Abstract Semantic Differencing via Speculative Correlation

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

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

Abstract

We address the problem of computing semantic differences between a program and a patched version of the program. Our goal is to obtain a precise characterization of the difference between program versions, or establish their equivalence. We focus on infinite-state numerical programs, 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 use a correlating abstract domain to compute a sound approximation of these relationships which captures semantic difference. This approximation can be computed over any interleaving of the two programs. However, the choice of interleaving can significantly affect precision. We present a *speculative search algorithm* that aims to find an interleaving of the two programs with minimal abstract semantic difference. This method is unique as it allows the analysis to dynamically alternate between several interleavings.

We have implemented our approach and applied it to real-world examples including patches from Git, GNU Coreutils, as well as a few handpicked patches from the Linux kernel and the Mozilla Firefox web browser. Our evaluation shows that we compute precise approximations of semantic differences, and report few false differences.

1. Introduction

Understanding the semantic difference between two versions of a program is invaluable in the process of software development [24]. A developer applying a patch is often interested in understanding the effect of the patch in terms of added/removed program behaviors. In particular, proving that a patch does not change observable behavior (i.e., equivalence checking), has numerous applications. These include translation validation [33], verifying superoptimization [36], finding protocol bugs [34], etc.

Semantic differencing is a fundamental problem [17], that has recently seen growing interest [16, 24–26, 30, 36], with applications including testing of concurrent programs [8], understanding software upgrades [22], differential assertion checking [26], and automatic generation of security exploits [6].

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

Existing Techniques The current state of the art applies only to small code and work at the granularity of a single function. We advance the state of the art as we are able to statically produce sound results over complex looping functions with vast syntactic difference - a contribution unmatched in related work. Existing techniques to semantic differencing mostly offer solutions based on under approximation, the most prominent of which is regression testing which provides limited assurance of behavior equivalence while consuming significant time and compute resources. Other approaches for computing semantics differences [32, 34] rely on symbolic execution techniques, may miss differences, and are generally unable to prove equivalence. Previous work for equivalence checking [15] rely on unsound bounded model checking techniques to prove (input-output) equivalence of two closely related numerical programs, under certain conditions (see Section 6 for more details).

Our Approach We present an approach based on abstract interpretation [11] for computing a *sound* representation of changed program behaviors and proving equivalence between a program and a patched version of the program. Our method focuses on abstracting relationships between variables in both versions allowing us to achieve a precise description of the difference and prove equivalence. We produce a characterization of difference in the form of linear

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
OOPSLA '14, October 20–24, 2014, Portland, OR, USA.
Copyright © 2014 ACM 978-1-4503-2585-1/14/10. . . $15.00.
http://dx.doi.org/10.1145/2660193.2660245

equations over program variables. Our approach computes an over approximation of the difference between the two versions, therefore guaranteeing equivalence when no difference is found.

Equivalence under Abstraction In contrast to techniques that limit loop iterations or the range of inputs [21, 25, 32, 34], our approach uses abstraction to handle infinite-state programs. The main challenge is to abstract program executions in a way that enables capturing difference. In particular, to establish equivalence under abstraction, the states of the two programs have to be abstracted *together* such that the *relationship* between them is preserved.

This leads to the notion of a correlating semantics in which P and P' are executed together and share a joint program state (in which we can track relationships between their values). Given the correlating semantics, there are many possible (joint) executions of P and P', depending on how their steps interleave. Because the programs manipulate disjoint parts of the shared state, the result of each of these different interleavings is exactly the same.

However, under abstraction, these different interleavings may have dramatically different results. Intuitively, the reason is that the abstraction of relationships between variables is only able to capture small (or structured) differences. Once the execution of one program goes too far beyond the other program, the abstraction loses precision, and would not be able to observe equivalence even if it is restored. This is true even without loops, but is of critical importance when loops are present.

The challenge of keeping precise differences under abstraction therefore becomes one of — *what interleaving of the executions of P and P' should we pick such that the abstraction is able to precisely capture the difference?*

The interleaving problem is cardinal in equivalence checking and other approaches include establishing a simulation relation using execution traces [36], program composition using syntactic similarity [30], relying on recursion rules [15] and brute-force searching (for non-looping code) [32, 34].

One trivial solution is to try all possible interleavings, which is similar to analyzing a concurrent program $P||P'$. However, this incurs an exponential cost, and severely limits the applicability of the technique. Another trivial solution is to use the sequential composition $P; P'$ (used e.g., in [38] to reason about information flow). However, this is typically not feasible under abstraction as the distance between program executions becomes too big for the abstraction to capture with sufficient precision.

Speculative Correlation Instead, we use abstraction-guided speculative correlation to find an interleaving of P and P' that tracks differences with sufficient precision. The basic idea is to break the abstract interpretation into short segments in which we speculatively try all interleavings, followed by a greedy choice of the segment that produced the minimal abstract difference as the one to be extended further.

Main Contributions The contributions of this paper are:

- We present a novel framework for computing abstract semantic differences between a program and a patched version of the program. The main idea is to use a speculative abstract semantics that attempts to find the right correlation between programs such that their equivalence can be captured under a given abstract domain.
- We present an abstract correlating semantics that executes a program P and its patched version P' within some bounded window of divergence, and capture relationships between variables in the two programs.
- We provide a *speculative search algorithm* that aims to find an interleaving of the executions of P and P' with the minimal abstract difference. This method is unique as it allows the analysis to dynamically alternate between several interleavings. This is fundamentally different from previous methods, which compute a single interleaving and use it.
- We have implemented our approach in a tool called SCORE and applied it to a number of real-world examples. Our evaluation shows that we compute precise approximations of semantic differences, and report only a few false differences.

2. Overview

We now informally describe our approach using an example.

2.1 Motivating Example

Fig. 1 shows two versions of the `print_numbers` function taken from the GNU Coreutils `seq.c` program. We use primes (') to denote functions, variables, and program points belonging to the newer version of the program. The functions print out a sequence of numbers, starting from `first` and ending with `last`, in intervals of `STEP` size (the functions have been slightly adapted to an integer-only version, and `STEP` has been factored out as a constant). Although the syntactic difference between versions is significant, these functions only differ in cases where in `print_numbers'` (v6.10), the last element in the sequence is equal (in string form, due to formating) to the next-to-last element. In this case, the `print_extra_number'` flag will be set to true (line 15') and the last print would be skipped (by the condition in line 16'). Our goal is to automatically compute some characterization of this difference, while proving that otherwise the functions are semantically equivalent. To the best of our knowledge, none of the existing techniques [15, 30, 32, 34, 36] are able to do so.

To precisely characterize the difference for the example of Fig. 1, it is sufficient to show that at the points of output (line 9 and line 10'), the variable `x` is equivalent in both versions, *in every iteration* other than possibly the last

```
1   static void
2   print_numbers (long first, long last, ...)
3   {
4     long i;
5     for (i = 0; /* empty */; i++) {
6       long x = first + i * STEP;
7       if (last < x) break;
8       if (i) fputs (separator, stdout);
9       printf (fmt, x);
10    }
11    if (i)
12      fputs (terminator, stdout);
13  }
```

Coreutils seq.c v6.9

```
1'  static void
2'  print_numbers'(long first, long last, ...)
3'  {
4'    bool out_of_range = (last < first);
5'    if (!out_of_range) {
6'      long x = first;
7'      long i;
8'      for (i = 1; /* empty */ ; i++) {
9'        long x0 = x;
10'       printf (fmt, x);
11'       if (out_of_range) break;
12'       x = first + i * STEP;
13'       out_of_range = (last < x);
14'       if (out_of_range){
15'         bool print_extra_number = !STREQ(STR(x0),STR(x));
16'         if (!print_extra_number) break;
17'       }
18'       fputs (separator, stdout);
19'     }
20'     fputs (terminator, stdout);
21'   }
22' }
```

Coreutils seq.c v6.10

Figure 1: Original (top) and patched (bottom) version of Coreutils `seq.c`'s `print_numbers` procedure

iteration, and that the last iteration differs only when the `print_extra_number'` flag is set (we focus on a single point of output to simplify the explanation).

Equivalence Check Using Separate Analysis is Unsound As a naïve attempt to achieve this, one could analyze each version of the program separately and compare the (abstract) results. However, this is unsound, as *equivalence under abstraction does not entail concrete equivalence*. For instance, if we use a standard relational abstraction to track the values of `i` and `i'` in the following code fragments separately:

```
for (i=0; i<2*n; i++){
  printf("\%d",i);
}
```

```
for (i=0; i<2*n; i+=2){
  printf("\%d",i);
}
```

we will arrive at $\{0 \leq i \leq 2n\}$ and $\{0 \leq i' \leq 2n'\}$ which would falsely suggest equivalence, since the left version prints all values in the range, and the right version prints only the even ones. Thus, under abstraction, anything short of *explicit equality* of variables cannot be used to soundly prove equivalence.

Abstracting Relationships to Prove Equivalence To establish equivalence under abstraction, we need to abstract *relationships between the values of correlating variables* in `print_numbers` and `print_numbers'`. Specifically, we need to track the relationship between the values of `x` and `x'`. This requires a joint representation in which these relationships can be tracked.

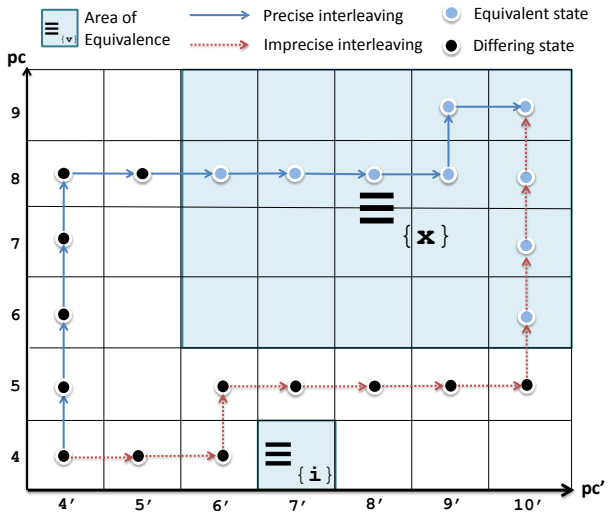

Figure 2: (Partial) Graphic description of an iteration of the speculative algorithm over `print_numbers` and `print_numbers'`

To address these challenges, we use a *correlating semantics* $[\![P, P']\!]_{\bowtie}$ that maintains a joint shared state for the two programs, and where executions of P and P' are interleaved. We use the $\bowtie$ notation for correlating semantics, states, etc. reflecting the correlation of variables from P and P'. The correlating semantics can be then abstracted using numerical domains [12, 27] to obtain shared abstract state. Such abstraction allows us, for instance, to show that when in the previous iteration $print\_extra\_number' = true$, the relationship between values of `x,x'` at the point (9, 10') of joint execution can be described by $\sigma^{\sharp}_{\bowtie} = \{\equiv_{\{first,x,last\}}, first \leq x \leq last\}$. We use the $^{\sharp}$ superscript to denote abstract entities and $\equiv_{\{first,x,last\}}$ to denote explicit equality between versions of variables i.e. $first = first'$,$x = x'$ and $last = last'$. This abstraction proves equivalence (under said clause), even though explicit values of `x` and `x'` were abstracted away through widening to overcome the loop (see Sec. 3.4).

Interleaving Determines Precision A key aspect of our technique is determining the interleaving in which the programs' instructions are analyzed. Looking at Fig. 1 we observe that to produce a precise result, our algorithm needs to analyze the programs together, carefully interleaving instructions of `print_numbers` and `print_numbers'`. We present a speculative algorithm, which performs a correlation-based search for an interleaving that will produce a minimal abstract difference (described in Sec. 4).

The abstraction drives the algorithm: The algorithm chooses an interleaving that minimizes abstract difference.

The algorithm speculatively performs analyses in all possible interleavings, and chooses an interleaving that leads to the minimal difference in its abstract state. Fig. 2 provides some intuition to how interleaving affects equivalence, and we elaborate on it later on as we describe *speculative exploration*.

This shows a key feature of our technique: interleaving of programs is not static, but is instead *computed dynamically in a speculative manner* during the analysis. In every iteration, the algorithm re-evaluates equivalence over all interleavings, and chooses a (possibly new) interleaving accordingly. This novel method is fundamentally different from previous methods where a single interleaving is chosen (as a simulation relation for instance) and analyzed. We instead allow for a dynamic interleaving, that may change throughout the analysis. We will first show how the choice of interleaving affects the precision of the resulting abstract difference and then present the algorithm that finds such interleaving.

2.2 Running Example

Tab. 1 depicts an analysis over an interleaving that is successful in forming a precise abstraction of equivalence and difference. This interleaving is shown as the solid line in Fig. 2. Tab. 1 also shows the abstract states throughout the analysis of this interleaving (column 3).

The table is comprised of `print_numbers` instructions (column 1), `print_numbers'` instructions (column 2) and the dual-program correlating state computed so far (column 3). The correlating state in column 3 is mapped to a pair of program counters, denoting values and equivalence of both programs' variables, at the specified locations. For instance, the correlating state appearing at the bottom of the second row: $(8,\ \texttt{ENTRY}') \mapsto \{\equiv_{\{STEP,first,last\}}, i = 0, x = first, x \leq last\}$ expresses that after analyzing up to line 8 in `print_numbers`, without advancing over `print_numbers'`, the resulting correlating state holds equivalence for the $STEP, first, last$ variables, alongside other constraints for $i, x, first$ and $last$. Nothing is known yet for $i', x', first'$ and $last'$ as none of `print_numbers'` lines were analyzed. We omitted most states for brevity, and noted only significant ones, for instance where equivalence was restored for a certain variable.

The analysis starts at the entry points of the two functions $(\texttt{ENTRY}\ , \texttt{ENTRY}')$, assuming equivalence over input. It then advances towards the line preceding the printing point of `print_numbers` at locations $(\texttt{ENTRY}, 8')$ and computes an abstract state where $x = first$. At this point, the analysis alternates to `print_numbers'`, interpreting instructions up to the printing point in line 10', where equivalence is restored ($x = x' = first$). It is now "safe" to advance towards the output points, as described by the subsequent row in the table.

As mentioned, the choice of interleaving affects precision: had the interleaving instead included further instructions from `print_numbers`, the analysis would report the loss of equivalence for variable x at line 9. This demonstrates how interleaving affects precision, as also observed by other approaches [30, 36].

The analysis continues from the output locations at $(9, 10')$ to analyze the rest of `print_numbers'` loop body and iteratively alternates between the versions', keeping a one-to-one loop iteration ratio, reflecting that analyzing both loops in a controlled manner is key for maintaining precision. Paths that break from the loop, when $print\_extra\_number'$ is $true$, are abstracted by states mapped to $(9, 19')$ (as they break from the loop). These paths are added to paths where $print\_extra\_number' = true$ at the end of the analysis, as one can see from the abstract state mapped to $(13, 22')$. We discuss partial disjunction later in this section.

We emphasize that our algorithm does not rely on advancing towards output locations (as one may suspect from Tab. 1). It searches for an optimal interleaving based on minimizing difference alone. The interleaving in Tab. 1 was chosen as it simplifies the explanation. Interleavings that do not alternate on output locations are equally valid (as long as they minimize difference).

Speculative Exploration in Search of Equivalence We now describe our speculative exploration algorithm which iteratively computes interleavings. We present an algorithm which computes analyses over all possible interleavings, within a given speculation window. It then proceeds to greedily select the interleaving that results in an abstract state with minimal abstract difference. Because our analysis is sound, and can never miss a difference, picking the minimal abstract difference *necessarily means better precision*. This process iteratively continues, until a fixed-point is reached.

The exploration process is essentially performing all possible analyses, in all possible interleavings, and comparing them to pick the one with minimal abstract difference. Each analysis advances an overall k number of steps over both programs, in a different interleaving from all other analysis, where k is the parameterized speculative windows size. We note that interleaving need not be balanced, i.e. one analysis can perform all k steps of P and zero steps over P'. We show that this novel dynamic method is superior to previous methods as it does not rely on the syntactic structure of the program [30] nor does it require any concrete data [36] to help establish a matching of program points.

Fig. 2 shows a (partial) graphical representation of an iteration of the speculative algorithm, running from the start of the programs $(4, 4')$ up to the first printing point $(9, 10')$ (this is in fact the first iteration of the algorithm). The vertical and horizontal axes correspond to instructions of `print_numbers` and `print_numbers'` respectively, i.e. moving up means advancing over a line from `print_numbers` and moving right means advancing over

print_numbers(...)	**print_numbers'(...)**	**Correlating State** $\sigma^{\sharp}_{\bowtie}$ adhering to a program counters in `print_numbers` and `print_numbers'`
ENTRY	ENTRY'	(ENTRY,ENTRY') $\mapsto \{\equiv_{\{STEP,first,last\}}\}$
4 long i; *// loop iteration #0* 5 for (i=0;/* empty */;i++){ 6 long x = first + i*STEP; 7 if (last<x) break; 8 if (i) fputs(separator,stdout);		(8,ENTRY') $\mapsto \{\equiv_{\{STEP,first,last\}}, i = 0, x = first, x \leq last\}$
	4' bool out_of_range' = (last'<x'); 5' if (!out_of_range') { 6' long x'=first'; 7' long i'; *// loop' iteration #0* 8' for (i'=1;/* empty */;i'++){ 9' long x0'=x';	(8,6') $\mapsto \{\equiv_{\{STEP,first,last,\mathbf{x}\}}, i = 0, x = first, x \leq last\}$ (8,9') $\mapsto \{\equiv_{\{STEP,first,last,\mathbf{x}\}}, i = 0, i' = 1, x = first, x \leq last, x'_0 = x'\}$
9 printf(fmt,x);	10' printf(fmt',x');	(9,10') $\mapsto \{\equiv_{\{STEP,first,last,\mathbf{x}\}}, i = 0, i' = 1, x = first, x \leq last, x'_0 = x'\}$
	11' if (out_of_range') break; 12' x' = first' + i' * STEP'; 13' out_of_range' = (last' < x'); 14' if (out_of_range') { 15' bool print_extra_number'=!STREQ(...); 16' if (!print_extra_number) break; 17' fputs(separator', stdout');	(9,**19'**) $\mapsto \{\equiv_{\{STEP,first,last\}}, i = 0, i' = 1, \neg print\_extra\_number', x'_0 = x', x = first, x' = x + STEP, x \leq last\}$ (9,17') $\mapsto \{\equiv_{\{STEP,first,last\}}, i = 0, i' = 1, print\_extra\_number', x'_0 = x', x = first, x' = x + STEP, x \leq last\}$
(i++;) *// loop iteration #1* 6 long x = first + i * STEP; 7 if (last<x) break; 8 if (i) fputs(separator,stdout); 9 printf(fmt,x);		(9,17') $\mapsto \{\equiv_{\{STEP,first,last,\mathbf{x},\mathbf{i}\}}, i = 1, print\_extra\_number', x'_0 = x', x = first + STEP, x \leq last\}$
	(i'++;) *// loop' iteration #1* 9' long x0'=x'; 10' printf(fmt',x'); 11' if (out_of_range') break; 12' x' = first' + i' * STEP'; 13' out_of_range' = (last' < x'); 14' if (out_of_range') { 15' bool print_extra_number'=!STREQ(...); 16' if (!print_extra_number') break; 17' fputs(separator', stdout');	(9,10') $\mapsto \{\equiv_{\{STEP,first,last,\mathbf{x}\}}, i = 1, i' = 2, x = first + STEP, x \leq last\}$ (9,**19'**) $\mapsto \{\equiv_{\{STEP,first,last\}}, i = 1, i' = 2, \neg print\_extra\_number', x \leq last, x = first + STEP, x' = x + STEP\}$ (9,17') $\mapsto \{\equiv_{\{STEP,first,last\}}, i = 1, i' = 2, print\_extra\_number', x \leq last, x = first + STEP, x' = x + STEP\}$
...	...	
// loop iteration #i: ... 9 printf(fmt,x); ...	*// loop' iteration #i:* ... 10' printf(fmt',x'); ...	(9,10') $\mapsto \{\equiv_{\{STEP,first,last,\mathbf{x}\}}, print\_extra\_number', i' = i + 1, x' = x + STEP, x \leq last\}$
// loop exit 10 } 12 fputs (terminator, stdout); 13 } *// function exit*	*// loop' exit* 19' } 20' if (i') fputs(terminator', stdout'); 22 } *// function' exit*	(13,22') $\mapsto \{\equiv_{\{STEP,first,last,\mathbf{x}\}}, i' = i + 1, print\_extra\_number', x \leq last\}$ $\vee$ $\{\equiv_{\{STEP,first,last\}}, i' = i + 1, \neg print\_extra\_number', x' = x + STEP, x \leq last\}$

Table 1: Analysis order (interleaving) and resulting correlating state for functions taken from Fig. 1

`print_numbers'`. The circles in the graph denote abstract states computed by the analysis, which correspond to program locations according to the numbering on the axis. Equivalent and differing states are colored differently. Areas in the graph marked by $\equiv_{\{v\}}$ denote states where v is equivalent ($v = v'$). Lastly, the arrows in the graph represent an advancement over a line of either program, interpreting that line and reaching a new abstract state at the new location. A path from $(4, 4')$ to $(9, 10')$ represents an interleaving of the two programs and the states over that path are the result of analyzing the programs in that interleaving.

Fig. 2 features two interleavings: a less precise one, marked by the dotted line, as it travels through more non-equivalent states and will falsely report difference at line 9 (as it reaches $(5, 9')$, equivalence for `x` has yet to be restored). The second interleaving, marked by a solid line, is a more precise one, and in fact corresponds to the interleaving shown in Tab. 1, up to the first printing points.

During an iteration of the speculative algorithm, it advances over all possible interleavings, up to the speculation window, effectively analyzing the two programs in all orders. It then assigns values to each interleaving according to equivalence criteria. Graphically, this would show as having all the possible paths of length $k = 12$, originating from $(4, 4')$ in Fig. 2. Some paths will reach beyond the boundaries of the graph since an interleaving can perform all steps over just one of the programs. Paths will also be limited by the programs structure (if P reached an exit point, the path will no longer advance over it).

The value assigning metric mainly relies on equivalence, but also contains a weighing element as certain program points (like labels containing back-edges) receive a higher score for equivalence (see Sec. 4). When equivalence is broken, the size of the difference is used, where the interleaving(s) with the minimal diff are scored.

Finally, once an interleaving is selected, the iteration completes and another speculative phase of the algorithm starts. For example, after the iteration of Fig. 2 is completed and the precise interleaving is chosen, the abstract state computed by the analysis will include all of the states along the chosen interleaving, and the next iteration will begin from $(9, 10')$, i.e., where the last iteration left off.

Partial Order Reduction Speculatively exploring for all possible program interleavings up to k steps results in performing 2^k analyses. To allow scalability, we employed a partial order reduction [31, 39, 40]. Given a speculative window value k, instead of exploring all interleavings, we choose a single representative adhering to the *number of steps* taken in each program. For example, if $k = 3$ then instead of exploring all 8 interleavings, only 4 will be analyzed: (i) 0 steps over P and 3 steps over P', (ii) 1 step over P and 2 steps over P', (iii) 2 steps over P and 1 step over P' and (iv) 0 steps over P and 3 steps over P'. This reduces the number of analyses in each speculative step to $k + 1$ resulting in a more scalable analysis. The reduction allows experimenting with larger values for k with minor loss of precision, since the (precision losing) *partition* operation is performed only in between speculative steps (Algorithm 1). The reduction is implemented in Function Speculate. Another means of achieving scalability is advancement over *program blocks* instead of lines. A block is a subsequent, non-branching group of instructions in the program. Thus, advancing over a speculative window of k means advancing over k blocks which translates to approximately $4k$ lines of code, in our experiments.

Separating Equivalent Paths From Differing Paths As one can see from the last row in Tab. 1, we use a partially disjunctive abstract domain that maintains a set of abstract numerical sub-states. This allows us to maintain several sub-states based on the set of equivalences they maintain. The ability to hold separate states for equivalent and differing paths is crucial for precision. Maintaining a fully disjunctive domain with complete path sensitivity does not scale, especially in the face of loops. Therefore we use a partitioning technique to abstract together sub-states according to an *equivalence criteria*. Sub-states that prove equivalence for the same set of variables will be abstracted (joined and widened) together.

The benefits of using equivalence-based partitioning can be seen in Tab. 1 (row 5 onwards) where in the computed abstract state, differing paths are separated from equivalent paths. States where `print_extra_number'` is $false$ are gathered at the loop exit (as they cause a `break` in the loop) alongside equivalent states where `print_extra_number'` is $true$. As described earlier, we use an equivalence criteria to partition and widen sub-states together, thus all the looping differing states are widened together, separately from the equivalent states, producing a precise result as seen in the last row of the table. Details of our partially disjunctive domain are discussed in Section 3.

Note that relying on syntactic methods to find such interleaving can prove to be challenging as `print_numbers'` includes refactoring as well as the actual patch that changes behavior, thus previous approaches [30] will fail in reaching this result.

2.3 Uninterpreted Functions

```
1  static int
2  get_sha1_basic(const char *str, int len,
3                 unsigned char *sha1,
4                 int warn_ambiguous_refs) {
5    ...
6    if (len > 0 && str[len-1] == '}') {
7      for (at = len - 2; at >= 0; at--) {
8        if (str[at] == '@' && str[at+1] == '{') {
9          if (upstream_mark(str + at, len - at) > 0) {
10           reflog_len = (len-1) - (at+2);
11           len = at;
12         }
13         break;
14       }
15     }
16   }
17   ...
18 }
```

Figure 3: `get_sha1_basic` procedure code fragment taken from Git's `sha1_name.c`

Fig. 3 depicts a code fragment, taken from the Git project `sha1_name.c` file. This fragment features a challenging set of operations, including array access by index and function calls. In order to successfully reason over such rich and complex code features, SCORE uses an uninterpreted functions modeling technique [15, 25].

Any operation that is not supported by the underlying APRON domain, including operations on non-integer data, is modeled by the analysis as an uninterpreted function. For example, Fig. 3 code fragment's array indexing and function call operations will be modeled as external functions and the following equivalence deduction rule will be applied during the analysis:

$$\textsc{Array Indexing}\quad \frac{\equiv_{array} \qquad \equiv_{idx}}{\equiv_{array[idx]}} \qquad \textsc{Function Call}\quad \frac{\equiv_{upstream\_mark} \qquad \equiv_{\{v_1,v_2\}}}{\equiv_{upstream\_mark(v_1,v_2)}}$$

We assume initial equivalence for arrays by denoting explicit equality on the array variable (this satisfies the $\equiv_{array}$ predicate in the rule). Once equivalence is broken for a certain index, for instance by assigning non equal values to it, equivalence is broken for the entire array. In case of actual functions, like `upstream_mark`, equivalence is assumed only if the function did not (syntactically) change over versions, or if equivalence was proven for it.

2.4 Path Boundedness

Analyzing Unbounded Paths with k-Bounded Syntactic Divergence We emphasize that k (our notation for speculative window size) bounds only the size of syntactic divergence explored throughout the analysis. It is in no way a bound for the length of paths analyzed by out approach. The analysis depicted in Tab. 1 shows an analysis for *unbounded* paths in looping code. The speculative window only determines how far can the analysis advance over one program before advancing over the other. For instance, in Fig. 2 the precise interleaving advanced four lines of code over the first program and only then turns to analyze five lines of the second program, etc. Furthermore, k only bounds the size of *local* syntactic change.

Consider the two procedures in Fig. 4 exhibiting several syntactic differences, each localized to a few consecutive lines of code. In this example, all lines hidden by ellipses are equal, while differing lines remain. The semantic effect of these lines was also omitted for brevity. The speculative analysis will produce a precise description of difference with k that is bounded only by the largest consecutive syntactic change. The local speculative exploration at each of the changed code fragments will result in a precise description of the change, as depicted by the linear equations in the third column. These are produced throughout the speculative analysis, as further described in Section 4. Therefore even though there are overall many syntactic changes on paths longer than k, the speculative analysis produces a precise result.

3. Speculative Correlating Semantics

In this section, we introduce our speculative correlating semantics which allows bounded representation of program difference through correlation of variables, within some divergence window. In Sec. 3.2, we define a concrete correlating semantics that tracks relationships between variables of both programs. Then, in Sec. 3.3, we show how to abstract this semantics using numerical domains. This sets the scene for the algorithm we present in Sec. 4.

3.1 Preliminaries

We use the following standard definitions for a program: Var, Val, Loc denote the set of program variable identifiers, variable values and program locations respectively.

A concrete program state σ is a pair $l \mapsto values$, mapping the set of program variables to their concrete value $values : Var \to Val$, at a certain program location $l \in Loc$. The set of all possible states of a program P is denoted by Σ_P. We also define $loc : \Sigma_P \to Loc$ which returns the program location of a state.

A program trace $\pi \in \Sigma_P^*$, is a sequence of states $\langle \sigma_0, ..., \sigma_n \rangle$ describing a single execution of the program. The set of all possible traces for a program is denoted by $[\![P]\!]$. We also define $last : \Sigma_P^* \to \Sigma_P$ which returns the last state in a trace.

3.2 Concrete Correlating Semantics

We define a concrete semantics that (i) maintains direct relationships between variables in P and P', using a *correlating concrete state* and (ii) define a notion of dual execution of P and P' within a restricted window of divergence, using *k-diverging correlating concrete traces* (by *k-diverging* we mean that one program may be at most k-steps ahead of the other).

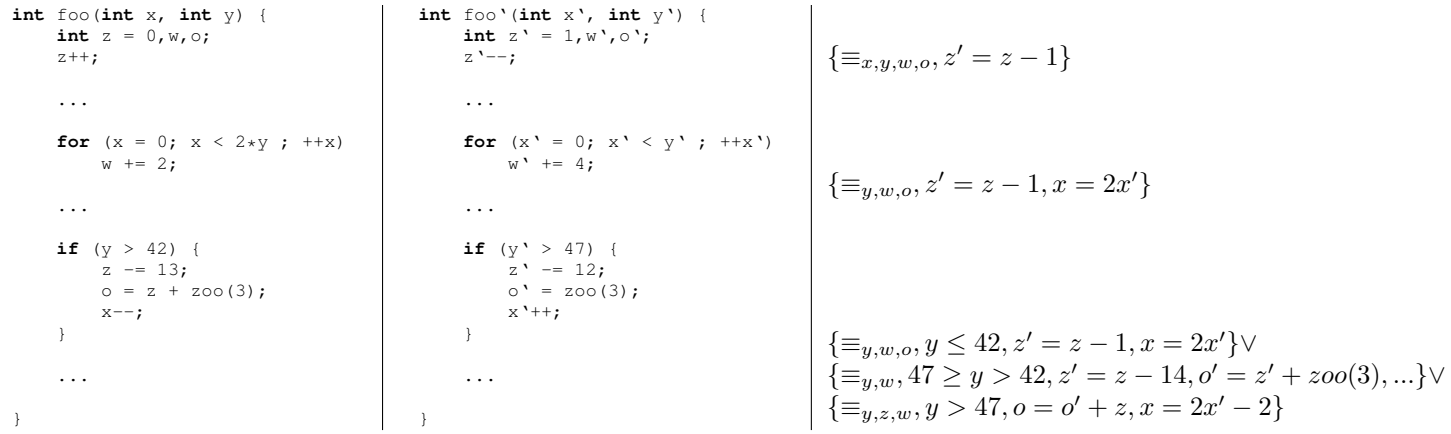

Figure 4: Procedures featuring several (overall changes $> k$) local syntactic changes (each change $< k$), along with correlating abstract state (column 3) adhering to executing the blocks from both programs

Correlating Concrete State A correlating concrete state $\sigma_{\bowtie}$ is a pair $(l, l') \mapsto values_{\bowtie}$, where $values_{\bowtie} : (Var \cup Var') \to Val$ is a joint mapping of variables, from both programs (P, P') to their values at program locations $(l, l') \in Loc \times Loc'$. The set of all correlating states is denoted by $\Sigma_{P \bowtie P'}$.

k-*Diverging Correlating Concrete Trace* A k-diverging correlating concrete trace $\pi^k_{\bowtie} \in \Pi^k_{\bowtie}$, is a sequence of correlating concrete states $\langle \sigma_{\bowtie_0}, ..., \sigma_{\bowtie_n} \rangle$ describing a bounded dual execution of P and P' where at any given point in the trace, the number of P instructions performed cannot exceed those of P' by more than k. We denote the set of k-diverging correlating traces of P and P' as $[\![P, P']\!]^k_{\bowtie}$. Informally, this can be thought of as a trace of a parallel execution $P||P'$ where one thread cannot go more than k steps ahead of the other.

Concrete Semantic Difference We define concrete semantic difference of a correlating state $\Delta_\sigma(varcor, \sigma_{\bowtie}) \subseteq \sigma_{\bowtie}$ to be all the mapped variables in $values_{\bowtie}$ that do not agree on a value with their correlated variable under a matching $varcor : Var \times Var'$. We assume $varcor$ to be either a matching by variable name, or supplied by the user. Next, we define concrete trace semantic difference. We further restrict our semantics and only consider traces that originate from equivalent input states, as we are interested in comparing executions of P and P' which originate from the same input values. Concrete semantic difference of a k-diverging correlating concrete trace $\Delta_\pi(labels, varcor, \pi^k_{\bowtie})$ is defined by the semantic difference of states along the trace of which the pair of labels (l, l') are in $labels$. Δ_π produces a sub-trace of $\pi^k_{\bowtie}$ according to matched labels. $labels$ usually holds the pair of exit labels, along with pairs of labels where output is emitted. We assume P, P' have the same number of output locations and thus can be matched by order, as this was the case in our experiments.

The semantics $[\![P, P']\!]^k_{\bowtie}$ is non-computable. In the next section, we define an abstract semantics that over-approximates it.

3.3 Abstract Correlating Semantics

In this section, we introduce our correlating abstract domain which allows bounded representation of program state while maintaining equivalence between correlated variables. We use a disjunctive domain to allow separate representation of different paths. Therefore, an abstract state in our domain is a set of abstract sub-states. This abstraction is similar to the trace partitioning domain [35].

We use relational abstract domains to hold variable information in the sub-states. In the following we assume an abstract relational domain $(D^\sharp, \sqsubseteq_D)$ equipped with join $\sqcup^D$, meet $\sqcap^D$ and widening ∇^D operations, for abstracting sets of concrete correlating states. We assume the abstract values in $D^\sharp$ are constraints over the P and P' variables, and do not go into further details regarding the particular abstract domain as it is a parameter of the analysis. In our experiments, we use the polyhedra abstract domain [12] and the octagon abstract domain [27].

Correlating Abstract State A correlating abstract program state $\sigma^\sharp_{\bowtie} \in (Loc \times Loc') \to 2^{D^\sharp}$, is a mapping from a pair of program locations to a set of sub-domain factoids, each representing a relational abstraction of the variables.

Abstract Transformers Our domain's abstract transformers, which define how each program statement affects an abstract state, are based on the abstract transformers of the underlying domain. Applying a statement s on an abstract correlating state $\sigma^\sharp_{\bowtie} = (l, l') \mapsto \{d_1 \vee ... \vee d_n\}$ will result in the application of the statement on each of the sub-states using the sub-domains's transformer i.e. $[\![s]\!]^\sharp(\sigma^\sharp_{\bowtie}) = (l, l') \mapsto \{[\![s]\!]^\sharp_{D^\sharp}(d_1) \vee ... \vee [\![s]\!]^\sharp_{D^\sharp}(d_n)\}$

Abstraction Function Finally, we define the abstraction function $\alpha^k : 2^{\Pi^k_{\bowtie}} \to 2^{D^\sharp}$ that abstracts sets of k-diverging

concrete correlating traces. We want our domain to abstract together traces that share the same path, such that each path is represented by a single sub-state in the disjunction. To achieve this, we first define an operation $path : \Sigma^*_{P \bowtie P'} \rightarrow (Loc \times Loc)^*$ which returns the path (sequence of labels) 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. We define the trace abstraction as follows (T is the set of traces to be abstracted):

$$\alpha^k(T) \triangleq \{\ (l,l') \mapsto \bigvee_{\substack{(l,l') = \\ loc(last(\pi^k_{\bowtie}))}} \bigsqcup^{D}_{path(\pi^k_{\bowtie}) \in path(T)} \alpha_{D^\sharp}(\ last(\ \pi^k_{\bowtie}\ )\ )\ \}$$

where $\alpha_{D^\sharp}$ is the abstraction function of the sub-domain.

We break down the definition as follows:

1. α^k groups together all trace prefixes $\pi^k_{\bowtie}$ from T that share the same path, as denoted by $path(\pi^k_{\bowtie}) \in path(T)$.
2. It then abstracts together (using the underlying join operation $\bigsqcup^D_{path(\pi^k_{\bowtie})}$) the concrete states at the end of the grouped traces as denoted by $\alpha_{D^\sharp}(last(\pi^k_{\bowtie}))$.
3. Finally, it disjuncts all paths that arrive at the same end locations (l,l') denoted by $(l,l') \mapsto \bigvee_{(l,l') = loc(last(\pi^k_{\bowtie}))}$.

Every pair of paths in P and P', will be represented by a single factoid of the sub-domain (denoted d_i), and all path pairs that arrive at locations (l,l') will be represented by the abstract state $(l,l') \mapsto \{d_1 \vee ... \vee d_n\}$. *The run of Algorithm 1 (Section 4) with parameters (P, P', k) effectively returns $\alpha^k(T_{(P,P')})$ where $T_{(P,P')}$ is the set of traces of a dual execution of P and P'*. Next, we address the fact that our abstract state may still be potentially unbounded as the number of paths in the program may be exponential and even infinite (due to loops).

Abstract Semantic Difference Given the abstraction, we define the abstract semantic difference $\Delta(labels, varcor, \alpha^k(T)) \subseteq \alpha^k(T)$ as the maplets $(l,l') \mapsto \{d_1 \vee ... \vee d_n\} \in \alpha^k(T)$ such that:

(i) $(l,l') \in labels$

(ii) $\exists d_i, (v,v') \in varcor.d_i \nvdash (v = v')$

This singles out abstract states where some factoids d_i cannot prove explicit equivalence $v = v'$ for matched variables under $varcor$. Note that using $d_i \vdash v \neq v'$ as the criteria is insufficient since as mentioned, anything short of explicit equality under the abstraction is unsound for proving equivalence. We assume the underlying domain D provides a means for performing this checks.

We note that in terms of fixed-point analysis, this means that our join operation $\sqcup$, that abstracts together states of converging paths is simply a disjunction operation, and the only application of the sub-domains $\sqcup^D$ operation will occur during partitioning, as we will next describe.

3.4 Dynamic Partitioning and Widening

As a first means of reducing state size, we define a special operation $\sqcup_{\bowtie}$ 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 class together (using the sub-domain join operation). This join criteria allows separation of equivalence preserving paths from differing ones, thus achieving better precision. Second, to allow a feasible bounded abstraction for programs with infinite number of paths, we define a widening operator $\nabla_{\bowtie}$ which utilizes the sub-domain's widening operator but chooses which sub-states are to be widened, according to the same equivalence criteria.

State Size Reduction Using Equivalence-Based Partitioning As mentioned, we must allow for reduction of state $\sigma^\sharp_{\bowtie} = (l,l') \mapsto \bigvee d_i$ with acceptable loss of precision. This reduction is achieved by joining the abstract sub-states in $\bigvee d_i$ (using the standard join of the sub-domain). However this can only be accomplished after first deciding which of the sub-states should be joined and then choosing the program locations for the partitioning to occur. Our partitioning strategy abstracts together sub-states according to *the set of variables which they preserve equivalence for*. This bounds the state size at $2^{|varcor|}$, where $varcor$ is the set of correlating variables we wish to track.

In order for our analysis to handle loops we require a means for abstracting an infinite number of paths in a bounded way. As our analysis iterates over a loop, sub-states may be added or transformed continuously, never converging. We therefore need to define a widening operator for our new domain. This problem has been addressed in other settings [3], and our approach can be viewed as a specialized form of widening that is tailored for tracking equivalences. We found that our partitioning strategy works well for widening, as it allows separation of equivalent looping paths from differing paths. Given two subsequent abstract states $\sigma^\sharp_{\bowtie^0} = (l,l') \mapsto \bigvee d^0_i$ and $\sigma^\sharp_{\bowtie^1} = (l,l') \mapsto \bigvee d^1_j$ corresponding to two subsequent iteration of a loop, the result of applying widening will be as follows:

$$\nabla(\sigma^\sharp_{\bowtie^0}, \sigma^\sharp_{\bowtie^1}) \triangleq (l,l') \mapsto \nabla^D_{\equiv(d^0_i) = \equiv(d^1_j)} (d^0_i, d^1_j)$$

Where $\equiv(d)$ returns the set of variables (from $varcor$) that are equivalent in d.

4. Speculative Search Algorithm

In this section, we present our speculative search algorithm. Given a speculation window size k, the algorithm explores all possible interleavings of the two programs up to k steps, and then employs an equivalence-based scoring heuristic to greedily select the abstract state with the minimal abstract difference. This abstract state is used as a basis for the next iteration of the speculative algorithm, which proceeds until reaching a fixed point. This novel dynamic approach allows

to dynamically explore different interleavings on each step of the analysis, instead of deciding on a fixed interleaving a priori.

Algorithm 1: Compute abstract difference

Input: Two programs P, P', Speculation window size k, Partition interval p, A matching $varcor : Var \leftrightarrow Var'$ of the correlating variables in both programs.

Output: A mapping $statespace : Loc \times Loc' \mapsto \sigma^{\sharp}_{\bowtie}$ from pairs of prog. locations to correlating abstract state.

```
1  num_speculations ← 0;
2  workset ← {(entry, entry')};
3  statespace ← {(entry, entry') ↦ {v ≡ v' | (v, v') ∈ (varcor ⋂ (In × In'))}};
4  while workset ≠ ∅ do
5      solutions ← Speculate(P, P', workset, statespace, k);
6      num_speculations += 1;
7      (workset, statespace) ← FindMinDiffSolution(P, P', varcor, solutions);
8      if num_speculations mod p = 0 then
9          statespace ← Partition(statespace);
10     statespace ← Widening(statespace);
11 return statespace;
```

4.1 Iterative Speculative Correlation

Algorithm 1 produces an abstraction of program difference in a form of a mapping from pairs of program locations of P, P' to correlating abstract states.

The input to the algorithm is: (i) two programs P and P', (ii) a speculation window k which determines how many speculative steps the algorithm may take, (iii) a mapping $varcor$, matching variable names between variables of P and P', describing which variables should be correlated. In our experiments, we show that matching variables that have the same name and appear in both programs is sufficient for producing precise results. However, in general, this mapping can be provided by the user or by other methods such as using data traces [36].

The algorithm operation is similar to standard abstract interpretation fixed-point algorithms [11]. It starts by adding the entry point of the two programs, denoted by $entry$ and $entry'$, to the empty $workset$, and initializes the $statespace$ by mapping these locations to correlating state that assume equivalence over input variables, denoted by In and In'. The algorithm iteration then starts, as it interprets program lines from the $workset$, creating and updating the $statespace$ with abstract states mapped to pairs of program locations, while adding lines where state has changed back to the $workset$, until a fixed-point is reached. Note

Function Speculate($P, P', workset, statespace, k$)

Input: As specified in Algorithm 1.

Output: A set $solutions : \{(workset_1, statespace_1), ..., (workset_n, statespace_n)\}$ specifying the work-set and state-space generated from advancing k steps over P, P' in all possible interleavings, under a partial order reduction

```
1  (workset_0, statespace_0) ← (workset, statespace);
2  for ((i, j) ← (0, k); i ≤ k; i++, j- -) do
3      for (t = 0; t < i; t++) do
4          (workset, statespace) ← Step(P, workset, statespace);
5      for (t = 0; t < j; t++) do
6          (workset, statespace) ← Step(P', workset, statespace);
7      solutions ← solutions ∪ {(workset, statespace)};
8      (workset, statespace) ← (workset_0, statespace_0);
9  return solutions;
```

Function Step($P, workset, statespace$)

Input: As specified in Algorithm 1.

Output: A solution $(workset, statespace)$ adhering to advancing a step over P

```
1  workset_res ← ∅;
2  foreach (loc, loc') ∈ workset do
3      if loc ∈ P then
4          // advancing over first graph
5          foreach succ ∈ successors(loc) do
6              σ ← statespace(loc, loc');
7              block ← blocks(P, loc, succ);
8              statespace(succ, loc') ← statespace(succ, loc') ∨ [[block]]^♯(σ);
9              workset_res ← workset_res ∪ {(succ, loc')};
10     else if loc' ∈ P then
11         // advancing over second graph
12         // the symmetrical case
13         ...
14 return (workset_res, statespace);
```

that Algorithm 1 differs from standard algorithms as it: (i) Operates over two programs instead of one. (ii) Performs k steps of the analysis instead of just one, i.e. it interprets $(0, k), (1, k-1), ...(k, 0)$ lines of the programs through `Speculate` resulting in $k+1$ $solutions$ (pairs of $workset$ and $statespace$). (iii) Dynamically chooses the

Function FindMinDiffSolution($P, P', vc, solutions$)

Input: As specified in Algorithm 1.
Output: A pair $(workset, statespace)$ containing the work-set and state-space of the most precise solution

```
1  factor ← |Loc| · |Loc'|;
2  foreach (l, l') ∈ P × P' do
3      equivalence(l, l') ← false;
4      min_delta ← ∞;
5      min_delta_solutions ← ∅;
6      foreach (workset, statespace) ∈ solutions do
7          score ← 0;
8          if ≡_{varcor} (statespace(l, l')) then
9              equivalence(l, l') ← true;
10             if has_backedge(l) ∨ has_backedge(l') then
11                 score(workset, statespace) += factor^2;
12             else
13                 score(workset, statespace) += factor;
14     if ¬equivalence(l, l') then
15         foreach (workset, statespace) ∈ solutions do
16             if |Δ(statespace(l, l'))| = min_delta then
17                 min_delta_solutions ← min_delta_solutions ⋃ {(workset, statespace)};
18             else if |Δ(statespace(l, l'))| < min_delta then
19                 min_delta ← |Δ(statespace(l, l'))|;
20                 min_delta_solutions ← {(workset, statespace)};
21     foreach (workset, statespace) ∈ min_delta_solutions do
22         score(workset, statespace) += 1;
23 max ← Max(Range(score));
24 return (workset, statespace) ∈ score^{-1}(max);
```

solution with the minimal abstract difference by applying `FindMinDiffSolution` which surveys the *statespace* of each solution and assigns a score to it according to an equivalence based weighed metric. (iv) Partitions *statspace* according to partition interval p (v) Applies widening to states mapped to blocks with back-edges.

Greedy Dynamic Selection of Interleaving We again emphasize an important dynamic feature of our speculative run: picking an interleaving over a range of program locations in one stage, does not determine the interleaving over these locations for the rest of the analyses. `Speculate` will always return the analysis of $k+1$ interleavings, and the one with minimal difference will be picked by `FindMinDiffSolution`. Therefore, the decision could vary in different stages of the run, based on the size of difference in the candidate solutions. The importance of this is exemplified in Tab. 1, where the order in which we analyze lines 4 through 9 in `print_number` and lines 4' to 11' in `print_number'` *changes between iterations of the algorithm* and the rows of the table. Next, we will describe the `Speculate` function and explain how an interleaving is explored and how all interleavings are returned in the forms of *solutions*.

State Size Reduction With Partitioning The number of speculative steps taken is kept in the $num\_speculations$ variable which is used to determine when to partition. Partitioning is crucial for performance as it reduces the size of the disjunction. We found that partitioning at parameterized constant intervals, is sufficient in producing scalable precise results. We experimented with different values for p to refine the result and achieve better precision. More complicated strategies for picking partition location include using the syntactic structures of the programs to find locations where these "converge" [30] however we did not find the need to apply other methods as precision was satisfactory.

Widening We maintain the number of visits to each pair of program locations and perform widening once a predetermined threshold has been reached. Widening is performed only if one of the lines in the pair (at least) has a back-edge reaching it, this is sufficient for reaching a fixed point as further discussed in Section 3.

4.2 Speculative Advancement Over All Interleavings

The `Speculate` function produces $k+1$ solutions, representing the result of analyzing P, P' in by advancing k steps over both programs, in different distributions of steps. The functions receives the two programs P, P', a speculative window size k and the state of the analysis so far captured by $(workset, statespace)$. It then proceeds to extend the solution by performing k steps over both programs, starting with 0 steps over the first and k over the second, continuing to perform $(1, k-1)$ steps, and so on as can be seen by the procedure's main loop.

Advancing a step over either program $P(')$ is performed by taking all location pairs in the $workset$ and advancing over $P(')$ locations as captured by the `Step` procedure. `Step` employs the abstract transformer described in Section 3.3 over the locations successors. For instance, if $(l, l') \in workset$ and l_{succ} is a successor of l (in P), then the effects of advancing from l to l_{succ} will be interpreted by: (i) Retrieving the state adhering to (l, l') from $statspace$. (ii) Retrieving the basic block (i.e. the edge) between l and

l_{succ} by applying $blocks(P, l, l_{succ})$. (iii) Applying the abstract transformer of the block on the retrieved state and joining with the state at $statspace(l_{succ}, l')$ (if such exists, otherwise $statspace(l_{succ}, l')$ will only hold the result). (iv) Adding (l_{succ}, l') to the resulting $workset_{res}$. The case of advancing from (l, l') to l'_{succ} in P' is symmetrical.

4.3 Comparing Abstractions to Find Minimal Difference

The final stage of the speculative iteration is the selection of a single solution out of the set of $solutions$, as performed by the function `FindMinDiffSolution`. Our goal is to find a solution, whose states differ minimally, since it necessarily means a more precise result (due to the soundness of the analysis). `FindMinDiffSolution` defines a scoring heuristic which ranks abstractions based on equivalence and minimal difference. It surveys all the possible pairs $(l, l') \in Loc \times Loc'$ and compares the abstract state mapped to that pair in $statespace(l, l')$ against all other states, awarding points according to the size of the difference.

For each pair of program locations (l, l'), the algorithm tries to find a solution which "proves equivalence" i.e. for all matched variables in $(v, v') \in varcor$, $statespace(l, l') \vdash v = v'$. This means that in all sub-states in the disjunctive abstract state $statespace(l, l')$, v is equal to v'. In the algorithm this is denoted by $\equiv_{varcors} (statespace(l, l'))$. In case such a solution is found, (l, l') is flagged using the $equivalent$ flag and other solutions will be given points there only if they also prove equivalence. This reflects the fact that the existence of equivalence for a pair of lines in one interleaving invalidates differences for that pair in other interleavings. Also, this is a design choice aimed at improving performance, as the computation and comparison of difference can be costly and will be omitted once equivalence is found.

In case no such solution is found for (l, l'), the function continues on to compare the size of difference. We experimented with different ways of comparing difference and arrived at using the number and (textual) size of the offending states in the disjunction as criteria, denoted by $|\Delta(statespace(l, l'))|$. We found this heuristic to be efficient yet accurate enough to produce precise results in reasonable analysis time. We also experimented with a formal method for comparing difference, via an algorithm for manipulating abstract states $\sigma^{\sharp}_{\bowtie}$ to extract disjunctive states that describe difference and can be compared over a lattice using the $\sqsubseteq$ operation. However, since this algorithm is computationally expensive and does not improve precision dramatically, we refrained from using it.

The existence of equivalence awards the solution $score$ a $factor = |Loc| \cdot |Loc'|$ amount of points, while the solution with minimal difference (in case equivalence was not found) is given a single point. This reflects that a solution with one equivalent state is preferred over a solution where all states have minimal difference. Solutions that did not arrive at (l', l) are not considered and do not get points for it.

We added a weighing component to our metric, where locations with entering back-edges (denoted by the $has\_backedge(l)$ predicate), receive $factor^2$ points, in case they prove equivalence. This change in order of magnitude was added to help direct the search towards lines with entering back-edges, that prove equivalence. Equivalence at these locations is important as *widening* is performed there. Since widening over-approximates specific numeric data, widening correlating states that do not already prove equivalence, will likely result in a non-restorable loss of equivalence. Thus, we will prefer solutions where abstract states at back-edge locations prove equivalence. This lowers the risk of losing equivalence once widening occurs.

5. Evaluation

In this section, we evaluate SCORE using a number of real world programs, and compare it to the state of the art in equivalence checking and semantic differencing. Our goal is to asses how SCORE measures against challenges presented by each method, and whether it can be applied to produce useful results. As benchmarks, we used several programs drawn from the Git source code [1], GNU Coreutils, as well as a few patches from the Linux kernel and the Mozilla Firefox web browser. We also collected benchmarks used as motivating examples in state of the art work, applied SCORE to them and present the comparison here.

5.1 Prototype Implementation

We implemented speculative correlation, as described in Algorithm 1, based on the LLVM and CLANG compiler infrastructure. We chose to analyze C code directly as it is more structured, has type information and keeps a low number of variables, as opposed to intermediate representation. Our analysis is intra-procedural and handles function calls, as well as operations not modeled by the numerical domain, as uninterpreted functions (Section 2.3). We used the APRON abstract numerical domain library and conducted our experiments using the Polyhedra domain [12]. We ran our experiments on an Intel(R) Core(TM) i7-2620M CPU @ 2.70 Ghz machine with 4GB installed RAM.

5.2 Benchmark Results

Tab. 2 summarizes the results of our analysis. The columns indicate the function name, length in lines of code, the number of lines added and/or removed by the patch, total number of loops, and the run time in minutes along with the *minimal* speculative window k and partition interval p values that produced minimal difference. The results are separated (by a horizontal line) according to project the function was taken from, as follows: GNU Coreutils, Linux kernel, Firefox, Git and related work representatives.

Results produced by SCORE are abstractions of the data dependencies and equivalence of variables in the program.

Table 2: Experimental Results

Function	#LOC	#Patch	#Loops	Time(k,p)
print_numbers	23	7-,13+	1	0:11 ($k=2,p=1$)
cache_fstatat	17	2-,4+	0	0:03 ($k=1,p=2$)
set_owner	51	2-,4+	0	0:02 ($k=2,p=1$)
fmt	42	5-,5+	1	0:22 ($k=2,p=2$)
md5sum	40	0-,3+	3	13:31 ($k=2,p=3$)
char_to_clump	111	2-,12+	3	19:09 ($k=2,p=4$)
savewd	86	0-,1+	0	0:46 ($k=1,p=2$)
addr	77	1-,2+	0	0:17 ($k=1,p=1$)
SetTextInternal	47	0-,3+	1	11:28 ($k=3,p=3$)
get_sha1_basic v1	145	3-,10+	2	118:01 ($k=4,p=2$)
get_sha1_basic v2	149	2-,20+	2	TO
get_path_prefix	22	2-,3+	1	29:12 ($k=3,p=4$)
boot_attr v1	77	7-,4+	0	8:08 ($k=4,p=2$)
boot_attr v2	74	5-,7+	0	6:04 ($k=4,p=2$)
read_attr	32	1-,4+	1	5:42 ($k=2,p=4$)
ll_binary_merge	37	8-,24+	1	0:53 ($k=1,p=1$)
write_zip_entry	340	1-,4+	3	7:32 ($k=2,p=1$)
DDEC	10	3-,3+	1	0:13 ($k=1,p=4$)
DSE	7	2-,3+	1	0:09 ($k=1,p=1$)
RegVer	10	4-,4+	1	0:07 ($k=1,p=1$
SymDiff	32	5-,4+	0	0:04 ($k=1,p=4$)

```
static int write_zip_entry(int zip_dir,
                           int zip_dir_size,
                           int zip_dir_offset,
                           int pathlen, ...) {
  ...
  /* make sure we have enough
     free space in the dictionary */
  int direntsize = ZIP_DIR_HEADER_SIZE +
                   pathlen + ZIP_EXTRA_MTIME_SIZE;
+ bool flag = false;
  while (zip_dir_size < zip_dir_offset + direntsize) {
    zip_dir_size += ZIP_DIRECTORY_MIN_SIZE;
-   zip_dir = xrealloc(zip_dir,zip_dir_size);
+   flag = true;
  }

+ if (flag)
+   zip_dir = xrealloc(zip_dir,zip_dir_size);
  ...
}
```

Figure 5: Patch for git `archive-zip.c`'s `write_zip_entry` procedure

SCORE supplies the resulting state space, as a mapping from all pairs of program locations to correlating state for those pairs (as described if Section 4). SCORE further reports whether differences were found in observable program locations (output and exit points) and prints out such states holding difference in textual form.

We experimented with a speculative window k of up to 4. We chose increasing partition interval sizes p as well, up to a maximum of 5. We capped run time at two hours for each SCORE run and reported the minimal p and k which resulted in the smallest difference (Δ) at output and exit locations. The size of speculation window k could usually be predicted by the size of the biggest addition or deletion of subsequent lines to the code. In some cases, like the `get_sha1_basic v2` benchmark, vast semantic differences required a large speculation window, and reported differences for smaller k sizes were unusable. The benchmark was therefore reported to time out with no acceptable precision output given.

Another interesting case is the 340-line long `write_zip_entry` benchmark depicted in Fig. 5. The patch here involves changing a few lines of code where the memory allocation to `zip_dir` is hoisted out of the loop (line marked with **-** in Fig. 5) and instead performed only after `zip_dir_size` has been determined (lines marked with **+** in Fig. 5). The appropriate lookahead window here is $k = 2$, to overcome additions by the patch. Running SCORE while partitioning at $p = 1$ (after each speculation) yields equivalence. Interestingly enough, SCORE was able to maintain equivalence for `zip_dir`, although the value for it was *widened* to overcome the loop. This is due to partitioning and widening both operating by equivalence: at the loops end, the analysis accumulates states holding the $zip\_dir = xrealloc(...), flag = true$ predicates, alongside states withholding these predicates but instead hold equivalence for `zip_dir` (states that did not enter the loop). These two categories of states were partitioned and widened separately due to equivalence criteria and equivalence was restored for `zip_dir` once the patched lines were reached. This demonstrates one of SCORE 's strong points: the ability to restore equivalence from equivalence classes.

Our experiments included other numerical domains such as Octagon [27], however we did not include the results as they exhibit poor precision and performance. Moving from Octagon to Polyhedra domain, a notable increase in precision was shown as the Polyhedra domain is able to capture more complex data relationships. Surprisingly, runs using the Octagon domain presented poor performance (run time) compared to the more expensive Polyhedra domain, with less precision. This is due to the domain's being less successful in capturing equivalences as it is built upon simpler linear inequalities. This means that more constraints were needed to represent variable equality, resulting in bigger states and a slower analysis.

SCORE produced results with high precision where *only* variables that indeed differ between versions were reported, and the description of the difference was useful for producing actual values for the differing variables. In cases where equivalence holds, no difference was reported. The `addr` and `SetTextInternal` benchmarks were taken from the net/sunrpc/addr.c module in the Linux kernel SUNRPC implementation v2.6.32-rc6 and Firefox 3.6 security advisory CVE-2010-1196 (adapted to C from C++) respectively. The results produced by SCORE can be directly used towards exploiting known security flaws mentioned in advisories from which these patches originate. Fig. 6 shows the patch made in the CVE-2010-1196 advisory, fixing a heap buffer overflow in the Firefox browser. Running SCORE on the `SetTextInternal` function yields the following output:

$$\Delta[EXIT, EXIT] =$$

```
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;
  memcpy (to + aOffset , aBuffer ,
          aLength * sizeof ( PRUnichar ));
    ...
}
```

Figure 6: Patched version of vulnerable `SetTextInternal` Firefox function

$$\{Ret = 0, Ret' = 1, newLength > 536870912\} \vee$$

$$\{Ret = 0, Ret' = 1, 0 > newLength > -3758096384\}$$

The `Ret` variable, was added by SCORE to instrument function return. Also, special handling was added for interpreting integer casting. The produced result is useful to the programmer, to ensure that vulnerable ranges have been covered and that the function ends for these ranges. The results could also be used by an attacker to deduce the vulnerability fixed by the patch.

5.3 State of The Art Comparison

Syntactic Correlation Based Techniques The approach described in [30] aims at proving C-code function equivalence and producing a textual representation of difference for equivalence checking, patch debugging etc. We evaluated SCORE on several benchmarks taken from [30]. Results show that SCORE yields results at similar or higher precision. One downside of this previous work, is illustrated by our motivating example from Fig. 1. The `seq.c` benchmark introduces considerable amounts of textual change which defeats the syntactic reliant method suggested in [30]. An integral part of this method involves creating a unification program, containing both versions, to be used by the analysis. The precision of the analysis relies on this unified program and its ability to bring together (syntactically) instructions that are "equivalent" in both programs. However, for the `seq.c` benchmark, this correlating program will be poorly formed, unable to syntactically match the versions, which will result in an imprecise result. Another shortcoming of this method is the addition of guards and the need to syntactically transform C code to guarded command form, a process proved to be challenging and erroneous. These guard variables are also incorporated into the domain, resulting in a more expensive analysis.

Data Driven Techniques Fig. 7 depicts the worked example from [36], describing a semantic preserving compiler optimization. This work aims to prove equivalence for looping assembly code segments for translation validation purposes. This dynamic method uses data traces in order to establish a simulation relation between code segments and then

```
int f(int x, int n){
  int i, z = 0;
  for (i=0; i!=n; ++i){
    x += z*5;
    z += 1;
    if (i >= 5)
      z += 3;
  }
  return x;
}
```

```
int f`(int x, int n){
  int i, z = 0;
  for (i=0; i!=n; ++i){
    x += z;
    z += 5;
    if (i >= 5)
      z += 15;
  }
  return x;
}
```

Figure 7: Original and patched version of [36] worked example

attempts to prove this relation by using off-the-shelf SAT solvers.

We ran SCORE on Fig. 7 to see whether equivalence can be established. SCORE reported the following result as the abstract state at the exit point of both programs, within 13.3 seconds ($k = 1, p = 8$) : $\{\equiv_{\{x,i\}}, i = n, z = 5z`, i \leq z\}$. SCORE was able to prove equivalence for `x` and capture the exact relationship of `z` and it's patched counterpart `z'`: $z = 5z`$. The strongpoint of [36] is the ability to produce this result on machine code, where syntactic differences are bigger.

Recursion Rule Based Techniques [15] applies a recursion rule to verify equivalence of recursive functions. This work uses recursive calls within candidate functions and assumes their equivalence as the basis of the recursive verification rule. It then tries to inductively prove equivalence by showing that all paths to the recursive call in both versions are equivalent, using bounded model checkers [10]. Although this technique is able to deal with recursion, it requires the recursive call to be nested under the exact same conditions in both programs, disallowing the use of the recursive rule in many cases. Our motivating example requires complex manual rewrite to adhere to such form, as well as many other benchmarks. As mentioned, SCORE assumes modular equivalence of function calls it encounters, thus, we were able to adapt it to implement the recursion rule from [15], by simply assuming equivalence on recursive calls and proving equivalence by showing equivalence over the recursive call parameters. The benchmark appears as `RegVer` in Tab. 2. [25] applies a similar path-proving technique, however they do not use the recursive rules as they do not handle loops. The strong point of this method, is that it handles input programs written in the Boogie verification language [4]. Boogie has translations from `C`, `C#` and x86 assembly, making [25] language agnostic. The benchmark appears under `SymDiff` in Tab. 2.

Directed Symbolic Execution Based Techniques Work such as [32] and [34] uses symbolic execution testing tools, such as KLEE [7] and DART [14], to find equivalence bugs. To prove equivalence between P and P', these techniques sequentially compose the programs (while assuring the same input) and then add an assertion checking if return values of both versions are equal. The new program is then fed

to an automatic test generation tool which in turn tries to explore all paths in the program, effectively exploring all path compositions of P and P' and checks if any of these paths break the assertion. This technique heavily relies on the effectiveness of the test generation tool and benefits from its features, as such the ability to reason over pointer and heap data. In addition to this method being dynamic, it is fundamentally unable to reason over looping programs, and loops are mitigated by a simple unrolling to some constant number of iterations which is of course un-sound. We drew one benchmark from [32], which appears as `DSE`, containing a loop. [34] benchmarks were omitted as they focused on pointer and heap data.

5.4 Quality of Semantic Diff

Next we examine the quality and usefulness of SCORE output. We compare the semantic diff produced by SCORE to several other methods, focusing on the most prominent tool for comparing programs– syntactic `diff`.

5.4.1 Equivalence of Refactored Looping Code

```
1  static void
2  print_numbers (long first, long last, ...)
3  {
4    long i;
5    for (i = 0; /* empty */; i++) {
6      long x = first + i * STEP;
7      if (last < x) break;
8      if (i) fputs (separator, stdout);
9      printf (fmt, x);
10   }
11   if (i)
12     fputs (terminator, stdout);
13 }
```

Coreutils seq.c original

```
1′ static void
2′ print_numbers`(long first, long last, ...)
3′ {
4′   bool out_of_range = (last < first);
5′   if (!out_of_range) {
6′     long x = first;
7′     long i;
8′     for (i = 1; /* empty */ ; i++) {
9′       long x0 = x;
10′      printf (fmt, x);
11′      if (out_of_range) break;
12′      x = first + i * STEP;
13′      out_of_range = (last < x);
14′      fputs (separator, stdout);
15′    }
16′    fputs (terminator, stdout);
17′  }
18′ }
```

Coreutils seq.c refactored

Figure 8: Output-equivalent versions of Coreutils `seq.c`'s `print_numbers` procedure

Fig. 8 depicts a modified version our motivation example (Fig. 1) where only the refactoring was applied to change code structure, while the semantic changing part was removed. The two versions of the `print_numbers` procedure are output equivalent as the printed variable `x` is equivalent at the print location. Running `diff` on the two versions would produce the entire text of the programs. This provides no insight into change semantics. SCORE output consists of: (i) an abstract description of versions state, in the form of linear equations, at all joint program locations and specifically at the output location (ii) for each variable in said state, the equivalence status is reported (all equivalent variables is explicitly reported as program equivalence). Thus, the output of SCORE for Fig. 8 is shown in Tab. 3. The output correctly describes the change in the programs, emphasizing the fact that `x` is equivalent at the output location. The state further holds the loop invariant for `x` value which is useful for program understanding. Previous techniques (described in Section 5.3) are unable to provide a useful description of the change since they are either inherently incapable of handling loops [32, 34]; rely on specific loop structure [15, 25]; rely on syntactic similarity [30] or work on smaller lower-level code [36].

Table 3: SCORE Output for Fig. 8

At Location (9,10'):
Equivalent: `first,last,x`
Non-Equivalent: `i,x0,out_of_range`
State = $\{first' - x0' + 2i - 2 = 0, first' + 2i - x = 0,$ $first' - x' + 2i = 0, -first' - 2i + last \geq 0, i - 1 \geq 0\}$

```
static int input_position = 0;
static int
char_to_clump (int chars_per_c,
               int width,
               bool untabify_input, ...) {
    int chars = 0, i;
    char * s;
    ...
    if (untabify_input) {
        i = width;
        while (i > 0) {
           *s++ = ' ';
           --i;
        }
        chars = 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;

    return chars;
}
```

Figure 9: Original and patched (+) version of Coreutils `pr.c`'s `char_to_clump` procedure

5.4.2 Previous Path Knowledge

In many cases, a locally applied patch could be affected by previously executed code as shown in Fig. 9 depicting the `pr` benchmark code taken from Coreutils. The lines added by the patch are marked by a boldface plus sign (+). Most of the benchmark code was omitted for brevity, except for the patch

Table 4: SCORE Output for Fig. 9

At Location (EXIT,EXIT'):
Equivalent: `width,chars_per_c,untabify_input,...`
Non-Equivalent: `input_position,chars`
State = $\{width = width' = -1, chars = 1, chars' = 0,$ $input\_position = -1, input\_position' = 0, ...\}$ $\vee$ $\{width = width' = -1, chars = chars' = 1,$ $input\_position \leq 0, input\_position' = 0, ...\}$

and a preceding loop involving values affected by the patch. Here, the semantics of the patch are related to previously executed code over many execution paths. Referring locally to the patch itself, as the result of a `diff` would, provides no information towards the change in variable values affected by the patch. SCORE instead provides a useful description of the change in variable values due to patching (Tab. 4). The output features variable values adhering to the two paths affected by the change. This result is more conducive towards helping the programmer understand and verify their patch.

6. Related Work

Equivalence checking and semantic differencing has received much attention lately where more and more works have been applied towards advancing the state of the art in some aspect. Similar to many verification problems, this problem has been presented nearly sixty years ago, with few mentions in the years following [17–20] as it is considered undecidable. Recently, work identifying program differencing as having vast security implications [6, 37] has surfaced, bringing the problem back to mainstream. Also, the considerable advancements made in the field of SAT solvers and it application to various verification problems, lead to a plethora of work leveraging off-the-shelf SAT solvers towards program equivalence [15, 25, 32, 34, 36]. Several works on the problem of equivalence of combinatorial circuits [9, 23, 28] made important contributions in establishing the problem of equivalence as feasible, producing practical solutions for hardware verification.

The idea of a *correlating semantics* and reasoning about correlated execution has been investigated in many contexts before (e.g., [2, 38]). In fact, our approach can be seen as an invariant generation technique for relational logics such as Relational Hoare Logic (RHL) [5].

An early work presenting a usable tool for semantic diff is [21]. They present a tool for computing data dependencies between input and output variables and comparing these dependencies along versions of a program for discovering difference. This method may falsely report difference as semantic difference may occur even if data dependencies have not changed. Furthermore, data dependencies offer little insight as to the meaning of difference i.e. input and output values. Nevertheless, this was an important first step in employing program analysis as a means for semantic differencing.

Aiken et. al. [36] present a data driven approach for translation validation. This approach requires the running of the program and data traces to help establish a simulation relation to later be proved by SMT solvers. Also, as far as we could discern, they are limited to small segments of assembly code, unable to handle bigger, higher language code. We compare to this work in our evaluation.

David et. al. [13] address equivalence of short non-branching sections of binary code (i.e. 'tracelets') in the context of binary code search. They define a notion of equivalence for binary code sections and apply rewriting and constraint solving techniques for finding equivalence as a basis for whole function similarity. These techniques can be further extended using speculative exploration to overcome syntactic obstacles and to apply to looping code. This can help find similarly even for highly re-factored, optimized or patched code.

Other work regarding translation validation [29, 33, 41], require establishing a simulation relation between the basic blocks of the translated code is found. This method is limited in the context of semantic differencing as, for instance, a simulation relation for examples such as Fig. 1 cannot be automatically established (it needs to be crafted manually as this is not one of the classic transformations).

Symbolic execution based methods [32, 34] offer practical equivalence verification techniques for loop and recursion free programs with small state space. These works complement each other in regards to reporting difference as one [32] presents an over approximating description of difference and the other [34] presents an under approximating description including concrete inputs for test cases demonstrating difference in behavior. These cannot be applied towards our setting as they handle loop free, finite state programs only.

[15] presents the notion of partial equivalence which allows checking for equivalence under specific conditions, using a recursive rule. They employ a technique based on theorem provers for proving an equivalence formula which embeds program logic (in SSA form) alongside the requirement for input and output equivalence and user provided constraints. As mentioned in Section 5, this work requires the rewrite of programs and loops recursive constructs, and applies only when the path conditions leading to the recursive call are the same in both candidate version, making it limited and inapplicable to our setting.

7. Conclusions

We presented a new abstract interpretation approach for program equivalence and differencing. Our approach is purely static, can prove equivalence and characterize differences for program with loops, and does not rely heavily on syntactic similarity to establish program correlation. The main idea

is to use a *speculative correlation* algorithm that guides the interleaving of the two programs based on the abstract difference between them. This algorithm is instantiated over a correlating numerical abstract domain. Our correlating domain uses a powerful numerical abstract domain to capture abstract program states (and differences) as linear inequalities between variables. We show that this approach is feasible and can be applied successfully to challenging real world patches.

Acknowledgement

This work was (partially) supported by the Israel Ministry of Science and Technology grant no. 3-9779. We would like to thank Omer Katz and the anonymous referees for their feedback and suggestions for this work.

References

[1] Github has surpassed sourceforge and google code in popularity. `http://readwrite.com/2011/06/02/github-has-passed-sourceforge`.

[2] D. Amit, N. Rinetzky, T. W. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *CAV*, pages 477–490, 2007.

[3] R. Bagnara, P. M. Hill, and E. Zaffanella. Widening operators for powerset domains. *STTT*, 8(4-5):449–466, 2006.

[4] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.

[5] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, pages 14–25, 2004.

[6] D. Brumley, P. Poosankam, D. X. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *IEEE Symposium on Security and Privacy*, pages 143–157, 2008.

[7] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[8] S. Chaki, A. Gurfinkel, and O. Strichman. Regression verification for multi-threaded programs. In *VMCAI*, pages 119–135, 2012.

[9] E. M. Clarke and D. Kroening. Hardware verification using ansi-c programs as a reference. In *ASP-DAC*, pages 308–311, 2003.

[10] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ansi-c programs using sat. *Formal Methods in System Design*, 25(2-3):105–127, 2004.

[11] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[12] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.

[13] Y. David and E. Yahav. Tracelet-based code search in executables. In *PLDI*, page 37, 2014.

[14] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[15] B. Godlin and O. Strichman. Regression verification. In *DAC*, pages 466–471, 2009.

[16] C. Hawblitzel, S. K. Lahiri, K. Pawar, H. Hashmi, S. Gokbulut, L. Fernando, D. Detlefs, and S. Wadsworth. Will you still compile me tomorrow? static cross-version compiler validation. In *ESEC/FSE 2013*, 2013.

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

[18] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *PLDI*, pages 234–245, 1990.

[19] S. Horwitz, J. Prins, and T. W. Reps. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345–387, 1989.

[20] J. W. Hunt and M. D. McIlroy. An algorithm for differential file comparison. Technical report, Bell Laboratories, 1975.

[21] D. Jackson and D. A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *ICSM*, pages 243–252, 1994.

[22] W. Jin, A. Orso, and T. Xie. Bert: a tool for behavioral regression testing. In *SIGSOFT FSE*, pages 361–362, 2010.

[23] A. Kuehlmann and F. Krohm. Equivalence checking using cuts and heaps. In *DAC*, pages 263–268, 1997.

[24] S. K. Lahiri, K. Vaswani, and C. A. R. Hoare. Differential static analysis: opportunities, applications, and challenges. In *FoSER*, pages 201–204, 2010.

[25] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *CAV*, pages 712–717, 2012.

[26] S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In *ESEC/FSE 2013*, 2013. .

[27] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[28] A. Mishchenko, S. Chatterjee, R. K. Brayton, and N. Eén. Improvements to combinational equivalence checking. In *ICCAD*, pages 836–843, 2006.

[29] G. C. Necula. Translation validation for an optimizing compiler. In *PLDI*, pages 83–94, 2000.

[30] N. Partush and E. Yahav. Abstract semantic differencing for numerical programs. In *SAS*, pages 238–258, 2013.

[31] D. Peled. All from one, one for all: on model checking using representatives. In *CAV*, pages 409–423, 1993.

[32] S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu. Differential symbolic execution. In *SIGSOFT FSE*, pages 226–237, 2008.

[33] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS*, pages 151–166, 1998.

[34] D. A. Ramos and D. R. Engler. Practical, low-effort equivalence verification of real code. In *CAV*, pages 669–685, 2011.

[35] X. Rival and L. Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.

[36] R. Sharma, E. Schkufza, B. R. Churchill, and A. Aiken. Data-driven equivalence checking. In *OOPSLA*, pages 391–406, 2013.

[37] Y. Song, Y. Zhang, and Y. Sun. Automatic vulnerability locating in binary patches. In *CIS (2)*, pages 474–477, 2009.

[38] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *SAS*, pages 352–367, 2005.

[39] A. Valmari. Stubborn sets for reduced state space generation. In *Applications and Theory of Petri Nets*, pages 491–515, 1989.

[40] P. Wolper and P. Godefroid. Partial-order methods for temporal verification. In *CONCUR*, pages 233–246, 1993.

[41] L. D. Zuck, A. Pnueli, Y. Fang, B. Goldberg, and Y. Hu. Translation and run-time validation of optimized code. *Electr. Notes Theor. Comput. Sci.*, 70(4):179–200, 2002.