Abstract Semantic Differencing for Numerical Programs

Nimrod Partush and Eran Yahav

Technion, Israel

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 using a partially-disjunctive abstract domain. To balance precision and cost of this representation, our domain over-approximates numerical information while preserving equivalence between correlated variables by dynamically partitioning the disjunctive state according to equivalence criteria.
We have implemented our approach in a tool called DIZY, and applied it to a number of 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 applying a patch 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., [11, 12, 10]) and has recently seen growing interest for various applications ranging from testing concurrent programs [5], understanding software upgrades [15], to automatic generation of security exploits [3].

Problem Definition We define the problem of *semantic differencing* as follows: Given a pair of programs (P, P') that agree on the number and type of input and output variables, for every execution π of P that originates from an input i and a corresponding execution π' of P' that originates from the *same input* i our goal is: (i) Check whether π and π' have the same output i.e. are output-equivalent, and (ii) In case of difference in output variables, provide a description of the difference.

Existing Techniques Existing techniques mostly offer solutions based on under approximation, the most 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 [22, 24] rely on symbolic execution techniques, may miss differences, and are generally unable to prove equivalence. Previous work for equivalence checking [9] rely on unsound bounded model checking techniques to prove (input-output) equivalence of two closely related numerical programs, under certain conditions (see Section 8 for more details).

Our Approach We present an approach based on abstract interpretation [7] for producing a *sound* representation of changed program behaviors and proving equivalence between a program and a patched version of the program. Our method focuses on abstracting relationships between variables in both versions allowing us to achieve a precise description of the difference and prove equivalence. Our solution is sound in the sense that it computes an over approximation of the difference between the two versions, therefore guaranteeing equivalence when no difference is found.

We focus on output equivalence in the final state. This is sufficient as mid-execution output can be modeled as added variables in the final state. This limitation also means that we assume all program executions to be finite (i.e. equivalence/difference holds if indeed both executions terminate). Note that the definition limits program difference to the final state which alleviates the need for matching the different stages of (P, P'). Finding equivalence/difference in earlier stages of the program requires program matching (we first need to find a suitable location in both programs for checking for equivalence, otherwise it has no meaning). The problem of program matching is orthogonal and can be addressed via various techniques ranging in complexity and precision - from syntactic diff [13] to execution indexing [29] and others. In this work we employ a simple matching strategy to achieve better precision as described in Section 6. We found this technique to be sufficient for our experiments.

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:

- Differencing requires correlation of *different program executions*. The abstraction must be able to capture and compare only the input-equivalent executions, and avoid comparing ones that are not input-equivalent.
- Equivalence of abstract output values does not entail concrete value equivalence.

To address these challenges, we introduce two new concepts: (i) *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) *correlating abstract domain* - a domain for tracking relationships between variables in P and variables in P' using $P \bowtie P'$.

Correlating Program We create a single program which captures the behavior of both the original program and its patched version. A *correlating program* $P \bowtie P'$ contains both programs flow and data, however program flow is arranged so to reflect a (simple) matching between the stages of the two programs. This matching is key for precision as otherwise we will not be able to maintain equivalence throughout the entire run of the program, particularly in the face of loops.

Correlating Abstraction Abstracting relationships allows us to maintain focus on differences while over-approximating (whenever necessary for scalability) equivalent behaviors. We abstract variables of both programs together, starting off by assuming equality over all matched variables (variable matching is discussed in Section 4). Thus we can reflect relationships without necessarily knowing the actual value of variables. We focus on numerical programs and use numerical domains such as Octagon [18] and Polyhedra [8] to capture the relationship between variables. Our current implementation does not track pointer equivalences, but such equivalences can be tracked by a using a correlating shape analysis domain [1]. To maintain equivalence as much as possible, our domain was designed to represent non-convex information (e.g. so we will not immediately lose equivalence taking a condition of the form $x \neq 0$ into account). We use a powerset domain of convex sub-states. Our domain uses a partitioning strategy that abstracts together states that have the same set of equivalent variables, thus avoiding exponential blowup (as explained in Section 5). This strategy helps us preserve equivalence even across widening. Therefore 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 novel approach for computing abstract semantic difference between a program P and a patched version of the program P'. We focus on numerical programs where the values of variables have no a-priori bounds.
- We reduce the problem of analyzing the two programs P, P' to the problem of analyzing a single *correlating program* $P \bowtie P'$ that captures the behavior of P and P'.
- We present a *correlating abstract domain* that captures an over-approximation of the difference between P and P' by tracking relationships between variables in $P \bowtie P'$. The domain applies a partitioning strategy for scaling the analysis while maintaining precision in equivalence.
- We have implemented our approach in a tool based on the LLVM compiler infrastructure and the APRON numerical abstract domain library, and applied it to several real-world programs. 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 illustrating example. In Section 7 we show how our approach is applied to real-world programs. Consider the two versions of a program for computing sign in Fig. 1, inspired by an example from [25]. For these programs, we would like to establish that the output of $sign$ and $sign'$ differs *only* in the case where $x = 0$ and that the difference is $sgn = 1 \neq sgn' = 0$.

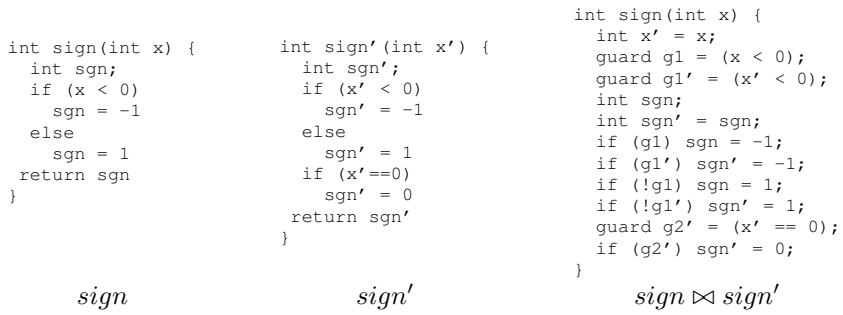

Fig. 1: Two simple implementations of the *sign* operation and their correlating program.

Separate Analysis is Unsound As a first naïve attempt to achieve this, 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 an interval domain [8] would yield that in both programs the result ranges in the same interval $[-1, 1]$, missing the fact that $sign$ never returns the value 0 where $sign'$ does.

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` and `sgn'`. This requires a joint representation in which these relationships can be tracked.

As our approach dictates the joint analysis of two programs for maintaining variable relationships, we need to determine an order in which the different stages of the programs are analyzed. One solution would be to analyze the programs sequentially. However, such an analysis will be forced to retain full path sensitivity, withholding over-approximation, since abstracting together paths will result in a non-restorable loss of equivalence. For example, analyzing $sign$ first will result in an abstract state where $\sigma = sgn \mapsto [-1, 1]$. As we continue on towards $sign'$, we could never restore in σ the fact that `sgn'` is equal to `sgn` for all paths except where x is zero.

Intuitively, establishing equivalence using the sequential composition $P; P'$ requires 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' does not allow maintaining variable relationships under abstraction.

Correlating Program To address these challenges, we construct a *correlating program* $P \bowtie P'$ where operations of P and P' are interleaved to achieve correlation throughout the analysis. Fig. 1 shows the correlating program $sign \bowtie sign'$. The programs were transformed to a guarded command language form to allow for interleaving. A key feature of the correlating program for closely related program versions is the ability to keep matched instructions, that appear in both versions, closely interleaved. This allows the analysis to better maintain relationships as the program executions are better aligned. Using the correlating program, we can directly track the relationship between `sgn` in $sign$ and its corresponding variable `sgn'` in $sign'$.

We note that the set of tracked relationships is determined by a matching of P and P' variables denoted VC and defined in Section 4. We match variables in the two versions using variable names as we found that these do not vary greatly over patches. However, this matching can also be provided by the user.

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.

Correlating Abstract Domain We introduce a *correlating abstract domain* that tracks relationships between corresponding variables in P and P' using the correlating program $P \bowtie P'$. Unfortunately, any domain with convex constraints will fail to capture the precise relationship between variables in many cases. For example, using the polyhedra abstract domain [8] to analyze the sign example from Fig. 1, the relationship between the `sgn` and `sgn'` variables 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 (as we do not explicitly know that $\equiv_{sgn}$), we still know nothing about the difference between the programs.

An obvious, but prohibitively expensive, solution to the problem is to use disjunctive completion, moving to a powerset domain where the abstract state is a set of convex objects (e.g., set of polyhedra). A state in such domain is a set of convex abstract representations (e.g., polyhedra [8] or octagon [18]). For example, analyzing $sign \bowtie sign'$ using a powerset domain would yield:

$$\sigma_1 = \{x = x' < 0, sgn = sgn' \mapsto -1\}, \sigma_2 = \{x = x' \mapsto 0, sgn \mapsto 1, sgn' \mapsto 0\}$$
$$\sigma_3 = \{x = x' > 0, sgn = sgn' \mapsto 1\}$$

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

Partitioning As the goal of this work is to distinguish equivalent from dissimilar 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.

For example partitioning the above-mentioned result of analyzing $sign \bowtie sign'$ according to our criteria would abstract behaviors σ_1 and σ_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 σ_2.

$$\sigma_1 = \{x = x', sgn = sgn' \mapsto [-1, 1]\}, \sigma_2 = \{x' = 0, sgn \mapsto 1, sgn' \mapsto -1\}$$

To reduce state size, we must perform partitioning dynamically during analysis. This cannot be achieved using a sequential composition $P; P'$. Intuitively, this is because an operation in P has to "wait" for its equivalent operation to occur in P'. To overcome this, our correlating program $P \bowtie P'$ interleaves P and P' commands, and informs the analysis when programs have reached a point where correlation may be established by annotating $P \bowtie P'$ with special markers called *correlation points* denoted CP and defined also in Section 6.

```
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;
}
```

Fig. 2: Two equivalent versions of a looping program for partial array summation.

Widening 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 [9, 16, 22, 24]. To overcome this, we define a widening operator for our domain, based on the convex sub-domain widening operator (e.g., interval, octagon, polyhedra). The main challenge here, as our state is a set of convex objects belonging to the sub-domain, 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. 2. Equivalence for these two small programs cannot be established soundly by approaches based on under approximation. To emphasize the importance of the correlating program, we will first show the result of an analysis of $sum; sum'$ which will be:

$$\sigma_1 = \{len = len' \leq 1, result = result' \mapsto 0\}, \sigma_2 = \{len = len' > 1\}$$

This loss of equivalence occurred due to the inability to 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 merged 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. 3, allows maintaining equivalence, as the loops are interleaved to allow establishing $\equiv_{result}$ as a loop invariant. This invariant survives the widening process to prove equivalence at the end as the result would be: $\sigma_1 = \{\equiv_{result}\}$. We note that we implicitly assume equivalence in array content for sum and sum'.

3 Preliminaries

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

- Var, Val, Loc denote the set of program variable identifiers, variable values and program locations respectively. Program locations are also denoted lab for label. The labels $begin$ and end mark the start and exit locations of the program.

```
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';
  }
}
```

Fig. 3: $sum \bowtie sum'$

- A concrete program state σ is a tuple $(loc, values) \in \Sigma$ mapping the set of program variables to their concrete value at a certain program location loc. 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 relation and Σ_0 is a set of initial states of the program.
- 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. The set of all possible traces for a program is denoted $[\![P]\!]$. We also define $last : \Sigma_P^* \rightarrow \Sigma_P$ which returns the last state in a trace.

We note that 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 assuming output-equivalence (for functions that were proven to be equivalent) or by inlining them (this work excludes recursion).

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*, but only those that originates from the same input. 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 with a mapping that establishes a correspondence between variables in P and P'. Thus concrete state differencing is restricted to comparing values of corresponding variables.

Definition 1 (Variable Correspondence). *A* variable correspondence $VC \subseteq Var \times Var'$*, is a partial mapping between two sets of program variables. The* VC *mapping can be taken as input from the user however, our evaluation indicates that is sufficient to use a name-based mapping for a program and a patched version:*

$$VC_{EQ} \triangleq \{(v, v') | v \in Var \wedge v' \in Var' \wedge name(v) = name(v')\}$$

Definition 2 (Concrete State Delta). *Given two concrete states* $\sigma \in \Sigma_P$*,* $\sigma' \in \Sigma_{P'}$*, and a correspondence* VC*, the concrete state delta is defined as:*

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

Informally, $\triangle_S$ means the "part of the state σ where corresponding variables do not agree on values (with respect to σ')". 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 $\triangle_S^- = \triangle_S(\sigma, \sigma')$ which means the values of the state that was "removed" in P' and $\triangle_S^+ = \triangle_S(\sigma', \sigma)$ which stands for the values "added" in P'. When there is no observable difference between the states we get that $\triangle_S^+(\sigma, \sigma') = \triangle_S^-(\sigma, \sigma') = \emptyset$, and say that the states are *equivalent* denoted $\sigma \equiv \sigma'$.

Example 1 *Consider two concrete states* $\sigma = (x \mapsto 1, y \mapsto 2, z \mapsto 3)$ *and* $\sigma' = (x' \mapsto 0, y' \mapsto 2, w' \mapsto 4)$ *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,* $\triangle_S^+ = \{(x' \mapsto 0)\}$.

We now use our notion of concrete state difference to define the difference between concrete program traces.

Definition 3 (Trace Delta). *Given two traces* $\pi \in [\![P]\!]$ *and* $\pi' \in [\![P']\!]$ *that* originate from equivalent input states*, we define the trace delta as simply the difference between the traces final states. Formally:* $\triangle_T(\pi, \pi') = \{\triangle_S(last(\sigma), last(\sigma'))\}$

The definition adheres to our problem definition in Section 1, where we defined program difference as difference between matched variables in the terminating state. Since $\triangle_T(\pi, \pi')$ is based on state difference, we define $\triangle_T^+$ and $\triangle_T^-$ similarly to their underlying states difference operations.

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.

4.2 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. This allows us to define the *correlating abstract semantics* which is key for successful differencing.

Definition 4 (Correlating Concrete State). *A correlating concrete state* $\sigma_{\bowtie} : Var \cup Var' \rightarrow Val$ *is a unified concrete state, mapping variables from both programs* (P, P') *to their values.*

Definition 5 (Correlating Concrete Trace). *A correlating trace* $\pi_{\bowtie}$*, is a sequence of correlating states ...,* $\sigma_{\bowtie_i}$*, ... describing an execution of* $P \bowtie P'$*.*

Note that an attribute of the correlating programs (as defined in Section 6) is that it restricts to traces that originate from equivalent input states i.e., $\sigma_{\bowtie_0} \equiv \sigma'_{\bowtie_0}$.

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 correlating 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 [25].

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 operator 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. 4.2.

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 $\Sigma_{P \bowtie P'}$. 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 [8] and the octagon abstract domain [18].

Definition 6 (Correlating Abstract State). *A correlating abstract program state $\sigma^\sharp \in Lab \rightarrow 2^{D^\sharp_{Guard} \times D^\sharp_{Var}}$, is a mapping from a correlating program label $l_\bowtie$ to a set of pairs $(ctx, data)$, 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 Guarded command language (GCL) format (see Section 6) from original program variables as there need not be any relationships between guard and regular variables.

5.2 Abstract Correlating Semantics

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

Table 1: Abstract transformers

Tab. 1 describes the abstract transformers. The table shows the effect of each statement on a given abstract state $\sigma^\sharp = l_\bowtie \mapsto S$. 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 only contains the operations described in Tab. 1 (this is achieved by the GCL format). We also assume that for $[\![g := e]\!]^\sharp$ operations, e is a logical operation with boolean value.

Next, we define the abstraction function $\alpha : 2^{\Sigma^*_{P \bowtie P'}} \rightarrow 2^{D^\sharp \times D^\sharp}$ that abstracts together a set of concrete correlating traces T. As in our domain traces are abstracted together if they share the exact same path, we first define an operation $path : \Sigma^*_{P \bowtie P'} \rightarrow Lab^*$ 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 follows:

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

where $\beta(\sigma) = \langle \beta_{D^\sharp}(\sigma|_{Guard}), \beta_{D^\sharp}(\sigma|_{Var})\rangle$ i.e. applying the abstraction function of the abstract sub-domain $\beta_{D^\sharp}$ on parts of the concrete state applying to $Guards$ (denoted

$\sigma|_{Guard}$) and $Vars$ (denoted $\sigma|_{Var}$) 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 correlating program will be represented by a single sub-state of the sub-domain. As a result, all *trace prefixes* that follow the same path to $l_{\bowtie}$ 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. 4 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 unbounded as the number of paths in the program may be exponential and even infinite (due to loops).

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

Fig. 4: Single path differentiation candidates

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 $\sigma^\sharp = l_{\bowtie} \mapsto S$ with acceptable loss of precision. This reduction via partitioning can be achieved by joining the abstract sub-states in S (using the standard join of the sub-domain). However this can only be accomplished after first deciding which of the sub-states should be joined and then choosing the program locations for the partitioning to occur. To choose a strategy, we start by taking a closer look at the final state of the fully disjunctive analysis of Fig. 1:

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

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 (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. To 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.

```
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' = 0;
    if (r') 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';
  }
}
```

Fig. 5: Patched `sum''` and correlating `sum` $\bowtie$ `sum''`

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. We have the widening operator of our sub-domain at our disposal, but we are faced with the question of how to lift this operator, i.e., which pairs of sub-states $\langle ctx, data \rangle$ from $\sigma^\sharp$ should be widened with which. This problem has been addressed in the path in other settings [2], and our approach can be viewed as a specialized form of lifting that is tailored for tracking equivalences. A first viable 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. 3, 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 to apply the strategy to the more complex $sum \bowtie sum''$ of Fig. 5. 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:

$$\sigma^\sharp = [d_1 = \langle(\neg r'), (len \leq max, result = 0, \equiv_{len,result})\rangle,$$
$$d_2 = \langle(r'), (len > max, retval' = -1, result = 0, \equiv_{len,result})\rangle]$$

As we further advance into the loop, d_1 will maintain equivalence but d_2 will continue to update the part of the state regarding untagged variables (since r' is $false$), 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-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:

$$\sigma_k^\sharp = [d_1 = \langle(\neg r', g, \equiv_g), (len \leq max, i = 2k+1, \equiv_{i,len,result})\rangle,$$
$$d_2 = \langle(r', \neg g, g'), (len > max, retval' = -1, result' = 0, i' = 2k+1, i = 1, \equiv_{len})\rangle]$$

And:

$$\sigma_{k+1}^\sharp = [d_1 = \langle(\neg r', g, \equiv_g), (len \leq max, i = 2k+3, \equiv_{i,len,result})\rangle,$$
$$d_2 = \langle(r', \neg g, g'), (len > max, retval' = -1, result' = 0, i' = 2k+3, i = 1, \equiv_{len})\rangle]$$

As we can identify the states predecessors by simply matching the guards. d_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 an abstract 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. When equivalence could not be proved, we report the offending states and apply a differencing algorithm for extracting of the delta. Fig. 4 shows an example of where our analysis is unable to prove equivalence, although part of the state does maintain equivalence (specifically for $x = 0$). This is due to the abstraction being too 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.

Definition 7 (Correlating Abstract State Delta). *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:

1. $d_\equiv$ is a state abstracting the concrete states *shared* by the original and patched program. Obtained by computing: $d_\equiv \triangleq d|_{V=V'} \equiv d \sqcap \bigwedge\{v = v' | (v, v') \in VC\}$.
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 $P \times P'$ 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 2 *Applying the algorithm on Fig. 4'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 note that as a sub-state is basically a conjunction of constraints, negating it by splitting to constraints and negating each individually reflects correctly the effect of negating a conjunction as we are left with a disjunction of negations, as seen in step 2. We also 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. 7 in Appendix A.

From this point forward any mention of "delta" (denoted $\triangle$) refers to the correlating abstract state delta ($\triangle_A$). 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 describe how to construct a correlating program $P \bowtie P'$. The process attempts to find an interleaving of programs for a more precise differentiation. The construction also instruments $P \bowtie P'$ with the required correlation points CP which define the locations for our partitioning. We also allow a user defined selection of CP.

6.1 Construction of $P \bowtie P'$

The idea of a correlating program is similar to that of self-composition [27], but the way in which statements in the correlating program are combined is designed to keep the steps of the two programs close to each other. 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.

The correlating program is an optimized reduction over $P \times P'$ where not all pairs of $(\sigma^\sharp, \sigma'^\sharp)$ are considered, but only pairs in a controlled execution, where correlating instructions in P and P' execute adjacently. This allows for superior precision.

The input for the correlation process are two C programs (P, P'). 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 guards) and control. Function calls are either inlined, in case equivalence could not be proven for them, or left as is, in case they are equivalent or are external system calls. Continuing the construction process, a syntactical diff [13] 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 $I_\triangle$ specifying for each command in I, I' whether it is an added command in P' (for I') marked $+$, a deleted command from P (for I) marked $-$, or a command existing in both versions marked $=$. 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 $I_\triangle$ and use it to construct the correlating program. We remind that since I, I' contain 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_G or P'_G should be taken next and put in the result. The construction goes as follows: iterate over $I_\triangle$ and for every command c (c') labeled l_c ($l_{c'}$):

- read P_G (P'_G) 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_G (P'_G).

The construction is now complete. We only add that at the start of the process, we strip P'_G of its prototype and add declarations for the tagged input variables, initializing

them to the untagged version (thus assuring $P \bowtie P'$ will only co-execute traces that originate from the same input for P and P'). 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 to use dynamic analysis and testing techniques such as fuzzing [21] and directed automated testing [4] on the correlating program in our future work.

7 Evaluation

We evaluated DIZY on a number of real world programs where the patches affect numerical variables. As benchmarks, we used several programs from the GNU core utilities, as well as a few handpicked patches from the Linux kernel and the Mozilla Firefox web browser. We also include results for illustrative examples used throughout the paper.

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 correlating them. This also allows 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 Interval, Octagon [18] and Polyhedra [8]. All of our experiments were conducted running on a Intel(R) Core-i7(TM) processor with 4GB.

7.2 Results

Tab. 2 summarizes the results of our analysis. The columns indicate the benchmark name, lines of code for the analyzed program, the number of lines added and removed by the patch, whether it required widening, and the result of each benchmark run alongside its run time in minutes. We included three different setting in the results: with and without partitioning and with an Interval, Octagon [18] and Polyhedra [8] abstract domains. Generally, the results are ordered in increasing order of precision from left to right. Results marked with ✓presented abstract states with acceptable precision i.e., mostly variables that indeed differ between variables were reported, and the description of the difference was useful for producing actual values for the differencing variables.

Table 2: Experimental Results

Name	**#LOC**	**#P**	**Widen**	Interval		Octagon		Polyhedra	
				Part	**No Part**	**Part**	**No Part**	**Part**	**No Part**
remove	16	4	N	✗(0)	✗(0)	✓(0:03)	✓(0:03)	✓(0:01)	✓(0:01)
copy	44	2	N	✗(0:33)	✗(0:33)	✓(0:23)	✓(3:11)	✓(0:07)	✓(0:47)
fmt	42	5	Y	✗(0:16)	✗(13:20)	✗(3:13)	TO	✓(0:22)	✓(1:46)
md5sum	40	3	Y	✓(0:04)	✓(0:15)	✓(5:24)	TO	✓(1:38)	✓(5:52)
pr	100	10	Y	✗(2:35)	TO	TO	TO	✓(18:49)	TO
savewd	86	1	N	TO	TO	✓(2:53)	✓(12:37)	✓(0:46)	✓(2:08)
seq	23	15	Y	✗(0:25)	✗(2:04)	✗(12:21)	TO	✗(3:24)	✗(8:12)
addr	77	1	N	✗(0:14)	✗(0:46)	✓(20:00)	TO	✓(6:46)	TO
nsGDDN	47	11	N	✗(0:02)	✗(0:21)	✗(0:24)	✗(1:56)	✓(0:11)	✓(0:35)
sign	8	2	N	✗(0)	✓(0)	✓(0)	✓(0)	✓(0)	✓(0)
sum	7	5	Y	✗(0:03)	✗(0:10)	✗(0:12)	✗(0:33)	✓(0:04)	✓(0:14)
nested	10	1	Y	✗(1:02)	TO	✗(0:35)	✗(1:37)	✓(0:12)	✓(0:30)

As precision increases, the resulting delta was more precise and contained more numerical information describing the difference. Results marked with ✗ produced false positives, reporting equivalent variables as different or providing too abstract of a description of the difference (i.e., $\top$). Results marked in TO represent runs that were stopped after 20 minutes. In either case, the results maintained soundness (equivalence was never reported falsely).

Runs without partitioning presented the most precise results with the most detailed abstract states describing the differencing paths. However this setting could not be applied towards all benchmarks since it leads to state explosion as shown by larger benchmarks that timed out. Applying partitioning allowed us to scale the analysis while maintaining precision. Results from runs that included partitioning described difference with less detail since some numerical data was abstracted away.

As expected, the Interval domain usually produced the fastest, least accurate results, while maintaining soundness as difference was reported for the appropriate variables but numerical data was almost completely abstracted away. In some case, like in the `copy` benchmark, Interval performed worse than Octagon and Polyhedra (in run time) for runs with partitioning. This is due to the Interval domain's limited ability to capture variable relationships which led to the partitioning algorithm failing in grouping together the different sub-states (as the equivalences they kept varied greatly). This resulted in a close to $2^{|VC|}$ number of equivalence groups.

Surprisingly, runs using the Octagon domain presented poor performance (run time), even compared to the more expensive Polyhedra domain, with less precision. This is due to the Octagon domain being less successful in capturing equivalences as it is built upon linear inequalities. This meant that more constraints were needed to represent variable equality, resulting in bigger states and a slower analysis.

The `addr` and `nsGDDN` benchmarks taken from the `net/sunrpc/addr.c` module in the Linux kernel SUNRPC implementation v2.6.32-rc6 and Firefox 3.6 security advisory CVE-2010-1196 (adapted to C from C++) respectively. The results produced by DIZY can be directly used towards exploiting known security flaws mentioned in advi-

sories from which these patches originate, as the resulting abstract state describes the difference between versions which is exactly the range of exploitable values.

```
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] != ')') {
    i--;
  }
  ...
}
```

Fig. 6: Original and patched version of coreutils `md5sum.c`'s `bsd_split_3` procedure

In the `md5sum` benchmark, all paths in the programs contain loops and only some of them maintain equivalence. Fig. 6 shows part of the benchmark that was patched to disallow 0-length inputs (patch line is marked with '+'). 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. As the partitioning maintains equivalence, the path where $s\_len = s\_len' \mapsto 0, ret \mapsto false, ret' \mapsto true$ will not be abstracted together with all other paths (that maintain equivalence). The offending path will be widened separately, precisely reporting difference in the final program state for the particular value.

The `seq` benchmark presented poor results, reporting difference on all variables although the semantic difference is small. This is due to the patch introducing a considerable amount of structural syntactic change to the code. We added the `nested` benchmark to demonstrate results for a simple nested loop program correlated with itself.

8 Related Work

Our work has been mainly inspired by recent work identifying program differencing as having vast security implications [3, 26] as well as advancements made in the field of under-approximations of program equivalence [9, 16, 22, 24].

The problem of program differencing is fundamental [10] and early work mainly focused on computing syntactical difference [13]. These solutions are an important stepping stone and we used syntactical diff as a means to achieve interleaving of programs in our correlating program. Another possibility for creating this program is to rely on the editing sequence that creates the new version from the original program [11].

We rely on classic methods of abstract interpretation [7] for presenting an over approximating solution for semantic differencing and equivalence. To achieve this we devised a static analysis over a correlating program. The idea of a correlating program is similar to that of self-composition [27] 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 [9, 22, 24], disregarding possible program correlation.

We base our analysis on numerical abstractions [8, 18] that allow us to reason about variables of different programs. The abstraction is further refined in a way similar to trace partitioning [25] with an equivalence-based partitioning criteria.

Jackson and Ladd [14] 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.

Several works on the problem of equivalence of combinatorial circuits [17, 19, 6] made important contributions in establishing the problem of equivalence as feasible, producing practical solutions for hardware verification.

Symbolic execution methods [22, 24] 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 [22] presents an over approximating description of difference and the other [24] 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 [9] 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.

In previous work regarding translation validation [23, 20, 30], in order to establish equivalence for a (looping) code fragment being translated or optimized by a compiler, a simulation relation between the basic blocks of the translated code is found. This method is limited in the context of semantic differencing as, for instance, a simulation relation for examples such as Fig. 2 cannot be automatically established (it needs to be crafted manually as this is not one of the classic transformations). However, the correlating program method we propose is generic enough to establish equivalence for many cases, without requiring special tailoring.

9 Conclusions

We presented an abstract interpretation approach for program equivalence and differencing. We defined a correlating program construct, that allows reasoning over both programs and establishing of equivalence. We defined a correlating abstract domain, that allows us to maintain variable relationships. This partially disjunctive domain allows to differentiate equivalent from differencing paths and we introduce a dynamic partitioning strategy to abstract together paths according to equivalence criteria and avoid exponential blowup. We also defined a widening operator for the disjunctive domain, which over approximates looping paths and is able to maintain equivalences for programs with unbound loops. We showed that this approach is feasible and can be applied successfully to challenging real world patches.

References

1. AMIT, D., RINETZKY, N., REPS, T., SAGIV, M., AND YAHAV, E. Comparison under abstraction for verifying linearizability. In *CAV'07*.
2. BAGNARA, R., HILL, P. M., AND ZAFFANELLA, E. Widening operators for powerset domains. *Int. J. Softw. Tools Technol. Transf. 8*, 4 (2006), 449–466.
3. 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.
4. 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.
5. CHAKI, S., GURFINKEL, A., AND STRICHMAN, O. Regression verification for multi-threaded programs. In *VMCAI'12*.
6. CLARKE, E. M., AND KROENING, D. Hardware verification using ansi-c programs as a reference. In *ASP-DAC* (2003), pp. 308–311.
7. 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).
8. COUSOT, P., AND HALBWACHS, N. Automatic discovery of linear restraints among variables of a program. In *POPL'78*, pp. 84–97.
9. GODLIN, B., AND STRICHMAN, O. Regression verification. In *DAC* (2009), pp. 466–471.
10. HOARE, C. A. R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (1969), 576–580.
11. HORWITZ, S. Identifying the semantic and textual differences between two versions of a program. In *PLDI '90* (1990), pp. 234–245.
12. HORWITZ, S., PRINS, J., AND REPS, T. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst. 11*, 3.
13. HUNT, J. W., AND MCILROY, M. D. An algorithm for differential file comparison. Tech. rep., Bell Laboratories, 1975.
14. JACKSON, D., AND LADD, D. A. Semantic diff: A tool for summarizing the effects of modifications. In *ICSM* (1994), pp. 243–252.
15. JIN, W., ORSO, A., AND XIE, T. BERT: a tool for behavioral regression testing. In *FSE'10* (2010), ACM, pp. 361–362.
16. KAWAGUCHI, M., LAHIRI, S. K., AND REBELO, H. Conditional equivalence. Tech. rep., MSR, 2010.
17. KUEHLMANN, A., AND KROHM, F. Equivalence checking using cuts and heaps. In *DAC* (1997), pp. 263–268.
18. MINÉ, A. The octagon abstract domain. *Higher Order Symbol. Comput. 19* (2006), 31–100.

19. MISHCHENKO, A., CHATTERJEE, S., BRAYTON, R. K., AND EÉN, N. Improvements to combinational equivalence checking. In *ICCAD* (2006), pp. 836–843.
20. NECULA, G. C. Translation validation for an optimizing compiler. pp. 83–95.
21. NETHERCOTE, N., AND SEWARD, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI'07*.
22. PERSON, S., DWYER, M. B., ELBAUM, S. G., AND PASAREANU, C. S. Differential symbolic execution. In *FSE'08*.
23. PNUELI, A., SIEGEL, M., AND SINGERMAN, F. Translation validation. In *TACAS'98*.
24. RAMOS, D., AND ENGLER, D. Practical, low-effort equivalence verification of real code. In *CAV'11*.
25. RIVAL, X., AND MAUBORGNE, L. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst. 29*, 5 (Aug. 2007).
26. SONG, Y., ZHANG, Y., AND SUN, Y. Automatic vulnerability locating in binary patches. In *CIS'09*.
27. TERAUCHI, T., AND AIKEN, A. Secure information flow as a safety problem. In *SAS'05*, Springer-Verlag, pp. 352–367.
28. VERDOOLAEGE, S., JANSSENS, G., AND BRUYNOOGHE, M. Equivalence checking of static affine programs using widening to handle recurrences. In *Proceedings of the 21st International Conference on Computer Aided Verification* (2009), CAV '09, pp. 599–613.
29. XIN, B., SUMNER, W. N., AND ZHANG, X. Efficient program execution indexing. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation* (2008), PLDI '08, pp. 238–248.
30. ZUCK, L., PNUELI, A., FANG, Y., GOLDBERG, B., AND HU, Y. Translation and run-time validation of optimized code. *Electr. Notes Theor. Comput. Sci. 70*, 4 (2002).

A Appendix

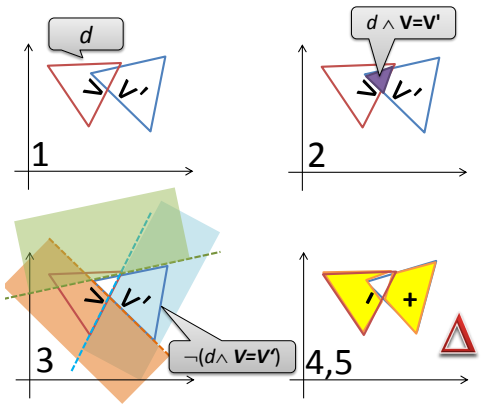

Fig. 7: Delta computation geometrical representation.