D^{\sharp}}$ on parts of the concrete state applying to $Guards$ (denoted $\sigma^{\natural}|_{Guard}$) and $Vars$ (denoted $\sigma^{\natural}|_{Var}$) separately. Our abstraction partitions trace prefixes π by path and abstracts together the concrete states reached by the prefix - $last(\pi)$, using the sub-domain.

Every path in the product program will be represented by a single sub-state of the sub-domain. As a result, all *traces prefixes* that follow the same path to $l_{\times}$ will be abstracted into a single sub-state of the underlying domain. This abstraction fits semantics differencing well, as inputs that follow the same path display the same behavior and will usually either keep or break equivalence together, allowing us to separate them from other behaviors (it is possible for a path to display both behaviors as in Fig. 8 and we will discuss how we are able to manipulate the abstract state and separate equivalent behaviors from ones that offend equivalence). Another issue to be addressed is the fact that our state is still potentially unbound as there may be an infinite number of paths in the program (due to loops).

5.3 Dynamic Partitioning

Performing analysis with the powerset domain does not scale as the number of paths in the correlated program may be exponential (we defer the case of unbound paths to widening of loops). We must allow for reduction of state $\sigma^{\sharp} = l_{\times} \mapsto S$ with acceptable loss of precision. This reduction via partitioning can be achieved by joining the abstract sub-states in S (using the standard precision losing join of the sub-domain). However this can only be accomplished

$[\![v := e]\!]^\sharp$	$l_\times \mapsto \{\langle ctx, [\![v := e]\!]^\sharp_{D^\sharp}(data)\rangle \mid \langle ctx, data\rangle \in S\}$
$[\![g := e]\!]^\sharp$	$l_\times \mapsto \{\langle [\![g := true]\!]^\sharp_{D^\sharp}(ctx), [\![e]\!]^\sharp_{D^\sharp}(data)\rangle \mid \langle ctx, data\rangle) \in S\} \cup \{\langle [\![g := false]\!]^\sharp_{D^\sharp}(ctx), [\![\neg e]\!]^\sharp_{D^\sharp}(data)\rangle \mid \langle ctx, data\rangle \in S\}$
$[\![$ **if** (g) $\{s_0\}$ **else** $\{s_1\}$ $]\!]^\sharp$	$l_\times \mapsto \{\langle [\![g = true]\!]^\sharp_{D^\sharp}(ctx), [\![s0]\!]^\sharp_{D^\sharp}(data)\rangle \mid \langle ctx, data\rangle) \in S\} \cup \{\langle [\![g = false]\!]^\sharp_{D^\sharp}(ctx), [\![s_1]\!]^\sharp_{D^\sharp}(data)\rangle \mid \langle ctx, data\rangle \in S\}$
$[\![$ **goto** lab $]\!]^\sharp$	$\sigma^\sharp$

Table 1: Abstract transformers using abstract transformers of the underlying domain $D^\sharp$. The table describe the effect of each statement on an abstract state $\sigma^\sharp = l_\times \mapsto S$.

after first deciding which of the sub-states shall be joined together and then choosing the program locations for the partitioning to occur. A trivial partitioning strategy is simply reverting back to the sub-domain by applying the join on all sub-states which may result in unacceptable precision loss as exemplified in Fig. 4. However, by taking a closer look at the final state of the same example ($\equiv_v$ means v and v' are equivalent in the state) :

$$\sigma^\sharp(fin) = [\langle(g1, \neg g2', \equiv_{g1}), (x > 0, sgn = 1, \equiv_{x,sgn})\rangle,$$
$$\langle(\neg g1, \neg g2', \equiv_{g1}), (x < 0, sgn = -1, \equiv_{x,sgn})\rangle,$$
$$\langle(\neg g1, g2', \equiv_{g1}), (x = 0, sgn = 0, sgn' = 1, \equiv_x)\rangle]$$

One may observe that were we to join the two sub-states that maintain equivalence on $\{x, sgn, g1\}$, it would result in an acceptable loss of precision (of losing the x related constraints). This is achieved by partitioning sub-states according to the set of variables which they preserve equivalence for. This bounds the state size at $2^{|VC|}$, where VC is the set of correlating variables we wish to track. As mentioned, another key factor in preserving equivalence and maintaining precision is the program location at which the partitioning occurs. The first possibility, which is somewhat symmetric to the first proposed partitioning strategy, is to partition at every join point i.e. after every branch converges. Let use examine $sign \bowtie sign'$ state after processing the first guarded instruction `if (G1) sgn = -1;` (we ignored $g2'$ effect at this point for brevity):

$$\sigma^\sharp = [\langle(g1, \equiv_{g1}), (x \geq 0, \equiv_{x,sgn})\rangle,$$
$$\langle(g1, \equiv_{g1}), (x < 0, sgn' = -1, \equiv_x)\rangle]$$

This suggests that partitioning at join points will perform badly in many scenarios, specifically here as we will lose all data regarding sgn. However if we could delay the partitioning to a point where the two programs "converge" (after the following `if (G1') sgn' = -1;` line), we will get a more precise temporary result which preserves equivalence. We accomplish this, we define special program locations we name *correlating points* which present places where programs have likely converged. These are a sub-product of the correlating program construction process described in Section 6.

Other viable program locations for partitioning are our *differencing points*. Partitioning at these locations is essentially more precise than at correlation points. We remind that diff-points are product program locations where both programs conceptually converge as they are about to emit output. In other words, both programs have finished "preparing" the next portion of output, therefore if equivalence exists - it must exist now, making differencing points a prime candidate for a partitioning point.

5.4 Widening

In order for our analysis to handle loops we require a means for reaching a fixed point. As our analysis iterates over a loop, sub-states may be added or transformed continuously, never converging. We therefore need to define a widening operator for our new domain, which will further over-approximate the looping state and arrive at a fixed point. We have the widening operator of our sub-domain at our disposable, but we are faced with the question of how to apply this operator, i.e. which pairs of sub-states $\langle ctx, data\rangle$ from $\sigma^\sharp$ should be widened with which. A first viable strategy, similar to the first partitioning strategy, is to perform an overall join operation on all pairs which will result in a single pair of sub-states and then simply apply the widening to this sub-state using the sub-domain's ∇ operator. If we examine applying this strategy to $sum \bowtie sum'$ from Fig. 6, we get that it will successfully arrive at a fixed point that also maintains equivalence as all sub-states maintain equivalence at loop back-edges. Now let us try applying the strategy to the more complex $sum \bowtie sum''$ as seen in Fig. 9. First we mention that as sum' introduces a return statement under the $len > max$ condition, the example shows an extra r' guard and $retval'$ variable for representing a return (this exists in all GCL programs but we omitted it so far for brevity). While analyzing, once we pass that first conditional, our state is split to reflect the return effect:

$$\sigma^\sharp = [s_1 = \langle(\neg r'), (len \leq max, result = 0, \equiv_{len,result})\rangle,$$
$$s_2 = \langle(r'), (len > max, retval' = -1, result = 0, \equiv_{len,result})\rangle]$$

As we further advance into the loop, s_1 will maintain equivalence but s_2 will continue to update the part of the state regarding un-tagged variables (since $r' = true$ in s_2 and it will not consider any of the commands guarded by r'), specifically it will change $result$ continuously, preventing the analysis from reaching fixed point. We would require widening here but using the naive strategy of a complete join will result in aggressive loss of precision, specifically losing all information regarding $result$. The problem originates from the fact that prior to widening, we joined sub-

```
unsigned max = ...;
int sum''(int arr[], unsigned len) {
  int result = 0;
  if (len > max)
    return -1;
  for (unsigned i = 1; i < len; i+=2)
    result += arr[i];
 return result;
}
```

```
unsigned max' = ...;
int sum(int arr[], unsigned len) {
  unsigned len' = len;
  int arr'[] = arr;
  int result = 0;
  int result' = 0;
  guard r' = (len' > max');
  if (r') retval' = -1;
  if (r') r' = 0;
  {
    unsigned i = 1;
    unsigned i' = 1;
l:  guard g = (i < len);
l': guard g' = (i' < len');
    if (g) result += arr[i];
    if (r') if (g') result' += arr'[i'];
    if (g) i+=2;
    if (r') if (g') i'+=2;
    if (g) goto l;
    if (r') if (g') goto l';
  }
}
```

Figure 9: Patched sum'' and correlating $sum \bowtie sum''$

states which adhere to two different loop behaviors: one where both sum and sum' loop together (that originated from $len < max$) and the other where sum' has exited but sum continues to loop ($len \geq max$). Ideally, we would like to match these two behaviors and widen them accordingly. We devised a widening strategy that allows us to do this as it basically matches sub-states that adhere to the same behavior, or loop-paths. This strategy dictates using *guards* for the matching. If two sub-states agree on their set of guards, it means they represent the same loop path and can be widened as the latter originated from the former (widening operates on subsequent iterations). In our example, using this strategy will allow the correct matching of states after consequent $k, k+1$ loop iterations:

$$\sigma^\sharp_k = [s_1 = \langle(\neg r', g, \equiv_g), (len \leq max, i = 2k+1, \equiv_{i,len,result})\rangle,$$

$$s_2 = \langle(r', g, \neg g),$$

$$(len > max, retval' = -1, result' = 0, i' = 2k+1, i = 1, \equiv_{len})\rangle]$$

And:

$$\sigma^\sharp_{k+1} = [s_1 = \langle(\neg r', g, \equiv_g), (len \leq max, i = 2k{+}3, \equiv_{i,len,result})\rangle,$$

$$s_2 = \langle(r', g, \neg g),$$

$$(len > max, retval' = -1, result' = 0, i' = 2k+3, i = 1, \equiv_{len})\rangle]$$

As we can identify the states predecessors by simply matching the guards. s_1 will be widened for a precise description of the difference shown as $\langle len = len' > max, retval' = -1, retval' = \top\rangle$.

5.5 Differencing for Abstract Correlating States

Given a state in our correlating domain, we want to determine whether equivalence is kept and if so under which conditions it is kept (for partial equivalence) or determine there is difference and characterize it. As our state may hold several pairs of sub-states, each holding different equivalence data, we can provide a verbose answer regarding whether equivalence holds. We partition our sub-states according to the set of variables they hold equivalence for and report the state for each equivalence partition class. Since we instrument our correlating program to preserve initial input values, for some of these states we will also be able to report input constraints thus informing the user of the input ranges that maintain equivalence. In the cases where equivalence could not be proved, we report the offending states and apply a differencing algorithm for extraction of the delta. Fig. 8 shows an example of where our analysis is unable to prove equivalence (as it is sound), although part of the state does maintain equivalence (specifically for $x = 0$). This is due to the abstraction being to coarse. We describe an algorithm that given a sub-state $d \in D^\sharp$, computes the differentiating part of the sub-state (where correlated variables disagree on values) by splitting it into parts according to equivalence. This is done by treating the relational constraints in our domain as geometrical objects and formulating delta based on that.

Correlating Abstract State Delta

DEFINITION 6. *Given a sub-state d and a correspondence VC, the correlating state delta $\triangle_A(d)$, computes abstract state differentiation over d. The result is an abstract state $\sqsubseteq d$ approximating all concrete values for variables correlated by VC, that differ between P and P'. Formally, the delta is simply the abstraction of the concrete trace deltas:*

$$\triangle_A(d)^+ \triangleq \alpha(\cup_{path}\triangle^+_T)$$

$$\triangle_A(d)^- \triangleq \alpha(\cup_{path}\triangle^-_T)$$

where deltas are grouped together by path and then abstracted.

The algorithm for the extraction of delta from a correlating state, is as follows:

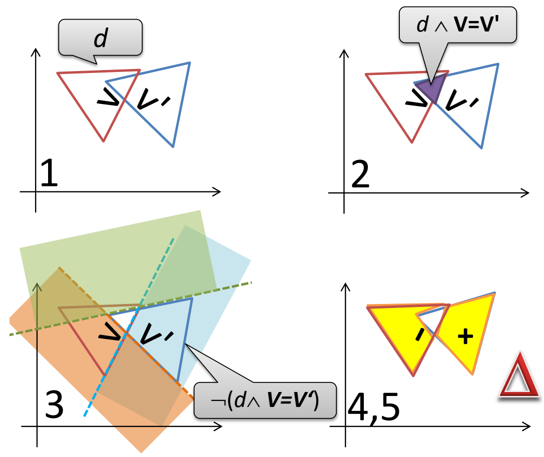

Figure 10: Delta computation geometrical representation.

1. $d_\equiv$ is a state abstracting the concrete states shared by the original and patched program. It is achieved by computing: $d_\equiv \triangleq d|_{V=V'} \equiv d \sqcap \bigwedge\{v = v' | VC(v) = v'\}$.
2. $\overline{d_\equiv}$ is the negated state i.e. $D^\sharp \setminus d_\equiv$ and it is computed by negating $d_\equiv$ (as mentioned before, all logical operations, including negation, are defined on our representation of an abstract state).
3. Eventually: $\triangle_A(d) \triangleq d \sqcap \overline{d_\equiv}$ abstracts all states in $P \times P'$ that where correlated variables values do not match.
4. $\triangle_A(d)^+ = \triangle_A(d)|_{V'}$ is a projection of the differentiation to display values of P' alone i.e. "added values".
5. $\triangle_A(d)^- = \triangle_A(d)|_V$ is a projection of the differentiation to display values of P alone i.e. "removed values".

EXAMPLE 4. *Applying the algorithm on Fig. 8's P and P' where $d = \{retVal' = 2retVal\}$ will result in the following:*

1. $d_\equiv = \langle retVal' = 0, retVal = 0\rangle$.
2. $\overline{d_\equiv} = [\langle retVal' > 0\rangle, \langle retVal' < 0\rangle, \langle retVal > 0\rangle, \langle retVal < 0\rangle]$
3. $\triangle_A(d) = [\langle retVal' = 2retval, retVal' > 0\rangle, \langle retVal' = 2retval, retVal' < 0\rangle, \langle retVal' = 2retval, retVal > 0\rangle, \langle retVal' = 2retval, retVal < 0\rangle]$
4. $\triangle_A(d)^+ = [\langle retVal' > 0\rangle, \langle retVal' < 0\rangle]$
5. $\triangle_A(d)^- = [\langle retVal > 0\rangle, \langle retVal < 0\rangle]$

We see that displaying the result in the form of projections is ill-advised as in some states differentiation data is represented by relationships on correlated variables alone, thus projecting will lose all data and we will be left with a less informative result. A geometrical representation of $\triangle_A$ calculation can be seen in Fig. 10.

From this point forward any mention of 'delta' (denoted $\triangle$) will refer to the correlating abstract state delta (denoted $\triangle_A$). We claim that $\triangle$ is a correct abstraction for the concrete state delta which allows for a scalable representation of difference we aim to capture.

6. Correlating Program

In this section we will describe how we construct our correlating program $P \bowtie P'$. The process of correlating attempts to find a better interleaving of programs for a more precise differentiation. The building process also instruments $P \bowtie P'$ with the required differencing points DP which allow reporting of difference and finds correlation points CP which define the locations for out partitioning. We also allow a user defined selection of DP and CP.

6.1 Construction of $P \bowtie P'$

The idea of a correlating program is similar to that of self-composition [24], but the way in which statements in the correlating program are combined is carefully designed to keep the steps of the two programs close to each other. Rather than having the patched program sequentially composed after the original program, our correlating program interleaves the two versions. Analysis of the correlating program can then recover equivalence between values of correlated variables even when equivalence is *temporarily* violated by an update in one version, as the corresponding update in the other version follows shortly thereafter. To adhere to our formal definitions, we note that the correlating program is a reduction of the product program, allowing a single interleaving of commands, where in $P \times P'$ all interleaving are possible (similarly to $P||P'$), thus all definitions apply to $P \bowtie P'$ as it is a single case of the generalized definitions.

We will generally describe the process of constructing the correlating program. The correlating program is an optimized reduction over $P \times P'$ where not all pairs of (σ, σ') are considered, but only pairs that result from a controlled execution, where correlating instructions in P and P' will execute adjacently. This will allow for superior precision.

The input for the correlation process are two program (P, P) in C. The first step involves transforming both programs to a normalized guarded instruction form (P^G, P'^G). Next, a vector of *imperative commands* I (and I' respectively) is extracted from each program for the purposes of performing the syntactic diff. An imperative command in our GCL format is defined to be either one of `v := e | goto l | f(...)` as they effectively change the program state (variable values, excluding guard) and control. Then a syntactical diff [12] is computed over the vectors (I, I'). One of the inputs to the diff process is VC as it is needed to identify correlated variables and the diff comparison will regard commands differing by variable names which are correlated by VC as equal. The result of the last step will be a vector I_{diff} specifying for each command in I, I' whether it an added command in P' (for I') marked $+$, a deleted command from P (for I) marked $-$, or a command existing in both versions $=$. This diff determines the order in which the commands will be interleaved in the resulting $P \bowtie P'$ as we will iterate over the result vector I_{diff} and use it to construct the correlating program. We remind that since I, I' are only the imperative commands, we cannot use it directly as $P \bowtie P'$. Instead we will use the imperative commands as markers, specifying which chunk of program from P or P' should be taken next and put in the result. The construction goes as follows: iterate over I_{diff} and for every command c (c') labeled l_c ($l_{c'}$):

- read P (P') up to label l_c ($l_{c'}$) including into block B_c (B'_c)
- for B'_c, tag all variables in the block.
- emit the block to the output.
- delete B_c (B'_c) from P (P').

The construction is now complete. We only add that at the start of the process, we strip P' of its prototype and add declarations for the tagged input variables, initializing them to the untagged version. As mentioned CP is also a product of the construction, and it's defined using $=$ commands: after two $=$ commands are emitted to the output, we add an instrumentation line, telling the analysis of the correlation point. One final observation regarding the correlating program is that it is a legitimate program that can be run to achieve the effect of running both versions. We plan to leverage this ability

```
size_t rpc_uaddr2sockaddr (const size_t uaddr_len, ...) {
  char buf[ RPCBIND_MAXUADDRLEN ];
    ...
-  if ( uaddr_len > sizeof ( buf ))
+  if ( uaddr_len > sizeof ( buf ) - 2:)
    return 0;
    ...
    (*)1
  buf [ uaddr_len ] = '\n';
  buf [ uaddr_len + 1] = '\0';
    ...
}
```

Figure 11: Linux kernel `addr.c` v2.6.32-rc6 module with patch.

to use dynamic analysis and testing techniques such as fuzzing [19] and directed automated testing [3] on the correlating program in our future work.

7. Evaluation

We evaluated DIZY on a number of challenging real world programs where the patches affect numerical variables. As benchmarks, we used several programs from the GNU core utilities (differencing versions 6.10 and 6.11), as well as a few handpicked patches taken from the Linux kernel and the Mozilla Firefox web browser. In most programs, DIZY was able to precisely describe the difference.

7.1 Prototype Implementation

We implemented a correlating compiler named CCC which creates correlating programs from any two C programs. We also implemented a differencing analysis for analyzing correlated programs. Both tools are based on LLVM and CLANG compiler infrastructure. We analyze C code directly since it is more structured, has type information and keeps a low number of variables, as opposed to intermediate representation. We also benefit from our delta being computed over original variables. As mentioned in Section 6, we normalize the input programs before unifying them for a simpler analysis. Our analysis is intra-procedural and we handle function calls by either modularly proving their equivalence and assuming it once encountered or, in case equivalence could not be proved, by inlining. Calls to external system functions do not change local state in our examples and thus were ignored. We used the APRON abstract numerical domain library and conducted our experiments using several domains including octagon [17] and polyhedra [7]. All of our experiments were conducted running on a Intel(R) Core-i7(TM) processor with 4GB.

For brevity, when presenting results we only show the relevant code fragments, however, our analysis has been applied to the full original program. A complete version of the results is available at `http://www.cs.technion.ac.il/~nimi/dizy`.

7.2 Results

Producing delta from abstract state Fig. 11 shows a patch made to the `net/sunrpc/addr.c` module in the Linux kernel SUNRPC implementation v2.6.32-rc6 which is aimed at removing an off-by-two array access out of bounds violation in the original program. Although simple, this example shows two advantages DIZY: (i) the ability to analyze programs without the need to run them and (ii) the ability to capture fine-grained differences. The result of analyzing the correlated program is shown in Fig. 12.

The difference is captured by one sub-state where the execution has ended in the patched program but continues in the original. We instrument the correlating program with a return flag to capture the difference as otherwise equivalence holds (none of the original variables change).

σ_1:
returned' = $true$
returned = $false$
uaddr_len' $\leq$ RPCBIND_MAXUADDRLEN - 2
uaddr_len' $\geq$ RPCBIND_MAXUADDRLEN

Figure 12: Difference for SUNRPC implementation

```
nsresult SetTextInternal (int textLength, int aCount,
                          int aLength, int aOffset,
                          PRUnichar * aBuffer) {
  PRInt32 newLength = textLength - aCount + aLength ;
  PRUnichar * to;
    ...
+   if ((unsigned)newLength > (1 « 29))
+   return NS_ERROR_DOM_DOMSTRING_SIZE_ERR;
+   (*)1
  memcpy (to + aOffset , aBuffer ,
          aLength * sizeof ( PRUnichar ));
    ...
}
```

Figure 13: Firefox `nsGenericDOMDataNode` module with patch.

σ_1:
returned' = $true$
returned = $false$
return value' = NS_ERROR

(a)

σ_1:	σ_2:
returned' = $true$	returned' = true
returned = $false$	returned = false
newLength > 536870912	newLength > -3758096384
	newLength < 0
return value' = NS_ERROR	return value' = NS_ERROR

(b)

Figure 14: Difference for Firefox nsGenericDOMDataNode, (a) with a single polyhedra; (b) set of polyhedra.

Another example, taken from CVE-2010-1196 advisory regarding Firefox's heap buffer overflow on 64-bit systems is shown in Fig. 13 (vulnerable part of the function only). Firefox 3.5 and 3.6 (up to 3.6.4) contain a heap buffer overflow vulnerability which is caused by an integer overflow. Due to the amount of data needed to trigger the vulnerability (> 8GB), this is only exploitable on 64-bit systems. The vulnerable code is found in `/content/base/src/nsGenericDOMDataNode.cpp` of the Mozilla code base and was adapted to C for analysis purposes.

Here, we need to describe a more complex and non-convex constraint that leads to difference. Running DIZY with partitioning produces the result shown in Fig. 14 (a).

The difference in state is described correctly as indeed the only change in values for the patch scenario would be the return value and the early return of the patched version. However, we did not preserve the conditional constraints as they are non-convex. Running the same analysis with no partitioning (this is feasible as the procedure does not loop) produces the result shown in Fig. 14 (b).

Now we see that the `(unsigned)newLength > (1 « 29)` constraint has been successfully encoded in two offending states, each holding a part of the problematic range.

Capturing complex delta Fig. 15 shows a patch made to the `char_to_clump` function in version 6.11 of coreutils. The patch replacing the execution of the line `input_position += width`, which originally executed unconditionally, with a conditional structure that in the new version, allows the line to execute only under certain complex conditions. Since the variables handled in this patch (the global `input_position` and return value `chars`) emit output, describing how their values changed and under which terms is important, especially as the patch cannot be easily parsed by a programmer to understand its meaning. The result of our analysis at the return point is show in Fig. 16. These results include information regarding initial values of parameters for improved precision (this is one of DIZY's features).

σ_1:	σ_2:	σ_3:
input_position$_0$ = 0	input_position$_0$ < -width	input_position$_0$ < -width
chars' = 0	input_position$_0$ < 0	input_position$_0$ > 0
input_position = width	input_position' = 0	input_position' = 0
input_position < 0	input_position < width	input_position > width
input_position' = 0	width < 0	input_position <= 0

Figure 16: Difference for coreutils pr.c `char_to_clump`

```
int logicalValue(int t) {
 if (!(curr - t >= 100)) {
   return old;
 } else {
   int val = 0;
   for (int i = 0 ;
     i < data.length; i++)
      val = val + data[i];
   old = val;
   return val;
 }
}
```

```
const int THRESHOLD = 100;
int logicalValue(int t) {
 int elapsed = curr - t;
 int val = 0;
 if (elapsed < THRESHOLD) {
   val = 1;
 } else {
   for (int i = 0;
     i < data.length; i++)
      val = val + data[i];
   old = val;
 }
 return val;
}
```

Figure 17: Two versions of the `logicalValue()` procedure taken from [20].

The result convey the difference in the output variable values alongside some of conditions under which the difference occurs. The result is composed of three sub-states featuring difference and adhere to two added paths in the patched program. The first sub-state belongs to the first branch in the added conditional: the difference is comprised of (i) the new value of `input_position` is 0 as opposed to it being `width` in the former version (the analysis took the `input_position += width` line into account and incorporated knowing that $input\_position = 0$ from the branch condition). The analysis also deduced that the old `input_position` is negative under the same input as the branch condition dictates that `width` is negative. (ii) `chars` in the new program is 0 under this path. The two other sub-states adhere to the second added path in the conditional and track a difference for `input_position` alone, basically stating that `input_position` under this path used to assume values in ranges $[-\inf, width]$ and $[width, 0]$ but now is simply 0. The splitting of this path into two cases is a result of expressing the non-convex $input\_position \neq 0$ condition from the first branch conditional using two sub-states. The result also describes constraints on the procedure's input under which the difference exist. Another product of the analysis, which we do not show here, are sub-states describing paths which the patch did not affect.

Maintaining Equivalence and Reporting Difference in Loops Fig. 17 shows two version of the java `logicalValue()` method taken from [20], adapted to C. This example features semantic preserving refactoring modification (introducing the `elapsed` variable and `THRESHOLD` constant, simplifying a conditional and moving the return statement out of branch block) and one semantic change where 1 is returned instead of old in case $curr - t < 100$). The challenge in this example is proving equivalence over the loop branch and reporting difference for the negated path. Using a separate analysis, we would have to deduce at the following loop

```
int input_position;

bool char_to_clump(char c) {
  int width;
    ...
  input_position += width;
    (*)1
    ...
  return chars;
}
```

coreutils pr.c v6.10

```
int input_position;

bool char_to_clump'(char c) {
  int width;
    ...
  if (width < 0 && input_position == 0) {
      chars = 0;
      input_position = 0;
  } else if (width < 0 && input_position <= -width) {
    input_position = 0;
  } else {
    input_position += width;
  }
    (*)1
    ...
  return chars;
}
```

coreutils pr.c v6.11

Figure 15: Original and patched version of coreutils `pr.c`'s `char_to_clump` procedure

σ_1:
curr - t < 100
return value = old
return value' = 1

Figure 18: Difference for `logicalValue()`

```
bool bsd_split_3 (char *s, size_t s_len,...) {
  int i = s_len;
  i--;
+ if (s_len == 0) return false;
  while (i && s[i] != ')') { (*)1
    i--;
  }
  ...
  (*)2
}
```

Figure 19: Original and patched version of coreutils `md5sum.c`'s `bsd_split_3` procedure

$(*)_1$:	
σ_1:	σ_2 (equivalent):
s_len = 0 s_len' = 0 i $\leq$ -1	s_len' = s_len i' = i s_len' - 1 $\geq$ i'
$(*)_2$:	
σ_1 (equivalent):	
s_len' = s_len i' = i s_len' - 1 $\geq$ i'	

(a)

$(*)_1$:		
σ_1:	σ_2 (equivalent):	σ_3 (equivalent):
s_len = 0 s_len' = 0 i $\leq$ -1	s_len' = s_len i' = i s_len' - 1 $\geq$ i'	s_len' = s_len i = 0 s_len' $\geq$ 1
$(*)_2$:		
σ_1 (equivalent):		
s_len' = s_len i' = i i = 0 s_len' $\geq$ 1		

(b)

Figure 20: Difference for `md5sum bsd_split_3`

invariant: $val = \Sigma_{i=0}^{data.length} data[i]$ in order to show equivalence. However, as our abstraction focuses on variable relationships and our correlating program allows us to interleave the two loops in lock-step, all our analysis needs to deduce is the $val = val'$ constraint. As we apply widening to converge, the constraint will be kept, allowing us to establish equivalence for the looping path. DIZY reports the state shown in Fig. 18 state for the exit point of `logicalValue()`. We note that in [20] the example was run by unrolling 2 steps of the loop.

Next we shall explore a different loop scenario where all paths in the programs contain loops and only some of them maintain equivalence. Fig. 19 shows part of coreutils md5sum.c `bsd_split_3` function that was patched in version version 6.11 to disallow 0-length inputs. Although this patch seems trivial, analyzing it is challenging as it affects the behavior of loops i.e. unbound path lengths. The main challenge in this example, is separating the path where s_len is 0, which results in the loop index i ranging within negative values (producing an array access out of bounds fault), from the rest of the behaviors that maintain equivalence, throughout the widening process which is required for the analysis to reach a fixed point. The result of our analysis with partitioning is shown in Fig. 20 (a) (per differencing points $(*)_1, (*)_2$).

We can see the analysis successfully reports a difference for the singularity point $s\_len = 0$ inside the loop, precisely describing the scenario where i' is negative. We can also see the other equivalent state existing within the loop which depicts the results of the widened analysis for all other paths (the $s\_len \neq 0$ constraint is not existing there due to partitioning as we will soon show). The differencing sub-state will be omitted once we move past the loop as the $i \leq -1$ constraint will not allow it to exist beyond the loop body thus we are left with the equivalent state alone after the loop which correctly expresses the fact that the programs are equivalent at this point (since both i's converged at 0). We can see that the result at the second differencing point has lost precision since it does not reflect the $i = 0$ constraint. The loss of this constraint is, again, due to partitioning as both sub-states that describe exiting the loop and the one describing entering the loop, hold equivalence for all variables and are joined to together and lose the extra constraint information. If we analyze the same example with no partitioning we get the result of Fig. 20 (b).

Which further separates the paths in the program, allowing for a different sub-state for the $i = 0$ and $i \neq 0$ substates (again, the $i \neq 0$ constraints was lost when joining together the $i > 0$ and $i < 0$ states as they both adhere to the same path and hold the same guard values). This extra precision is beneficial, but we still managed to supply a satisfactory result using the more scalable partitioning by equivalence technique.

8. Related Work

Our work has been mainly inspired by recent work identifying program differencing as having vast security implications [2, 23] as well as advancements made in the field of under-approximations of program equivalence [8, 15, 20, 21].

the problem of program differencing is fundamental [9] and early work mainly focused on computing syntactical difference [12]. These solutions are an important stepping stone and we used syntactical diff as a means to achieve optimized interleaving of programs in out correlating program for better analysis results. Another possibility for creating this program is to rely on the editing sequence that creates the new version from the original one [10].

Jackson and Ladd [13] proposed a tool for computing data dependencies between input and output variables and comparing these dependencies along versions of a program for discovering difference. This method may falsely report difference as semantic difference may occur even if data dependencies have not changed. Furthermore, data dependencies offer little insight as to the meaning of difference i.e. input and output values. Nevertheless, this was an important first step in employing program analysis as a means for semantic differencing.

Important work in the problem of equivalence of combinatorial circuits [5, 16, 18] made important contributions in establishing the problem of equivalence as feasible, producing practical solutions for hardware verification.

We rely on classic methods of abstract interpretation [6] for presenting an over approximating solution for semantic differencing and equivalence. To achieve this we devised a static analysis over a newly defined construct we call a correlating program. The idea of a correlating program is similar to that of self-composition [24] except that we compose two different programs in a interleaving designed to maintain a close correlation between them. The use of a correlating construct for differencing is novel as previous methods mainly use sequential composition [8, 20, 21], disregarding possible program correlation.

We base our analysis on a relational abstraction [7, 17] that allows us to reason about variables of different programs. The abstraction is further refined towards a disjunctive domain, similar to trace partitioning [22] and we use an equivalence based partitioning criteria, which is apt to our purposes.

Symbolic execution based methods [20, 21] offer practical equivalence verification techniques for loop and recursion free programs with small state space. These works complement each other in regards to reporting difference as one [20] presents an over approximating description of difference they call differential summaries and the other [21] presents an under approximating description including concrete inputs for test cases demonstrating difference in behavior. An interesting question is how could these methods be combined iteratively to achieve better precision. Also, this work can be used to complement our work in cases where equivalence could not be proven and the description of difference can be leveraged for the extraction of concrete input that leads to offending states.

Bounded model checking based work [8] presents the notion of partial equivalence which allows checking for equivalence under specific conditions, supplied by the user but are bound by loops. They employ a technique based on theorem provers for proving an equivalence formula which embeds program logic (in SSA form) alongside the requirement for input and output equivalence and user provided constraints.

[1] introduced a correlating heap semantics for verifying linearizability of concurrent programs. In their work, a correlating heap semantics is used to establish correspondence between a concurrent program and a sequential version of the program at specific linearization points.

References

[1] AMIT, D., RINETZKY, N., REPS, T., SAGIV, M., AND YAHAV, E. Comparison under abstraction for verifying linearizability. In *CAV'07*.

[2] BRUMLEY, D., POOSANKAM, P., SONG, D., AND ZHENG, J. Automatic patch-based exploit generation is possible: Techniques and implications. In *S&P'08*, pp. 143–157.

[3] CADAR, C., DUNBAR, D., AND ENGLER, D. R. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI* (2008), pp. 209–224.

[4] CHAKI, S., GURFINKEL, A., AND STRICHMAN, O. Regression verification for multi-threaded programs. In *VMCAI'12*.

[5] CLARKE, E. M., AND KROENING, D. Hardware verification using ansi-c programs as a reference. In *ASP-DAC* (2003), pp. 308–311.

[6] COUSOT, P., AND COUSOT, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *POPL* (1977), pp. 238–252.

[7] COUSOT, P., AND HALBWACHS, N. Automatic discovery of linear restraints among variables of a program. In *POPL'78*, pp. 84–97.

[8] GODLIN, B., AND STRICHMAN, O. Regression verification. In *DAC* (2009), pp. 466–471.

[9] HOARE, C. A. R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (1969), 576–580.

[10] HORWITZ, S. Identifying the semantic and textual differences between two versions of a program. In *PLDI '90* (1990), pp. 234–245.

[11] HORWITZ, S., PRINS, J., AND REPS, T. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst. 11*, 3.

[12] HUNT, J. W., AND MCILROY, M. D. An algorithm for differential file comparison. Tech. rep., Bell Laboratories, 1975.

[13] JACKSON, D., AND LADD, D. A. Semantic diff: A tool for summarizing the effects of modifications. In *ICSM* (1994), pp. 243–252.

[14] JIN, W., ORSO, A., AND XIE, T. BERT: a tool for behavioral regression testing. In *FSE'10* (2010), ACM, pp. 361–362.

[15] KAWAGUCHI, M., LAHIRI, S. K., AND REBELO, H. Conditional equivalence. Tech. rep., MSR, 2010.

[16] KUEHLMANN, A., AND KROHM, F. Equivalence checking using cuts and heaps. In *DAC* (1997), pp. 263–268.

[17] MINÉ, A. The octagon abstract domain. *Higher Order Symbol. Comput. 19* (March 2006), 31–100.

[18] MISHCHENKO, A., CHATTERJEE, S., BRAYTON, R. K., AND EÉN, N. Improvements to combinational equivalence checking. In *ICCAD* (2006), pp. 836–843.

[19] NETHERCOTE, N., AND SEWARD, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI'07*.

[20] PERSON, S., DWYER, M. B., ELBAUM, S. G., AND PASAREANU, C. S. Differential symbolic execution. In *SIGSOFT FSE* (2008), pp. 226–237.

[21] RAMOS, D., AND ENGLER, D. Practical, low-effort equivalence verification of real code. In *Computer Aided Verification*, vol. 6806 of *LNCS*. Springer, 2011, pp. 669–685.

[22] RIVAL, X., AND MAUBORGNE, L. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst. 29*, 5 (Aug. 2007).

[23] SONG, Y., ZHANG, Y., AND SUN, Y. Automatic vulnerability locating in binary patches. In *International Conference on Computational Intelligence and Security, CIS 2009* (2009), pp. 474–477.

[24] TERAUCHI, T., AND AIKEN, A. Secure information flow as a safety problem. In *Proceedings of the 12th international conference on Static Analysis* (2005), SAS'05, Springer-Verlag, pp. 352–367.