Counterexample-Guided Correlation Algorithm for Translation Validation

SHUBHANI GUPTA, Indian Institute of Technology Delhi, India
ABHISHEK ROSE, Indian Institute of Technology Delhi, India
SORAV BANSAL, Indian Institute of Technology Delhi, India

Automatic translation validation across the unoptimized intermediate representation (IR) of the original source code and the optimized executable assembly code is a desirable capability, and has the potential to compete with existing approaches to verified compilation such as CompCert. A difficult subproblem is the automatic identification of the correlations across the transitions between the two programs' respective locations. We present a counterexample-guided algorithm to identify these correlations in a robust and scalable manner. Our algorithm has both theoretical and empirical advantages over prior work in this problem space.

CCS Concepts: • **Software and its engineering → Formal software verification**; *Compilers*.

Additional Key Words and Phrases: Translation Validation, Certified Compilation, Automatic Verification, Equivalence Checking

ACM Reference Format:
Shubhani Gupta, Abhishek Rose, and Sorav Bansal. 2020. Counterexample-Guided Correlation Algorithm for Translation Validation. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 221 (November 2020), 29 pages. https://doi.org/10.1145/3428289

1 INTRODUCTION

Translation validation (TV) [Pnueli et al. 1998] verifies the result of every compilation; this approach stands in contrast with certified compilation where the compiler's logic is verified for all possible input programs [Leroy 2006]. Unlike certified compilation where the compiler usually needs to be written from scratch, TV has the potential to reuse an existing off-the-shelf compiler and validate its input-output behavior for every compilation. For most programs/compiler-transformations, it usually suffices to restrict oneself to bisimilarity checking, where the algorithm proceeds by correlating the transitions (or paths) in the two programs and identifying inductive relational predicates (or invariants) between variables (state-elements) of the two programs at the endpoints of the correlated transitions [Pnueli et al. 1998]. We call the endpoints of the correlated transitions, correlated *PCpairs*, given that they are formed by pairing two program locations or PCs of the respective programs. If these correlations and relational invariants ensure equivalent observable behavior (e.g., identical sequence of I/O events, identical return value and returned heap state), then we have obtained a proof (or witness) of equivalence (and bisimilarity). This proof, involving correlations and invariants, can be represented either as a (bi)simulation relation [Milner 1971; Necula 2000; Pnueli et al. 1998] or as a product program [Zaks and Pnueli 2008], both of which are equivalent representations.

Authors' addresses: Shubhani Gupta, Indian Institute of Technology Delhi, India, shubhani@cse.iitd.ac.in; Abhishek Rose, Indian Institute of Technology Delhi, India, abhishek.rose@cse.iitd.ac.in; Sorav Bansal, Indian Institute of Technology Delhi, India, sbansal@iitd.ac.in.

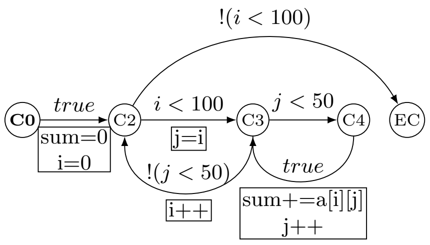
2475-1421/2020/11-ART221
https://doi.org/10.1145/3428289

```
int a[100][50];
C0: void nestedLoop(){
C1:   int sum = 0;
C2:   for(int i=0; i<100; i++) {
C3:     for(int j=i; j<50; j++) {
C4:        sum += a[i][j];
C5:     }
C6:   }
C7:   return sum;
C8: }
```

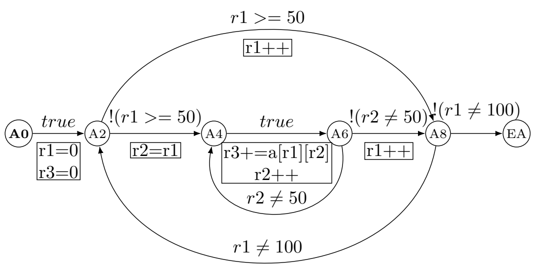
(a) C program

```
A0: nestedLoop:
A1:   r1 = 0; r3 = 0
A2:     if (r1 >= 50) goto A7
A3:     r2 = r1
A4:       r3 += a[r1][r2]
A5:       r2++
A6:       if (r2 != 50) goto A4
A7:     r1++
A8:     if (r1 != 100) goto A2
A9:   ret r3
```

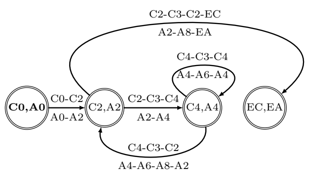
(b) (Abstracted) Assembly program



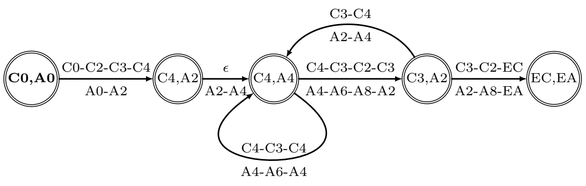
(c) C program CFG. C0 is the entry node and EC is the exit node



(d) Assembly program CFG. A0 is the entry node and EA is the exit node



(e) Product-Graph possibility #1



(f) Product-Graph possibility #2

Node	Invariant
(C0,A0)	$\mathtt{H_C} = \mathtt{H_A}$
(C2,A2)	$(\mathtt{sum}, \mathtt{i}, \mathtt{H_C}) = (\mathtt{r3}, \mathtt{r1}, \mathtt{H_A})$
(C4,A4)	$(\mathtt{sum}, \mathtt{i}, \mathtt{j}, \mathtt{H_C}) = (\mathtt{r3}, \mathtt{r1}, \mathtt{r2}, \mathtt{H_A})$
(EC,EA)	$(\mathtt{sum}, \mathtt{H_C}) = (\mathtt{r3}, \mathtt{H_A})$

(g) Inductive invariants for product-CFG #1

Node	Invariant
(C0,A0)	$\mathtt{H_C} = \mathtt{H_A}$
(C4,A2)	$(\mathtt{sum}, \mathtt{i}, \mathtt{j}, \mathtt{H_C}) = (\mathtt{r3}, \mathtt{r1}, 0, \mathtt{H_A})$
(C4,A4)	$(\mathtt{sum}, \mathtt{i}, \mathtt{j}, \mathtt{H_C}) = (\mathtt{r3}, \mathtt{r1}, \mathtt{r2}, \mathtt{H_A})$
(C3,A2)	$(\mathtt{sum}, \mathtt{i}, \mathtt{j}, \mathtt{H_C}) = (\mathtt{r3}, \mathtt{r1}, \mathtt{r1}, \mathtt{H_A})$
(EC,EA)	$(\mathtt{sum}, \mathtt{H_C}) = (\mathtt{r3}, \mathtt{H_A})$

(h) Inductive invariants for product-CFG #2

Fig. 1. A C program (left) and an equivalent abstracted assembly program (right).

Figures 1a and 1b show an equivalent pair of programs, and in figs. 1c and 1d, the programs are represented as Control Flow Graphs (CFG) where each node represents a PC and each edge represents a transition. Each CFG edge is labeled with an *edge condition* (which specifies the condition under which that edge is taken) and a *transfer function* (which specifies the new state at the destination of the edge). In the figure, the edge condition is shown above the transition edge, and the transfer function is shown inside a box below the transition edge. In this example and in the rest of the paper, we use the following correspondence between a program's PC labels and its CFG's node names: (1) For a general statement, the PC label represents the start of the corresponding statement in the program; (2) For a `for` loop construct, the PC label corresponds to the condition test which is also the loop head; (3) EC and EA represent program exits for the two programs.

In figs. 1e and 1f, we show two distinct product programs: a product program correlates the transitions in one program with the transitions in another program, as though they execute correspondingly in lockstep. We use the CFG representation to show the product program, which we also call a *product-CFG*; multiple product-CFGs are possible for any same pair of programs. Each edge in a product-CFG encodes the PC-transition correlations across the two programs. Given a product-CFG, equivalence checking involves inferring inductive invariants at each node of the product-CFG (e.g., using an off-the-shelf invariant inference algorithm) and then checking if the inferred invariants are strong enough to prove equivalent observable behavior at intermediate program locations (e.g., identical arguments to calls to an external function) or at program exit (e.g., identical return values and identical heap states). Figures 1g and 1h show the inferred inductive invariants at each node in the two product-CFGs in figs. 1e and 1f respectively. In this example, both product-CFGs can prove equivalent observable behavior of the two programs at exit because both the return value `sum` and the heap $\mathtt{H_C}$ in the C program are identical to corresponding state-elements `r3` and $\mathtt{H_A}$ in the assembly program.

There exists a trade-off between the amount of computational effort spent in identifying the "right" product-CFG and the effort spent in identifying the required inductive invariants. *For most programs/compiler-transformations, there* **exists** *a product-CFG where the required invariants (to prove equivalence) are formed by simply relating the bitvector and array values through equality, inequality, and affine relations.* This claim has been observed and assumed by multiple independent prior research efforts [Churchill et al. 2019; Dahiya and Bansal 2017a; Gupta et al. 2018], and we refer to this as **Observation-A** in the rest of the paper. Thus, if we fix our invariant inference algorithm, the problem of translation validation reduces to identifying the required product-CFG.

Each edge in a product-CFG encodes correlated transition *paths* in the two respective programs. Another relevant observation (**Observation-B**) is that *for most programs/compiler-transformations, we can bound the maximum length of a path that needs to be correlated within a single CFG edge.* For example, compilers bound the *unroll factor* while transforming programs. In the rest of the paper, we use C to denote the source program (in C language in our setting) and A to denote the compiled program (in assembly language in our setting). To bound correlated path lengths, we introduce a parameter, μ_C, which represents the maximum number of times a PC may appear in a program path of C that is correlated through a product-CFG edge. We only count a PC to "appear" in a program path if it is present as the head of an edge in that path. For example, the path `(C4-C3-C4)` can be correlated at $\mu_C = 1$ because both `C3` and `C4` appear only once in it. On the other hand, the path `(C4-C3-C4-C3-C4)` cannot be correlated at $\mu_C = 1$ because `C4` appears twice in it. While the use of μ_C lends finiteness to the space of potential product-CFGs, this space is still exponential in the size of the program. This search space is further reduced due to the following two observations:

- **Observation-C**: It usually suffices to restrict the correlated PCs (that constitute the PCpairs) to the heads (first instruction) and tails (last instruction) of the basic blocks of one program's CFG to the loop heads of the other program's CFG. Intuitively, even if the "ideal" product-CFG (that yields a provable bisimulation) required a PC p in the middle of a basic block to be correlated, a product-CFG that instead correlates either the head or the tail of the corresponding basic block (that contains p) also yields a provable bisimulation in most cases. Further, we expect the loop heads to have correlations with the head or tail of some basic block in the other program. In other words, we rely on our invariant inference procedure to be able to bridge this gap between the ideal correlation and a correlation that only considers loop heads in one program and basic block heads and tails in the other.

- **Observation-D**: For a given PCpair, it is rare for an outgoing path in A to be correlated with more than one paths in C such that these (two or more) correlated paths in C have different endpoints, but not vice-versa. Intuitively, this is so because optimizers may specialize program paths in C to yield two or more versions of the same path in the optimized implementation (e.g., loop splitting, peeling, unrolling, unswitching). Conversely, it is relatively rare for an optimizer to combine two different paths in unoptimized C into a single path in optimized A. This latter category of "despecializing" transformations are usually only relevant while optimizing for code size, and are relatively rare.

These four observations (A-D) prune the search space of product-CFGs in the quest for identifying a provable bisimulation. Even so, the number of potential product-CFGs are in the order of 10^{14} for $\mu_C = 8$ for the pair of programs in fig. 1. If we optimistically assume that for a given product-CFG, it takes an average of only one second to infer the invariants (and check equivalence of observables), a naive exhaustive search algorithm would require close to 10^6 years to compute equivalence for this small example.

1.1 Contributions

We present a counterexample-guided algorithm, called Counter[1], to efficiently search this space of potential product-CFGs to yield a provable bisimulation relation. We demonstrate the effectiveness of Counter by statically computing proofs of equivalence across unoptimized LLVM IR programs and their optimized and vectorized x86 assembly code. The x86 code is generated using `-O3 -msse4.2` flags in modern production compilers. The programs used to evaluate Counter are drawn from the TestSuite for Vectorizing Compilers (TSVC) [Maleki et al. 2011] and also include complex loop nest patterns drawn from the LORE repository [Chen et al. 2017]. To our knowledge, Counter is the first algorithm that can compute equivalence across both vectorizing transformations and register realloaction in the presence of multiple loops with potential nesting and control-flow in both programs. An online demo of the equivalence checking tool based on Counter is available at [cou 2020].

1.2 Related Work

This identification of the product-CFG (or correlation) for the construction of a bisimulation proof has been investigated by multiple researchers previously. Early efforts used simple branch correlation heuristics [Necula 2000; Zuck et al. 2003] to construct the product-CFG. Subsequently, data-driven heuristics were proposed [Sharma et al. 2013] to identify a one-to-one correspondence between the loop heads in C and A. All such approaches are inadequate for transformations involving loop peeling and unrolling.

[Dahiya and Bansal 2017a] used a brute-force approach to attempt to correlate program paths in C and A, wherein the product-CFG is constructed incrementally and invariants are inferred on the partially-constructed product-CFG at each step. A new edge (representing a correlation of paths in C and A) is added to the incrementally-constructed product-CFG only if the *path conditions* (the weakest conditions under which the paths are taken) are provably identical (based on the invariants inferred so far). While this approach can theoretically handle a much larger set of transformations, it has significant practical limitations. First, the number of paths that need to be correlated can be exponential in the size of the program and in the unroll factor. The experiments reported in their work discounted the possibility of loop unrolling (causing false equivalence failures); even

[1]The name Counter is intended to represent two facts about the algorithm: (1) it uses counterexamples to identify the most promising correlation, and (2) it counts the number of related variables across the two programs to rank the potential correlations in the order of their promise.

after this simplification, they reported equivalence failures due to timeouts for a large number of equivalence checks. Second, they required the correlated paths to have identical path conditions, which we find is too restrictive as it cannot accommodate all transformations that involve code specialization, e.g., loop unswitching.

[Churchill et al. 2019] addressed these shortcomings in [Dahiya and Bansal 2017a]'s work through a data-driven approach. They first "guess" an *alignment predicate* (AP) that must hold at all nodes of the required product-CFG. Then, using concrete execution traces (on identical inputs) on C and A, they construct a candidate product-CFG (which they call the *Program Alignment Automaton* or PAA) — the execution traces are employed to determine potentially correlated transitions by identifying a correspondence between PCs such that the machine states satisfy the guessed AP at those correlated PCs. By construction, the language accepted by the PAA includes the concrete execution traces (for all available inputs) on C and A. In other words, the PAA represents the product-CFG as *guessed* through the available concrete execution traces. The primary idea is to extrapolate the behavior of the two programs on a small set of concrete traces by using a "good" AP guess, to all possible executions on C and A. This approach is best-effort because: (a) it requires execution traces with adequate path coverage on both C and A; we find that adequate coverage may require traces that exhibit an exponential number of distinct behaviors. (b) It relies on a good AP guess: an AP that is too strong would ignore the desired PAA while an AP that is too weak would result in too many satisfying PAAs, of which most would be incapable of yielding a provable bisimulation. In their paper, the authors synthesize potential APs through a syntactic grammar, and for each potential AP, they pick the first satisfying PAA to see if it can yield a provable bisimulation. This approach was evaluated on mostly single-loop source programs with no control flow within the loop bodies of the source program. In their paper [Churchill et al. 2019], the authors acknowledge that they leave the problem of synthesizing the required AP for programs with multiple loops for future work.

Our approach to identifying the product-CFG neither requires high-coverage execution traces, nor relies on the availability of an alignment predicate. Instead we are able to use a combination of a brute-force search (a la [Dahiya and Bansal 2017a]) and a counterexample-guided best-first strategy (somewhat inspired from [Churchill et al. 2019]) to guide this search. Our experiments demonstrate that Counter can compute equivalence across vectorizing transformations in the presence of register reallocation across different loops, and even in the presence of multiple loops and complex control-flow in both programs.

2 MOTIVATING EXAMPLES

We motivate our algorithm by using examples to critique [Churchill et al. 2019]'s *semantic program alignment* (SPA) algorithm, which we think is the closest competing algorithm in terms of its capabilities. To understand SPA through an example: if for a given input, program C takes path η_C and program A takes path η_A such that the AP is satisfied by the machine states of C and A at the endpoints of η_C and η_A respectively, then a PAA transition (product-CFG edge) that correlates the two paths, η_C and η_A, is added to the PAA. In other words, an edge $e = (\eta_C, \eta_A)$ is added to the PAA only if the states at the two (start and stop) endpoints are related by AP and the programs C and A are known to take these paths η_C and η_A respectively for the same input (for all available concrete inputs). By construction, the start nodes (`(C0,A0)` in fig. 1) and exit nodes (`EC`, `EA`) of the two programs are always correlated in the PAA. Inductive invariants are then inferred on the PAA (which is identical to a product-CFG) and the equivalence proof is completed if the inferred invariants guarantee observable equivalence.

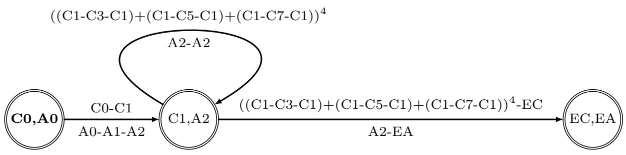
Our first criticism of this approach is that a path is correlated in the PAA only if it is seen to be taken in one of the concrete execution traces. In general, the number of paths can be exponential

```
int LEN, a[LEN], b[LEN];
int c[LEN], d[LEN];
C0: void s441() {
C1:   for (int i = 0; i < LEN; i++) {
C2:     if (d[i] < 0) {
C3:       a[i] += b[i] * c[i];
C4:     } else if (d[i] == 0) {
C5:       a[i] += b[i] * b[i];
C6:     } else {
C7:       a[i] += c[i] * c[i];
C8:     }
C9:   }
C10:}
```

(a) C program.
LEN is a positive multiple of 4.

```
A0:  s441:
A1:   r1 = 0
A2:    xmm1 = a[r1 .. r1+3]
A3:    xmm2 = xmm1 + b[r1 .. r1+3]*c[r1 .. r1+3]
A4:    xmm3 = xmm1 + b[r1 .. r1+3]*b[r1 .. r1+3]
A5:    xmm4 = xmm1 + c[r1 .. r1+3]*c[r1 .. r1+3]
       // pcmpgtd
A6:    xmm0 = (d[r1] < 0), .. , (d[r1+3] < 0)
A7:    xmm1 = xmm0 ? xmm2 : xmm1  // pblendvb
       // pcmpeqd
A8:    xmm0 = (d[r1] == 0), .. , (d[r1+3] == 0)
A9:    xmm1 = xmm0 ? xmm3 : xmm1  // pblendvb
       // pcmpgtd
A10:   xmm0 = (d[r1] > 0), .. , (d[r1+3] > 0)
A11:   xmm1 = xmm0 ? xmm4 : xmm1  // pblendvb
A12:   a[r1 .. r1+3] = xmm1
A13:   r1 += 4
A14:   if (r1 != LEN) goto A2
A15:  ret
```

(b) (Abstracted) Assembly as generated by GCC


((C1-C3-C1)+(C1-C5-C1)+(C1-C7-C1))^4
A2-A2
C0,A0
C0-C1
A0-A1-A2
C1,A2
((C1-C3-C1)+(C1-C5-C1)+(C1-C7-C1))^4-EC
A2-EA
EC,EA

(c) Product-CFG

Fig. 2. Example program taken from TSVC suite

in the size of the program and in the unroll factor, and thus this approach may require traces with an exponential number of distinct behaviors to arrive at the required product-CFG (or PAA).

Consider the example program from TSVC suite listed in fig. 2. The loop body in fig. 2a is unrolled four times and vectorized in the assembly code in fig. 2b. The product-CFG required for this pair of programs is shown in fig. 2c, where we use a *series-parallel digraph* to represent a set of paths, where the + operator indicates parallel composition, and the serial composition is denoted by simply concatenating the edges consecutively in a string. ϵ represents the empty path and a numeric superscript P^n is used to indicate P serially composed with itself n times. For example, (C1-(C2-C3)^2-(ϵ+C3)) represents a set of two paths, namely (C1-C2-C3-C2-C3) and (C1-C2-C3-C2-C3-C3).

For ease of exposition, we show an assembly program with four unrolling in fig. 2 even though the actual unrolling that can be performed by a compiler can be eight or even higher. Thus, a single edge in the assembly program (A2-A2) (edge from A2 to A14 and back to A2) needs to be correlated with all the possible $3^4 = 81$ (or $3^8 = 6561$ in case of 8 unrolling) paths in the unrolled loop body of C. Thus, for an unrolling of eight, SPA requires traces with at least 6561 distinct behaviors (where each behavior corresponds to the traversal of a different path in C) to be able to propose the required PAA. On the other hand, Counter statically identifies this product-CFG without requiring access to even a single execution trace. We find that none of the 28 benchmarks used to evaluate SPA [Churchill et al. 2019] involved control flow inside the loop bodies of the source program, and so this exponential-path problem could not get exposed during SPA's evaluation.

Our second criticism of the SPA approach is that it depends heavily on the availability of the required AP (alignment predicate). An AP that is too strong would prune out the required PAA, while a weak AP may result in too many spurious potential PAAs and the algorithm would

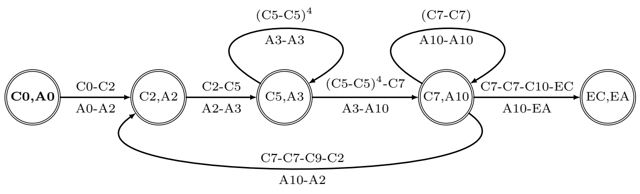
```
int in1[LEN][LEN], in2[LEN];
int out1[LEN], out2[LEN];
C0: void kernel_mvt() {
C1:   int i, j;
C2:   for (i = 0; i < LEN; i++) {
C3:     int sum1 = out1[i];
C4:     int sum2 = out2[i];
C5:     for (j = 0; j < LEN; j++)
C6:       sum1 += in1[i][j] * in2[j];
C7:     for (j = 0; j < LEN; j++)
C8:       sum2 += in1[j][i] * in2[j];
C9:     out1[i] = sum1;
C10:    out2[i] = sum2;
C11:  }
C12: }
```

(a) C program.

LEN is a positive multiple of 4.

```
A0: kernel_mvt:
A1:   r1 = 0
A2:    r2 = 0, r3 = out1[r1], r4 = out2[r1], xmm0 = 0
A3:      xmm0 += in1[r1][r2..r2+3] * in2[r2..r2+3]
A4:      r2 += 4
A5:      if (r2 != LEN) goto A3
A6:    xmm0 += (xmm0 >> 8) // shift right by 8 bytes
A7:    xmm0 += (xmm0 >> 4) // shift right by 4 bytes
A8:    r3 += xmm0[31:0]
A9:    r5 = 0
A10:     r4 += in1[r5][r1] * in2[r5]
A11:     r5++
A12:     if (r5 != LEN) goto A10
A13:   out1[r1] = r3, out2[r1] = r4
A14:   r1++
A15:   if (r1 != LEN) goto A2
A16: ret
```

(b) (Abstracted) Assembly as generated by GCC.



(c) Product-CFG

Fig. 3. A C program (left) and an equivalent abstracted assembly program (right).

be forced to arbitrarily pick one (or some) of the potential spurious PAAs. To see this with an example, consider the pair of programs in fig. 3 taken from the Polybench suite [pol 2019]. The C program contains three loops with two-level maximum nesting depth. The GCC assembly for this program also has three loops where the first inner loop has been unrolled four times. An ideal AP should precisely identify the correlated PCs; in this example, the ideal AP is different for each loop. For example, the ideal AP for the outer loop may need to relate variables i and r1 while the two inner loops may need to relate variables j and r2, and j and r5 in their respective APs. However, the SPA algorithm accepts a single AP, which in this case may need to be something like: $(\mathtt{i} = \mathtt{r1}) \land ((\mathtt{j} = \mathtt{r2}) \lor (\mathtt{j} = \mathtt{r5}))$.

We must discuss how SPA behaves if we choose a less-than-ideal AP in more detail. In fig. 3, if the AP is too strong, such as $(\mathtt{i} = \mathtt{r1}) \land (\mathtt{j} = \mathtt{r2})$, we will miss the required correlation, e.g., we will never be able to correlate any intermediate PCs in the second inner loop. On the other hand, if the AP is too weak, such as $\mathtt{H}_\mathtt{C} = \mathtt{H}_\mathtt{A}$, then we would get a lot of spurious correlations, e.g., a single iteration of the C program's loop would get correlated with a single iteration of the assembly program's loop which would be incorrect.

Based on the discussion in this section, we believe that while the SPA algorithm provides interesting insights into data-driven correlations for equivalence checking, we identify the following improvement opportunities.

(1) An algorithm that needs to identify correlations for each program path individually is not scalable because the number of potential program paths is exponential in program size and unroll factor. To tackle this problem, our algorithm identifies correlations for a set of program paths, or *pathset*, in a single step. Counter attempts to correlate a pathset in *C* with a pathset in *A*, where a single pathset may potentially represent a large number of program paths.

(2) In the presence of multiple loops, the correlation must be developed incrementally: each loop may have its own alignment properties and using a single alignment predicate (AP) for the whole program is unlikely to succeed in general.

Our algorithm uses the following ideas to address these limitations:

(1) We use a series-parallel digraph representation for pathsets to correlate them efficiently across C and A in a single step. This representation enables linear-sized SMT proof obligations while determining correlations across pathsets, even when a single pathset may contain an exponential number of individual paths. The discharge of linear-sized SMT proof obligations, that are generated due to the control-flow transformations performed by a typical compiler, is typically fast, even though it remains worst-case exponential-time.
In contrast to this approach, an algorithm that attempts to correlate each individual path separately (for an exponential number of paths) would require an exponential amount of time, even for computing equivalence across trivial transformations.
(2) The Counter algorithm incrementally constructs the product-CFG through counterexample-guided pruning and ranking subprocedures, and does not require an alignment predicate.

3 ALGORITHM

3.1 Equivalence Definition

A translation validator is generally supposed to check that if the source program C is *safe* (e.g., does not exhibit *undefined behavior* or UB), then the observable behaviors produced by the compiled program A refine the observable behaviors produced by C. In our setting, the source program C is unoptimized LLVM IR translated from C source code, and the compiled program is x86 assembly. To tackle UB in C, we introduce a special "error" state in C which is reached whenever UB gets triggered. In this setting, the equivalence problem can be overapproximately reduced to the checking of *weak trace equivalence*: C and A are equivalent, if for all inputs (a) *either* both programs produce identical (potentially infinite) sequence of observable events; (b) *or* C exhibits UB on that input. This is overapproximate because we do not assume the absence of non-terminating behaviors in C (which is often UB).

Observable events include return values of the program (e.g., exit code of the `main` procedure) and the heap states (which includes the regions belonging to the global variables) at exit. Observable events also include any intermediate calls to *undefined procedures*, i.e., a C language procedure (aka function) whose definition is not available in the same compilation context as its caller (e.g., `printf`), because the compiler and the validator must conservatively assume that such callee procedures can potentially result in I/O events (e.g., through system calls). Because we disable interprocedural optimizations, all callee procedures are undefined in our setting. For the C language, UB includes runtime errors related to type and memory safety.

3.2 Partially-Constructed Product-CFG, Associated Invariants, and Counterexamples

The Counter algorithm starts with a *partial product-CFG* that has only a single start node `(C0,A0)` and incrementally expands it till we obtain a *complete product-CFG* (where all possible program paths are correlated). For every partial product-CFG, we also compute the inductive invariants (through our invariant inference algorithm) at each of its node. In addition to the inductive invariants, we also maintain a set of concrete machine states (data) that may be observed at each product-CFG node. A concrete machine state in the product-CFG is formed by combining the individual abstract machine states of C and A, and assigning concrete values to its state elements. We do not rely on actual execution traces, but generate these concrete machine states through SMT queries (such as those that are made during invariant inference). For example, the potential machine states at

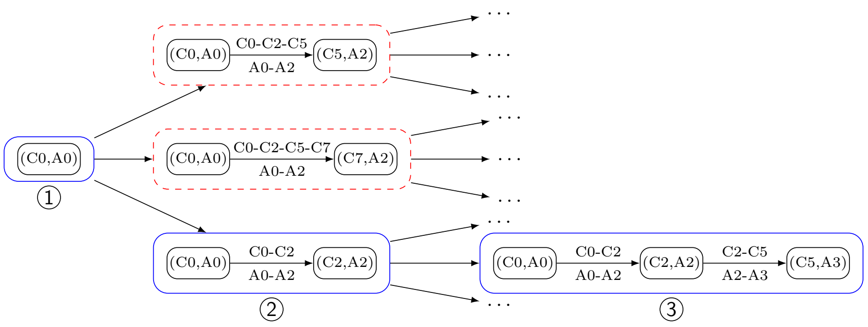


Fig. 4. Backtracking search tree

Table 1. Dataflow formulation for Inference of Inductive Invariants.

Domain	$\left\{ (\mathtt{INV}_n, \Gamma_n) \;\middle|\; \mathtt{INV}_n$ is a conjunction of predicates drawn from grammar in 5b, Γ_n is a set of counterexamples $\right\}$
Direction	Forward
Transfer function	f_ω as specified in fig. 5a
Meet operator $\otimes$ $(\mathtt{INV}_n, \Gamma_n) \leftarrow (\mathtt{INV}^1_n, \Gamma^1_n) \otimes (\mathtt{INV}^2_n, \Gamma^2_n)$	$\Gamma_n \leftarrow \Gamma^1_n \cup \Gamma^2_n, \qquad \mathtt{INV}_n \leftarrow \mathit{StrongestInvCover}(\Gamma_n)$
Boundary condition	$\mathtt{out}[n^{start}] = (\mathtt{Pre}, \{\})$ $\quad$ ($\mathtt{Pre}$ = precondition at start node)
Initialization to $\top$	$\mathtt{in}[n] = (\mathtt{False}, \{\})$ for all non-start nodes

the start node (`C0,A0`) may be identified through SMT queries that assert the invariants at that node (e.g., equivalence of input values and heap states). Because these concrete states are created from the *models* generated through SMT queries, we also refer to them as *counterexamples*. By construction, a counterexample at a product-CFG node must satisfy the inductive invariants at that node. In the context of a product-CFG, a node corresponds to a PCpair and these terms are used interchangeably in the rest of the paper.

Notice that a counterexample need not necessarily be an actually occurring concrete machine state for real inputs. As long as it satisfies the inferred inductive invariant at the respective node, it suffices for our purposes.

3.3 Backtracking Search Tree

At every step of the algorithm, we pick a partially-constructed product-CFG and add another edge (and potentially a node) to it (to expand it). This is repeated until a complete product-CFG is obtained (at which point we check if the inductive invariants ensure observable equivalence). However, at every step, there are several choices on which partial product-CFG to pick, and also which edge to add in that product-CFG. A snapshot of the search tree for the pair of programs in fig. 3 is depicted in fig. 4. Each node in the search tree represents a partially-constructed product-CFG, and the outgoing edges at a node represent the potential possibilities for the newly added edge. Even if the number of possibilities at each step is finite, the full search space remains exponentially large. To lend robustness to this procedure, we use a backtracking-based exhaustive search in this exponentially-large space. We show that a counterexample-based pruning and ranking strategy can efficiently search this space to identify a provable bisimulation relation for several types of aggressive vectorization transformations.

```
Function $f_\omega(\mathtt{INV}_n, \Gamma_n)$
    $\Gamma^{can}_{n^d} \hookleftarrow \Gamma_{n^d} \cup p_\omega(\Gamma_n)$;
    $\mathtt{INV}^{can}_{n^d} \hookleftarrow StrongestInvCover(\Gamma^{can}_{n^d})$;
    while $\mathtt{SAT}(\mathtt{INV}_n \land \neg\mathtt{WP}_\omega(\mathtt{INV}^{can}_{n^d}), \gamma_n)$ do
        $\gamma_{n^d} \hookleftarrow p_\omega(\gamma_n)$;
        $\Gamma^{can}_{n^d} \hookleftarrow \Gamma^{can}_{n^d} \cup \{\gamma_{n^d}\}$;
        $\mathtt{INV}^{can}_{n^d} \hookleftarrow StrongestInvCover(\Gamma^{can}_{n^d})$;
    end
    return $(\mathtt{INV}^{can}_{n^d}, \Gamma^{can}_{n^d})$;
```

(a) Transfer function f_ω for invariant inference.

$$Inv \rightarrow \textstyle\sum_i c_i v_i = c \;\big|\; \mathtt{H_C} =_\Delta \mathtt{H_A} \;\big|\; \pm v \le 2^c \;\big|\; r_1 \le r_2 \;\big|\; r_1 < r_2$$

(b) Predicate grammar for constructing invariants. v represents a bitvector program variable (including registers and stack slots) and c represents a bitvector constant. $\mathtt{H_C} =_\Delta \mathtt{H_A}$ equates the heap states of programs C and A except addresses in Δ. r represents a register in A.

Fig. 5. Transfer function and Predicate grammar for Invariant Inference DFA in table 1. *StrongestInvCover()* computes the strongest invariant cover for a set of counterexamples. p_ω represents the concrete execution function for edge ω. γ_n is the counterexample returned by the SMT solver for a $\mathtt{SAT}()$ query.

3.4 Invariant Inference and Counterexample Generation

At every step of the algorithm, we add an edge to a partial product-CFG to obtain a new partial product-CFG. For each new partial product-CFG thus obtained, we infer inductive invariants at every product-CFG node. Our invariant inference procedure is a forward Data-Flow Analysis (DFA) as described in table 1. The values computed through the DFA are represented by a tuple $(\mathtt{INV}_n, \Gamma_n)$ where $\mathtt{INV}_n$ denotes an invariant at node n and Γ_n denotes a set of counterexamples at node n. For an edge $\omega = n \rightarrow n^d$ from node n to node n^d, the transfer function involves:

(1) Identifying the strongest invariant $\mathtt{INV}_{n^d}$ that is weaker than the strongest-postcondition of the invariant at n, $\mathtt{INV}_n$ across edge ω, and
(2) Adding counterexamples to Γ_{n^d} based on the proof obligations generated during invariant inference.

The DFA's meet operator involves computing the union of the counterexample sets Γ_n and then updating the strongest invariant cover, $\mathtt{INV}_n$, accordingly. The boundary condition initializes the invariants at the start node $(\mathtt{C0},\mathtt{A0})$ or n^{start} to the precondition that asserts the equality of heap states and input arguments while considering ABI and calling conventions; similarly the boundary condition initializes the counterexample set at the start node n^{start} to the empty set. (As an optimization, it is also possible to seed $\Gamma_{n^{start}}$ at the start node with a small set of counterexamples).

An invariant $\mathtt{INV}_n$ is formed by conjuncting atomic predicates drawn from the grammar shown in fig. 5b. The atomic predicates relate the variables across programs C and A. Notice that for the assembly program A, we consider the registers and stack slots as its "variables"; also for vector registers like `xmm`, we also consider 32-bit subwords of such registers as separate variables.

The evaluation of the transfer function f_ω of an edge $\omega = n \rightarrow n^d$ on an invariant $\mathtt{INV}_n$ (and counterexample set Γ_n) at node n involves a fixed-point procedure as shown in fig. 5a. To compute $(\mathtt{INV}_{n^d}, \Gamma_{n^d}) = f_\omega(\mathtt{INV}_n, \Gamma_n)$, we start by propagating the input counterexample set Γ_n over the edge ω using its concrete execution function p_ω to add output counterexamples to the existing set Γ_{n^d} and obtain a new candidate set $\Gamma^{can}_{n^d}$. $\mathtt{INV}^{can}_{n^d}$ denotes the candidate output invariant at n^d and is computed as the *strongest invariant cover* of $\Gamma^{can}_{n^d}$. The strongest invariant cover of $\Gamma^{can}_{n^d}$ is the strongest possible invariant, that can be generated by conjuncting predicates drawn from our grammar (fig. 5b), and that is satisfied by all counterexamples in set $\Gamma^{can}_{n^d}$ e.g., the strongest invariant cover for an empty set is `false`. At each step, we generate a proof obligation which can be represented through a relational Hoare triple [Benton 2004; Hoare 1969] as $\{\mathtt{INV}_n\}\omega\{\mathtt{INV}^{can}_{n^d}\}$.

This Hoare triple states that if the machine starts at node n such that it satisfies $\{\mathtt{INV}_n\}$, and the edge ω is executed, then the resulting machine state would satisfy $\{\mathtt{INV}^{can}_{n^d}\}$. To discharge proof obligations, the Hoare triple is lowered to a propositional boolean logic formula of the form $\mathtt{INV}_n \Rightarrow \mathrm{WP}_\omega(\mathtt{INV}^{can}_{n^d})$ where $\mathrm{WP}_\omega(\mathtt{INV}^{can}_{n^d})$ computes the weakest precondition of $\mathtt{INV}_{n^d}$ across ω. The proof obligation is discharged through an off-the-shelf SMT solver with quantifier-free bitvector, array and uninterpreted function theories. If the proof succeeds, $\mathtt{INV}^{can}_{n^d}$ holds the required invariant ($\mathtt{INV}_{n^d}$) and $\Gamma^{can}_{n^d}$ holds the output set of counterexamples at n^d (Γ_{n^d}) both of which are returned. Otherwise we obtain a counterexample (model) that we add to $\Gamma^{can}_{n^d}$, and also recompute $\mathtt{INV}^{can}_{n^d}$ as the strongest invariant cover of the new $\Gamma^{can}_{n^d}$. This computation is guaranteed to converge, as the newly generated counterexample in $\Gamma^{can}_{n^d}$ would strictly weaken $\mathtt{INV}^{can}_{n^d}$ at each step, and thus we are guaranteed to eventually converge in a finite number of steps (because the semi-lattice formed by the candidate invariants has finite height).

For the affine invariant, the computation of the strongest invariant cover involves computing the strongest affine cover [Müller-Olm and Seidl 2005]. For all other invariants in our grammar, we use a Houdini-like approach [Flanagan and Leino 2001] where a conjunction of all atomic predicates (from the grammar) is used and predicates that are not satisfied by the current set of counterexamples are eliminated to weaken the candidate invariant.

The DFA construction ensures that we obtain the Maximum Fixed Point (MFP) solution for the inductive invariants at each node. A very useful by-product of this procedure is that we also obtain counterexamples at each product-CFG node.

3.5 Counterexample Propagation

In addition to helping our invariant inference procedure, counterexamples in the partial product-CFG also help in identifying the most promising future correlations, as we discuss later. For both these reasons, more counterexamples at each node are desirable. Thus, we *propagate* every generated counterexample in the product-CFG in the forward direction to populate the Γ sets at downstream nodes. The propagation of a counterexample is quite similar to interpreted execution of the product-CFG on a concrete machine state, with two operational differences. First, counterexamples need not have concrete valuations for all live program variables — they just contain valuations for those variables that were a part of the SMT query that generated it. Thus, during propagation, if the program reads a variable that is not already present in the counterexample, a random value is generated for that variable and added to the counterexample. Second, the potential presence of UB in the *C* program may interfere with counterexample propagation. During propagation, if a counterexample triggers UB, we do not propagate the counterexample any further — in other words, the counterexample transitions to the special "error" state meant to catch UB. Thus, a counterexample differs from inputs derived from real program traces: while real traces must never trigger UB, a counterexample generated through an SMT query has no such requirement.

Even though counterexamples cannot replace real execution traces, a counterexample is still useful because it satisfies the inferred invariant at the node at which it was generated, *and* it does not trigger UB on the paths on which it is propagated. Thus it is safe to consider a counterexample at par with a real concrete machine state for these smaller program segments where it does not trigger UB, because our reasoning power in these smaller segments is constrained by the inferred invariants in any case.

To ensure termination, we bound the maximum number of times a counterexample propagation may encounter a node — we call this the *propagation bound* which is set to 3 in our equivalence checking tool. If the counterexample visits a node more than thrice during propagation, we do not propagate it any further. Using a propagation bound of 3 is meaningful because it is small enough to result in efficient propagations, and yet it is usually able to produce new useful counterexamples from an existing counterexample (e.g., over a loop edge) without requiring more SMT queries. Each time a new counterexample gets generated (through an SMT query), counterexample propagation is attempted on it. Additionally, whenever a new edge $\omega = n \rightarrow n^d$ is added to the partial product-CFG, the counterexamples at n are propagated to n^d and further, subject to propagation bound.

Whenever a counterexample set is updated, we recompute its strongest invariant cover. Notice that the invariant cover for a counterexample set need not be inductively provable, but serves as a lower bound on the inductive invariant (which would be computed later). As we will see later (section 3.9), the recomputed invariant cover is helpful in pruning and ranking the candidate partial product-CFGs to implement a best-first exploration of the search space. It is important to note that the computation of the strongest invariant cover is significantly cheaper than an SMT query; thus pruning based on such computation (to save an SMT query) is meaningful.

3.6 Anchor Nodes in A and C

To reduce the number of possibilities at each step of the search tree, we restrict the nodes (PCs) of a program that may be correlated in the product-CFG, also called that program's anchor nodes.

We restrict A's anchor nodes to one of the following three possibilities:

(1) The start and exit nodes of A.
(2) Any node in A that is incident (incoming or outgoing) to a CFG edge that involves a call to an undefined function. An undefined function is a function whose definition is not available in the current compilation (and validation) unit.
(3) Loop heads in A for loop bodies that do not already contain an anchor node on one of the cyclic paths[2].

Calls to undefined functions may result in observable events such as I/O, and thus they can be easily correlated by matching their potential observable events. Considering loop heads as anchor nodes ensures that there is at least one PC in every cyclic path in A that may get correlated in the product-CFG.

Similarly, we restrict C's anchor nodes too. Unlike A, the anchor nodes in C include all basic block heads and tails. Further, all nodes that are incident to an edge with a function call are included in the set of anchor nodes for C. Our choice of anchor nodes for A and C appeals to Observation-C in section 1, so that we still expect the space of potential product-CFGs to contain the required solution.

In fig. 3b, `A0`, `A2`, `A3`, `A10`, and `EA` form the set of anchor nodes for program A. Similarly, in fig. 3a, `C0`, `C1`, `C2`, `C3`, `C5`, `C6`, `C7`, `C8`, `C9`, `C11`, and `EC` form the set of anchor nodes in C.

3.7 (μ, δ)-Unrolled Full Pathsets and Their Correlation Criterion

3.7.1 Pathset. A pathset ξ is a set of execution paths with the following two requirements:

(1) All execution paths in ξ start at the same node and end at the same node. For example, in fig. 2a, the paths `(C1-C3-C1)` and `(C1-C5-C1)` may be a part of the same pathset; but paths `(C1-C3-C1)` and `(C1-C5-EC)` may not be a part of the same pathset.
(2) All execution paths in a pathset should be pairwise mutually-exclusive, i.e., if one path is taken, another path in the same pathset cannot be taken simultaneously. For example, in fig. 2a, the paths `(C1-C3-C1)` and `(C1-C5-C1-C3-C1)` may be a part of the same pathset. However, the paths `(C1-C3-C1)` and `(C1-C3-C1-C5-C1)` may not be a part of the same pathset because the first path `(C1-C3-C1)` is a prefix of the second path `(C1-C3-C1-C5-C1)` indicating that if the second path is taken, the first path is also taken simultaneously.

[2]We use depth-first search to identify the loop heads on cyclic paths; we do not assume reducible CFGs in our work

A product-CFG edge correlates a pathset in C with a pathset in A. For example, in fig. 2, the product-CFG correlates a pathset in C formed by $3^4 = 81$ paths (represented as series-parallel digraph `((C1-C3-C1)+(C1-C5-C1)+(C1-C7-C1))`^{4}) with pathset `(A2-A2)` (single path) in program A.

3.7.2 (μ, δ)-Unrolled Full Pathset. To tackle different possible unrollings, we introduce the notion of a *(μ, δ)-unrolled full pathset.* A (μ, δ)*-unrolled full pathset* between two nodes s and t is the set of all possible paths from s to t such that no node other than t is repeated more than μ times on any one path and t is repeated exactly δ times on all paths. We denote this as $\text{FP}^{\mu,\delta}_{s \rightsquigarrow t}$. We are only interested in pathsets where $\delta \le \mu$.

For example, in fig. 3a, $\texttt{FP}^{2,2}_{\texttt{C2} \rightsquigarrow \texttt{C7}}$ contains paths `(C2-C5-C7-C7)`, `(C2-C5-C5-C7-C7)` and `(C2-C5-C7-C2-C5-C7)`.

Further, we define $\texttt{FPsets}^{\mu}_{s \rightsquigarrow t} = \{\texttt{FP}^{\mu,\delta}_{s \rightsquigarrow t} | 1 \le \delta \le \mu\}$. $\texttt{FPsets}^{\mu}_{s \rightsquigarrow t}$ is a *set of μ pathsets*, where each element is the pathset $\texttt{FP}^{\mu,i}_{s \rightsquigarrow t}$ (for $1 \le i \le \mu$).

In fig. 3a, $\texttt{FPsets}^{2}_{\texttt{C5} \rightsquigarrow \texttt{C7}}$ is a set of two elements: $\{\texttt{FP}^{2,1}_{\texttt{C5} \rightsquigarrow \texttt{C7}}, \texttt{FP}^{2,2}_{\texttt{C5} \rightsquigarrow \texttt{C7}}\}$. Here, $\texttt{FP}^{2,1}_{\texttt{C5} \rightsquigarrow \texttt{C7}}$ contains `(C5-C7)`, `(C5-C5-C7)`, and `(C5-C5-C5-C7)`. Similarly, $\texttt{FP}^{2,2}_{\texttt{C5} \rightsquigarrow \texttt{C7}}$ contains `(C5-C7-C7)`, `(C5-C5-C7-C7)`, `(C5-C5-C5-C7-C7)`, `(C5-C7-C2-C5-C7)`, `(C5-C5-C7-C2-C5-C7)`, and `(C5-C7--C2-C5-C5-C7)`. Note that $\texttt{FP}^{2,1}_{\texttt{C5} \rightsquigarrow \texttt{C7}}$ and $\texttt{FP}^{2,2}_{\texttt{C5} \rightsquigarrow \texttt{C7}}$ are disjoint (they do not have a common path). In general, two full pathsets with different values of δ are disjoint by definition. Further, all paths within a single pathset (say $\texttt{FP}^{2,2}_{\texttt{C5} \rightsquigarrow \texttt{C7}}$) are pairwise mutually-exclusive by construction.

3.7.3 Criterion for Correlating Pathsets. By definition, if an edge $\omega = (\xi_C, \xi_A)$ is traversed in the product-CFG, it implies that one of the paths in ξ_C is traversed in program C and one of the paths in ξ_A is traversed in program A. While this property defines the general space of potential correlations, we restrict this space further through a *correlation criterion* to achieve better tractability. We restrict correlations by requiring that ξ_C can be correlated with ξ_A through a product-CFG edge $\omega = (\xi_C, \xi_A)$ only if the following property holds: *if any of the paths in ξ_A is traversed in program A, then one of the paths in ξ_C in program C must be traversed.* For example, our algorithm does not allow $\xi_A =$`(A2-A2)` to be correlated with $\xi_C =$`(C1-C3-C1)`^{4} for the pair of programs in fig. 2 because if ξ_A is traversed in A, then paths outside ξ_C may be traversed in C (e.g., `(C1-C5-C1)`^{4}).

This restriction that mandates that ξ_A must represent a specialization of ξ_C is a direct reflection of Observation-D introduced in section 1. We argue, through empirical evaluation, that this restriction significantly reduces the product-CFG search space without hurting robustness.

To compare this restriction with previous work, the SPA algorithm [Churchill et al. 2019] has no such requirement: given the required AP (alignment predicate), it could potentially correlate any C path with any A path in theory. However, for the APs and the transformations considered in their paper, there is no program-pair and associated PAA (or product-CFG) that violates this requirement. This is unsurprising because 27 out of 28 programs evaluated in their work have a single loop in the program and none of them have control flow within the loop body. This restriction allows us to scale to larger programs, while still retaining robustness to a very high degree.

3.8 Enumerating Candidate Correlations

For a given product-CFG and for a node $n = (n_C, n_A)$ in that product-CFG, we are interested in identifying the outgoing product-CFG edges $\omega = n \rightarrow n^d = (\xi_C, \xi_A)$. We first enumerate the possibilities for ξ_A, and for each ξ_A, we enumerate possibilities for ξ_C.

3.8.1 Enumerating Outgoing Pathsets ξ_A in A. To enumerate outgoing pathsets ξ_A starting at n_A in A, we restrict our attention to only those pathsets that end at one of the *nexthop anchor nodes* of n_A. A nexthop anchor node of n_A is an anchor node which can be reached from n_A through a program path in A without having to go through any other anchor node in A.

To enumerate outgoing pathsets ξ_A at n_A, we use a two-step procedure. First, for each nexthop anchor node h_A of n_A, we compute $\texttt{FPsets}^{1}_{n_A \rightsquigarrow h_A}$ (i.e., unroll factor = 1, indicating no unrolling). Notice that for a fixed h_A, $\texttt{FPsets}^{1}_{n_A \rightsquigarrow h_A}$ will be a singleton set whose only element represents the full pathset $\text{FP}^{1,1}_{n_A \rightsquigarrow h_A}$. In our second step, we remove those paths from $\text{FP}^{1,1}_{n_A \rightsquigarrow h_A}$ that contain edges incident to any other anchor node in A (except h_A). The resulting pathset ξ_A is our correlation candidate. For example, in fig. 3b, $\text{FP}^{1,1}_{\texttt{A3} \rightsquigarrow \texttt{A3}}$ can be represented as `((A3-A3)+(A3-A10-A2-A3))`. After the second step, ξ_A reduces to only `(A3-A3)` (the other path incident to `A10` and `A3` is removed).

By construction all these candidate outgoing pathsets in A, ξ_A, are mutually exclusive. In the complete product-CFG, we thus need to correlate all such ξ_A pathsets. Our algorithm proceeds by correlating one such ξ_A at each step. To reduce backtracking, we pick these nexthop nodes h_A (and associated candidate pathsets) in *Reverse Post-Order* (RPO). In fig. 3c, starting at product-CFG node `(C5,A3)`, the nexthop anchor nodes in program A (in reverse post-order) are `A3` and `A10`.

3.8.2 Enumerating Outgoing Pathsets ξ_C in C. To enumerate pathsets in C starting at n_C, we consider all anchor nodes w_C, and compute the full pathsets $\texttt{FPsets}^{\mu_C}_{n_C \rightsquigarrow w_C}$ and add all such pathsets to an accumulator κ_{n_C}, i.e., $\kappa_{n_C} = \{\epsilon\} \cup (\bigcup_{w_C} \texttt{FPsets}^{\mu_C}_{n_C \rightsquigarrow w_C})$ (ϵ represents the empty path). Each non-ϵ element in κ_{n_C} represents a full pathset (at an unroll factor μ_C) to some node w_C in C. Notice that unlike the pathsets enumerated for A, here we do not restrict w_C to be the nexthop of n_C.

Consider the example in fig. 3a and assume we are enumerating pathsets in C starting at node `C5` for $\mu_C = 2$. Based on the procedure described in this section, we would consider all possibilities for w_C, which are `C2`, `C3`, `C5`, `C6`, `C7`, `C8`, `C9`, `C11`, and `EC` (basically all anchor nodes in C except those that are not reachable from `C5`). For each possible w_C, we compute $\texttt{FPsets}^{\mu_C}_{n_C \rightsquigarrow w_C}$ as described in section 3.7.2. The elements of this enumerated κ_{n_C}represents the candidates for ξ_C.

3.9 Counterexample-Guided Pruning and Ranking

At every PCpair in a partial product-CFG, for every enumerated outgoing pathset in A, we need to enumerate pathsets in C as candidate correlations. To keep our backtracking search tractable, it is crucial to carefully prioritize more promising correlations over others. This prioritization allows us to realize a *best-first search.* At a high level, we first enumerate all possibilities for pathsets in C (in an arbitrary order) and then *prune* out certain possibilities because the current set of counterexamples decisively indicate that those possibilities cannot yield a provable bisimulation (sections 3.9.1 and 3.9.2). Next, we use counterexamples to *rank* the remaining possibilities from most promising to least promising (sections 3.9.3 to 3.9.5).

For the following discussion in this section, we consider a PCpair $n = (n_C, n_A)$ in the product-CFG and consider a newly-created correlation between pathset ξ_C (starting at node n_C in C) and ξ_A (starting at node n_A in A). The counterexample set at node n is Γ_n. To represent the candidate correlation, an outgoing product-CFG edge $\omega = n \rightarrow n^d = (\xi_C, \xi_A)$ is added to the product-CFG. Further, counterexamples in Γ_n are propagated across $\omega = n \rightarrow n^d$ to add counterexamples at node n^d, and potentially n^d's successors, subject to the propagation bound. During counterexample propagation, as the counterexample sets are updated, the strongest invariant covers at n^d (and other downstream nodes) are recomputed. In our following discussion on the ranking procedure, we assume that we are comparing two partial product-CFGs π_1 and π_2 which are otherwise identical, but differ in the most-recently added ω edge. We later generalize this discussion to comparison between arbitrary partial product-CFGs.

3.9.1 Pruning Based on Paths Taken by Counterexamples. If there exists a counterexample $\gamma \in \Gamma_n$ such that, when propagated on the outgoing paths, γ takes a path in ξ_A but does not take any of the paths in ξ_C, then that candidate correlation is discarded. This pruning strategy appeals to our correlation criterion (section 3.7.3) which requires that if program A takes a path in ξ_A, then program C must take one of the paths in ξ_C. If an existing counterexample violates this criterion for a candidate correlation, that candidate correlation is evidently incorrect.

3.9.2 Pruning Based on Heap Relations. If the heap states $\mathtt{H_C}$ and $\mathtt{H_A}$ are not related by the recomputed invariant cover at n^d (or any other downstream node), we eliminate that candidate product-CFG. This is based on the premise that the heap states need to be correlated at program exit, and if they are not correlated at an intermediate PCpair, then there is little hope for them to be correlated at exit. Thus it is safe to eliminate all such partial product-CFGs.

3.9.3 Ranking on Number of Affine-Related Live Bitvector Variables in A. To compare two candidate correlations, represented as two candidate partial product-CFGs, π_1 and π_2, we count the number of live bitvector variables in A that are related through affine relations at n^d. If the number of live bitvector variables in program A related through affine relations in $\mathtt{INV}_{n^d}$ is more in π_1 than in π_2, then π_1 is ranked higher than π_2, and vice-versa. Notice that n^d may be different in π_1 and π_2.

This ranking strategy is based on the heuristic that an incorrect correlation would likely cause some of the live bitvector variables in A to not have any relation to the program values at the correlated PC in C.

3.9.4 Ranking on Number of Affine-Related Live Bitvector Variables in C. If the first ranking step results in a tie (i.e., the number of affine-related live bitvector variables in A is identical in both correlations), we compare the number of live bitvector variables in program C that are related through affine relations in $\mathtt{INV}_{n^d}$: if the number of live bitvector variables in C with affine relations is more in π_1, then π_1 is ranked higher, and vice-versa.

This ranking strategy is also based on the heuristic that a correct correlation is likely to relate more C variables than an incorrect correlation.

3.9.5 Static Heuristic as Tie-Breaker. If both correlations behave identically on the two ranking criteria listed above, we use the following *static heuristic* as a tie-breaker:

- Recall that all correlated pathsets in C are (μ, δ)-unrolled full pathsets. Also recall that for the pathset ξ_C between nodes n_C and n_C^d, the node n_C^d is repeated exactly δ times on all paths (indicating δ unrollings). We prioritize the correlation that correlates a pathset with a lower value of δ. This heuristic is based on the observation that most program transformations do not involve unrolling, and so it is more efficient on average to prioritize correlations at smaller unroll factors.
- If the unroll factors of the two candidate correlated pathsets ξ_C are identical, then we prioritize the correlation that has ξ_C with a longer *pathset length*. The pathset length is the length of the shortest path in that pathset. This tie-breaker is meaningful because longer paths would generally entail stronger path conditions and thus have a higher likelihood of failing our correlation criterion (described in section 3.7.3) in case of an incorrect correlation. In other words, this heuristic of prioritizing the longer path over shorter paths resembles a "fail fast" strategy.

So far, we have described the ranking strategy in the context of a single correlation (of ξ_A with ξ_C). However, a product-CFG is made up of multiple edges, each denoting a separate correlation. For our best-first search algorithm, we need to compare one product-CFG with another even if they may involve multiple different correlations. To allow such comparisons, we extend these

```
Function computeRank(π)
    $r^A \leftarrow 0 \quad r^C \leftarrow 0$
    foreach node $n = (n_C, n_A) \in$ nodes($\pi$) do
        $live^A \leftarrow getLiveVariables(n_A, A) \qquad live^C \leftarrow getLiveVariables(n_C, C)$
        $r^A \leftarrow r^A + getVariablesWithNoAffineRelations(live^A, INV_n)$
        $r^C \leftarrow r^C + getVariablesWithNoAffineRelations(live^C, INV_n)$
    end
    return $(r^A, r^C)$
Function staticHeuristic($\pi_1$, $\pi_2$)
    $\xi_1^C \leftarrow lastCorrelatedSrcPathset(\pi_1) \qquad \xi_2^C \leftarrow lastCorrelatedSrcPathset(\pi_2)$
    $\mu_1 \leftarrow getUnrollFactorForPathset(\xi_1^C) \qquad \mu_2 \leftarrow getUnrollFactorForPathset(\xi_2^C)$
    $len_1 \leftarrow getPathSetLength(\xi_1^C) \qquad len_2 \leftarrow getPathSetLength(\xi_2^C)$
    return $(\mu_1, -len_1) \le (\mu_2, -len_2)$
Function comparePromiseForProductCFGs($\pi_1$, $\pi_2$)
    $(r_1^A, r_1^C) \leftarrow$ computeRank($\pi_1$) $\qquad (r_2^A, r_2^C) \leftarrow$ computeRank($\pi_2$)
    if $(r_1^A, r_1^C) \ne (r_2^A, r_2^C)$ then
        return $(r_1^A, r_1^C) < (r_2^A, r_2^C)$
    end
    return staticHeuristic($\pi_1$, $\pi_2$)
```

Fig. 6. Comparison function used to rank product-CFGs during best-first search. The comparison operators $<, \le$ for tuples compare lexicographically starting with the first element.

ideas to the whole product-CFG by accumulating the number of live bitvector variables (in A and C) that have not been correlated at every PCpair, and then comparing these accumulated counts. This comparison function is shown in fig. 6; *comparePromiseForProductCFGs()* compares two product-CFGs for their relative promise towards yielding a provable bisimulation; it returns true iff π_1 holds more promise than π_2.

3.10 Pruning and Ranking Algorithms Through Examples

Consider the example program pair in fig. 3, and suppose we are considering the partial product-graph ③ in fig. 4. The algorithm would next try to correlate the pathset (A3-A3) in A starting at PCpair (C5,A3), based on RPO.

For $\mu_C = 4$, the enumerated pathsets for C (κ_{n_C}) would include the ϵ path and full pathsets starting from C5 and ending at one of the following nine reachable anchor nodes in C nodes: C2, C3, C5, C6, C7, C8, C9, C11, and EC. In other words, we get 34 candidate pathsets, one for each element in the set $\{\{\epsilon\} \cup \mathtt{FPsets}^4_{C5 \rightsquigarrow C2} \cup \mathtt{FPsets}^4_{C5 \rightsquigarrow C3} \cup \mathtt{FPsets}^4_{C5 \rightsquigarrow C5} \cup \mathtt{FPsets}^4_{C5 \rightsquigarrow C6} \ldots\}$. To keep our following discussion simpler, we restrict our attention to only those pathsets that end at loop heads in C (even though the algorithm considers all pathsets and still operates efficiently). The loop heads in C are C5, C7, and C2, and hence the candidate pathsets that we will consider for (A3-A3) belong to $\{\{\epsilon\} \cup \mathtt{FPsets}^4_{C5 \rightsquigarrow C5} \cup \mathtt{FPsets}^4_{C5 \rightsquigarrow C7} \cup \mathtt{FPsets}^4_{C5 \rightsquigarrow C2}\}$, for a total of 13 (out of 34) possibilities.

3.10.1 Pruning Based on Paths Taken by Counterexamples. Recall that our path condition pruning checks that starting at PCpair (C5,A3), if pathset ξ_A is correlated with pathset ξ_C, then a counterexample at (C5,A3) that takes one of the paths in ξ_A must also take one of the paths in ξ_C (to satisfy our correlation criterion in section 3.7.3). Also we must have at least one (usually a few) counterexamples at (C5,A3) that must have been added either during invariant inference or counterexample propagation or both. Let's assume that one of the counterexamples at (C5,A3) is $\{$ i $\mapsto 0$, sum1 $\mapsto 10$, j $\mapsto 0$, r1 $\mapsto 0$, r2 $\mapsto 0$, r3 $\mapsto 10$, xmm0 $\mapsto 0$, out1 $\mapsto 0$, in1 $\mapsto 200$, in2

$\mapsto 2000$, LEN $\mapsto 12$, $\texttt{H}_\texttt{C}$[3] $\mapsto (0 \mapsto 10, 200 \mapsto 1, 204 \mapsto 2, 208 \mapsto 3, 212 \mapsto 4, 2000 \mapsto 5, 2004 \mapsto 6, 2008 \mapsto 7, 2012 \mapsto 8, () \mapsto 0)$, $\texttt{H}_\texttt{A} \mapsto (0 \mapsto 10, 200 \mapsto 1, 204 \mapsto 2, 208 \mapsto 3, 212 \mapsto 4, 2000 \mapsto 5, 2004 \mapsto 6, 2008 \mapsto 7, 2012 \mapsto 8, () \mapsto 0), \ldots\}$.
This counterexample would traverse the path (A3-A3) in program A because it satisfies the corresponding path condition, $(\texttt{r2} + 4 \neq \texttt{LEN})$. However, of the 13 candidate pathsets in C, only 5 would be taken by this counterexample (those that end at C5 at different unrollings and ϵ path). Thus, path condition pruning reduces the candidate correlations from 13 to 5 based on just a single counterexample in this case.

3.10.2 Counterexample-Guided Ranking. After path condition pruning, we are left with five candidate correlations for (A3-A3), namely ϵ, (C5-C5), $\texttt{(C5-C5)}^2$, $\texttt{(C5-C5)}^3$, and $\texttt{(C5-C5)}^4$. Thus, we create five different product-CFGs, each with a *different* newly added product-CFG edge $\omega = (\texttt{C5}, \texttt{A3}) \rightarrow (\texttt{C5}, \texttt{A3})$ (the difference is in the correlated pathset), and add all the five product-CFGs to the backtracking search tree. Further, for each of the newly created product-CFGs, we propagate the counterexamples at (C5,A3) across ω, adding more counterexamples to (C5,A3). Notice that these added counterexamples would be different for each of the product-CFGs because they would have traversed different program paths in C before being added to the product-CFG node (C5,A3).

For example, for the product-CFG that correlates (A3-A3) with $\texttt{(C5-C5)}^2$, the counterexample discussed in section 3.10.1 would yield a new propagated counterexample at (C5,A3) obtained by simply propagating it once over the newly added product-CFG edge $\omega = ((\texttt{A3} - \texttt{A3}), (\texttt{C5} - \texttt{C5})^2)$:
$\{$ sum1 $\mapsto 27$, j $\mapsto 2$, r2 $\mapsto 4$, r3 $\mapsto 10$, xmm0 $\mapsto$ 0x20000000150000000C00000005, $\ldots\}$.
Similarly, for the product-CFG that correlates (A3-A3) with $\texttt{(C5-C5)}^4$, the counterexample after propagating once would be:
$\{$ sum1 $\mapsto 80$, j $\mapsto 4$, r2 $\mapsto 4$, r3 $\mapsto 10$, xmm0 $\mapsto$ 0x20000000150000000C00000005, $\ldots\}$.

After propagation, we compute the strongest invariant cover for the updated set of counterexamples at (C5,A3) for each of the five candidate product-CFGs. Now, we claim that for the correct correlation, the updated invariant cover at (C5,A3) would likely relate more variables in A to variables in C through affine relations. Conversely, for incorrect correlations, the invariant cover at (C5,A3) would likely relate fewer variables in A. To see this more concretely, consider the live variables $\texttt{xmm0}_0$, $\texttt{xmm0}_1$, $\texttt{xmm0}_2$, and $\texttt{xmm0}_3$ in A, where $\texttt{xmm0}_i$ is shorthand for xmm0[(32*i+31):(32*i)]. For the incorrect correlations, only some of these four variables may get related by the invariant cover at (C5,A3) — e.g., for the product-CFG that correlates (A3-A3) with $\texttt{(C5-C5)}^2$, the variables $\texttt{xmm0}_0$ and $\texttt{xmm0}_1$ would be related to sum1 through (sum1 = $\texttt{xmm0}_0$ + $\texttt{xmm0}_1$ + r3), but $\texttt{xmm0}_2$ and $\texttt{xmm0}_3$ would not get related to any variables in C. However, for the product-CFG that correlates (A3-A3) with $\texttt{(C5-C5)}^4$, all four parts of xmm0 would get related to sum1 through the invariant cover (sum1 = $\texttt{xmm0}_0$ + $\texttt{xmm0}_1$ + $\texttt{xmm0}_2$ + $\texttt{xmm0}_3$ + r3). For this reason, this latter product-CFG would rank higher than the others in our algorithm. Thus, ranking helps the algorithm in arriving at the correct correlation in the first attempt during the best-first search procedure in this example.

3.10.3 Pruning Based on Heap Relations. To demonstrate our pruning based on heap relations, we consider a partial product-CFG such that it has almost all the required correlations, as shown in fig. 3c, except that it has not yet correlated the pathset (A10-A2) starting at PCpair (C7,A10). Among the various correlations for (A10-A2), two candidates are (C7-C2) and (C7-C2-C5-C7-C2). However we see that while the first candidate updates the heap only once (through writes to out1[i]

[3]The concrete state for memory array is represented using mappings of the form (addr $\mapsto$ data), which implies that the value (byte) stored in memory at address "addr" is "data" and (() $\mapsto$ data) represents the default value "data" for the remaining address space.

and `out2[i]`), the second candidate updates the heap twice. Thus, the latter candidate correlation (`C7-C2-C5-C7-C2`) is likely to generate an invariant cover (for the propagated counterexamples) that does not relate the two heap states $\mathtt{H_C}$ and $\mathtt{H_A}$, causing it to be pruned out. In general, such pruning is very effective in the presence of writes to heaps where the addresses of the writes cannot be characterized at compile time. This pruning strategy is somewhat similar, albeit more flexible and general, to the correlation strategy used in Necula's translation validator [Necula 2000] where the algorithm required one-to-one correspondence between accesses to the heap for correlation.

3.10.4 Contrast with SPA Algorithm. Both SPA and Counter are data-driven because they rely on data (or counterexamples) to predict (or prioritize) the correlations through path correlations or relations on machine state values. However, Counter represents a significant generalization and improvement over the SPA approach because:

- Counter does not restrict the correlation condition to be one of the enumerated alignment predicates. Instead it takes a more flexible approach where it simply uses the number of relations in the strongest invariant cover for the concrete counterexamples known so far. This flexibility in Counter eliminates the dependence on the availability of the required alignment predicate (which is quite complex for the example pair of programs in fig. 3).
- Instead of proposing a single product-CFG (or PAA), Counter formulates the algorithm as a best-first search strategy to avoid getting stuck inside a local search subspace. However, our experiments demonstrate that the algorithm converges to the required product-CFG in the first attempt with a very high probability (section 4).
- Correlation of pathsets (instead of individual paths) avoids the explosion in the number of required correlations. Our experiments confirm that our method of using full pathsets at different unroll factors (section 3.7.2) along with our correlation criterion (section 3.7.3) does not compromise robustness.

3.11 Putting It All Together

We provide pseudo-code to tie all the sub-procedures into a single algorithm in fig. 7. The top-level procedure ***bestFirstSearch()*** takes as arguments the two programs, C and A, and returns either a product-CFG that is a provable bisimulation (if proof found) or `null` (if the proof was not found). This top-level procedure initializes a product-CFG π_{init} with the start node (`C0,A0`) and initializes the *frontier* Ω of the best-first search tree by adding a candidate correlation (as a product-CFG edge) to π_{init} (through call to *expandProductCFG()*). At each step, the best-first search picks the most promising product-CFG π_{cur}, based on the ranking strategy described in fig. 6, from the frontier Ω (*removeMostPromising()*) and "expands" it by adding another candidate correlation through a new product-CFG edge to it. There are multiple possibilities for the new candidate correlations, and all the newly created product-CFGs (if any) are added back to the frontier. On return from the *expandProductCFG()*, π_{cur} is checked to see if it already yields a provable bisimulation (*ProductCFGisProvableBisim()*).

The ***expandProductCFG()*** function first checks that the correlation criterion (section 3.7.3) is met for all the edges in the input product-CFG π, through the subroutine *checkCriterionForEdges()*. *checkCriterionForEdges()* internally checks the following condition for every product-CFG edge $\omega = n \rightarrow n^d = (\xi_C, \xi_A)$: $Inv_n \Rightarrow (pscond_{\xi_A} \rightarrow pscond_{\xi_C})$ where $pscond_{\xi}$ represents the *path-set condition* of pathset ξ, which is equivalent to the disjunction of the path-conditions (i.e., the weakest condition under which the path must be taken) of the individual paths in ξ. The *inferInvariantsAndCounterExamples()* subroutine implements the invariant inference procedure described in section 3.4. The *findIncompleteNode()* function identifies a PCpair $n = (n_C, n_A)$ in π that has not yet correlated all outgoing paths from n_A in program A. If no such PCpair exists, the product-CFG is

```
Function expandProductCFG($\pi$, C, A)
    if $\neg$ checkCriterionForEdges($\pi$) then
        return {}
    end
    inferInvariantsAndCounterExamples($\pi$)
    if $\neg$ (($n_C, n_A$) $\leftarrow$ findIncompleteNode($\pi$)) then
        return {}
    end
    $\Sigma \leftarrow$ {}
    $\xi_A \leftarrow$ getNextPathsetRPO($n_A$, A)
    $\Psi \leftarrow$ getCandCorrelations(($n_C, n_A$), C, $\xi_A$, $\mu_C$)
    foreach $\xi_C \in \Psi$ do
        if actionsAreCompatible($\xi_C$,$\xi_A$) then
            $\pi_{new} \leftarrow \pi$
            addEdgeAndPropCEs($\pi_{new}$, ($\xi_C$, $\xi_A$))
            if CEsSatisfyCorrelCriterion($\pi_{new}$) $\wedge$
            InvRelatesHeapsAtEachNode($\pi_{new}$) then
                $\Sigma \leftarrow \Sigma \cup \pi_{new}$
            end
        end
    end
    return $\Sigma$
Function bestFirstSearch(C, A)
    $\pi_{init} \leftarrow$ initProductCFG(C, A)
    $\Omega \leftarrow$ expandProductCFG($\pi_{init}$, C, A)
    while $\pi_{cur} \leftarrow$ removeMostPromising($\Omega$) do
        $\Omega \leftarrow \Omega \cup$ expandProductCFG($\pi_{cur}$, C, A)
        if ProductCFGisProvableBisim($\pi_{cur}$) then
            return $\pi_{cur}$;
        end
    end
    return null
```

```
Function getFullPathset(s, P, $\mu$, v)
    fpset $\leftarrow$ { }
    foreach $o \in$ successors(s, P) do
        if (v[o] $\geq \mu$) then
            continue //skip this edge
        end
        //$\odot$ is the serial composition operator
        pth $\leftarrow$ ($s \rightarrow o$) $\odot$ getFullPathset(o, P, $\mu$, v[o]++)
        fpset $\leftarrow$ fpset $\cup$ { pth }
    end
    return makeSeriesParallelGraph(fpset)
Function getFullPathsetAtAllDeltas(s, t, P, $\mu$)
    v $\leftarrow$ { } //visited map (count of each visited node)
    paths $\leftarrow$ getFullPathset(s, P, $\mu$, v)
    return splitByNumOccurrencesOfLastPC(paths, t)
Function getNextPathsetRPO(($n_C, n_A$), A, $\pi$)
    foreach $h_A \in$ nexthops of $n_A$ in A in RPO do
        $\xi_A \leftarrow$ getFullPathsetAtAllDeltas($n_A$, $h_A$, A, 1)
        if notAlreadyCorrelated($\pi$, ($n_C, n_A$), $\xi_A$) then
            return
            eliminatePathsWithOtherAnchorNodes($\xi_A$)
        end
    end
    NotReached()
Function getCandCorrelations(n, C, $\xi_A$, $\mu_C$)
    $\kappa_C \leftarrow \{\epsilon\}$
    foreach all anchor nodes $w_C$ in C do
        S $\leftarrow$ getFullPathsetAtAllDeltas($n_C$, $w_C$, C, $\mu_C$)
        $\kappa_C \leftarrow \kappa_C \cup$ S
    end
    return cartesianProduct($\kappa_C$, {$\xi_A$})
```

Fig. 7. Pseudo-code of the algorithm.

already complete; in which case, we return to the caller which will eventually check if the inferred invariants ensure equivalent observable behavior (*ProductCFGisProvableBisim()*).

If there exists an "incomplete node" (i.e., a node which requires a new correlation for an outgoing A path), $n = (n_C, n_A)$, we identify the next pathset ξ_A starting at node n_A in A in RPO that has not yet been correlated. We enumerate all possible correlations in C for ξ_A starting at node n_C in a set of pathsets Ψ. For each of the pathsets $\xi_C \in \Psi$, we first check if the actions are compatible with ξ_A (*actionsAreCompatible()*), i.e., ξ_A should execute a function call iff ξ_C executes a function call, as an undefined function call could potentially produce an "observable action". If the action compatibility check passes, a new product-CFG edge is created which additionally includes a new product-CFG edge encoding the new candidate correlation between ξ_C and ξ_A. Counterexamples are propagated on the newly added edge (*addEdgeAndPropCEs()*) before applying our pruning criteria: *CurrentCEsSatisfyCorrelCriterion()* implements pruning based on paths taken by counterexamples, and *InvRelatesHeapsAtEachNode()* implements pruning based on heap relations. The newly-related product-CFG π_{new} is added to the frontier (Σ) only if both these subroutines return `true`.

The ***getCandCorrelations()*** subprocedure enumerates the potential correlations at node $n = (n_C, n_A)$ for pathset ξ_A in A by enumerating the outgoing pathsets from n_C in C (including the empty path ϵ). Similarly, ***getNextPathsetRPO()*** identifies the first nexthop anchor node, h_A, of n_A in A in RPO such that the full pathset from n_A to h_A (at unroll factor 1) has not already been correlated in π (*notAlreadyCorrelated()*). Both *getCandCorrelations()* and *getNextPathsetRPO()* employ the ***getFullPathsetAtAllDeltas()*** subroutine to enumerate the full pathset starting at node s and ending at node t in CFG P for a given unroll factor μ. *getNextPathsetRPO()* additionally eliminates those paths that are incident to other anchor nodes in A (*eliminatePathsWithOtherAnchorNodes()*). The *getFullPathsetAtAllDeltas()* subroutine returns $\texttt{FPsets}^{\mu}_{s \rightsquigarrow t}$ (set of pathsets) where each individual pathset represents $\text{FP}^{\mu,\delta}_{s \rightsquigarrow t}$, one for each value of δ such that $1 \le \delta \le \mu$. The helper function (***getFullPathset()***) enumerates the successor nodes of s in P (*successors()*), recursively calling itself for each successor node after updating the *visited map* v (by incrementing the count of node in v). The returned pathsets from the successor nodes are then composed in parallel and compacted at each step into a series-parallel digraph representation (*makeSeriesParallelGraph()*). Finally, the returned pathset is split into multiple pathsets such that each new pathset has an identical number of occurrences of t in all its constituent paths (*splitByNumOccurrencesOfLastPC()*).

For the ranking procedure to be effective, we would ideally prefer that the set of of counterexamples Γ_n has high coverage — e.g., the matrix formed by bitvector counterexamples as its rows should have a large number of linearly-independent rows. Thus, it is important that all counterexamples generated (or propagated) so far are considered before the ranking procedure is used. To implement this, we recompute the promise of each candidate correlation after every addition to the counterexample set. For example, if the call to *inferInvariantsAndCounterExamples()* changes the rank of π_{cur} (due to the newly inferred counterexamples), we discard it in favour of the new most promising correlation. For simplicity, this optimization has not been shown in the pseudocode in fig. 7.

4 EVALUATION

4.1 Implementation

For evaluation, we compare equivalence between unoptimized LLVM IR generated from the input C program and a corresponding optimized x86 assembly implementation (compiler-generated or manually programmed). We have implemented symbolic executors for both LLVM IR and x86, to convert the two representations to C and A CFGs. For the x86 executable, we use the symbol table and the relocation headers to identify the locations and (potentially offseted) accesses to global variables. The edge condition and transfer function for each CFG edge is encoded as an SMT expression involving bitvectors (for variables and registers), arrays (for memory states), and uninterpreted functions (for undefined procedure calls).

Before we run the *bestFirstSearch()* procedure in fig. 7 on the obtained CFGs, C and A, we run some static analyses on the individual CFGs:

(1) For A, we perform a forward dataflow analysis to identify all register values that are always at a constant offset from the input stack pointer at function entry. We classify such stack locations as *stack slots* and consider their values as bitvector variables during invariant inference and alias analysis.
(2) We run a forward points-to, intra-procedural, flow-sensitive, field-insensitive, untyped dataflow analysis to compute aliasing information for each program value at each program point in both C and A; the range of our points-to set includes each global variable (separately), a single stack region, and a single *allocated-heap* region[4]. Our sound and overapproximate points-to analysis algorithm is somewhat similar to the algorithm described in [Dahiya and Bansal 2017b; Debray et al. 1998] and involves identifying C's *based-on* relationships (§6.7.3.1 in [ISO 2011]) and tracking the flow of values. Our memory model for C is based on [Besson et al. 2014]'s proposal. We conservatively assume that the input arguments to a procedure can point to either the globals or the allocated-heap.

[4]We distinguish "allocated-heap" from our use of the word "heap" in the paper; the latter also includes global variables.

(3) We run a few standard dataflow analyses including available-expressions and liveness for both C and A; for available expressions, we encode the results as node invariants to supplement our invariant inference algorithm.
(4) We run a must-reach definitions analysis [Aho et al. 2006] on C and during invariant inference, consider only those SSA bitvector variables at a node n whose definition must reach n.

For the restricted class of programs we consider (e.g., no address-taken local variables), our points-to analysis is able to categorize every memory access as either accessing only the stack region, or definitely not accessing the stack region (i.e., accessing global(s) or allocated-heap). This allows us to model these two regions as separate arrays during discharge of proof obligations, just before transmitting the proof query to the SMT solvers.

For every SMT proof obligation generated by Counter, three off-the-shelf SMT solvers are spawned in parallel: `z3-4.8.7`, `Yices2-45e38fc`, and `cvc4-1.7`. For `unsat` results, we return as soon as the first solver finishes. For `sat` results, we opportunistically try and collect multiple counterexamples (to aid our counterexample-driven procedures): we wait for the first solver to finish; if the first solver finishes with a `sat` result in time t, then we wait till time $2*t$ and return the counterexamples generated by all solvers that finished in time $2*t$ (thus doubling our query times in the worst case). To avoid SMT solver timeouts, we employ the *query decomposition* technique from [Gupta et al. 2018]. In addition to improving efficiency and providing more counterexamples, employing multiple SMT solvers also improves the reliability of our verifier because it allows cross-checking of the results of one SMT solver against another. In fact, we found a bug in Yices during our experiments, which was fixed immediately upon reporting [yic 2020].

4.2 Experimental Setup

We evaluate Counter on a set of benchmarks involving extensive loop and vectorization optimizations, which include all the benchmarks used in the SPA paper [Churchill et al. 2019]. In addition to the examples in figs. 1 to 3, our evaluation involves two sets of benchmarks: the first set of benchmarks includes programs (C functions) from the TestSuite for Vectorizing Compilers (TSVC) [Maleki et al. 2011], and the second set of benchmarks is a set of 27 distinct vectorizable loop patterns that we have mostly taken from the LORE repository [Chen et al. 2017]. These programs usually operate on statically-allocated fixed size global arrays of integers. We compiled all programs using recent versions of production compilers, namely, GCC-8, Clang/LLVM-11, and ICC-18.0.3 with `-O3 -msse4.2` compiler flags to generate optimized x86 binaries. For our experimental evaluation with each compiler, we select only those functions that are vectorized by that compiler. For each of these compiler-function pairs, we attempt to prove equivalence across the unoptimized LLVM IR (generated by `clang -O0`) and assembly programs generated by an optimizing compiler. We use a global timeout of five hours for each function, an SMT-solver timeout of five minutes for each SMT proof query, and a memory limit of 12GB for a single equivalence check.

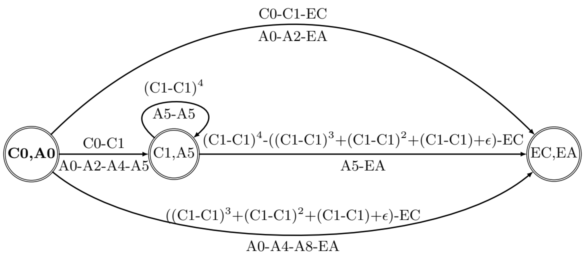
Both GCC and ICC perform loop unrolling with a maximum unroll factor of four, while LLVM uses an unroll factor of eight. Thus, we need $\mu_C = 8$ for GCC and ICC and we need $\mu_C = 16$ for LLVM. Notice that the required value of μ_C needs to be at least twice the unroll factor used by the compiler, to be able to handle the cool-down loops in the vectorized assembly programs. This can be seen using a simple vectorization example shown in fig. 8. In the assembly program A of this example, the path from the loop head to the program exit (`A5-EA`) involves four unrolled iterations of the original loop body followed by up to three residual iterations before reaching program exit. Thus, this path will be correlated with seven iterations of the original loop in C (which can be captured only at $\mu_C \geq 7$) when the compiler used an unroll factor of four. In general, we find that $\mu_C = 2 * \mu^o$ suffices where μ^o represents the unroll factor used by the optimizing compiler.

```
C0: void init1d(int n) {
C1:   for (int i = 0; i < n; i++)
C2:     a[i] = b[i];
C3: }
```

(a) C program

```
A0:  init1d:
A1:  r1 = 0
A2:  if (n <= 0) ret
A3:  r2 = n % 4, r3 = n - r2, r4 = 0
A4:  if (r3 == 0) goto A8
A5:    a[r4, .. r4+3] = b[r4, .. r4+3]
A6:    r4 += 4
A7:    if (r4 != r3) goto A5
A8:  if (r2 >= 1) { a[r4] = b[r4]; r4++ }
A9:  if (r2 >= 2) { a[r4] = b[r4]; r4++ }
A10: if (r2 == 3) { a[r4] = b[r4]; r4++ }
A11: ret
```

(b) (Abstracted) Assembly code


C0-C1-EC
A0-A2-EA
(C1-C1)^4
A5-A5
C0,A0
C0-C1
A0-A2-A4-A5
C1,A5
(C1-C1)^4-((C1-C1)^3+(C1-C1)^2+(C1-C1)+ϵ)-EC
A5-EA
EC,EA
((C1-C1)^3+(C1-C1)^2+(C1-C1)+ϵ)-EC
A0-A4-A8-EA

(c) Product-CFG

Fig. 8. C program, its abstracted assembly after loop unroll and vectorization, and the product-CFG

Table 2. Results for TSVC functions and LORE loop nest patterns.

		TSVC functions demonstrated by prior work			TSVC functions not demonstrated by prior work			LORE Loop Nests			
								All loops have memory write		At least 1 loop with no memory write	
		gcc	llvm	icc	gcc	llvm	icc	$\mu^o 4$	$\mu^o 8$	$\mu^o 4$	$\mu^o 8$
	Total/Failing functions	28/1	28/0	28/3	28/7	30/4	60/22	11/0	11/0	16/0	16/0
	Avg/Max ALOC	16/44	19/51	19/40	25/64	31/72	29/95	19/28	24/53	29/48	37/100
	Avg/Max # of product-CFG nodes	3/4	3.1/5	3.2/5	3.5/5	3.4/5	3.5/6	4.4/7	4.4/7	4.8/7	5/8
	Avg/Max # of product-CFG edges	3.2/7	3.3/7	3.6/9	4/7	3.9/8	4.2/11	5.2/9	5.2/9	5.9/9	6.8/18
	Avg # of total CEs / node	17	27	16	18	28	18	16	24	20	30
	Avg # of gen. CEs / node	13	22	11	12	22	13	10	17	10	16
BFS	Avg equivalence time (seconds)	209	70	15	201	3842	110	107	2243	131	676
	Avg # of paths enumerated	44	89	53	95	160	103	179	344	232	469
	Avg # of paths pruned	28	45	32	49	80	54	66	98	130	251
	Avg # of paths expanded	3.3	3.9	3.8	4.6	5.3	5.8	7.1	8.5	8.9	14.8
DFS	Memory/timeout reached	0	2	0	1	6	1	0	1	12	16
	Avg # of paths enumerated	173	3904	315	5776	14992	2635	301	561	17518	27727
	Avg # of paths expanded	35	252	52	518	913	262	111	208	4582	3781
	Avg # of paths expanded DFS/BFS	11	65	14	113	172	45	16	24	515	255

4.3 Results

Benchmark Categories. Table 2 tabulates the results of our experiments. We divide our benchmarks into three categories: (1) 28 TSVC functions that were a part of the benchmarks evaluated by [Churchill et al. 2019]; (2) TSVC functions for which Counter is the first to automatically generate equivalence proofs (they were not included in [Churchill et al. 2019] benchmarks); and (3) loop nest patterns taken from the LORE repository. For TSVC benchmarks, we present results for all three compilers. For LORE loop nests, we use one representative pattern for a set of structurally-similar program/transformation pairs, irrespective of the compiler that generated it. The space of additional transformations performed in this category (that are not covered by the first two categories) include loop splitting, loop fusion for bounded number of iterations, loop unswitching, and summarization of loop with small and constant bounds. The number of loops per function and maximum loop nesting depth varies between one and three for the loop patterns in this last category.

Table 3. List of passing vectorized TSVC functions. ✗denotes equivalence check failure for that function-compiler pair and ⊗ denotes that the function is not vectorized by that particular compiler.

TSVC functions demonstrated by prior work								TSVC functions not demonstrated by prior work							
Name	ALOC			Name	ALOC			Name	ALOC			Name	ALOC		
	gcc	llvm	icc		gcc	llvm	icc		gcc	llvm	icc		gcc	llvm	icc
s000	11	13	15	s243	21	51	✗	s111	28	⊗	⊗	s271	⊗	43	19
s1112	12	17	✗	s251	11	15	15	s1111	20	⊗	30	s2710	⊗	⊗	44
s112	22	8	12	s3251	44	50	39	s1115	⊗	✗	28	s2711	⊗	47	21
s121	18	32	20	s351	28	17	✗	s1119	✗	14	⊗	s2712	⊗	43	19
s122	17	17	24	s452	14	19	18	s113	20	⊗	23	s272	⊗	⊗	26
s1221	9	8	13	s453	11	13	15	s114	⊗	⊗	50	s273	⊗	53	25
s1251	13	12	17	sum1d	15	16	18	s116	⊗	17	⊗	s274	⊗	⊗	23
s127	17	18	23	vdotr	17	19	21	s1161	⊗	⊗	46	s276	⊗	⊗	29
s1281	17	15	21	vpv	9	11	13	s119	27	31	28	s293	⊗	⊗	13
s1351	9	11	13	vpvpv	10	13	14	s1213	⊗	⊗	37	s311	15	15	19
s162	43	37	40	vpvts	12	15	16	s124	18	24	20	s3111	19	20	24
s173	9	8	15	vpvtv	10	13	14	s125	24	20	25	s319	22	30	27
s176	✗	21	22	vtv	9	11	13	s1279	⊗	47	22	s352	⊗	22	⊗
s2244	24	47	28	vtvtv	10	13	14	s128	20	⊗	23	s4115	⊗	⊗	32
								s131	15	29	20	s421	25	48	29
								s132	28	43	26	s423	33	46	29
								s1421	24	25	40	s441	28	⊗	34
								s171	⊗	29	✗	s442	⊗	⊗	49
								s174	64	35	52	s443	17	⊗	25
								s2233	39	✗	✗	s471	28	26	✗
								s252	⊗	18	⊗	va	8	9	12
								s253	⊗	⊗	24	vbor	✗	✗	95
								s254	⊗	8	12	vif	⊗	72	17

Success and Failures. Among the first set of benchmarks (28 TSVC functions that were evaluated by [Churchill et al. 2019]), and across optimizations performed by the three compilers (for a total of 84 program pairs), Counter is able to compute equivalence proofs for all but four of these program pairs (see row `Failing functions` of table 2). Three of the remaining four program pairs — `s176 compiled with GCC`, `s1112`, and `s243 compiled with ICC` — involve non-bisimilar transformations, namely loop tiling and interchange, which are beyond the scope of Counter's algorithm. We confirm that the SPA algorithm is also unable to compute equivalence proofs across these three program pairs, and the reason these were reported as successful equivalence checks in their paper [Churchill et al. 2019] is because the authors used older compiler versions (which did not perform such transformations for these functions). One program pair (`s351` when compiled with ICC) involves loop re-rolling which is out of scope for Counter. The next table row shows the average and maximum Assembly Lines of Code (ALOC) in the optimized assembly program across all these functions. It is important to note that the C source code for all these 28 functions involve only a single loop and involve no control flow within their loop bodies.

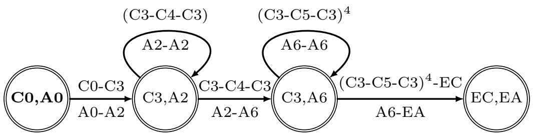
We next consider the remaining TSVC functions (our second set of benchmarks) and report only those program-pairs that involve some form of vectorization in their optimized assembly code. There are 28, 30, and 60 such program pairs (where vectorization was involved) for GCC, LLVM, and ICC respectively. Of these, Counter is able to compute equivalence for all but 7, 4, and 22 program-pairs respectively. The primary cause for equivalence failures is the presence of non-bisimilar transformations, namely loop interchange, fission, fusion, tiling, acceleration; an unbounded number of memory writes are reordered through such transformations and so these

```
C0: int loopSplitting() {
C1:   int sum = 0;
C2:   int mid = LEN/2;
C3:   for (int i = 0; i < LEN; i++) {
C4:     if (i < mid) sum += c[a[i]];
C5:     if (i >= mid) sum += b[i];
c6:   }
C7:   return sum;
C8: }
```

(a) C program.
LEN is a positive multiple of 4.

```
A0: loopSplitting:
A1:   r1 = 0; r2 = 0;
A2:     r2 += c[a[r1]]
A3:     r1++
A4:     if (r1 != mid) goto A2
A5:   r1 = &b[mid]; r3 = &b[LEN]; xmm0 = 0
A6:     xmm0 += *r1, .., *(r1+12)
A7:     r1 += 16
A8:     if (r1 != r3) goto A6
A9:   xmm0 += (xmm0 >> 8) // shift right by 8 bytes
A10:  xmm0 += (xmm0 >> 4) // shift right by 8 bytes
A11:  r2 += xmm0[31:0]
A12:  ret r2
```

(b) (Abstracted) Assembly code after loop splitting and unswitching



(c) Product-CFG

Fig. 9. C code before and after optimizations for an example loop nest

equivalences are difficult to establish through bisimulation relations. Only four of these failures (one each in GCC and LLVM, and two in ICC) are due to SMT solver timeouts during invariant inference. It is interesting to note that ICC is able to perform vectorization for a larger number of programs (60 vs. 30) and that most such vectorizations involve non-bisimilar transformations (20 failures). Importantly, compilations of these TSVC functions are among the most challenging program-pairs for equivalence checking. Among the function-compiler pairs for which Counter successfully computes equivalence for: (1) six functions contain more than one loop in function body; (2) six functions have nested loops of depth up to 2; (3) one function has both nested loops and multiple outer loops; (4) nineteen functions have control flow inside the loop body; (5) eight functions use multi-dimensional arrays potentially involving non-regular memory accesses; and (6) eleven functions have at least one loop without a memory write. Recall that correlation identification becomes easier if all program loops contain updates to the heap memory, because our pruning based on heap relations is then able to reduce the search space. Table 3 lists out the TSVC functions for which equivalence could be established by Counter.

Our third set of benchmarks include 16 different loop nest patterns. For each of these 16 patterns, we test two variations: one where the loop bodies involve a memory write, and another where at least one of the loop bodies does not involve a memory write. Among the 16 variations that involve a memory write in the loop bodies, the compilers produce non-bisimilar transformations for five of them. Thus we show results for 11 loop nest patterns where loop bodies have memory writes, and 16 loop nest patterns where the loop bodies don't have memory writes. Further, for each loop nest variation, we test across two different unroll factors ($\mu^o = 4$ and $\mu^o = 8$). The patterns with unroll factor 8 are due to compilations generated by LLVM or by GCC with the appropriate pragma switch. Counter is able to compute the required equivalence proof for all these (16+11)*2=54 program pairs. As an example of the complexity of transformations involved, fig. 9 shows the product-CFG generated by Counter for the program-pair involving multiple transformations including loop splitting, loop unswitching, unrolling, and vectorization. Most of these source programs (16 out of 27 total) have multiple loops with potential nesting (and different variables in each loop); we find that SPA's grammar is inadequate for guessing the alignment predicate in these cases. Moreover, 17 benchmark programs use multi-dimensional arrays which are out of scope for the SPA algorithm. Six benchmark programs have control flow inside the loop body for both source and generated assembly program, and it may be difficult and expensive to identify execution traces with adequate code coverage in such programs (as required by SPA).

Equivalence Checking Statistics. Table 2 also shows the average and maximum number of nodes and edges in the product-CFG generated by Counter for each benchmark category. Further, it shows the average number of counterexamples per final product-CFG node (`Avg # of total CEs/node`) and the average number of counterexamples that were generated (not propagated) per node through SMT queries (`Avg # of gen. CEs/node`). The next row (in category `BFS` which stands for best-first search) lists the average time taken to generate an equivalence proof in each category (`Avg equivalence time`). The next three rows demonstrate statistics for the best-first search (BFS) algorithm: we list the number of correlation possibilities that were created (`paths enumerated`) before the complete product-CFG was found, the number of correlation possibilities that were remaining after pruning (`paths pruned`) and the number of correlation possibilities which were actually expanded further (`paths expanded`). This last metric is a measure of the effectiveness of our ranking strategy: the table shows that the average number of paths expanded is small, and usually close to the average number of total product-CFG edges. Because each time a product-CFG correlation is expanded, we add an edge to the product-CFG: this confirms that in most cases, the correct correlation is ranked and picked first at each step of the backtracking search. In other words, our ranking strategy ensures that there is minimal backtracking, if any.

Comparison with a Static Strategy. The three rows labeled `DFS` (for depth-first search) demonstrate the results with a backtracking-based strategy relying on the static heuristic, where counterexample-guided pruning and ranking is omitted. This static backtracking strategy resembles the depth-first search approach, also used in [Dahiya and Bansal 2017a], as one part of the search tree is exhausted (depth-first) before another part of the subtree is attempted. We find that the average number of paths expanded in DFS is up to 515x more than the average number of paths expanded in BFS (last row in table 2); this is evidently due to the extra backtracking that occurs in the DFS strategy. In fact, the DFS strategy runs out of either time or memory resources for 39 of the 219 program pairs for which BFS is able to successfully establish equivalence (`Memory/timeout reached`). It is worth noting that these improvements produced by our pruning and ranking strategies are more pronounced in programs involving loops which do not update memory in their loop bodies. For loops that update memory in their bodies, the *InvRelatesHeapsAtEachNodes()* check (in fig. 7) allows early backtracking in situations where an incorrect correlation is chosen.

Comparison of SMT Solvers. Recall that Counter uses three off-the-shelf SMT solvers — Z3, Yices2, and CVC4 — that execute in parallel to discharge each SMT proof obligation. It is interesting to note that different SMT solvers exhibit significantly different behavior in our experiments: while Yices2 is usually much faster at discharging SMT queries (around 98% of queries are first answered by Yices2), the other two solvers actually produce "better" counterexamples (satisfying assignments) for our equivalence procedure.

To see this with an example: let's say at some node n in the partial product-CFG, there exist two *unconstrained* and *independent* 32-bit variables x and y. Also assume that we obtain $(0, 0)$ (short for `{x ↦ 0,y ↦ 0}`) as the first satisfying assignment (counterexample) through an incoming edge at n. The strongest invariant cover inferred by our algorithm for this counterexample will be $(x = 0) \wedge (y = 0)$. Now assume that the next query to the SMT solver generates an assignment that does not satisfy this inferred invariant, and let's say we get another counterexample $(0, 2^{31})$. With these two counterexamples obtained so far, the strongest invariant cover will now be weakened to $(x = 0) \wedge (2y = 0)$. Repeating this, let's assume that the third counterexample that we obtain through

```
C1: void *memccpy(void *dst, const void *src,
                  int c, size_t count) {
C2:   char *a = dst;
C3:   const char *b = src;
C4:   while (count--) {
C5:     *a++ = *b;
C6:     if (*b == c)
   // missing (unsigned char) type casts; (*b) will
   // get sign-extended before comparison and thus
   // may not be equal to c when sign bit is set
C7:         return (void *)a;
C8:     b++;
C9:   }
C10:   return 0;
C11: }
```

```
const char src[] = { 255, 128 };
char dst[2] = { 'A', 'B' };
memccpy(dst, src, 255, 2);
if (dst[1] != 'B')
  printf("BUG!")
```

Fig. 10. diet libc bug. Left: memccpy function. Right: Sample input for triggering the bug.

the SMT solver query is $(0, 2^{30})$; now, the new invariant cover would become $(x = 0) \wedge (4y = 0)$. This pattern of counterexamples where the satisfying assignments are successively decreasing powers of 2 can potentially go on — notice that for this pattern of counterexamples returned by the SMT solver, we would require 64 SMT queries before reaching the desired invariant, i.e., True (recall that x and y are unconstrained and independent). On the other hand, if the SMT solver had returned counterexamples $(0, 0)$, $(3, 5)$, and $(5, 7)$ in the first three SMT queries, we would have inferred the required invariant True within just three queries. Thus, the speed of our algorithm also depends on the "quality" of counterexamples returned by the SMT solver.

It turns out that Yices2 is more prone to the former behavior (returning counterexamples that involve decreasing powers of 2), even though it is faster in discharging the individual queries than Z3 and CVC4. This observation motivated our opportunistic counterexample collection scheme described in section 4.1, wherein we opportunistically try to collect counterexamples from multiple solvers for sat results.

Other Explorations. In addition to these benchmarks, we have applied Counter for verifying equivalence of several benchmarks including all the examples used in previous papers on equivalence checking [Churchill et al. 2019; Dahiya and Bansal 2017a; Kiefer et al. 2018]. We have also applied Counter to verify *libc* string functions implementations, and in one such experiment, we compared the OpenBSD [ope 2020] libc implementation against diet libc [die 2020]. Through this exercise, we uncovered three subtle and serious bugs in diet libc implementation, one of them shown in fig. 10; these bugs were acknowledged and fixed by diet libc developers immediately upon reporting. All of the three bugs were related to missing type casts in the C code. Surprisingly, these bugs had escaped years of testing and deployment.

4.4 Limitations

The Counter algorithm is not without limitations. The primary limitation arises from the assumption associated with Observation-D in section 1. Consider the example pair of programs in fig. 11. In this example, the two near-identical loops in C are transformed into a single loop in A. Here, the path (A1-A2) in A needs to be correlated with two distinct full pathsets in C: (C1-C2) and (C1-C4). Notice that this violates Observation-D because the two pathsets in C have different endpoints, C2 and C4. Because our algorithm only correlates a pathset in A with a single full pathset in C (section 3.7.3), it will be unable to identify the required product-CFG in this case. It is possible to relax this correlation condition in our algorithm and allow a pathset in A to correlate with multiple pathsets in C. Such choices should carefully balance the algorithm's common-case running

```
C1: if (a)                              A1:  mn = a ? m : n;
C2:   while (i < m) S;                  A2:  while (i < mn) S;
C3: else
C4:   while (i < n) S;
```

Fig. 11. Example program-pair where Counter will report false equivalence failure

time against its ability to handle these corner cases. We justify our chosen correlation criterion by observing that such corner cases are rare in practice.

Further, because our algorithm is only interested in identifying bisimulation relations, a whole class of transformations that do not preserve program-structure (*aka* non-bisimilar transformations) are out of scope for our algorithm. Some examples of non-bisimilar transformations are loop fusion, loop fission, loop interchange, and loop tiling.

Finally, Counter is unable to compute equivalence if the unrolling performed in the compiler transformation is out of range of the μ_C value considered in the algorithm. Larger μ_C values result in larger sets of possible correlations at each step, and thus potentially make the equivalence checking algorithm slower. As we discuss in our experiments, we find that $\mu_C = 16$ suffices for the transformations produced by GCC, LLVM, and ICC compilers. We must point out that Counter only unrolls paths in C through μ_C, and does not unroll paths in A — this may be inadequate for computing equivalence across certain types of loop re-rolling transformations.

During an equivalence check, most of the time is spent in discharging SMT proof obligations. We believe that future work towards making this proof effort more efficient would enable the equivalence checker scale to larger programs and across more complex transformations.

5 MORE RELATED WORK AND CONCLUSION

Translation validation across a selected set of transformation passes within a compiler has been previously demonstrated in different contexts [Barrett et al. 2005; Kanade et al. 2009; Kundu et al. 2009; Leung et al. 2015; Lopes and Monteiro 2016; Necula 2000; Poetzsch-Heffter and Gawkowski 2005; Stepp et al. 2011; Tate et al. 2009; Tristan et al. 2011; Zaks and Pnueli 2008; Zuck et al. 2003, 2005]. There have also been efforts aimed at various other applications of equivalence checking [Felsing et al. 2014; Lahiri et al. 2012; Strichman and Godlin 2008]. Our work fits in the category of recent efforts on comparing equivalence at the assembly level [Churchill et al. 2019; Dahiya and Bansal 2017a,b; Sharma et al. 2013], broadly categorized as the blackbox equivalence checking approach. Unlike [Sharma et al. 2013], we do not assume access to concrete execution traces on user-provided test inputs. Unlike [Dahiya and Bansal 2017a,b] which compares equality of edges for correlation, our correlation algorithm is more general because we use a one-way implication check on pathsets. Throughout the paper, we compare extensively with the semantic alignment paper [Churchill et al. 2019]. In section 4, we also compare with the depth-first search approach used in [Dahiya and Bansal 2017a]. Producing witnesses [Namjoshi and Zuck 2013] during optimization is another competing approach to a blackbox equivalence checking approach like ours; unlike a witness-based approach that places extra burden on the compiler developers, a blackbox approach is more automated. To our knowledge, Counter is the first algorithm that achieves robust and efficient equivalence checking across vectorizing transformations in the presence multiple loops in both programs with potentially distinct register allocations.

REFERENCES

2019. Polybench/C. https://sourceforge.net/projects/polybench/.
2020. [ONLINE-DEMO] Online demo of the equivalence checker. http://compiler.ai/.
2020. diet libc webpage. https://www.fefe.de/dietlibc/.

2020. OpenBSD libc sources. https://github.com/openbsd/src/tree/master/lib/libc.

2020. Yices2 bug report. https://github.com/SRI-CSL/yices2/issues/146.

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Clark Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore Zuck. 2005. TVOC: A Translation Validator for Optimizing Compilers. In *Computer Aided Verification*, Kousha Etessami and Sriram K. Rajamani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 291–295. https://doi.org/10.1007/11513988_29

Nick Benton. 2004. Simple Relational Correctness Proofs for Static Analyses and Program Transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL ¿04)*. Association for Computing Machinery, New York, NY, USA, 14¿25. https://doi.org/10.1145/964001.964003

Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2014. A Precise and Abstract Memory Model for C Using Symbolic Values. In *Programming Languages and Systems*, Jacques Garrigue (Ed.). Springer International Publishing, Cham, 449–468. https://doi.org/10.1007/978-3-319-12736-1_24

Z. Chen, Z. Gong, J. J. Szaday, D. C. Wong, D. Padua, A. Nicolau, A. V. Veidenbaum, N. Watkinson, Z. Sura, S. Maleki, J. Torrellas, and G. DeJong. 2017. LORE: A loop repository for the evaluation of compilers. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*. 219–228. https://doi.org/10.1109/IISWC.2017.8167779

Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic Program Alignment for Equivalence Checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 1027–1040. https://doi.org/10.1145/3314221.3314596

Manjeet Dahiya and Sorav Bansal. 2017a. Black-Box Equivalence Checking Across Compiler Optimizations. In *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings*. 127–147. https://doi.org/10.1007/978-3-319-71237-6_7

Manjeet Dahiya and Sorav Bansal. 2017b. Modeling Undefined Behaviour Semantics for Checking Equivalence Across Compiler Optimizations. In *Hardware and Software: Verification and Testing - 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings*. 19–34. https://doi.org/10.1007/978-3-319-70389-3_2

Saumya Debray, Robert Muth, and Matthew Weippert. 1998. Alias Analysis of Executable Code. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '98)*. ACM, New York, NY, USA, 12–24. https://doi.org/10.1145/268946.268948

Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating Regression Verification. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. ACM, New York, NY, USA, 349–360. https://doi.org/10.1145/2642937.2642987

Cormac Flanagan and K.RustanM. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *FME 2001: Formal Methods for Increasing Software Productivity*. Lecture Notes in Computer Science, Vol. 2021. Springer Berlin Heidelberg, 500–517. https://doi.org/10.1007/3-540-45251-6_29

Shubhani Gupta, Aseem Saxena, Anmol Mahajan, and Sorav Bansal. 2018. Effective Use of SMT Solvers for Program Equivalence Checking Through Invariant-Sketching and Query-Decomposition. In *Theory and Applications of Satisfiability Testing – SAT 2018*, Olaf Beyersdorff and Christoph M. Wintersteiger (Eds.). Springer International Publishing, Cham, 365–382. https://doi.org/10.1007/978-3-319-94144-8_22

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576¿580. https://doi.org/10.1145/363235.363259

ISO. 2011. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, Geneva, Switzerland. 683 (est.) pages. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853

Aditya Kanade, Amitabha Sanyal, and Uday P. Khedker. 2009. Validation of GCC Optimizers Through Trace Generation. *Softw. Pract. Exper.* 39, 6 (April 2009), 611–639. https://doi.org/10.1002/spe.v39:6

Moritz Kiefer, Vladimir Klebanov, and Mattias Ulbrich. 2018. Relational Program Reasoning Using Compiler IR. *J. Autom. Reason.* 60, 3 (March 2018), 337–363. https://doi.org/10.1007/s10817-017-9433-5

Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. 2009. Proving Optimizations Correct Using Parameterized Program Equivalence. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. ACM, New York, NY, USA, 327–337. https://doi.org/10.1145/1542476.1542513

Shuvendu Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebelo. 2012. SymDiff: A language-agnostic semantic diff tool for imperative programs. In *Computer Aided Verification (CAV '12) (Tool description)*. Springer. https://doi.org/10.1007/978-3-642-31424-7_54

Xavier Leroy. 2006. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM symposium on Principles of Programming Languages*. ACM Press, 42–54. https://doi.org/10.1145/1111037.1111042

A. Leung, D. Bounov, and S. Lerner. 2015. C-to-Verilog translation validation. In *Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE International Conference on*. 42–47. https://doi.org/10.1109/MEMCOD.2015.7340466

Nuno P. Lopes and José Monteiro. 2016. Automatic Equivalence Checking of Programs with Uninterpreted Functions and Integer Arithmetic. *Int. J. Softw. Tools Technol. Transf.* 18, 4 (Aug. 2016), 359–374. https://doi.org/10.1007/s10009-015-0366-1

Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. 2011. An Evaluation of Vectorizing Compilers. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT '11)*. IEEE Computer Society, Washington, DC, USA, 372–382. https://doi.org/10.1109/PACT.2011.68

Robin Milner. 1971. *An Algebraic Definition of Simulation Between Programs*. Technical Report. Stanford, CA, USA.

Markus Müller-Olm and Helmut Seidl. 2005. Analysis of Modular Arithmetic. In *Programming Languages and Systems*, Mooly Sagiv (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 46–60. https://doi.org/10.1145/1275497.1275504

KedarS. Namjoshi and LenoreD. Zuck. 2013. Witnessing Program Transformations. In *Static Analysis*, Francesco Logozzo and Manuel Fähndrich (Eds.). Lecture Notes in Computer Science, Vol. 7935. Springer Berlin Heidelberg, 304–323. https://doi.org/10.1007/978-3-642-38856-9_17

George C. Necula. 2000. Translation Validation for an Optimizing Compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. ACM, New York, NY, USA, 83–94. https://doi.org/10.1145/349299.349314

Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '98)*. Springer-Verlag, London, UK, UK, 151–166. https://doi.org/10.5555/646482.691453

Arnd Poetzsch-Heffter and Marek Gawkowski. 2005. Towards Proof Generating Compilers. *Electron. Notes Theor. Comput. Sci.* 132, 1 (May 2005), 37–51. https://doi.org/10.1016/j.entcs.2005.03.023

Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2013. Data-driven Equivalence Checking. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. ACM, New York, NY, USA, 391–406. https://doi.org/10.1145/2509136.2509509

Michael Stepp, Ross Tate, and Sorin Lerner. 2011. Equality-based Translation Validator for LLVM. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (Snowbird, UT) *(CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 737–742. https://doi.org/10.1007/978-3-642-22110-1_59

Ofer Strichman and Benny Godlin. 2008. Regression Verification - A Practical Way to Verify Programs. In *Verified Software: Theories, Tools, Experiments*, Bertrand Meyer and Jim Woodcock (Eds.). Lecture Notes in Computer Science, Vol. 4171. Springer Berlin Heidelberg, 496–501. https://doi.org/10.1007/978-3-540-69149-5_54

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: a New Approach to Optimization. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages* (Savannah, GA, USA). ACM, New York, NY, USA, 264–276. https://doi.org/10.1145/1480881.1480915

Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. 2011. Evaluating Value-graph Translation Validation for LLVM. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. ACM, New York, NY, USA, 295–305. https://doi.org/10.1145/1993498.1993533

Anna Zaks and Amir Pnueli. 2008. CoVaC: Compiler Validation by Program Analysis of the Cross-Product. In *Proceedings of the 15th International Symposium on Formal Methods* (Turku, Finland) *(FM '08)*. Springer-Verlag, Berlin, Heidelberg, 35–51. https://doi.org/10.1007/978-3-540-68237-0_5

Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. 2003. VOC: A Methodology for the Translation Validation of Optimizing Compilers. 9, 3 (mar 2003), 223–247. https://doi.org/10.3217/jucs-009-03-0223

Lenore Zuck, Amir Pnueli, Benjamin Goldberg, Clark Barrett, Yi Fang, and Ying Hu. 2005. Translation and Run-Time Validation of Loop Transformations. *Form. Methods Syst. Des.* 27, 3 (Nov. 2005), 335–360. https://doi.org/10.1007/s10703-005-3402-z