Abstract Semantic Differencing for Numerical Programs

```
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.

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. Overview

Consider the simple example program of Fig. 1, inspired by an example from [**?**]. For this example, we would like to establish that the output of $sign$ and $sign'$ only differ in the case where $x = 0$ and that the difference is $sgn = 1 \neq sgn' = 0$. An optimal abstract characterization of behavior is shown in Fig. 2 as the abstraction precisely captures and describes the states of equivalence σ_1, σ_3 while the difference is captured by σ_2.

As a first naive attempt 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

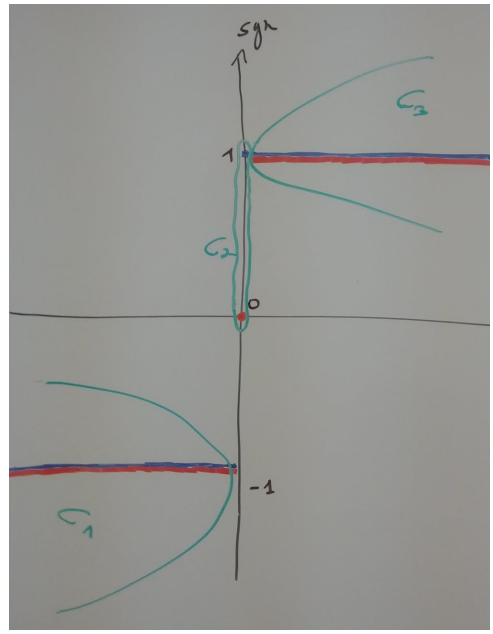

Figure 2: Abstract characterization of $sign$ (blue) and $sign'$ (red) behaviors

a interval analysis [**?**] 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 as depicted in Fig. 3. Furthermore, this result entirely ignores how x affects the value of sgn thus we would have no means to differentiate equivalent inputs from offending ones (e.g. we will get the same result for the $-1 * sign$ function).

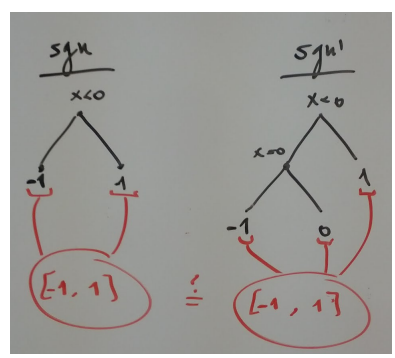

Figure 3: Interval analysis unsound comparison for $sign$ and $sign'$

To establish equivalence under abstraction, we need to abstract relationships between the values of variables in $sign$ and $sign'$ under the assumption of equivalence of input. Specifically, we need to track the relationship between the values of `sgn` in both versions and see whether we can establish their equivalence. Tracking relationships dictates performing a *joint analysis* that employs a *correlating ab-*

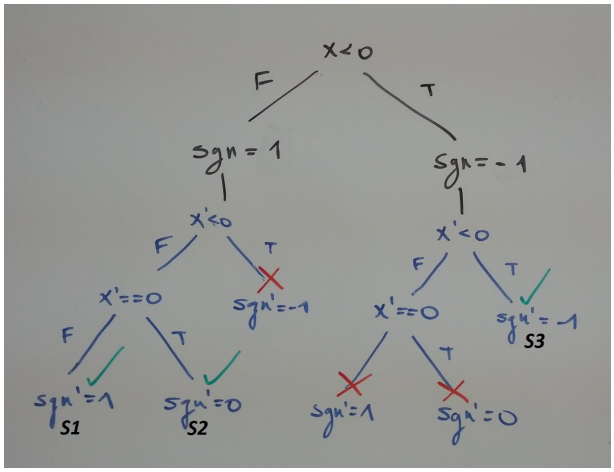

Figure 4: Joint $sign; sign'$ analysis

straction that will allow us to bind variables of both programs in one abstract state.

A correlating-oriented abstraction is well suited for proving equivalence as it allows focusing on relationships between versions of variables while abstracting away other (numerical) information allowing us to scale better. Most importantly, such an abstraction guarantees that equivalence will be reported soundly: as in a separate analysis we abstracted $\langle sgn \mapsto -1\rangle$ and $\langle sgn \mapsto 1\rangle$ towards an interval $\langle sgn \mapsto [-1, 1]\rangle$, and again for sgn' values, separately, which cannot assure equivalence, we will now abstract $\langle sgn = sgn' \mapsto -1\rangle$ and $\langle sgn = sgn' \mapsto 1\rangle$ as $\langle sgn = sgn'\rangle$ which soundly assures equivalence although all other variables information has been abstracted away.

We present an abstraction over dual program state, thats able to correlate paths that originate from the same input as well as produce a precise characterization of equivalence and difference as depicted in Fig. 2. We use a disjunctive completion powerset domain, abstracting together information of both sets of variables. For example, we produce the following constraints for the $sign$ programs: $\langle x = x' < 0, sgn = sgn' = -1\rangle \vee \langle x = x' > 0, sgn = sgn' = 1\rangle \vee \langle x = x' = 0, sgn = sgn' = 0\rangle$. These constraints are a precise abstract characterization of equivalence and difference in $sign$ and $sign'$ and we will describe how we arrive at this result.

We abstract the dual program state by analyzing both programs, sequentially (for now), and updating the shared state with data regarding both sets of variables. We allow direct relationships between versions of variables, this will be of upmost importance later on when we over approximate paths for scalability. In order to correlate paths by input and arrive at a precise disjunction (Fig. 2), we initially assume input equivalence $\vec{i} = \vec{i'}$. As we advance through the analysis of P, we will accumulate the disjunction of all possible path constraints in its final state (this is similar to trace partitioning []). At this point, as we continue to analyze P', each disjunct representing a path in P will be further refined and conjunct with all of P' paths. This will produce a precise disjunction for differencing as each path in P will be split and conjuncted with all of P' paths, while avoiding considering conjunctions that disagree on input due to our input equivalence assumption. An illustration of the joint analysis for the $sign$ example can be seen in Fig. 4 including markings for feasible and infeasible paths.

Essentially, our analysis aims to establish correspondence between paths in P and P' by first analyzing all of P paths and then attempting to correlate with P' path. Analyzing over $P; P'$ means in the worst case remembering the states along each P-path and relating them to states in the corresponding P'-path. This approach is similar to the symbolic execution approach [] where all possible correlating paths are explored individually and output is examined to determine difference whilst attempting to reach full coverage. Much like this approach, this abstraction is unfeasible for most cases, especially for programs with an unbound number of paths e.g. **loops**. To avoid this we move to a partially disjunctive domain, partitioned by *equivalence criteria*.

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 $2^{|V|}$ disjunctions in the abstract state, where V is the set of correlated (output?) variables. This criteria can be refined, by adding to V, to provide a more precise result or alternatively can become more coarse by allowing only certain equivalence classes of 2^V.

For example partitioning the result of Fig. 4 according to our criteria would abstract behaviors $S1$ and $S3$ 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 lose of precision is acceptable as it is complemented by the offending state $S2$. Still, not much is gained from this partitioning, as it is performed at the final state, where we may have already reached an exponential amount of disjunctions.

To truly gain a reduction of state size, we must perform partitioning dynamically, as the analysis is executed i.e. at earlier program locations. This cannot be achieved using a sequential composition $P; P'$. Looking at Fig. 4 we immediately see that equivalence holds only at final states. Intuitively, this is caused due to a command in one program having to "wait" for it's equivalent command to arrive from the second program. To overcome this, we present the correlating program denoted $P \bowtie P'$ which allows for earlier partitioning by "saving the need to wait" as it interleaves P and P' commands in an optimized manner, and informs the analysis that it need not wait any further and partitioning is permitted. Fig. 5 depicts the analysis of $sign \bowtie sign'$ (shown in Fig. 6) where the partitioning location is marked in red. We will further describe the specifics of creating $P \bowtie P'$ in Section **??** and only shortly say that the interleaving is chosen according to a syntactic diff process over a guarded command language version of the programs.

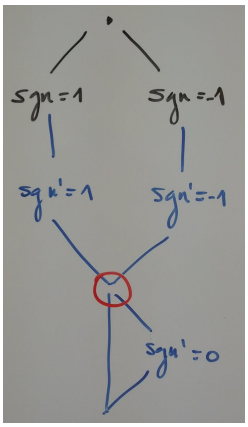

Figure 5: $sign \bowtie sign'$ analysis

Although we achieved a reduction is state size using partitioning, we have yet to account for programs with an unbound number of paths, created by loops. Unbound path lengths means a potentially unbound analysis as all paths are abstracted. This is mainly where previous approaches fall short []. To overcome this, we define a widening operator for our domain, based on the convex subdomain 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. Optimally, we would like to pair objects that adhere to the same "looping path" meaning we would want to match π_i's abstraction with a π_{i+1} that results from taking another step in the loop. This basically requires en-

```
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 6: Correlating program $sign \bowtie sign'$.

coding path information along with the sub-state abstraction. This information is acquired by simply 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 (for easier matching and better precision we separate guards from other variables in our implementation). **? simple loop example** .

? maybe show a bigger example that better shows how we benefit from partitioning

? talk about how merging early in paths may affect equivalence and differencing behaviors further along

Say how the correlating program is crucial for the success of the analysis in the case of loops

also need to say that there is the problem of choosing differencing points

2. Preliminaries

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

Program Location / Label 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.

Concrete State 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.

Program 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 to 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 inlining and we exclude recursion for now.

Concrete Trace 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.

3. Concrete Semantics

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

3.1 Concrete State Differencing

Comparing two different programs P and P' under our concrete semantics means comparing *traces*. A trace is composed of concrete states thus we first need a way to compare states. We define a state $\sigma^\natural$ as a mapping $Var \rightarrow Val$, and a difference between states would be one state mapping a different value to the same variable. But as variables may differ between programs, we require a correspondence between variables in P (i.e. Var) and those in P' (i.e. Var') and the difference will be checked over matched variables.

Variable Correspondence A variable correspondence $VC \in Var \times Var'$, is a partial mapping between 2 sets of program variables. Any variable in Var may be matched with any in Var' and vice versa. Naturally, the comparison between states will occur between values mapped to corresponding variables, as described by VC.

Our analysis may receive a user defined correspondence or use a standard name-base matching: $VC_{EQ} \triangleq \{(v, v') | v \in Var \wedge v' \in Var' \wedge name(v) = name(v')\}$ and vice versa. Our experience suggests that for most patches, the set of variables does not change over subsequent versions ($Var = Var'$) except for perhaps the addition of new variables or removal of an old one. Therefore we concluded that for our purposes, matching variables by name is sufficient for finding a precise difference.

Concrete State Delta Given two concrete states $\sigma^\natural \in \Sigma_P^\natural, \sigma'^\natural \in \Sigma_{P'}^\natural$ and a variable correspondence VC, we define the concrete state delta $\triangle_S(\sigma^\natural, \sigma'^\natural) : \Sigma_P^\natural \times \Sigma_{P'}^\natural \rightarrow 2^{Var \times Val}$ as the part of the state $\sigma^\natural$ where corresponding variables do not agree on values (with respect to $\sigma'^\natural$). Formally: $\triangle_S(\sigma^\natural, \sigma'^\natural) \triangleq \{(var, val) | (var, var') \in VC \wedge \sigma^\natural(var) = val \neq \sigma'^\natural(var')\}$. In case there is no observable difference in state we get that $\triangle_S(\sigma^\natural, \sigma'^\natural) = \emptyset$. As state delta is directional, we notice that unless it is empty, $\triangle_S$ 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'.

Example For instance, given two 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, $\triangle_S^+ = (x' \mapsto 0)$.

We defined a notion for difference between states but this is insufficient to describe difference between program traces. Program equivalence is defined as behavior, or output, equivalence given the **the same input**. Therefore we are only interested in comparing traces that originate from the same input (for the input variables correlated by VC). Every mention of trace differentiation will assume the traces agree on input. The way to differentiate traces is by differentiating their states, but which states? 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. We define it as following:

Trace Diff Points Given two traces $\pi \in [\![P]\!]$ and $\pi' \in [\![P']\!]$ that originate from the same input, 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: $DP_\pi \subseteq \{(i, i')|i \in 0..|\pi|, i' \in 0..|\pi'|\}$. 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.

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

Trace Delta Given two traces $\pi \in [\![P]\!]$ and $\pi' \in [\![P']\!]$ that originate from the same input, and a state correspondence DP_π we define the trace delta $\triangle_T(\pi, \pi') : [\![P]\!] \times [\![P']\!] \to (Loc \to 2^{Var \times Val})$ as state differentiations between all corresponding states in π and π'. Formally, for every $(i, i') \in DP_\pi$ such that $\triangle_S(\sigma_i^\natural, \sigma_{i'}^{\prime\natural}) \neq \emptyset$, $\triangle_T$ will contain the mapping $i \mapsto \triangle_S(\sigma_i^\natural, \sigma_{i'}^{\prime\natural})$, thus the result will map certain states in π to their state delta with the corresponding state' in π' (deemed interesting by DP_π). We define $\triangle_T^+$ and $\triangle_T^-$ in a similar way.

One possible choice for DP_π would be the endpoints of the two traces $\{(fin, fin')\}$ (assuming they are finite) meaning differentiating the final states of the executions or formally: $\triangle_{fin}^- \triangleq \triangle_{fin}(\pi, \pi') = \{fin \mapsto \triangle_S(\sigma_{fin}^\natural, \sigma_{fin'}^{\prime\natural})\}$ (we will also be interested in $\triangle_{fin}^+$). The final state delta may not always be sufficient for truly describing the difference between said 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). $\triangle_{fin}$ will only compare final state values and could miss what happened during the execution. This can be overcome by instrumenting the semantics such that the state will contain 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) where the values existed by, for instance, adding temporary variables, allowing for a complete differentiation at the end point. This may substantially complicate the selection of VC as it will require a matching between 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.

Defining the DP over any two traces is a daunting task since traces of separate (although similar) programs can vastly differ. If we take a look at two versions of a program depicted in Fig. **??** 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. However, one can get the sense that using program location as a means of correlation, one can produce a meaningful result that describes how the values of i range differently in the new version P. For example, if we look at all the possible values for i in 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 $2x$. We will discuss differentiation of sets of states later on as we describe the collecting semantics.

3.2 Differencing at Program Labels

Trace Delta using Program Label Given two traces π, π' and two program labels l, l' we define a trace delta based on all trace locations (states) that are labeled l, l'. 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 $\pi'_{l'}$. Formally: $\triangle_{L_{set}}(\pi_l, \pi'_{l'}) \triangleq \{\sigma^\natural \in ran(\pi_l) | \neg\exists \sigma'^\natural \in ran(\pi'_{l'}) s.t. \triangle_S(\sigma^\natural, \sigma'^\natural) = \emptyset\}$. For example consider Fig. **??**, for π, π' that originate from $x = 2$ then $\triangle_{L_{set}}(\pi_{lab}, \pi'_{lab'}) = \emptyset$ and $\triangle_{L_{set}}(\pi'_{lab'}, \pi_{lab}) = \{(i' \mapsto 3), (i' \mapsto 4)\}$. We see that this notion of $\triangle$ indeed captures a useful description of difference.

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

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

Figure 7: P, P' differentiation candidates

The problem of choosing DP is now reduced to the matching of labels as the trace indexing correspondence DP_π defined in Definition 3.1 is induced by the definition over labels. as 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} as a matching of labels between programs. Formally: $DP_{Lab} \subseteq \{(l, l')|l \in Lab, l' \in Lab'\}$. From this point on any mention of the diff-points correspondence DP will refer to label diff-points DP_{Lab}. We address the question of selection of DP_{Lab} in **??**.

Now, we will move past the concrete semantics towards *abstract semantics*. This is required as it is unfeasible to describe difference based on traces. However, we need to adjust our concrete semantics before we can correctly define this as the 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.

3.3 Concrete Correlating Semantics

In this section we shortly define, based on previous definitions, the correlating state and trace which bind the executions of both programs, P and P', together and we define the notion of delta there. Essentially, these will be states and traces of the product program $P \times P'$ but **only traces that originate from the same input are considered**. This allows us to then define the correlating abstract semantics which is key for successful differencing.

Correlating Concrete State A correlating concrete state $\sigma_\times^\natural : Var \cup Var' \to Val$ is a join of concrete state, mapping variables from a pair of programs P and P' to their values. The set of all possible correlating states is denoted $\Sigma_{P \times P'}^\natural$.

Correlating Concrete Trace A correlating trace $\pi_\times$, is a sequence of correlating states $..., < \sigma_{i_\times}^\natural >, ...$ describing a dual execution of the two programs where at every point each of the program can perform a single step. The $label_\times$, $last_\times$ and $pre_\times$

operations are defined similarly. We denote by $[\![P \times P']\!]$ the set of all traces in $P \times P'$. Again, we restrict to traces that originate from the same input i.e. $\sigma_0^\natural =_{VC} \sigma_0'^\natural$.

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.

4. 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. This comes at the cost of an acceptable lose of precision of other state information. We represent variable information using standard relational abstract domain. 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 [4]. To assure we only consider correlated paths from the product programs (that agree on input), our abstraction will initially assume equality on all inputs. This power-set domain records precise state information but does not scale due to exponential blow-up of number of paths. To reduce 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 lossy join operation). This equivalence criteria allows separation of equivalence preserving paths thus achieving better precision. We start off by abstracting the correlating trace semantics in **??**.

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 variable for return flag). We assume the abstract values in $D^\sharp$ are constraints over the variables and guards (we denote $D^\sharp_{Guard}$ for abstraction of guards and $D^\sharp_{Var}$ for abstracting original variables), and do not go into further details about 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 [] and the octagon abstract domain [].

Correlating Abstract State A correlating abstract program state $\sigma^\sharp \in Lab_\times \to 2^{D^\sharp_{Guard}\times D^\sharp_{Var}}$, is a set of pairs $\langle ctx, data\rangle \in \Sigma^\natural$ 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 (anything that is not a guard) also using the domain $D^\sharp$. We separate abstractions over guard variables added by the transformation to GCL and original program variables as there need not be any relationships between guard and regular variables.

Correlating Abstract Semantics Tab. 1 describe 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 only the aforementioned operations (specifically GCL format). We also assume that for $[\![g := e]\!]^\sharp$ operations, e is a

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

Figure 8: single path differentiation candidates

logical operation with binary value. Next, we define the abstraction function $\alpha : 2^{\Sigma^*_{times}} \to 2^{D^\sharp\times D^\sharp}$ for a set of concrete traces $T \subseteq \Sigma^*_{times}$. As in our domain traces are abstracted together if they share the exact same path, we first define an operation $path : \Sigma^*_{times} \to Lab^*$ 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: $\alpha(T) \triangleq \{\sqcup_{path(\pi)=p}\beta(last(\pi))|p \in path(T)\}$ 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 $Guards$ and $Vars$ 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. **??** 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).

Canonization Performing analysis with the powerset domain does not scale as the number of paths in the correlated program may be exponential. We must allow for reduction of state $\sigma^\sharp = l_\times \mapsto S$ with acceptable loss of precision. This reduction, or canonization as we call it, can be achieved by joining the abstract sub-states in S (using the standard precision losing join of the sub-domain) but to perform this we must first answer the following: (i) Which of the sub-states shall be joined together i.e. what is the *Canonization Strategy* and (ii) At which program locations should the canonization occur i.e. what is the *Canonization Point*. A trivial canonization strategy (we name *Join-All*) 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. 6. However, by taking a closer look at the final state of the same example $\sigma^\sharp(fin) = [\langle(g1 = 1, g2' = 0, \equiv_{g1}), (sgn = 1, \equiv_{x,sgn})\rangle, \langle(g1 = 0, g2' = 0, \equiv_{g1}), (x < 0, sgn = -1, \equiv_{x,sgn})\rangle, \langle(g1 = 0, g2' = 1, \equiv_{g1}), (x = 0, sgn = 0, sgn' = 1, \equiv_x)\rangle]$, 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 led us to devise a canonization strategy (we name *Join-Equiv*) that partitions sub-states by the set of variables which they preserve equivalence for. This bounds the super-state size at $2^{|V|}$, where V 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 canonization occurs. The first possibility, which is somewhat symmetric to the first canonization strategy, is to canonicalize at every join point i.e. after every branch converges. We name this canonization point selection as *At-Join*. A quick look at Fig. 6's state after processing the first guarded in-

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

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 $\sigma^\sharp = l_\times \mapsto S$.

struction `if (G1) sgn = -1;`, i.e. $\sigma^\sharp = [\langle(g1 = 0, \equiv_{g1}), (x \geq 0, \equiv_{x,sgn})\rangle, \langle(g1 = 1, \equiv_{g1}), (x < 0, sgn' = -1, \equiv_x)\rangle]$, (we ignored $g2'$ effect at this point for brevity) suggests that applying canonization strategy 1, will perform badly in some scenerios, specifically here as we will lose all data regarding sgn (remember, we do not assume anything regarding canonization strategy therefore the canonization by equivalence will not necessarily save us here). However if we could delay the canonization 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. Therefore we define another canonization point we call *At-Converge* which are defined during the correlating program construction process described in Section 5.4 and are basically locations where the two (guarded) programs have syntactically converged (we found two lines that are syntactically equal, sans tagging). A final strategy we define uses our *differencing points* as canonization points and delay applying canonization until then. We name this canonization point selection as *At-Diff*. We remind that diff-points are product program locations where both programs conceptually converge as they are about to emit output i.e. both programs have finished "preparing" the next portion of output therefore, if equivalence exists, it must exist now, therefore this is a prime candidate for a canonization point. Our evaluation includes applying each of our strategies along with each of the canonization points. Intuitively, the results should range from least precise using the <Join-All,At-Join> strategy and point to most precise in the <Join-Equiv,At-Diff> scenario and this is indeed the case as we show in Section 6 (not taking into account the no-canonization scenario which is naturally most precise).

Widening In order for our analysis to handle loops we require a means for reaching a fixed point. As our analysis advances over a loop and state is transformed, it may keep changing and never converge unless we apply the widening operator to 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 again we are faced with the question of how we apply this operator, i.e. which pairs of sub-states $\langle(\rangle$ctx,data) from $\sigma^\sharp$ should be widened with which i.e. which *Widening Strategy* should we apply. A first viable strategy, similar to the first canonization strategy, is to perform an overall join operation on all pairs which will result in a single pair of sub-states and then simply applying the widening to these sub-states using the sub-domain's ∇ operator. **finish this once we have a loop example for the overview**

4.1 Correlating Abstract State Differencing

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. **??** 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 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 $\sqsubseteq rd$ 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 $\alpha(\cup_{path}\triangle^+_T), \alpha(\cup_{path}\triangle^-_T)$ where deltas are grouped together by path and then abstracted. But it is not clear as to how we compute this set differencing on the correlating abstraction. Instead, we consider the geometric representation of the domain and applied operations there for the extraction of delta, as following:

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".

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

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

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

Figure 9: Delta computation geometrical representation.

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. 9.

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.

5. Semantics of Correlating Program $P \bowtie P'$

The idea of a correlating program is similar to that of self-composition [2, 5], but the way in which statements in the union 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 union program interleaves the two versions. Analysis of the union 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.

check whether Aiken paper SAS'05 does some sort of interleaving as well

5.1 Definition of Difference

5.2 General Product Program

A simple approach for a joint analysis is to construct a product program $P \times P'$ where at every point during the execution we can perform a program step (as defined in Definition **??**) of either programs. The product program has a duo-state (σ, σ') and each step updates (σ, σ') accordingly. The product program can also be seen as a concurrent run $P||P'$ where every interleaving is possible. The product program emphasizes the fact that, as described in Section 1, the notion of $\triangle$ is unclear without an established variable and label correspondence. Choosing the location where $\triangle$ is checked is a key part of identifying differences. Consider Fig. **??**, which presents a product automata of the simple program with itself, we see that even in this trivial program, although it is clear that $\triangle = \emptyset$, checking for difference in any of the non-correlating states will result in a false difference being reported. As this example demonstrates, selecting a correct label correspondence is crucial for a meaningful delta, we will elaborate on our approach for choosing DP in **??**.

```
void foo() {
    int x = 0;
}
```

Figure 10: Program P

5.3 Program Correspondence and Differential Points

Selecting the point where $\triangle$ is computed is vital for precision. As mentioned, a natural selection for diff points would be at the endpoints of traces but that loses meaning under the collecting semantics. A possible translation of this notion under the collecting semantics would be to compute delta between *all* the endpoints of the two programs i.e. $DP = \{(fin, fin') | fin \in exit(P), fin' \in exit(P')\}$ somehow differentiating the final states of the programs. This approach is problematic for two reasons:

1. Comparing all endpoints results in a highly imprecise delta. This is shown by the simple exercise of taking program with 2 endpoints and comparing it with itself.
2. This choice for DP may result in missing key differences between versions. If at some point during the calculation existed a delta that failed reaching the final state - it will be ignored. An interesting example for this is an array index receiving different bounds after a patch (but later overwritten so that it is not propagated to some final state).

Alternatively, the brute force approach where we might attempt to capture more potential diffs by selecting a diff-point after every line, will result in a highly inaccurate result as, for instance in Fig. **??**, many diffs will be reported although there is no difference. Finally, we must be careful with the selection of DP as it affects the soundness of our analysis: we might miss differences if we did not correctly place diff-points in locations where delta exists. **Our approach employs standard syntactic diff algorithm ?? for producing the correlation. This selection for DP assures soundness** . The Diff approach works well since two versions of the same software (and especially those that originate from subsequent check-ins to a code repository) are usually similar. Another important factor in the success of the diff is the guarded instruction format for our programs (as defined in Definition **??**). Transforming both programs to a our format helps remove a lot of the "noise" that a patch might introduce yet it is superior to low lever intermediate representation as it retains many qualities (such as variable names, conditions, no temporaries, etc.). **See Appendix ?? for examples illustrating said benefits and qualities** . There are alternate ways for creating the correspondence such as **graph equivalence, etc.** , this could be a subject of future research. Calculating delta according to DP over the product automata is a complex task as it allows both programs to advance independently. We formulate the *correlating program* as a restricted product automata where we advance the programs while keeping the correlation allowing for a superior calculation of delta using our correlating abstract domain later defined in Section 4.

5.4 The Correlating Program $P \cup P'$

We will generally describe the process of constructing the correlating program. A more elaborate and formal description of the algorithm can be found in Algorithm **??**. The correlating program is an optimized structure where not all pairs of (σ, σ') are considered, but only pairs that result from a controlled execution, where correlating instructions (according to DP) in P and P' will execute together. This will allow for superior precision. As said, the main idea is to create one program which contains both versions. The correlating starts out as (exactly) the older version P (after being converted to our guarded instruction form). Afterwards a syntactic diff with P' (also transformed to guarded mode) is computed (the programs are not combined just yet). In fact, this is the point where DP is created as the diff supplies us with the correlation between labels we desire. Then P''s instructions are interleaved into the guarded P while maintaining the correlation found by the diff (matched instructions will appear consequentially). Just before a patched instruction is interleaved into the correlating, all variables that appear in it are tagged, as to make sure that the patched instructions will only affect patched variables. Thus we maintain the semantics of running both programs correctly while achieving a new construct that will allow us to analyze change more easily and precisely. **Fig. ?? holds a complete correlating program of the program in Fig. ?? and it's patched version, a graphic description as a controlled automata is shown in Fig. ??** . Note that if we view the general correlating program as a concurrent program, then this optimized program can be viewed as a partial-order reduction

applied over the concurrent program. 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. This ability allows us to use dynamic analysis and testing techniques such as fuzzing **??** and directed automated testing **??** which may produce input that lead to states approximated by $\triangle$.

5.5 Analyzing Correlating Programs

Analyzsis of a guarded correlating program has certain caveats. In

```
1  l:  guard g = (i>0);
2      if (g) i--;
3      if (g) goto l;
```

Figure 11: example program illustrating guard analysis caveat

order to correctly analyze the program in Fig. **??** we need our analysis to assume $(i > 0)$ whenever taking the true branch on the `if (g)` instruction and $(i <= 0)$ when taking the false branch. However, since the `i-` instruction invalidates this assumption we would need to update the guard assumption to $(i > -1)$ which would complicate the analysis as we would need to consider updating the guard assumption while widening etc. Our solution simply incorporates the guard's assumption the first time it encounters the guard and allows it to flow to the rest of the nodes. We are not in danger of losing the assumption during the following join as our join employs a partition-by-equivalence strategy and will not join the two states where $g, i > 0$ and $\neg g, i <= 0$.

6. Evaluation

We mainly tested our tool on the GNU core utilities, differencing versions 6.10 and 6.11. Our benchmark is composed of ? patched programs which include changes to numerical variables. Our tool was able to precisely describe the difference. We also tested our tool on a few handpicked patches taken from the Linux kernel and the Mozilla Firefox web browser.

We implemented a correlating compiler named *CCC* which creates correlating programs from any two C programs as well as a differencing oriented dataflow analysis solver for analyzing correlated programs, both tools use the 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 **??**, we normalize the input programs before unifying them for a simpler analysis. We employed the APRON abstract numerical domain library and conducted our experiments using several domains including interval, octagon and polyhedra. We found that octagon and polyhedra domain provide the same level of precision in our experiments (while interval produces many false positives).

All of our experiments were conducted running on a Intel(R) Core-i7(TM) processor with 4GB. The results we present have been trimmed down for brevity, the full version of the results can be found at **link to full results**

6.1 Results

Producing delta from abstract state Fig. 12 depicts 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 error in the original program. Analyzing the correlated program for the two versions will produce the following result:

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

Original

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

Patched

Figure 12: Original and patched version of Linux kernel `net/sunrpc/addr.c` v2.6.32-rc6 module

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

The difference is captures 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).

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 (upto 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 a *By-Equiv* canonization strategy would produce the following result for this patch:

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

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 canonization (this is feasible as the procedure does not loop) will produce:

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

Now we see that the `(unsigned)newLength > (1 « 29)` con-

```
nsresult SetTextInternal (int textLength, int aCount,
                          int aLength, int aOffset,
                          PRUnichar * aBuffer) {
  PRInt32 newLength = textLength - aCount + aLength ;
  PRUnichar * to;
    ...
    (*)1
  memcpy (to + aOffset , aBuffer ,
          aLength * sizeof ( PRUnichar ));
    ...
}
```

Original

```
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 ));
    ...
}
```

Patched

Figure 13: Original and patched version of Mozilla Firefox `/content/base/src/nsGenericDOMDataNode.cpp` v3.5-3.6.4 module

straint has been successfully encoded in two offending states, each holding a part of the problematic range.

Capturing complex delta Fig. 14 depicts a patch made to the `char_to_clump` function in version 6.11. 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. Our analysis produces the following description for the differencing point at the return point:

σ_1:	σ_2:	σ_3:
chars' = 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 $\leq$ 0

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 (the offending variables appear before each sub-state) 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. Running the analysis again while saving the initial values of variables and parameters will produce an even more precise result as following:

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

This result 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. 15 shows two version of the java `logicalValue()` method taken from [**?**], 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 $currentTime - t < 100$). The challenge in this example is of course 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 invariant: $val = \Sigma_{i=0}^{data.length} data[i]$ in order to show equality. 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 $val = valw$ constraint. As we apply widening to converge, the constraint will be kept, allowing us to establish equivalence for the looping path. Thus DIZY will report the following differencing state for the exit point of `logicalValue()`:

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

We note that the example was run in [**?**] 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. 16 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 (which result in 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 is as follows, per differencing point, with the *By-Equiv* canonization strategy:

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

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

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

```
int logicalValue(int t) {
  if (!(currentTime - 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 = currentTime - 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 15: Two versions of the `logicalValue()` procedure taken from [**?**]

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

coreutils md5sum.c v6.10

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

coreutils md5sum.c v6.11

Figure 16: Original and patched version of coreutils `md5sum.c`'s `bsd_split_3` procedure

widened analysis for all other paths (the $s\_len \neq 0$ constraint is not existing there due to canonization 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 (**we do not account for overflow at this point in our work as ???**) 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 canonization 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. **describe the canonization for the example?** If we analyze the same example with the *By-Guards* canonization strategy we get:

$(*)_1$:		
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		

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 canonization by equivalence technique.

7. Related Work

Bounded symbolic execution in CLang As prior work we used the CLang infrastructure [1] static analysis graph reachability engine in order to perform a simple and bounded state differentiation exploration. We used the existing infrastructure and it's abstract representation facilities to simply record every location where the 2 versions of the variables differ. This of course was not sufficient since it only presents a bounded solution and we will show the limitations of this method by example.

Existing work on patch-based exploit generation Brumley, Poosankam, Song and Zheng [**?**] is the prominent work addressing patch-based analysis. We differ from this work in the following aspects:

1. First, the problem definition in said work is different from our own. They aim to find an $exploit$ for vulnerabilities fixed by a certain patch. Furthermore, this exploit is defined in relevance to a $securitypolicy$ which can differ. While our goals are similar to those of [**?**], we achieve them by solving 2 extended problems of a) recording the delta between new variable values and old ones and b) producing input from said values. These problems are a superset of the problem described in [**?**] and solving them has the potential for a much more complete and sound result.
2. We aim to find differentiation between every variable changed by the patch and analyze that differentiation while they concentrate on input sanitation alone. Thus if in the patched program some variable has changed in a way that does not involve input validation, it will be disregarded: for instance if an array index variable i to a buffer B is patched by adding an assignment $i = sizeof(B)-1$, it will be ignored in the previous work while we will record that the old version of i can no longer have values greater than $sizeof(B) - 1$ and may use it for exploit generation.
3. We perform our analysis on the source code of the program and patch instead of the binary. Working on a higher level gives us much more data thus potentially allowing for more results.

Kroening and Heelan [**?**] main focus focus was producing an exploit from given input that is known to trigger a bug. No patch is involved in the process. Our goal is to produce said input from the corrected software thus [**?**] can be used to create an exploit from our results.

Song, Zhang and Sun [**?**] also relate to the patch-based exploit generation problem but their main focus is on finding similarities between versions of the binary to better couple functions from the original program with their patched counter-part, a problem that was not addressed in [**?**]. Also their method of recognizing possible exploits is degenerate and relies on identifying known input validation functions that were added to a certain path - a method that could be easily overcome.

Oh [**?**] presented a new version for the DarunGrim binary diffing tool aimed at better reviewing patched programs and specifically finding patches with security implications. The goal of the tool is to help researches who manually scan patches for the purpose of producing intrusion prevention system signatures. The tool relies mainly on syntactic analysis of patterns to produce a security implication score for procedures patches making them a candidate for manual inspection. [**?**] used the DarunGrim binary diffing tool EBDS for their experiment.

Person, Dwyer, Elbaum and Pasareanu [**?**] introduced an extension and application of symbolic execution techniques that computes a precise behavioral characterization of a program change called differential symbolic execution. As we also implemented bounded symbolic execution as our preliminary work we will discuss this method in comparison to our own.

Godlin and Strichman [**?**] developed a method for proving the equivalence of similar C programs under certain restrictions based on and existing functional verification tool. This was a basis for future work regarding equivalence and we intend to base our work upon these advances.

Kawaguchi, Lahiri and Rebelo [3] defined the concept of *conditional equivalence* meaning under which conditions (inputs) are 2 different versions of a program equivalent (i.e. produce the same output). Their goal is to keep software changes from breaking procedure contracts and changing module behavior too drastically and they achieve this by computing the conditions under which the behavior is preserved. This work indirectly addresses our problem and we believe we can leverage their techniques for producing the inputs that break the equivalence while focusing on bug triggering rather than contract breaking.

[?]

Determining corresponding components As suggested in [**?**], one possibility is to rely on the editing sequence that creates the new version from the original one. Another option is using various syntactic differencing algorithms as a base for computing correspondence tags.

their idea for computing correspondence, is to minimize the "size of change". They have two different notions of size of change.

[**?**] 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.

A. Worklist

A.1 Points to Hammer

1. a special kind of self composition, where correlated steps are kept together. This is particularly important when handling loops.

A.2 TODO

1. run on uc-klee examples.
2. Consider describing each sub-state as a single (possibly looping) path of execution in both programs that originated from the same input.

A.3 Questions

1. how come you don't need the "product program"?
2. what are the theorems that you provide? (no reason to have definitions if there are no theorems).
3. what abstract domains can we use as "underlying domains" for our abstraction? Do we have any particular requirements from the abstract domains (one requirement is being relational).
4. what makes a "patched version of a program" different from just saying "a different program"? In other words - what are the requirements on the difference between P and P'?
5. can we claim that our abstraction "forgets" paths along which equivalence is established, but keeps apart paths along which the is a difference, hoping that it will re-converge later?
6. (intuition only) what if we correlate badly and lose soundness? one can propose a 2 "trick" programs that correlating them our way gives a result that loses difference.

```
if (input % 2 == 0) goto 2 else  goto 4;
s := input+2;
goto 5;
s := input+3;
if (s>input) goto 6 else goto ERROR;
ptr := realloc(ptr,s);
// use ptr[0], ptr[1], ... ptr[input-1]
```

Figure 17: Example from [**?**]

7. WHY DO WE CHECK DIFFERENCE ABOVE THE SUB-STATE LEVEL? doesn't that mean we compare different paths? isn't that bad?

A.4 Useful text fragments

Differential static analysis is useful for regression debugging [**?**], and may also lead to an automated approach for patch-based exploit generation [**?**]. Such automated exploit generation enables the organization releasing a patch to estimate the attack surface exposed by its release. Furthermore, it enables the organization releasing the patch to reduce vulnerability to manual and automated patch-based exploitation.

References

[1] clang: a c language family frontend for llvm, 2007.

[2] BARTHE, G., D'ARGENIO, P. R., AND REZK, T. Secure information flow by self-composition. In *Proceedings of the 17th IEEE workshop on Computer Security Foundations* (Washington, DC, USA, 2004), CSFW '04, IEEE Computer Society, pp. 100–.

[3] LAHIRI, S., HAWBLITZEL, C., KAWAGUCHI, M., AND REBÊLO., H. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *CAV* (2012).

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

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