Abstract Semantic Differencing for Numerical Programs

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 original program and its patched version. Towards that end, we first construct a *union program* in which these relationships can be identified, and then use a *correlating abstract domain* to compute a sound approximation of these relationships. To 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 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** . We evaluate over 50 patches and show that for these programs, the tool often manages to establish equivalence, reports useful approximation of semantic differences when differences exists, and reports only a few false differences.

1. Introduction

When applying a patch to a procedure, the programmer has very limited means for acquiring a description of the change the patch made to the procedure behavior.

Given a program P and a patched version of the program P' our goal is to determine the difference between ⟦P ⟧ and ⟦P' ⟧.

Our goal is to describe semantic difference between two programs (program versions) P, P' at certain program points i.e. given two programs and two locations in the programs, we want to produce all program states that are possible at the given location in one program but do not exist in the other program and vice versa.

Existing Techniques Existing techniques for proving patch equivalence will only supply the programmer with a binary answer [4] as to the (input-output) equivalence a program but no description of the difference is supplied. Further work [6] allows refining the equivalence proof by providing a set of constraints under which equivalence is desired but requires the programmer to manually deduce these. Other techniques for describing difference [**?** **?**] which rely on symbolic execution supply unsound results as they are limited by loops and essentially cover a subset of program behavior. We present a novel approach which allows for a sound description of difference for programs with loops. Our technique employs methods of abstract interpretation for over-approximating the difference in behaviors, by focusing the abstraction on the *relationships* between program behaviors i.e. between variables values (data) and conditionals (path) in the two versions.

In contrast to existing techniques, our approach allows checking for equivalence in every point of execution, and for every variable, while previous approaches focus only on input-output equivalence. This enables detection of key differences that impact the correctness of the patch: if the changed behavior includes a bug manifested by a local variable (for instance: array index out of bounds), we will detect and describe it while previous work only detected it when propagated to the output and equivalence may have been reported although a bug was introduced. This also provides a challenge as we need to carefully choose the program locations where we check for difference otherwise we will spuriously detect difference.

Correlating Program Abstracting relationships allows us to maintain focus on difference while omitting (whenever necessary for scalability) parts of the behavior that does not entail difference. In order to monitor these relationships we created a *correlating program* which captures the behavior of both the original program and its patched version. Instead of designing a correlating semantics that is capable of co-executing two programs, we chose to automatically construct the correlating program such that we can benefit from the use of standard analysis frameworks for analyzing the resulting program. Another advantage of this new construct, is that you may apply other methods for equivalence checking directly on it [9] as the correlation allows for a much more fine-grained equivalence checking (between local variables and not only output).

Correlating Abstractions Our abstraction holds data of both sets of variables, joined together and is initialized to hold equality over all matched variables. This means we can reflect relationships without necessarily knowing the actual value of a variables (we can know that $x_{old} = x_{new}$ even though actual values are unknown). We ran out analysis over the correlating program while updated the domain to reflect program behavior.

To establish equivalence between correlated variables and precisely capture differences, our domain has to maintain correlating information even when other information is abstracted away.

Since some updates may result in non-convex information (e.g. taking a condition of the form $x \neq 0$ into account), our domain has to represent non-convex information, at least temporarily. We address this by working with a powerset domain of a convex representation. To avoid exponential blowup, our join operator may over-approximate numerical information as long as equivalence between correlated variables is preserved.

In some cases, it would have been sufficient to use alternative domains that are capable of representing richer information, such as interval polyhedra [3], or other numerical domains that can represent non-convex information (e.g., [**?**]). The recent donut domain [**?**] may be of particular interest for this purpose. However, the general principle of having to preserve correlating information even when information about the values is abstracted away, holds in all of these cases.

In this paper, we present a technique based on abstract interpretation, that is able to compute an over-approximation of the difference between numerical programs or establish their equivalence when no difference exits. The approach is based on two key ideas: (i) create a *union program* that captures the behavior of both the original program and its patched version; (ii) analyze the union program with a *correlating domain* that captures relationships between values of variables in the original program and values of variables in the patched version.

The idea of a union program is similar to that of self-composition [2, 10], 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

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

1.1 Main Contributions

The main contributions of this paper are as follows:

- we phrase the problem of semantic differential analysis as an analysis of a union program — a single program that represents an original program and its patched version.
- we present an approach for analyzing differences over the union program using a correlating abstract domain. Our approach is sound — if there is a difference at a differentiation point, we cannot miss it. However, since we over-approximate differences, our approach may report false differences due to approximation.
- We have implemented our approach in a tool based on the LLVM compiler infrastructure and the APRON numerical abstract domain library, and evaluated it using over 50 patches from open-source software including GNU core utilities, Mozilla Firefox, and the Linux Kernel. Our evaluation shows that the tool often manages to establish equivalence, reports useful approximation of semantic differences when differences exists, and reports only a few false differences.

mention classical work like [? ?]

2. Overview

In this section, we informally describe our approach with a simple example program.

2.1 Motivating Example

Fig. 1 shows a procedure P and two patched versions of it: P_1 - an equivalent refactored version, and P_2 which introduces a bug fix (changing program behavior). We aim to find semantic differences between the original P and each of its patched versions. Ideally, when comparing P and P_1 we would be able to establish their equivalence, and when comparing P and P_2 (by location) we would be able to observe that:

- **(1)**: $s\_len_P \geq 0$ but $s\_len_{P2} > 0$.
- **(1)**: $i_P \geq -1$ but $i_{P2} > -1$.
- **(2)**: $i_P \neq 0$ but $i_{P2} > -1$.

```
while (i) {     while (i) {
 i -= 1;         i -= 2;
}               }
```

Figure 2. Example small non-equivalent loops

- **(3)**: no difference (except for the difference in `s_len` already reported in **(1)**).

We hold off addressing the question of how these program locations, which we call *Differencing Points* were selected and matched to produce an optimal result as it is an outcome of our program correlation algorithm later discussed in Section **??**.

Separate Analysis is not Sound To achieve this result using abstract interpretation, one might consider performing a separate analysis on each of the procedures and afterwards comparing the results (at the selected program points) to check for difference. However, as we are dealing with over-approximations, it would be impossible to claim equivalence based on two separate abstract states. Say we wish to compare P and P1 at location **(2)**. Optimally, the separate analysis result at that location will be $[\![P]\!]^\sharp = [\![P1]\!]^\sharp = \{i \neq 0\}$ (we assume the domain may hold non-convex data and elaborate on this issue later on in this section). Though it would be tempting to claim that the programs are equivalent at this point, it would be wrong, as each program may have arrived at the constraints by entirely different means. This can be easily seen by looking at the two loops in Fig. 2 where a separate analysis on each will yield the same $\{i \neq 0\}$ result where equivalence clearly does not exist. This key observation, which basically states that *equality under abstraction does not assure concrete equality*, dictates the use of a combined analysis as otherwise we can never hope to establish equivalence.

Correlating Program One option for performing this combined, correlating analysis is by defining a special correlating semantics which requires performing a dual analysis on both programs, whilst maintaining abstract information regarding the two sets of variables together. This is a viable option, however we chose a different approach where we would simply construct a single correlating program, denoted $P \bowtie P'$ (for the correlation of a program P and it's patched version P'), that will hold the variables and statements of the two programs. These will be interleaved in such a way where matching statements (that appear in both versions) will be adjacent, thus allowing the analysis to maintain equivalence. We transform the programs into our own guarded command language beforehand to achieve a better correlation. The variables of the programs are kept separated by tagging all variables from the newer version. A example correlating program for P and P1 from Fig. 1 can be seen in Fig. **??**. We opted for the correlating program solution since it allows us to employ standard analysis frameworks [1] and abstract domains [5]. Another advantage is that the correlating program building process supplies us with a matching of program locations, thus we are able to check for difference at appropriate locations. Lastly, since $P \bowtie P'$ is a syntactically correct program, that contains the semantic of both programs, we are able to use existing techniques of symbolic execution and equivalence checking, as used in previous work [4, 9**?**], **to achieve potentially better results** , as we allow for a more fine-grained checking and differencing (as opposed to input-output checking). We will elaborate on these issues, and on the process of building the correlating program in Section **??**.

Correlating Abstract Domain Having defined a facility for performing a joint analysis, we need to define our abstraction in such a way where we are able to maintain equivalence (under abstraction) if such exists, or provide a precise description of the difference while maintaining soundness. Considering a standard relational ab-

```
void foo(int arr[],
    unsigned len) {
 int i = len;
 i--;
    (1)
 while (i) {
    (2)
  arr[i] = i;
  i--;
 }
    (3)
}
```

P

```
void foo(int arr[],
    unsigned len) {
 int i = len;
 i--;
    (1)
 while (i) {
    (2)
 arr[i] = i-;
 }
    (3)
}
```

P1

```
void foo(int arr[],
    unsigned len) {
 int i = len;
 if (len == 0) return;
 i--;
    (1)
 while (i) {
    (2)
  arr[i] = i;
  i--;
 }
    (3)
}
```

P2

Figure 1. Example looping code and patched versions

```
1   void foo(int arr[], unsigned len) {
2       int arr'[] = clone(arr,len);
3       int len' = len';
4       int i = len;
5       int i' = len';
6       i--;
7       i'--;
8       (1)
9   l:  guard g = (i);
10  l': guard g' = (i');
11      (2)
12      if (g) arr[i] = i;
13      if (g') arr'[i'] = i'--;
14      if (g) i--;
15      if (g) goto l;
16      if (g') goto l';
17      (3)
18  }
```

Figure 3. $P \bowtie P1$ (from Fig. 1)

```
if (x>3) y = 4;
```

C1

```
z = 4;
if (x>3) y = z;
```

C2

```
if (x>3) y = 4;
z' = 4;
if (x'>3) y' = z';
```

$C1 \bowtie C2$

Figure 4. Example conditional code and patched version

```
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;
```

Figure 5. Patch taken from corutils 6.11 pr.c

straction, the only case where equivalence in variables is assured, is when both versions of the variable equal the same concrete value. As this is usually not the case (especially for unknown inputs) we explicitly force the abstraction to initially assume equivalence until proven otherwise. For example, when analyzing Fig. 3, while forcing initial equivalence, we will arrive at location **(2)** (the start of the loop) knowing that $i = i'$. The analysis will then advance over lines line 12 and line 13 and will temporarily lose equivalence but keep the $i = i' + 1$ constraint which will be used to restore equivalence as it moves past line 14. This is a key feature of our analysis where we allow *temporary equivalence divergence* with the ability to later restore it.

Correlating Abstract Domain Must Be Non-Convex As stated, explicit equivalence over abstraction is key for proving equivalence. Consequently, we cannot use a standard convex abstract domain [5], as it will result in lose of equivalence over conditionals. Let us examine the two pieces of equivalent code in Fig. 4. The optimal result here would be that $[\![P \bowtie P']\!]^\sharp \models \equiv_V$, but the use of a convex domain here can never yield this result. This is due to the fact that when we analyze C1's conditional, and get a state for each path: $\sigma_T = \{x > 3, y = 4\}$ and $\sigma_F = \{x <= 3, y = y'\}$, the following join operation under the convex domain will result in loss of all equivalence information as $\sigma_T \sqcup \sigma_F = \top$ in a convex domain. As a solution, our domain allows delaying the join operation to a later point. This means that our domain is basically a powerset domain, holding a set of convex sub-states, and our join operation is simply set union. Performing the same analysis, now with the new domain, will result in a super-state $\Sigma_1 = [\{x > 3, y = 4\}, \{x <= 3, y = y'\}]$ right after $C1 \bowtie C2$'s line 1. The analysis will now be able to restore analysis as it will pass line 2 to update $\Sigma_2 = [\{x > 3, y = 4, z' = 4\}, \{x <= 3, y = y', z' = 4\}]$ and finally after analyzing the second conditional in line 3, we will receive $\Sigma_3 = [\{x > 3, y = 4, z' = 4, y' = 4\}, \{x <= 3, y = y', z' = 4\}]$ which holds equivalence (this required pruning of unfeasible sub-states, like ones where $x <= 3$ and $x > 3$ which we perform). An appropriate concern at this point would be exponential explosion due to the potential growth in sub-states at each conditional which will be discuss promptly. An important added benefit of the powerset domain is in precision: since we keep much more refined data regarding variable correlation, we will be able to produce a more precise description of the difference. An indication for this can be drawn from Fig. 5, where using the new domain will give us a refined state which we will later use to produce a precise description of difference, something that we could not achieve with the standard convex domain.

Reducing Powerset State - 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 super-state $\Sigma = [\sigma_1, ..., \sigma_n]$ with acceptable loss of precision. This reduction, or canonization as we call it, can be achieved by joining the sub-states σ_i (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. 4. However, by taking a closer look at the final state of the same example $\Sigma_3 = [\{x > 3, y = 4, z' = 4, y' = 4\}, \{x <= 3, y = y', z' = 4\}]$, one may observe that were we to join the two sub-states in Σ_3 at this point, we would be able to preserve the coveted

$y = y'$ constraint (with the acceptable cost 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. 4's state after processing line 1 - $\Sigma_1 = [\{x > 3, y = 4\}, \{x <= 3, y = y'\}]$, while applying canonization strategy 1, shows that this may perform badly as we will lose all data regarding y. However if we could delay the canonization to a point where the two programs "converge" (at the end), we will get a more precise result which preserves equivalence. We use our *differencing points* as these convergence points and delay applying the canonization strategy until then. We name this canonization point selection as *At-Diff*. 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 7 (not taking into account the no-canonization scenario which is naturally most precise).

Handling Loops - 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 sub-states σ_i from Σ should be widened with which σ'_j in Σ' or 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 σ_i and σ'_j to result is a single convex sub-state and then simply applying the widening to these sub-states using the sub-domain's ∇ operator. If we examine applying this strategy to $P \bowtie P1$ from Fig. 3, we see that it may successfully arrive at a fixed point that also maintains equivalence. Now let us try applying the strategy to $P \bowtie P2$ as seen in Fig. 6. First we mention that as P2 introduces a return statement under the $len = 0$ condition, the example shows an extra r guard for representing a return (this exists in all GCL programs but we omitted it so far for brevity). While analyzing, once we pass line 8, our state is split to reflect the return effect $\Sigma = [\sigma_1 = \{i' = len' = 0, r = 1, r' = 0, \equiv_{i,len}\}, \sigma_2 = \{i' = len', i' \neq 0, \equiv_{r,i,len}\}$. As we further advance into the loop, σ_2 will maintain equivalence (and in fact will not require widening?) but σ_1 will continue to update the part of the state regarding un-tagged variables (since $r' = 1$ in σ_1 and it will not consider any of the commands guarded by r'), specifically it will decrement i 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 i. The problem originates from the fact that prior to widening, we joined sub-states which adhere to two different loop behaviors: one where both P and P2 loop together (that originated form $len \neq 0$) and the other where P2 has existed but P continues to loop ($len = 0$). Ideally, we would like to match these two behaviors and widen them accordingly. We devised a widening strategy that allows to do this as it basically matches sub-states that adhere to the same behavior, or loop-paths. We do this by using guard 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 for a precise description of the difference conserved in the state $\sigma_1 = len = len' = 0, -\inf < i < 0, i' = 0, r = 1, r' = 0$ showing that the original program continued to access the array at illegal $-\inf < i < 0$ indexes, where P2 stopped and exited. We also define the notion of *Widening Point* meaning at what point in the loop we widen and experiment with applying at back-edges and differencing points. **maybe add a nice picture of behaviors in P and P2 and how they are widened** .

```
1   void foo(int arr[], unsigned len) {
2       guard r = 1;
3       guard r' = 1;
4       int arr'[] = clone(arr,len);
5       int len' = len';
6       int i = len;
7       int i' = len';
8       if (r') if (len' == 0) r' = 0;
9       (1)
10      if (r) i--;
11      if (r') i'--;
12  l:  guard g = 0;
13  l': guard g' = 0;
14      if (r) g = (i);
15      if (r') g' = (i');
16      if (r) if (g) arr[i] = i;
17      if (r') if (g') arr'[i'] = i';
18      if (r) if (g) i--;
19      if (r') if (g') i'--;
20      if (r) if (g) goto l;
21      if (r') if (g') goto l';
22  }
```

Figure 6. $P \bowtie P2$ (from Fig. 1)

Producing Diff Finally, we need to give meaning to state difference and find an abstraction for precisely capturing the difference. **add pretty diff calculation picture from poster**

knowing equivalence is important, but knowing the actual difference (abstract characterization of it) is useful

3. Preliminaries

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

Program A program P,...We are able to convert any program to this format...We exclude recursion for now.

Program Location A program location $loc \in Loc$,

Program Label A program label $lab \in Lab$, is a unique identifier for a certain location in a program. Every location has a label (usually the program counter). We also define two special labels for the start and exit locations of the program as $begin$ and fin respectively.

Concrete State A concrete program state is a tuple $\sigma \equiv \langle loc, values \rangle \in \Sigma$ mapping the set of (integer) program variables to their concrete (integer) 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 $[\![P]\!]$.

Concrete Trace A program trace $\pi \in \Sigma^*$, is a sequence of states $\sigma_0, \sigma_1, ...$ 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 $\Pi \subseteq \Sigma^*$. We refer to these semantics as concrete state semantics. We also define the following standard operations on traces:

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

4. Concrete Semantics Delta

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

4.1 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. Since a state σ is a mapping $Var \rightarrow Val$ we first need a correspondence between variables in P (i.e. Var) and those in P' (i.e. Var').

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.

In our analysis, the correspondence can be determined by the user but our experience suggests that in most cases the set of variables stays the same ($Var = Var'$) over subsequent versions and in cases where Var does change, it's by 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. Thus we define our standard correlation to be: $VC_{EQ} \triangleq \{(v, v')|v \in Var \wedge v' \in Var' \wedge name(v) = name(v')\}$ and vice versa.

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

For instance, given two states $\sigma : \{x = 1, y = 2, z = 3\}$ and $\sigma' : \{x' = 0, y' = 2, w' = 4\}$ and assuming our default VC then $\triangle_S^- = \{x = 1\}$ 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}. Respectively, $\triangle_S^+ = \{x' = 0\}$.

We defined a notion for difference between states but this is insufficient to describe difference between whole runs of programs i.e. traces. Naturally, we are only interested in traces that originate from **the same input** and 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 state correspondence for choosing the states to be differentiated within the two traces. We define it as following:

Trace Diff Points Given two traces π and π', we define a trace index correspondence relation named trace diff points denoted DP_π as a matching of indexes specifying states in where concrete state delta should be computed. Formally: $DP_\Pi \equiv \{(i, i')|i \in 0..|\pi|, i' \in 0..|\pi'|\}$. 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.

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

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

Figure 7. P, P' differentiation candidates

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 traces π, π' of programs P, P' respectively, and a state correspondence DP_Π we define the trace delta $\triangle_T(\pi, \pi')$ as state differentiations between all corresponding states in π and π'. Formally, for every $(i, i') \in DP$ such that $\triangle_S(\sigma_i, \sigma'_{i'}) \neq \emptyset$, $\triangle_T$ will contain the mapping $i \mapsto \triangle_S(\sigma_i, \sigma'_{i'})$, 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: $\triangle_{in}^- \equiv \triangle_{Fin}(\pi, \pi') = \{fin \mapsto \triangle_S(\sigma_{fin}, \sigma'_{fin'})\}$ (we will also be interested in $\triangle_{Fin}^+$). It is clear that the delta is not sufficient for truly describing the difference between said traces as it will only compare final state values and will miss out on 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 it will require a matching between all of these temporary, indexed, variables that will somehow produce meaning. 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. 4.1 and the following traces generated from the input $x = 2$: $\pi = \{x = 2, i = 0\}\{i = 1\}\{i = 2\}$ and $\pi' = \{x' = 2, i' = 0\}\{i' = 1\}\{i' = 2\}\{i' = 3\}\{i' = 4\}$ (we omit labels and only mention parts of the trace where variable values change), 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. 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 may get a meaningful result that i in the patched version can range from $x + 1$ up to $2x$. We will discuss differentiation os sets of states later on as we describe the collecting semantics.

4.2 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 traces 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: $\triangle_{L_{set}}(\pi_l, \pi'_{l'}) \triangleq \{\sigma \in ran(\pi_l) | \neg\exists\sigma' \in ran(\pi'_{l'}) s.t. \triangle_S(\sigma, \sigma') = \emptyset\}$. For example, if we look at Fig. 4.1, for π, π' that originate from $x = 2$ then $\triangle_{L_{set}}(\pi_{lab}, \pi'_{lab'}) = \{\}$ and $\triangle_{L_{set}}(\pi'_{lab'}, \pi_{lab}) = \{\{i' = 3\}, \{i' = 4\}\}$. We see that this notion of $\triangle$ indeed captures a useful description of difference.

The problem of choosing DP is now reduced to the matching of labels as the trace indexing correspondence DP_{Π} defined in Definition 4.1 is no longer needed here. 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: $DP_{Lab} \equiv \{(l, l') | l \in Lab, l' \in Lab'\}$. 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 a meaningful way in Subsection 6.2.

Now, we will move past the concrete semantics towards a *collecting semantics* as a step towards abstraction. 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 collecting 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.

Collecting Semantics of a Program Label Given a label $l \in Lab$ in a program P we define the collecting semantics of a program label $states(l) \subseteq [\![P]\!]$ as all the concrete states that are possible at that label (i.e. exist in some trace reaching that label). Formally:

1. $at \equiv Lab \rightarrow (\Pi \rightarrow 2^{[\![P]\!]^*})$ for a given label l and trace π, is the set of prefixes of π that end in a state labeled l formally: $at(l, \pi) \equiv \{\pi * | \pi* \in pre(\pi) \wedge last(\pi*) = l\}$.
2. $states(l) \equiv \{last(at(l, \pi)) | \pi \in \Pi\}$.

Collecting Semantics Delta We will use the correspondence of labels in P and P' (DP), to compute the collecting semantics delta. The delta will be computed individually for each pair of matched labels by using $states(l)$ and $states(l')$. This is somewhat similar to the label-based trace delta as now we handle *sets* of states (instead of a series). We define collecting semantics delta by simply applying the state delta Definition 4.1 between each of the states in both states and removing from the result matched states or formally: $\triangle_C(states(l), states(l')) \equiv \{\sigma \in states(l) | \neg\exists\sigma' \in states(l') \cdot \triangle_S(\sigma, \sigma') = \emptyset\}$

For example, given two sets of states $C : \{\sigma_1 : \{x = 0, y = 0\}, \sigma_2 : \{x = 1, y = 2\}\}$ and $C' : \{\sigma'_1 : \{x = 0, y = 0\}, \sigma'_2 : \{x = 4, y = 5\}\}$ and using VC_Eq then $\triangle^-_C = \{\{x = 1, y = 2\}$ and $\triangle^+_C = \{\{x = 4, y = 5\}\}$. Note that $\triangle^+_C$ now obtains the meaning of "lost states" as in states which existed in the previous version and removed by the patch (similarly $\triangle^-_C$ here means "new states").

We must remember however, that the sets of states to be compared are potentially unbounded which means that the delta we compute may be unbounded too. Therefore we must use an abstraction over the collecting semantics that will allow us to represent the collecting semantics in a bounded way.

5. Abstract Correlating Semantics

In this section, we introduce our correlating abstract domain which allows bounded representation of union program state while focusing on maintaining equivalence between correlated variables. This comes at the cost of an acceptable lose of precision of other numerical information of the variables. We represent variable information using standard numerical abstract domain. To allow for temporary divergence of equivalence (due to union program structure) we keep a set of abstracts (as divergence is non-convex). As we will show, this allows for restoration of equivalence later on (if indeed equivalence holds) and in the case we are unable to converge, we will record the precise state information (which will produce a more precise diff) before aggressively joining the abstracts set into one abstract, continuing the analysis and avoiding exponential blow-up. We start off by abstracting the collecting semantics in Subsection **??**.

In the following, we assume an abstract numerical domain $ND = \langle NC, \sqsubseteq_{ND} \rangle$ equipped with operations $\sqcap_{ND}$ and $\sqcup_{ND}$, where NC is a set of numerical constraints over the variables in Var, and do not go into further details about the particular abstract domain. We also assume that the numerical domain ND allows for a sound over-approximation of the concrete collecting semantics (given a sound interpretation of program operations).

Correlating Abstract State A correlating abstract program state $\sigma^\natural$, is a tuple $\langle l, G^{nc} \rangle \in \Sigma^\natural$, where G^{nc} is a **group** of numerical constraints, each capturing relationships between numerical variables of both the original and patched programs P and P'. The semantics of $G^{nc} = nc_1, nc_2, ..., nc_n$ is $nc_1 \wedge nc_2 \wedge ... \wedge nc_n$ where each nc_i is a disjunction of numerical constraints. Let use explicitly define the operations of the domain:

- $G_1 \sqsubseteq_{CD} G_2 \iff \forall nc_1 \in G_1 \exists nc_2 \in G_2 : nc_1 \sqsubseteq_{ND} nc_2$
- $G_1 \sqcap_{CD} G_2 \equiv nc_1 \sqcap_{ND} nc_2 | nc_1 \in G_1 \wedge nc_2 \in G_2$
- $G_1 \sqcup_{CD} G_2 \equiv G_1 \cup G_2$

One major advantage of our correlating domain over using two separate domains, is the ability to preserve equivalence in the face of non-linear operations - this argument may be too thin to include .

For example, if we take our motivating example from Fig. **??** and annotate it with our correlating domain, we will get after line XXX the state XXX and after line XXX the state XXX. This example emphasizes the need for our correlating domain to hold a group of numerical constraints since Although precise, the sets of abstracts produced by interpreting our example using our correlating domain are not very informative. One cannot easily deduce whether equivalence is kept or if the state holds a difference. To answer this question we define the *correlating abstract state delta*.

5.1 Correlating Abstract State Differencing

Given a state in out correlating domain, we want to compute the version difference, if exists, at that state. However, since our input now is one correlating state holding information of both versions of variables, there is no straightforward way of defining the difference (unlike previous delta definitions Definition **??**, Definition **??**). We overcame this by treating the numerical constraints in our domains as geometrical objects and formulating delta based on that.

Correlating Abstract State Delta Given two abstract states and a correspondence VC, the correlating state delta $\triangle_A(\sigma^\natural, \sigma'^\natural)$, computes abstract state differentiations between $\sigma^\natural$ and $\sigma'^\natural$. The result is an abstract state $\sqsubseteq \sigma^\natural$ approximating all concrete values possible in P but not in P' (regarding variables that match in VC). Formally, the delta is simply $\sigma^\natural \setminus \sigma'^\natural$ but since this concept is vague to the reader and furthermore, does not exist in most domain implementation (and specifically in the ones we used) we break it down to a simpler multi-step operation as following:

1. $U \equiv \sigma^\natural \sqcap \sigma'^\natural$ is the joint state of the original and patched program. No precision is lost while joining states as they operate on different variables. This state, as well as the ability to cleanly separate variables, is achieved by symbolically executing a *union program* as defined in Section 6.
2. R is a state abstracting the concrete states shared by the original and patched program. It is achieved by computing: $R \equiv U|_{V=V'} \equiv U \sqcap \bigwedge\{v = v' | VC(v) = v'\}$.
3. $R|_V$ is the projected state where all the variables from Var' are eliminated from R.
4. $\overline{R|_V}$ is the negated state i.e. $D \setminus R|_V$ and it is computed by negating $R|_V$ (as mentioned before, all logical operations, including negation, are defined on our representation of an abstract state).
5. Eventually: $\triangle(\sigma^\natural, \sigma'^\natural) \equiv \sigma^\natural \sqcap \overline{R|_V}$ meaning it is part of the original program state $\sigma^\natural$ that does not appear in $\sigma'^\natural$ i.e. appears in the negation of R (which is the intersection of both abstract states).

Algorithm 1: Compute Abstract Difference.

```
**Input**: Abstract states $\sigma$, $\sigma'$
**Output**: Abstract difference - $\triangle(\sigma, \sigma')$
$U \leftarrow \sigma \sqcap \sigma'$
$R \leftarrow U \sqcap \bigwedge\{v = v' | VC(v) = v'\}$
$\triangle(\sigma, \sigma') \leftarrow \sigma \sqcap \overline{R|_V}$
**return** $\triangle(\sigma, \sigma')$
```

A geometrical representation of $\triangle_A$ calculation can be seen in Fig. **??**

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

5.2 Minimizing Divergence

One can see from our motivating example that it is not feasible to allow our correlating domain to keep diverging and double in size with every conditional as it will exponentially blow up the analysis run-time and memory. Instead, we employ an equivalence conserving canonization technique such that after every join will either (i) check for equivalence in in the joined state and if it is kept, join all of $G_n c$'s abstracts into one abstract, potentially losing precision but preserving equivalence. (ii) see that equivalence is not kept and allow it to converge, for now. In order for use to truly maintain equivalence after option (1) is executed, our domain must not lose precision of variable equality over join. Option (ii) entails that once equivalence is broken, our analysis will quickly explode in memory as we will no longer be able to minimize sets of constraints by joining all. We solve this by performing our canonicalization at the next diff-point, but make sure to record the "exploded" state before joining all abstracts as it is potentially *more precise* and may be used to produce a more informative delta (without losing soundness).

6. Semantics of Union Program

6.1 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 3) 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 2, 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. 6.1, 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 Subsection 6.2.

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

Figure 8. Program P

6.2 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. 6.1, 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 3). 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 5.

6.3 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 $\triangle$.

6.4 Analyzing Correlating Programs

Analyzsis of a guarded correlating program has certain caveats. In order to correctly analyze the program in Fig. 6.4 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 $\neg g, i <= 0$.

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

Figure 9. example program illustrating guard analysis caveat

7. Evaluation

We mainly tested our tool on the GNU core utilities, differencing versions 6.10 and 6.11. This benchmark included 40 patches where most of the patches (35) were a one-line patch aimed at updating the version information string in the code. Our analysis easily showed equivalence for these programs. About 10 of these patches included actual changes to numerical variables and we were 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 union compiler named *ucc* which creates union programs from any two C programs as well as a differencing oriented dataflow analysis solver for analyzing union programs, both tools use the LLVM and CLang compiler infrastructure. We analyze C code directly thus benefiting from a low number of variables as there are no temporary values as there might appear in an intermediate representation. We also benefit from our delta being computed over original variables. As mentioned in Section 6, we normalize the input programs before unifying them for a simpler analysis.

Analysis of some of our benchmarks required the use of widening. We applied a basic widening strategy which widens all cfg blocks once reaching a certain threshold. All of our experiments were conducted running on a Intel(R) Core-i7(TM) processor with 4GB.

7.1 Results

Table 1. Experimental Results

Name	Domain	#Added	#Deleted	#DiffPoints	#Diffs
dd.i	ppl	52	54	87	4
id.i	ppl	15	6	26	0
pr.i	ppl	10	3	13	1
su.i	ppl	2	2	2	0
env.i	ppl	2	2	3	0
seq.i	ppl	10	10	15	4
nice.i	ppl	3	3	36	0
test.i	ppl	3	3	14	0
chmod.i	ppl	3	3	7	0
nohup.i	ppl	2	2	18	0
paste.i	ppl	24	17	4	0
rmdir.i	ppl	3	0	3	0
users.i	ppl	3	4	30	0
chroot.i	ppl	3	3	22	0
md5sum.i	ppl	15	7	32	28
runcon.i	ppl	2	2	3	0
lbracket.i	ppl	3	3	14	0
setuidgid.i	ppl	7	4	34	12
chown-core.i	ppl	3	3	10	0

Tab. 1 summarizes the results of our analysis. The columns indicate the benchmark name and description, lines of code for the analyzed program, the number of lines added and removed by the patch, the number of diff-points generated, the numerical domain used, and the number of differences found in the analysis (i.e. number of diff-points where $\triangle \neq \varnothing$). In our benchmarks, we focused on computing intra-procedural difference between the two versions of procedures. Procedure calls presented difficulty as they potentially change global variables and local variables through pointers. We overcame this by either (i) assuming equivalence (alone) once we encounter a call to a procedure we already established as equivalent or (ii) warn that all results regarding variables touched by the procedure is un-sound. **In the majority of our benchmarks we identified calls only to library and system procedures thus we could omit their effect as they do not change variables beyond those given as parameter or those being assigned the return value.** All differences reported describe, in constraints over variables, an existing delta at that program point.

Identifying delta down the line One advantage of our analysis method is the ability to identify differences in variables that were not directly affected by the patch. Fig. 7.1 shows part of the `bsd_split_3` function that was patched by the line marked by a comment. Note that although the patch directly restricts the value range of `s_len` to above zero, our analysis is able to identify the effects on the index variable `i` and also report a lost program state of `i>-1` further down the line...

```
static bool
bsd_split_3 (char *s, size_t s_len, unsigned char **hex_digest, char **file_name) {
  size_t i;

  if (s_len == 0) return false; // Patch

  *file_name = s;

  /* Find end of filename. The BSD 'md5' and 'sha1' commands do not escape
     filenames, so search backwards for the last ')'. */

  i = s_len - 1;
  while (i && s[i] != ')')
    i--;

  if (s[i] != ')') return false;

  s[i++] = '\0';

  while (ISWHITE (s[i])) i++;

  if (s[i] != '=') return false;
  i++;

  while (ISWHITE (s[i])) i++;

  *hex_digest = (unsigned char *) &s[i];
  return true;
}
```

Figure 10. md5sum.c bsd_split_3 function patch

Non-convex delta

Maintaining equivalence in loops

8. 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 [8] 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 [7] 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. Appendix

A.1 Algorithm 2 : Convert P To Guarded Instruction Format

The algorithm is constructive i.e. it takes a procedure P and outputs the new lines for a guarded version of P. The original P is not part of the output.

- *Stage 0:* Output P's signature.
- *Stage 1:* Convert all `while` constructs to `if` and `goto` constructs.
- *Stage 2:* For each non branch instruction I:

  If I is a declaration, output it under a new block.

  Otherwise collect all branch conditions C under which I executes. Produce the code line: `if (`$\bigwedge C$`) I;`

We take a small but sufficient example to demonstrate the algorithm:

```
void foo(char Out[], int n) {
    char In[42];
    int i;
    if (n >= 42) n = 41;
    while (i < n) {
        int j = i + 1;
        In[i] = Out[j];
        i++;
    }
}
```

Stage 1 Output:

```
void foo(char Out[], int n) {
    char In[42];
    int i;
    if (n >= 42) n = 41;
l:  if (i < n) {
        int j = i + 1;
        In[i] = Out[j];
        i++;
        goto l;
    }
}
```

Stage 2 Output:

```
void foo(char Out[], int n) {
    char In[42];
    int i;
    if (n >= 42) n = 41;
    {
        int j;
l:      if (i < n) j = i + 1;
        if (i < n) In[i] = Out[j];
        if (i < n) i++;
        if (i < n) goto l;
    }
}
```

Figure 11. Algorithm 1 application example

Note that as required, the loop is implemented in means of branches and goto and that the branches do not exceed the 1 level of nesting allowed.

A.1.1 Algorithm 2 Assumptions

1. We only deal with `while` loop statements.
2. Logical statement (branch conditions) do not have side-effects i.e. they do not change variable values.

A.2 Algorithm 1 Example

We run algorithm 1 on the previous example with the following patch to line 4:

```
- if (n >= 42) n = 41;
+ if (n >= 42) n = 40;
```

We start off from the output of algorithm 2 i.e. a "guarded" program that performs simple buffer handling:

```
void foo(char Out[], int n) {
    char In[42];
    int i;
    if (n >= 42) n = 41;

    {
        int j;
l:      if (i < n) j = i + 1;
        if (i < n) In[i] = Out[j];
        if (i < n) i++;
        if (i < n) goto l;
    }

}
```

The result:

```
void foo(char Out[], int n, char Out’[], int n’) {
    char In[42], In’[42];
    int i, i’;

    if (n >= 42) n = 41;
    if (n’ >= 42) n’ = 40;

    {
        int j,j’;
l:      if (i < n) j = i + 1;
l’:     if (i’ < n’) j’ = j’ + 1;
        if (i < n) In[i] = Out[j];
        if (i’ < n’) In’[i’] = Out’[j’];
        if (i < n) i++;
        if (i’ < n’) i’++;
        if (i < n) goto l;
        if (i’ < n’) goto l’;
    }
}
```

Figure 12. Algorithm 1 Application Example

B. Worklist

B.1 Points to Hammer

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

B.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.

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

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] CHEN, L., MINÉ, A., WANG, J., AND COUSOT, P. Interval polyhedra: An abstract domain to infer interval linear relationships. In *Proceedings of the 16th International Symposium on Static Analysis* (Berlin, Heidelberg, 2009), SAS ’09, Springer-Verlag, pp. 309–325.

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

[5] JEANNET, B., AND MINÉ, A. Apron: A library of numerical abstract domains for static analysis. In *CAV* (2009), pp. 661–667.

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

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

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

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

[10] 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.