Inequivalence Checking across C Programs

Thesis submitted by

Jai Arora
2018CS50219

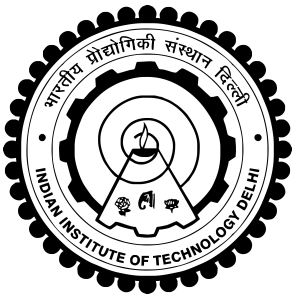
under the guidance of

Prof. Sorav Bansal

in partial fulfilment of the requirements
for the award of the degree of

Bachelor and Master of Technology

July 2023



Department Of Computer Science and Engineering
INDIAN INSTITUTE OF TECHNOLOGY DELHI

THESIS CERTIFICATE

This is to certify that the thesis titled **Inequivalence Checking across C-Programs**, submitted by **Jai Arora (2018CS50219)**, to the Indian Institute of Technology Delhi, for the award of **Bachelor and Master of Technology** in **Computer Science and Engineering**, is a record of bona fide work carried out by him under our supervision at the Department of Computer Science and Engineering, Indian Institute of Technology Delhi. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Sorav Bansal
Associate Professor
Dept. of Computer Science
Indian Institute of Technology
Delhi
New Delhi - 110016

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Sorav Bansal, who introduced me to research in Compilers and Verification. His wealth of knowledge, enthusiasm and perseverance are amazing and have been very motivating. I have much appreciated the independence he gave me in working on this project, while also giving me constant feedback on the areas where I needed it the most. Working with him and learning from him these past two years has been a wonderful and enriching experience.

Furthermore, I am grateful to my family for their unwavering support, understanding, and encouragement. Their love and belief in me have been the driving force behind my determination to overcome challenges along this academic journey.

I would like to thank Shubhani Gupta, Abhishek Rose and Indrajit Banerjee for generously sharing their time and expertise on various things. In particular, I would like to thank Shubhani for being extremely helpful and always making time to discuss things with me whenever I was facing technical challenges in my research.

Finally, I am grateful to all my friends for making this journey enjoyable. I am thankful for the wonderful memories we have made and shall cherish them.

Jai Arora

ABSTRACT

We have designed and implemented an automatic and sound approach for **Inequivalence Checking**, which is based on the Equivalence Checker `Counter`. Our approach first tries to prove the input programs to be equivalent, while collecting *inconsistencies* throughout the search. Once the equivalence checker terminates, we try to find input that would trigger these inconsistencies, which could also lead to observational inequivalence of the two programs. We designed a Data-Flow Analysis for the same, which is guaranteed to converge. We evaluate our approach on multiple C library functions, spanning different implementations. We found multiple bugs in these functions which were previously uncaught. We also find that our approach compares to the state of the art fuzzing techniques.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

CFG	Control Flow Graph
TFG	Tranfer Function Graph
UB	Undefined Behaviour
DFA	Data-Flow Analysis

NOTATION

$\vee$	Logical Or Operator
$\wedge$	Logical And Operator
`WP`	Weakest Precondition Function
μ	`unroll-factor` used by `Counter`
ν	`loop-bound`
k	`ce-bound`

Chapter 1

Introduction

An Equivalence checker is a proof finder. In contrast, an Inequivalence checker would be interested in identifying a distinguishing input that proves that the input programs are inequivalent. So, it's a bug finder.

Both Equivalence checking and Inequivalence checking are undecidable problems in general. A typical tool (assuming it is sound) would first try to find an equivalence proof. If found, we are done. Else, it would try and find a distinguishing input. If found, we are done. Else, we have neither found equivalence nor inequivalence, and we simply give up.

An Equivalence Checker, will either return with a *proof of equivalence* or just return *failure* (in which case we don't know if the programs are inequivalent or equivalent). With the Inequivalence Checker, we can get more information about the input programs.

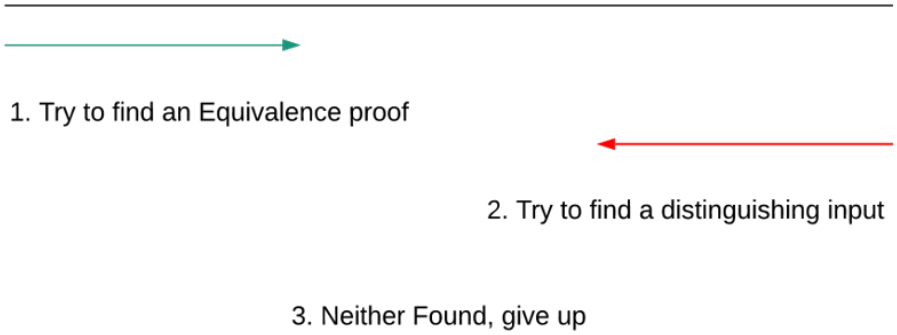


Figure 1.1: Equivalence and Inequivalence Checking

Following are two well-known problems where Inequivalence Checking can be used:

- Translation Validation: The problem of Equivalence Checking has a well-known and important application in Translation Validation. Because compilers are so complex and may have bugs, a Translation Validator checks the source code against the machine code for equivalence, and has the potential to compete with existing approaches to verified compilation such as `CompCert`.

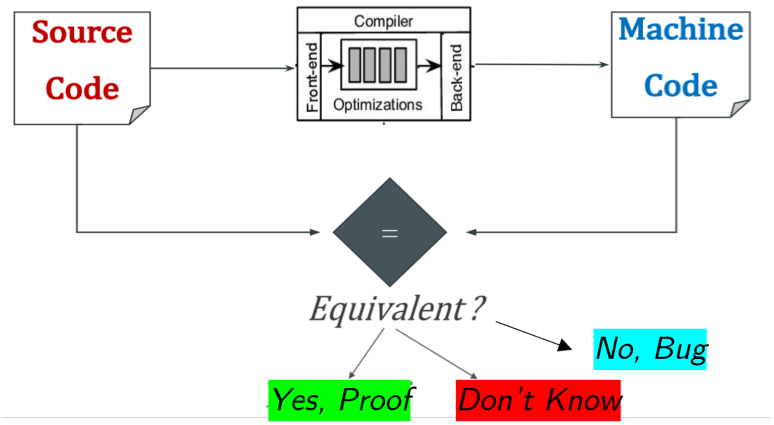


Figure 1.2: Translation Validation

On the other hand, an Inequivalence checker could be used in Translation Invalidation, where you try to detect a bug in the compilation.

- Differential Testing: Differential testing validates a set of systems with the same semantics, by comparing their output for a given input. As shown in the diagram, you pass the same input to a series of systems, and their outputs become oracles for each other.

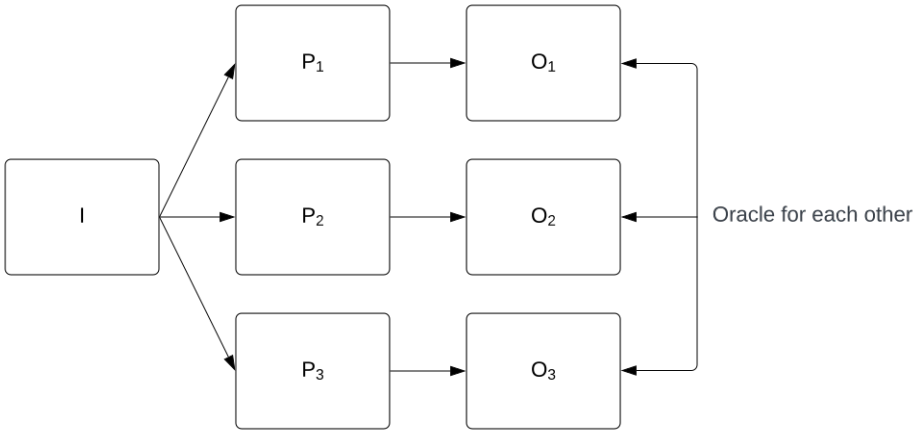


Figure 1.3: Differential Testing Setup

If there is an input I, for which $P_j(I) \neq P_k(I)$, then you have a bug. One could use the equivalence/inequivalence checker in this setting as well.

We propose an automatic and sound framework on top of the Equivalence Checker tool `Counter` [1], to check if the given two programs are Inequivalent.

Chapter 2

Background

2.1 Data-Flow Analysis

Code Optimization could either be *local* (code improvement within a basic block) or *global* (code improvement across basic blocks). Most global optimizations are based on Data-Flow Analyses [2], which are algorithms to gather information about the flow of data along program execution paths.

The results of data-flow analyses all have the same form: for each instruction in the program, they specify some property that must hold every time that instruction is executed. The analyses differ in the properties they compute. For example, a constant-propagation analysis computes, for each point in the program, and for each variable used by the program, whether that variable has a unique constant value at that point. This information may be used to replace variable references by constant values, for instance.

The aim of this section is to study a common framework for data-flow problems, abstractly. This will allow us to prove properties for an entire family of data-flow problems, once and for all. The framework also helps us identify the reusable components of the algorithm in our software design.

A Data-Flow Problem $(D, V, \otimes, F)$ is defined by:

- A direction D of the data-flow, which is either `FORWARD` or `BACKWARD`
- A Semilattice, which includes a domain of values V and a meet operator $\otimes$
- A family F of transfer functions from V to V

2.1.1 Semilattices

A *semilattice* $S = \langle V, \otimes \rangle$ consists of a set of values V and a binary meet operator $\otimes$ on V such that for all $x, y, z \in V$:

- $x \otimes x = x$ (`Idempotent`)
- $x \otimes y = y \otimes x$ (`Commutative`)
- $x \otimes (y \otimes z) = (x \otimes y) \otimes z$ (`Associative`)

A semilattice has a *top* element, denoted $\top$, such that,

$$\forall x \in V,\ \top \otimes x = x$$

Optionally, a semilattice may have a *bottom* element, denoted $\bot$, such that

$$\forall x \in V,\ \bot \otimes x = \bot$$

The Partial Order for a Semilattice

A relation $\leq$ is a *partial order* on a set V if for all $x, y, z \in V$,

- $x \leq x$ (`Reflexive`)
- $x \leq y$ and $y \leq x$ implies $x = y$ (`Antisymmetric`)
- $x \leq y$ and $y \leq z$ implies $x \leq z$ (`Transitive`)

The pair $(V, \leq)$ is called a *poset*. We can also define a $<$ relation for a poset as:

$$x < y \text{ if and only if } x \leq y \text{ and } x \neq y$$

It is useful to define a partial order $\leq$ for a semilattice $\langle V, \otimes \rangle$. For all $x, y \in V$, we define

$$x \leq y \text{ if and only if } x \otimes y = x$$

As the meet operator $\otimes$ is idempotent, commutative and associative, the defined partial order $\leq$ is reflexive, antisymmetric and transitive.

Height of a Semilattice

We may learn something about the rate of convergence of a data-flow analysis problem by studying the *height* of the associated semilattice. The *height* of a semilattice is the largest number of $<$ relations that would fit in an ascending chain $x_1 < x_2 < \cdots < x_n$.

Showing convergence of an iterative data-flow algorithm is much easier if the semilattice has finite height. Clearly, a lattice consisting of a finite set of values will have a finite height; it is also possible for a lattice with an infinite number of values to have a finite height.

2.1.2 Transfer Functions

The family of transfer functions F in a data-flow problem has the following properties:

- Every function $f \in F$ has the form $f : V \to V$
- F contains the identify function:

$$\exists f \in F,\ f(x) = x\ \forall x \in V$$

- F is closed under composition:

$$f_1, f_2 \in F \text{ implies that } f_1 \circ f_2 \in F$$

where $f_1 \circ f_2(x) = f_2(f_1(x))$

Monotone Frameworks

Recall that we require the semilattice $\langle V, \otimes \rangle$ to have a finite height to guarantee convergence of the analysis. We need an additional condition for the convergence guarantee -- monotonicity of the data-flow framework.

Formally, a framework $(D, V, \otimes, F)$ is monotone if and only if for all $x, y \in V$ and $f \in F$

$$x \leq y \text{ implies } f(x) \leq f(y)$$

Equivalently, a framework is monotone if and only if for all $x, y \in V$ and $f \in F$

$$f(x \otimes y) \leq f(x) \otimes f(y)$$

2.1.3 DFA Fixed-Point Iteration Algorithm

INPUT: A Data-Flow framework with the following components:

- A data-flow graph, with specially labeled `ENTRY` and `EXIT` nodes
- A direction of the data-flow D
- A set of values V
- A meet operator $\otimes$
- A set of functions F, where $f_B \in F$ is the transfer function for basic block B
- A constant value $v_{\texttt{ENTRY}}$ or $v_{\texttt{EXIT}}$, representing the boundary condition for forward and backward frameworks, respectively

OUTPUT: Values in V for $\texttt{IN}[B]$ and $\texttt{OUT}[B]$ for each block B in the data-flow graph

METHOD:

Iterative algorithm for a `FORWARD` Data-Flow problem:

```
OUT[ENTRY] = v_ENTRY
for (each basic block B other than ENTRY): OUT[B] = ⊤
while (changes to any OUT occur) {
   for (each basic block B other than ENTRY) {
      IN[B] = ⋀_{P∈pred(B)} OUT[P]
      OUT[B] = f_B(IN[B])
   }
}
```

Iterative algorithm for a `BACKWARD` Data-Flow problem:

```
IN[EXIT] = v_EXIT
for (each basic block B other than EXIT): IN[B] = ⊤
while (changes to any IN occur) {
   for (each basic block B other than EXIT) {
      OUT[B] = ⋀_{S∈succ(B)} IN[S]
      IN[B] = f_B(OUT[B])
   }
}
```

If the algorithm described above converges, then the result is a solution to the data-flow equations. Moreover, if the framework is monotone and the semilattice has a finite height, then the algorithm is guaranteed to converge.

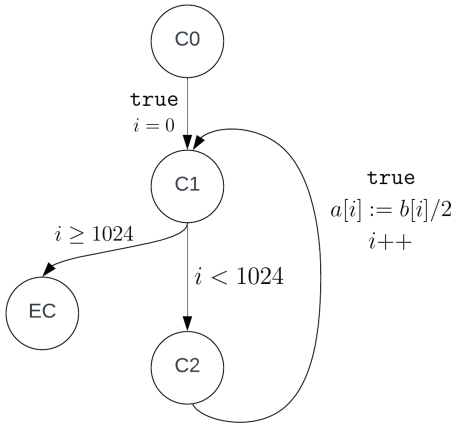
2.2 Control-Flow Graph Representation

This section presents the details for the abstract framework named as Control-Flow Graph (CFG) used for program representation in this thesis.

CFG is a directed graph with nodes and edges. Each node in the CFG representation of a program corresponds to a program location or program counter (PC) and is denoted by the symbol n. Each edge in the CFG corresponds to transition from one program location to another and is denoted using $\omega[n \rightarrow n']$ from node n to node n'. Each edge (representing a transition) is labeled with an edge-condition that must be true to trigger that transition, a transfer function that specifies how the (abstract) machine's state is modified across that transition, and an action that indicates the program's potential interactions with the environment (e.g., program exit, unspecified procedure call). The CFG for a program has a start node ($n_{\texttt{START}}$) at which it begins execution, and a (potentially empty) set of exit nodes.

```
int a[1024], b[1024];
C0: void divP() {
C1:     for (int i = 0; i < 1024; ++i) {
C2:         a[i] = b[i] / 2;
C3:     }
EC: }
```

(a) C Program



(b) C Program CFG. C0 is the entry node and EC is the exit node

Figure 2.1: An Example C-Language Program and its Control Flow Graph Representation

Figure 2.1 shows an example C-language program and its control-flow graph representation. The example program sets the elements in the global array a to half of the value of elements in the global array b. In the CFG representation shown in fig. 2.1b, the edge conditions and the transfer function are shown for each edge. As an example, the edge condition for the edge (C2 $\rightarrow$ C4) is ($i < 1024$) and the transfer function for the edge (C4 $\rightarrow$ C2) is (i++). In this example CFG, there is a single exit edge (C2 $\rightarrow$ EC) and the action α associated with this edge returns the memory region for global variables a and b. The labels (C0), (C2), (EC), etc., in the C-program denote the program locations or PCs and are used to represent the nodes in the CFG. The CFG representation used in this work is similar to the Transfer Function Graph (TFG) representation used by prior work [1].

2.3 Counter: A Black-Box Equivalence Checker

This section presents the details for Counter [1], which is a counterexample-guided correlation algorithm for Translation Validation. Our proposed approach builds on top of Counter.

Translation validation (TV) 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. 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. 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 or as a product program, both of which are equivalent representations.

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 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).

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. `Counter` is a counterexample-guided algorithm, which efficiently searches this space of potential product-CFGs to yield a provable bisimulation relation. It uses an incremental approach for constructing the required product-CFG, where an edge is added at each step to the partial product-CFG constructed so far. It is based on a best-first search procedure that uses counterexample-guided pruning to reduce the search space of candidate product-CFGs and counterexample-guided ranking to prioritize remaining correlation candidates. It also consists of a scalable yet static invariant inference algorithm, *Sifer*, to infer precise and expressive invariants between programs that have large syntactic gap across them. It is implemented as a data-flow analysis and thus is incremental in itself.

Counter 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.

Chapter 3

Inequivalence Checker

In this chapter, we describe our proposed automatic and sound Inequivalence Checking procedure, built on top of the Equivalence Checking tool `Counter`. This is implemented as a part of the `eq32` tool.

Currently, our approach can handle `C`-to-`C` and `C`-to-`Assembly` Inequivalence Checking.

3.1 Overview

When given a source (denoted as *src*) and a destination (denoted as *dst*) program, `eq32` first converts them into their respective CFGs and then uses `Counter` to incrementally construct a Product CFG. We first try to prove that the programs are equivalent.

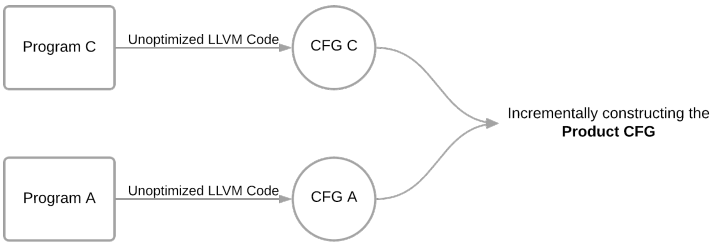


Figure 3.1: Inequivalence Setup

`Counter` uses a backtracking based best-first search to try to find a Complete Product-CFG. An example snapshot of the search tree is shown below. 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.

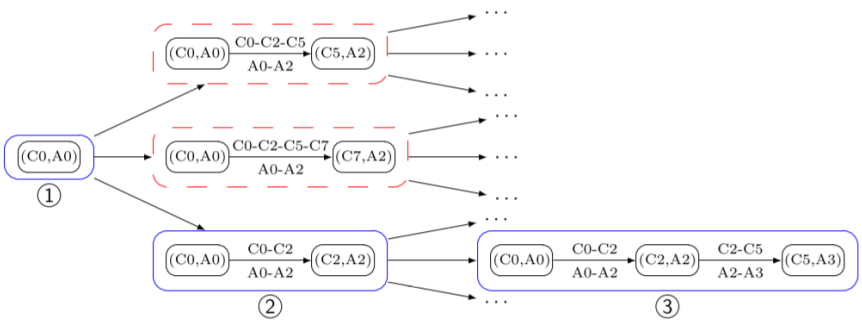


Figure 3.2: A snapshot of the backtracking search tree

During this search, we try to find CFGs where there is an *inconsistency*. For instance, the return values being unequal at the exit node could be an *inconsistency*. More inconsistencies are described in Section 3.2. For inconsistency that we collect, we use a data-flow analysis to *propagate* it backwards. In the end, we can construct an inequivalence condition at the start-*PCpair*. If the condition is satisfiable, then we would have found a *potential* bug.

The process is described in more detail in Sections 3.2-3.8.

3.2 Collecting Inconsistencies

As discussed in Section 3.1, `eq32` takes the input programs and converts them into their CFGs. The CFGs of the individual programs are referred to as Transfer Functions Graphs (TFGs). We then try to incrementally construct a product-CFG in an attempt to prove equivalence.

At each step of the backtracking search, we identify a candidate correlation. Let the correlation by represented by an outgoing product-CFG edge $\omega = n \to n^d = (\xi_{\texttt{src}}, \xi_{\texttt{dst}})$. We try to figure out if the correlation is *inconsistent*. We will call these inconsistencies as *failconds* (short for *failure conditions*), and the node as *failcond-pc*. In the following subsections, we discuss the different kinds of failure conditions.

Exit Node Inconsistencies

This is applicable when the candidate product-CFG has an exit *PCpair* (a node correlating the exit *PC*s of the input TFGs). If the inferred invariants at the exit node are not strong enough to prove equivalent observable behaviour, then we have an *inconsistency*. The following cases may arise:

- Return values are unequal: If the inferred invariants do not ensure that the return values are equal, then our *failcond* is $\texttt{ret}_{\texttt{src}} \neq \texttt{ret}_{\texttt{dst}}$
- Heap states are unequal: If the inferred invariants do not ensure that the heap states at exit are equal, then our *failcond* is $\texttt{H}_{\texttt{src}} \neq \texttt{H}_{\texttt{dst}}$

In both these cases, our *failcond-pc* will be the exit node.

Heap Inconsistencies

If the heap states $\texttt{H}_{\texttt{src}}$ and $\texttt{H}_{\texttt{dst}}$ are not related by the invariants at the node n^d, then `Counter` eliminates that 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 node, then there is little hope for them to be correlated at exit.

We consider this as an *inconsistency*. Our *failcond* would be $\texttt{H}_{\texttt{src}} \neq \texttt{H}_{\texttt{dst}}$ and *failcond-pc* would be n^d.

Correlation Inconsistencies

By definition, if an edge $\omega = (\xi_{\texttt{src}}, \xi_{\texttt{dst}})$ is traversed in the product-CFG, it implies that one of the paths in $\xi_{\texttt{src}}$ is traversed in `src` program and one of the paths in $\xi_{\texttt{dst}}$ is traversed in `dst` program.

While this property defines the general space of potential correlations, `Counter` restricts this space further through a correlation criterion to achieve better tractability. It restricts correlations by requiring that $\xi_{\texttt{src}}$ can be correlated with $\xi_{\texttt{dst}}$ through a product-CFG edge $\omega = (\xi_{\texttt{src}}, \xi_{\texttt{dst}})$ only if the following property holds: if any of the paths in $\xi_{\texttt{dst}}$ is traversed in `dst` program, then one of the paths in $\xi_{\texttt{src}}$ in `src` program must be traversed.

This correlation criteria is checked for all the edges in the product-CFG. For an edge $\omega = n \rightarrow n^d = (\xi_{\texttt{src}}, \xi_{\texttt{dst}})$, we check if the following condition holds,

$$\texttt{INV}_n \Rightarrow (pscond_{\xi_{\texttt{dst}}} \rightarrow pscond_{\xi_{\texttt{src}}})$$

where $pscond_\xi$ represents the path-set condition of ξ and $\texttt{INV}_n$ represents the inferred invariants at node n. If this check fails, then this condition becomes our *failcond* and the *failcond-pc* would be n.

We may encounter a lot of such *failconds* in the search, so we store the inconsistent product-CFG and the *failcond* whenever we encounter one. Note that in the last two cases, the product-CFG would be incomplete. We would like to determine if this inconsistency is due to the imprecision of our equivalence checking or because the programs are inequivalent.

3.3 Propagating the Inconsistencies

Now we will assume that the equivalence checker has finished its search, and we have collected some *failconds*. We would like to determine if these inconsistencies are due to the imprecision of our equivalence checking or because the programs are inequivalent.

One way is to compute the weakest predicate P_n at each node n that is stronger than the weakest precondition of the *failcond*. In other words, if P_n is satisfied at node n, then the

failcond would definitely get triggered in the product-CFG. If $P_{\texttt{START}}$ (i.e. the predicate at the start node) is non-empty, then we have a *potential* bug.

We will see the working of this approach with a small example. Consider the following acyclic input programs:

$$\begin{aligned} \texttt{src} &\stackrel{df}{=} \lambda x{:}int[3 * x] \\ \texttt{dst} &\stackrel{df}{=} \lambda x'{:}int[x' + 10] \end{aligned} \tag{3.1}$$

Clearly the programs are inequivalent. The product-CFG for these programs would be a single-edge CFG as shown in Figure-3.3. The *failcond* in consideration will be $x \neq x'$, since we won't be able to ensure that the return values are equal. The *failcond-pc* will be (EC, EA), the exit $PCpair$.

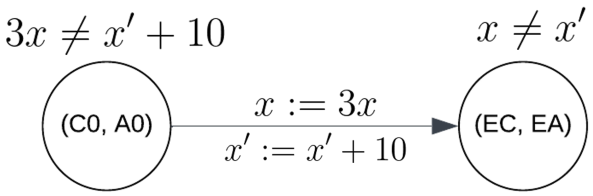


Figure 3.3: Product-CFG for the programs shown in equation 3.1

We then propagate the inconsistency backwards by simply computing the weakest precondition along the product-CFG edge, and we get the condition $3x \neq x' + 10$. Now, to check if it is satisfiable, we will add an extra condition that the inputs were equal at the start $(x = x')$.

$$3x \neq x' + 10 \bigwedge x = x' \Rightarrow 3x \neq x + 10 \Rightarrow x \neq 5$$

We get $x \neq 5$ as a sufficient condition for inequivalence, and it is easy to check that any value of x satisfying this condition will trigger a bug for these two programs. This intuition will form the basic idea of our analysis discussed in the next section. Since the input programs may have loops, the product-CFG may have loops in it, and it is not easy to simply compute the weakest precondition across a loop.

3.4 Data-Flow Analysis for Inequivalence Checking

To apply the idea discussed in Section 3.3 to general programs, we design a backwards data-flow framework (discussed in section 2.1) which uses the Kildall's worklist algorithm to compute the maximum fixed-point (MFP) of the data-flow formulation. The worklist

algorithm is an optimized algorithm as compared to the naive iterative algorithm for solving the data-flow formulation for a given CFG. It is based on the premise that input data-flow value at a CFG node is directly determined by the output values at its predecessor nodes and will remain same if the output value of any of the predecessors has not changed. Therefore, instead of re-computing the values for all nodes at each iterative step, the Kildall's worklist algorithm maintains a list of nodes to be processed as a worklist. The algorithm initializes the worklist with the start nodes (or exit nodes in case of a backward data-flow analysis). In each iteration, a node is removed from the worklist and the output value is computed for that node. If the newly computed output value is different from the previous output value for that node, its successors are added to the worklist. For efficiency, a node should not be present in the worklist more than once. We discuss our framework in the following subsections.

3.4.1 Notation

A desired property of the data-flow framework is convergence, the conditions for which are discussed in Section-2.1. For convergence, we need the paths in our analysis to have a bounded length. Each straight-line path is represented as a list of edges. Let ν be a parameter. Our set of representable paths will be parameterised by ν. Consider the set $\mathcal{P}_\nu$ defined as:

$$\mathcal{P}_\nu \stackrel{df}{=} \{\text{paths with frequency of each edge } \leq \nu\} \tag{3.2}$$

Note that the size of this set is finite. We denote this parameter ν by `loop-bound`. To give an example, the path $e_1 \cdot e_2 \cdot e_1$ lies in $\mathcal{P}_2$, while the path $e_1 \cdot e_2 \cdot e_1 \cdot e_1$ does not belong to set the $\mathcal{P}_2$, since the frequency of the edge e_1 is 3.

Let f be the *failcond* in consideration. Define a set of expressions $\mathcal{E}$ by the following grammar:

$$\mathcal{E} \rightarrow \bot \mid \texttt{WP}(f, p),\ p \in \mathcal{P}_\nu \mid e_1 \vee e_2 \tag{3.3}$$

The size of $\mathcal{E}$ is also finite, although it is exponential in $|\mathcal{P}_\nu|$. Each expression in $\mathcal{E}$ can be represented as a finite disjunction of weakest preconditions of f across some paths in $\mathcal{P}_\nu$. So, we can represent each expression by a set of paths in $\mathcal{P}_\nu$. The representation is defined inductively as follows:

$$\begin{aligned} \bot &\equiv \{\} \\ f &\equiv \{\epsilon\} \\ \texttt{WP}(f, p) &\equiv \{p\} \\ e_1 \vee e_2 &\equiv S_1 \cup S_2,\ \text{where } e_1 \equiv S_1,\ e_2 \equiv S_2 \end{aligned}$$

Note that the sets $\mathcal{P}_\nu$ and $\mathcal{E}$ are dependent on the product-CFG, the parameter ν and the

failcond f. So we will assume that they are fixed.

3.4.2 Domain of DFA values

The values computed through the DFA are represented by a tuple (p, e), where $p \in \mathcal{P}_\nu$ and $e \in \mathcal{E}$. The semantics of these values are:

- If we have a value (p, e) at a node n, then if we take the path p and e holds at the *end* of the path, then the *failcond* f will definitely get triggered
- The above point implies that the path p should start at the node n

We will maintain these semantics as *invariants* throughout our analysis. The set of values V for our framework is defined below. The set contains a special value $\top$ and the number of elements in this set is again finite.

$$V = \{(p, e) \mid p \in \mathcal{P}_\nu,\ e \in \mathcal{E}\} \cup \{\top\}$$

3.4.3 Initialization of DFA Values

The direction of our DFA is `BACKWARDS`. The boundary condition initializes the value at the *failcond-pc* to the pair (ϵ, f), where ϵ denotes the empty path. This is consistent with the semantics discussed in Section-3.4.2. For each node n other than the *failcond-pc*, we initialize its value to $\top$, since we do not have any information at the other nodes.

3.4.4 Meet Operator

Consider two values (p_1, e_1) and (p_2, e_2) at node n. We can infer the following from their semantics:

- p_1 starts at n and if we take the path p_1 and e_1 holds at the end of p_1, then f will definitely get triggered
- p_2 starts at n and if we take the path p_2 and e_2 holds at the end of p_2, then f will definitely get triggered

We want to combine these two values using the meet operator $\otimes$, so that it is consistent with the value semantics. Without loss of generality, assume that p_1 and p_2 have the following structure.

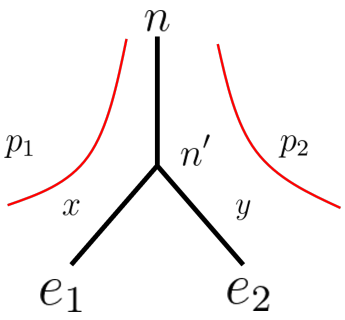


Figure 3.4: The paths p_1 and p_2 have some common prefix, after which they diverge

Given this structure, the most intuitive path in the value after meet would be the longest common prefix of p_1 and p_2. To make sure we maintain the value semantics, we need some expression at the intersection node n', which would be $\texttt{WP}(e_1, x) \vee \texttt{WP}(e_2, y)$.

Note that we also need to make sure that $\texttt{WP}(e_1, x) \vee \texttt{WP}(e_2, y) \in \mathcal{E}$. Let $e_1 \equiv S_1$ and $e_2 \equiv S_2$, then

$$\texttt{WP}(e_1, x) \vee \texttt{WP}(e_2, y) \equiv (x \cdot S_1) \cup (y \cdot S_2) \tag{3.4}$$

where $p \cdot S$ represents all paths of S prefixed with p. In equation-3.4, some paths may lie outside $\mathcal{P}_\nu$ due to the prefix operation. Define a function $\texttt{WP}_\nu$ inductively as follows:

$$\texttt{WP}_\nu(\bot, p) = \texttt{WP}(\bot, p)$$

$$\texttt{WP}_\nu(\texttt{WP}(f, q), p) = \begin{cases} \texttt{WP}(f, p \cdot q) & p \cdot q \in \mathcal{P}_\nu \\ \bot & p \cdot q \notin \mathcal{P}_\nu \end{cases}$$

$$\texttt{WP}_\nu(e_1 \vee e_2, p) = \texttt{WP}_\nu(e_1, p) \vee \texttt{WP}_\nu(e_2, p)$$

This functions filters the paths that lie outside $\mathcal{P}_\nu$. We can now define the meet operator as follows:

$$(p_1, e_1) \otimes (p_2, e_2) \stackrel{df}{=} (\texttt{LCP}(p_1, p_2),\ \texttt{WP}_\nu(e_1, x) \vee \texttt{WP}_\nu(e_2, y)) \tag{3.5}$$

where $\texttt{LCP}(p_1, p_2)$ denotes the longest common prefix of p_1 and p_2.

3.4.5 Transfer Function

The transfer function f_ω for a path ω in the CFG is described here. Let (p, e) be the value that needs to be transferred. Consider the following two cases:

- $\omega \cdot p \in \mathcal{P}_\nu$: in this case, we can simply return $(w \cdot p, e)$ and the semantics are still maintained
- $\omega \cdot p \notin \mathcal{P}_\nu$: in this case, we cannot return $(w \cdot p, e)$ because this value does not lie in the DFA domain.

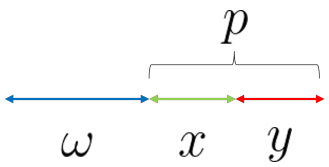


Figure 3.5: Divide p into two parts such that $\omega \cdot x \in \mathcal{P}_\nu$

Let $\omega \cdot x$ be the maximal prefix of $\omega \cdot p$ such that $\omega \cdot x \in \mathcal{P}_\nu$. We return this path and then shift the expression to some other node. The returned value is $(\omega \cdot x, \texttt{WP}_\nu(e, y))$

The transfer function f_ω is summarized below.

$$f_\omega((p,e)) = \begin{cases} (\omega \cdot p, e) & \omega \cdot p \in \mathcal{P}_\nu \\ (\omega \cdot x, \texttt{WP}(e,y)) & \omega \cdot p \notin \mathcal{P}_\nu, \text{ where } p = x \cdot y \text{ and } \omega \cdot x \text{ is the} \\ & \qquad \text{maximal prefix of } \omega \cdot p \text{ which lies in } \mathcal{P}_\nu \end{cases} \tag{3.6}$$

3.4.6 Characteristics of the Algorithm

The overall DFA Formulation is shown below.

Domain	$\{(p,e) \mid p \in \mathcal{P}_\nu,\ e \in \mathcal{E}\} \cup \{\top\}$
Direction	`BACKWARDS`
Boundary Condition	$\texttt{in}[\textit{failcond-pc}] = \{(\epsilon, \textit{failcond})\}$
Initialization to Top	$\texttt{in}[n] = \top$
Transfer Function	f_ω as specified in Equation-3.6
Meet Operator $\otimes$	As specified in Equation-3.5

Table 3.1: Data-Flow Formulation for Inequivalence Checking

The meet operator described in Section-3.4.4 is idempotent, commutative and associative. The semilattice $\langle V, \otimes \rangle$ has a finite height and the family of transfer function described in Equation-3.6 is monotonic. These properties ensure that our DFA Framework will converge.

3.5 Getting Distinguishing Inputs

After our Data-Flow Analysis has converged, let (p, e) denote the value computed at the start *PCpair*. Then using the DFA value semantics, if we take the path p and e is true at the end of the path, then the *failcond* f will definitely get triggered. So, our inequivalence condition would be $\texttt{WP}(e, p)$. We also need to assert at the start that both the programs

have equal input arguments and equal input heap states. Our final inequivalence condition becomes:

$$\texttt{WP}(e, p) \wedge \texttt{arg}_{\texttt{src}} = \texttt{arg}_{\texttt{dst}} \wedge \texttt{H}_{\texttt{src}} = \texttt{H}_{\texttt{dst}}$$

The condition is then encoded as an SMT condition and sent to three off-the-shelf SMT solvers spawned in parallel -- `z3`, `Yices2` and `cvc4`. For `unsat` results, we return as soon as the first solver finishes. For `sat` results, we opportunistically try and collect multiple counterexamples.

Note: The value (p, e) at the start *PCpair* is equivalent to $\texttt{WP}(e, p)$. Let $e \equiv S$, then $\texttt{WP}(e, p) \equiv p \cdot S$. So, we have essentially collected the set of paths $p \cdot S$ and compute the weakest precondition of f across each of the paths in $p \cdot S$.

3.6 Validating the Counterexamples

As discussed in Section-3.2, our product-CFG would be incomplete if the *failcond* arises from a heap inconsistency or a correlation failure, in which case the *failcond-pc* will not be the exit *PCpair*. By soundness of our Data-Flow framework, any counterexamples that we get will definitely trigger the failure condition f. But f may not imply observational inequivalence, so we need to verify the counterexamples that we get. So, we translate the counterexample across both of the input programs and check if they lead to unequal return values or unequal heap states at the end.

The execution traces of the counterexamples triggering these *failconds* could be really large, so the translation could take a lot of time. So, as a heuristic, we bound the magnitude of all inputs. More specifically, for each input argument a, we put the following constraint:

$$\texttt{lshr}(a, \texttt{DWORD\_LEN - k}) \;==\; 0$$

This constrains the top k bits of a to be zero. We denote this parameter k by `ce-bound`. This also enforces signed integer values to be always positive. Note that this is only a heuristic and does not guarantee that the counterexamples would have finite execution traces after this. For instance, `strlen(const char* s)` takes a pointer to a string as input. If we restrict the top-k bits of `s` to 0, then only the magnitude of the pointer address gets bounded. The actual runtime of the function depends on the characters in the string pointed to by `s`.

3.7 Ranking the product-CFGs

We may encounter a lot of inconsistencies while incrementally constructing the product-CFG. We collect the inconsistent product-CFGs and the respective *failconds* and then run

the Data-Flow Framework (Section 3.4-3.6) on each of those product-CFGs. Since the number of failed-CFGs could be exponential in the size of the input programs, we need to rank them based on some heuristic so that we try out the more promising candidates first. We use the following insights for our ranking heuristic:

- Correlation Failures: if our *failcond* arises from a correlation inconsistency as discussed in Section-3.2, then we rank them lower than the other inconsistencies. The premise for this is that we are less likely to get distinguishing inputs from correlation failures as compared to the ones which directly correspond to observational inequivalence
- Observing the best-first search tree pattern: the basic idea is that for a particular failed-CFG g, the higher the relative height of the next candidate, the better its rank would be.

  In other words, if we have come far down in the search tree and then we fail and move to a higher location, then this is a good candidate for inequivalence checking. Any candidate that is good for equivalence checking, is also good for inequivalence checking.

  $$\texttt{rank}(g) \propto \texttt{depth(g)} - \texttt{depth(next\_candidate(g))}$$

  In the example shown below, `root.A3.B1.C1` would be better ranked than `root.A1` as its successor is higher.

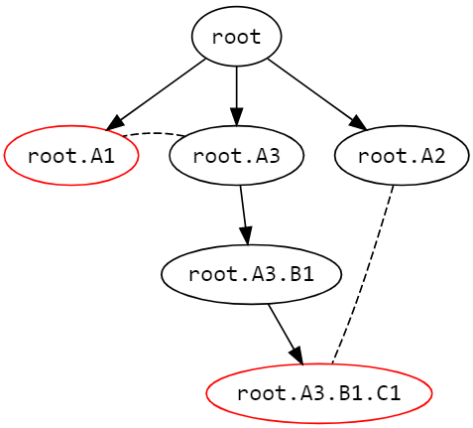


Figure 3.6: An example search tree indicating the failed-CFGs and the next candidates

3.8 Putting all components together

Using the algorithms discussed in Section 3.2-3.7, we now combine them into one procedure as follows:

Algorithm 1: Inequivalence Checking Algorithm based on `Counter`

```
1  Function ineqChecking(src, dst, μ, ν_max):
2      π, Ω ← bestFirstSearch(src, dst, μ);
3      if π ≠ null then
4          return EQUIV, {};
5      end
6      heuristic_sort(Ω);
7      ν ← 1;
8      while ν ≤ ν_max do
9          foreach cg ∈ Ω do
10             Γ_ce ← computeIneq(cg, ν);
11             if Γ_ce ≠ φ and valid(Γ_ce) then
12                 return INEQ, Γ_ce;
13             end
14         end
15         ν ← ν * 2;
16     end
17     return FAIL, {};
```

The inputs to the top-level procedure `ineqChecking()` are input programs *src* and *dst*, the unroll-factor μ as used by `Counter` and the maximum value of loop-bound $\nu_{\texttt{max}}$. It returns two values, the first being the result returned by the tool (one of `EQUIV`, `INEQ` or `FAIL`) and the second being a set of distinguishing inputs for the two programs.

We first call `bestFirstSearch()`, the top level procedure of `Counter` to check for equivalence. If the procedure returns a non-empty, complete product-CFG, then we return `EQUIV`. If not, then the product-CFG would be empty and we would have a set of failed-CFGs, Ω. We then sort Ω by the ranking heuristic described in Section-3.7.

The algorithm then needs to run the DFA on each of the failed-CFGs. Choosing an appropriate value of ν could be tricky, so we first start from a low value (1 in this case), and double it in each iteration till we reach $\nu_{\texttt{max}}$. `computeIneq()` then runs the data-flow analysis for the given *cg* and the loop bound ν, and returns a set of counterexamples using the algorithm described in Section-3.5. If the set of counterexamples is non-empty and valid (checked using Section-3.6), then we return `INEQ`. If we do not find any valid counterexamples from any failed-CFG, then we return `FAIL`.

Chapter 4

Evaluation

4.1 Experimental Setup and Benchmark Selection

We evaluate our inequivalence checking approach on functions from 9 different C library implementations which are listed here:

- *glibc v2.37* [3]: the GNU C Library
- *dietlibc v0.34* [4]: an alternative small implementation of the C standard library (MMU-less)
- *BSDlibc*: various implementations distributed with BSD-derived operating systems such as *FreeBSD* [5], *NetBSD* [6] and *OpenBSD* [7]
- *klibc v2.0.11* [8]: primarily for booting Linux systems
- *musl v1.2.3* [9]: another lightweight C standard library implementation for Linux systems
- *newlib v4.3.0* [10]: a C standard library for embedded systems (MMU-less)
- *µClibc-ng v1.0.42* [11]: an embedded C library

We considered the implementations of 52 C library functions from these 9 libraries, which broadly fall into 3 categories:

- `string` functions: `strlen`, `strlcpy`, `strrchr`, `strcasecmp`, ...
- `memory` functions: `memcpy`, `memccpy`, `memset`, `memrchr`, ...
- `wide-char` functions: `wcschr`, `wcslen`, `wcsncat`, `wcsncpy`, ...

Some of these libraries have multiple implementations of the same function. For instance, *newlib* has two implementations of `memrchr`: `SMALL` (when optimising for code size) and `FAST` (when optimising for runtime).

For each function, our goal is to use the equivalence checker and the inequivalence checker to categorize all implementations into equivalence classes. We do so by running our tool for each possible pair in the list of implementations, and keeping track of equivalence classes. Initially, all implementations are in a separate equivalence class. If two implementations are proven

equivalent, then we merge their equivalence classes, else if they are proven inequivalent, then we mark them as inequivalent.

For each selected implementation pair, we convert them into their unoptimized LLVM IR (generated by `clang -O0`). Since we don't know whether a given pair is equivalent or inequivalent, choosing an appropriate value of the unroll-factor μ and the loop-bound ν could be tricky. For an equivalent pair, choosing a low value of unroll-factor may not be sufficient. While for an inequivalent pair, if we choose a high value of unroll-factor, then we may spend a lot of time trying to prove its equivalence and then run our inequivalence procedure. So, we start from a low value of unroll factor and increase it iteratively. This is described by the algorithm below:

Algorithm 2: Inequivalence Benchamrking

Input: The input programs *src*, *dst*, max unroll factor $\mu_{\texttt{max}}$, max loop bound $\nu_{\texttt{max}}$

```
1  μ ← 1;
2  while μ ≤ μ_max do
3  |   res, Γ_ce ← ineqChecking(src, dst, μ, ν_max);
4  |   if res ∈ {EQUIV, INEQ} then
5  |   |   return res;
   |   //rerun by switching src and dst
6  |   res, Γ_ce ← ineqChecking(dst, src, μ, ν_max);
7  |   if res ∈ {EQUIV, INEQ} then
8  |   |   return res;
9  |   μ ← μ * 2;
10 end
11 return FAIL;
```

`Counter`'s correlation algorithm is asymmetric since it allows for a one-to-many mapping from nodes in *src* to nodes in *dst*. So, while benchmarking, it is essential to re-run the tool by switching the source and destination programs. We used a timeout of 60 minutes for each call to `ineqChecking()`, with a memory limit of 32GB. We set $\mu_{\texttt{max}}$ to 32, $\nu_{\texttt{max}}$ to 16 and `ce-bound` (Section-3.6) to 22 (which would set the top 22 bits of all input arguments to 0).

After finishing benchmarking, we generate equivalence checking graphs (shown in the later sections), which have the following properties:

- Implementations in the same node are in the same equivalence class, and hence are observationally equivalent
- A *blue edge* between classes denotes that they are inequivalent
- A *dashed red edge* between classes denotes that we don't know if they are equivalent or inequivalent

4.2 Results

In this section, we will discuss some of the functions and how their implementations have been sorted into equivalence classes.

`memccpy`

The `memccpy` specification [12] states the following

> `void *memccpy(void *s1, const void *s2, int c, size_t n);`
>
> The `memccpy()` function shall copy bytes from memory area `s2` into `s1`, stopping after the first occurrence of byte `c` (converted to an `unsigned char`) is copied, or after `n` bytes are copied, whichever comes first. If copying takes place between objects that overlap, the behavior is undefined.

The equivalence classes computed for `memccpy` are shown below.

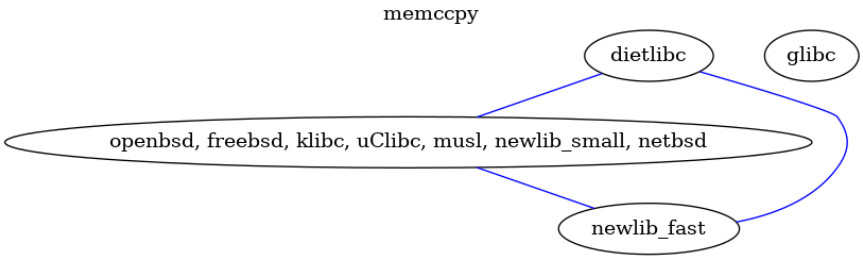


Figure 4.1: Equivalence Classes for `memccpy`

We found 4 equivalence classes. The one with `freebsd` is the *correct* implementation because they follow the specifications. We found two bugs in `dietlibc` and `newlib_fast` (discussed in the later sections). `glibc` is in a disconnected component since it has function calls and our tool cannot handle this case.

`wcschr`

The `wcschr` specification [13] states the following

> `wchar_t* wcschr (wchar_t* ws, wchar_t wc);`
>
> Returns a pointer to the first occurrence of the wide character `wc` in the C wide string `ws`. The terminating null wide character is considered part of the string. Therefore, it can also be located in order to retrieve a pointer to the end of a wide string.

The equivalence classes computed for `wcschr` are shown below.

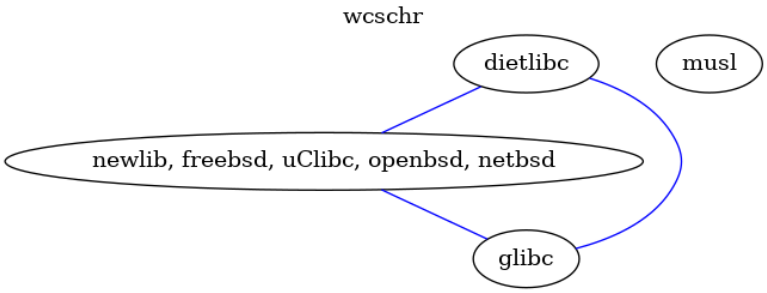


Figure 4.2: Equivalence Classes for `wcschr`

We found 4 equivalence classes. The one with `freebsd` is the *correct* implementation because they follow the specifications. We found one bug in `dietlibc` (discussed in the later sections). The indicated bug with `glibc` is invalid because we find a counterexample where the input pointer was `null`, which does not correspond to a valid C input. `musl` is in a disconnected component since it has function calls and our tool cannot handle this case.

`swab`

The `swab` specification [14] states the following

> `void swab(const void *restrict from, void *restrict to, ssize_t n);`
>
> The `swab()` function copies `n` bytes from the array pointed to by `from` to the array pointed to by `to`, exchanging adjacent even and odd bytes.
>
> This function does nothing when `n` is negative. When `n` is positive and odd, it handles `n-1` bytes as above, and does something unspecified with the last byte. (In other words, `n` should be even.)

The equivalence classes computed for `swab` are shown below.

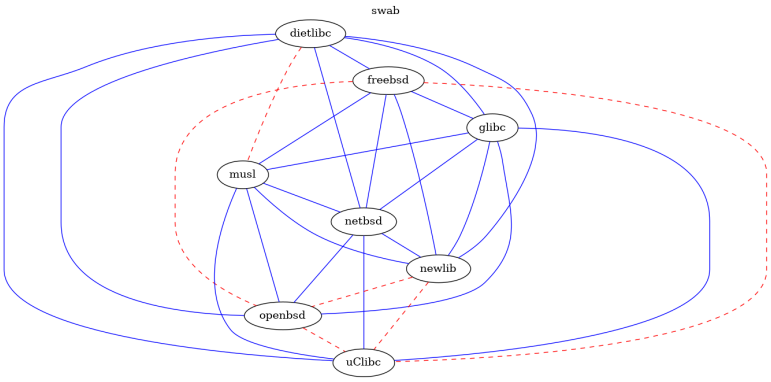


Figure 4.3: Equivalence Classes for `swab`

We found multiple equivalence classes. The inequivalence between `freebsd` and `newlib` occurs due to the undefined behaviour mentioned in the specification (`n` should be even, so it is not a valid bug). We also found a bug in `netbsd` (discussed later). The other inequivalences occur due to the following two reasons:

- Incorrect swapping: if the `from` and `to` memories coincide, then it reduces to in-place swapping of bytes in the input memory. `dietlibc` has the following code inside the byte-swap loop:

```
...
for (i=0; i<nbytes; i+=2) {
    d[i]=s[i+1];
    d[i+1]=s[i];
}
...
```

  The swapping without a temporary variable, which would be incorrect if `from == to`.

- Direction of swapping: if the input memories overlap (and don't coincide), then the direction of the copy matters. For instance, if `from` < `to`, then `from` will overwrite itself while doing a forward copy, while a backwards copy will not. Some of the implementations (`freebsd`, `musl`, ...) do a forward copy, while `glibc` does a backwards copy, leading to a number of inequivalences as shown in Figure 4.3.

Note that since memory overlap is also undefined behaviour for `swab`, most of these inequivalences may not be valid.

4.3 Undefined Behaviour

We found 40 *blue edges* across all the functions that we tried. But not all the *blue* edges may correspond to valid bugs. Currently, we say that a given pair of programs in *inequivalent* if there is an input that leads to unequal return values or unequal heap states at the end. But if we have the function specifications and the C language specifications into account, then the programs may not be inequivalent. For instance, as discussed in Section-4.2, if the input memories overlap, then the behaviour is undefined.

We encountered 3 types of Undefined Behaviour conditions in the benchmarks:

- Memory Area Overlap: This applies to benchmarks like `memccpy` [12], `memcpy`, `strcpy`, `swab`, etc. The C-Language specification [15] also mentions these conditions

- Valid Pointer Values: The C-Language specification [15] states the following `string` function conventions

  > Unless explicitly stated otherwise in the description of a particular function in this subclause, pointer arguments on such a call shall still have valid

> values, as described in 7.1.4. On such a call, a function that locates a character finds no occurrence, a function that compares two character sequences returns zero, and a function that copies characters copies zero characters.

This applies to benchmarks like `strlen`, `strnlen`, `wcschr`, `stpncpy`, etc.

- Number of bytes should be even: This applies to `swab`, that the number of input bytes should be even

Our tools allows us to put preconditions at the start of the programs (referred to as `assumes`). The solution to this problem is to put assumes corresponding to each Undefined Behaviour condition in the benchmarking. After putting the assumes, we found that 18 *blue* edges are still valid, and we have reported bugs in 11 function implementations so far.

As for the other bugs triggered due to Undefined Behaviour, they could still be useful to us. Programmers rarely care about function specifications. Failure to comply with these specifications has been a significant source of bugs in the past. Most notably, in `glibc 2.13`, a performance optimization of `memcpy()` on some platforms (including `x86-64`) included changing the order in which bytes were copied from `src` to `dest`. This change revealed breakages in a number of applications that performed copying with overlapping areas. Under the previous implementation, the order in which the bytes were copied had fortuitously hidden the bug, which was revealed when the copying order was reversed [16].

4.4 Bugs Found

This section summarizes the valid bugs that we found across different C library implementations. The bug-reports can be found here.

4.4.1 `klibc`

We found a bug in the `strrchr` implementation of `klibc`. Linux manpage [17] for `strrchr()` specifies the following:

> `char *strrchr(const char *s, int c);`
>
> The `strrchr()` function returns a pointer to the last occurrence of the character `c` in the string `s`. The terminating null byte is considered part of the string, so that if `c` is specified as '\0', this function returns a pointer to the terminator.

`klibc`'s implementation does not follow this and thus gives wrong output when `c` is '\0'. An example input is shown below:

```
const char src[] = {128, '\0'};
char *ret = strrchr(src, 0);
if (!ret) {
          printf("BUG!\n");
}
```

We reported this bug and it was fixed in `klibc v2.0.12`.

4.4.2 `netbsd`

We found a bug in the `swab` implementation of `netbsd`. The Linux manpage [14] for `swab()` specifies the following:

> `void swab(const void *restrict from, void *restrict to, ssize_t n);`
>
> The `swab()` function copies `n` bytes from the array pointed to by `from` to the array pointed to by `to`, exchanging adjacent even and odd bytes.
>
> This function does nothing when `n` is negative. When `n` is positive and odd, it handles `n-1` bytes as above, and does something unspecified with the last byte. (In other words, `n` should be even.)

`netbsd`'s implementation stores the half of `n`, the number of swaps operations to be made, in a variable `len`. It rounds `len` to a multiple of 8, and then unroll the loop to execute 8 steps in a single iteration. The rounding operation is done incorrectly, as shown below. As it can be seen, it always decrements the value of `len`, and then checks if it is a multiple of 8. This leads to one less swap than expected.

```
/* round to multiple of 8 */
while ((--len % 8) != 0)
    STEP;
```

An example input is shown below:

```
const char src[] = {90, 91, 1, 2};
char dst[4] = {'A', 'B', 'C', 'D'};
swab(src, dst, 4);
// Expected values at dst: {91, 90, 2, 1}
if (dst[0] != 91 || dst[1] != 90 || dst[2] != 2 || dst[3] != 1) {
    printf("BUG!\n");
}
```

We reported this bug and it was fixed on their official git repository. When our fix was accepted, the NetBSD devs replied with the following:

> Evidently the previous definition, presumably tightly optimized for 1980s-era compilers and CPUs, was too hard to understand, because it was incorrectly tested for two decades and broken for years.

This just shows that the bugs are usually very subtle and hard to catch, and sometimes, the specifications are not properly followed. They also added the following comment on their implementation after the fix [18]:

> According to POSIX (2018), the behaviour is undefined if src and dst overlap. However, there are uses in-tree (xsrc/external/mit/xfwp/dist/io.c) that rely on swab(ptr, ptr, n) to do the swabbing in-place. So make sure this works if src == dst.

So, even though the `src == dst` triggers UB, we still might want to enforce a specified behaviour of the function when UB is triggered.

4.4.3 newlib

We found a bug in the memccpy FAST implementation of newlib. This implementation is enabled when the macros PREFER_SIZE_OVER_SPEED and __OPTIMIZE_SIZE__ are not defined. The memccpy specification [12] states the following:

> void *memccpy(void *s1, const void *s2, int c, size_t n);
>
> The memccpy() function shall copy bytes from memory area s2 into s1, stopping after the first occurrence of byte c (converted to an unsigned char) is copied, or after n bytes are copied, whichever comes first. If copying takes place between objects that overlap, the behavior is undefined.

The FAST implementation converts c to signed char (stored in endchar) and computes a mask as follows:

```
for (i = 0; i < LITTLEBLOCKSIZE; i++)
    mask = (mask << 8) + endchar;
```

This is used to detect endchar in one long word and is supposed to represent a word whose each byte has the same value as endchar. But if the input character lies in the extended ASCII set, then endchar is negative, which leads to an incorrect mask computation. This

leads to the character `c` not being detected, more bytes are copied to `dst` than expected. We reported this bug and our fix was accepted.

An example input is shown below:

```
const char src[] = {1, 2, 3, 4, 5, 192, 6, 7};
char dst[8] = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H'};
memccpy(dst, src, 192, 8);
if (dst[7] != 'H') {
    printf("BUG!\n");
}
```

4.4.4 `dietlibc`

Most of the bugs in `dietlibc` were due to missing `char*` or `unsigned char*` typecasts. The bug reports can also be found here.

`memchr`

We found a bug in the `memchr` implementations present in the `contrib/` directory of `dietlibc` (different from the ones present in the `lib/` directory. The bugs in this implementation was earlier reported by [1]). The Linux manpage for `memchr()` [19] states the following:

> `void *memchr(const void *s, int c, size_t n);`
>
> The `memchr()` function scans the initial `n` bytes of the memory area pointed to by `s` for the first instance of `c`. Both `c` and the bytes of the memory area pointed to by `s` are interpreted as `unsigned char`.

Both the `SMALL` (with the `WANT_SMALL_STRING_ROUTINES` macro defined) and the `FAST` (without the `WANT_SMALL_STRING_ROUTINES` macro defined) implementations do not follow this specification, which led to a bug when `c` is greater than 255.

An example input is shown below (compiled with the `-m32` flag). Here, the character `257` is interpreted as a `signed int`. So, the equality check with the characters in `src` never succeeds.

```
// We should find the occurance of 1
const char src[] = {1, 0, 0, 0, 0};
if (!memchr(src, 257, 5)) {
```

```
        printf("BUG!\n");
}
```

`strcmp`

We found a bug in the `strcmp` implementations of `dietlibc`. The Linux manpage for `strcmp()` [20] states the following:

> `int strcmp(const char *s1, const char *s2);`
>
> The `strcmp()` function compares the two strings `s1` and `s2`. The locale is not taken into account. The comparison is done using `unsigned` characters.

Both the `SMALL` (with the `WANT_SMALL_STRING_ROUTINES` macro defined) and the `FAST` (without the `WANT_SMALL_STRING_ROUTINES` macro defined) implementations do not follow this specification, which led to a bug.

An example input is shown below. Here, the character `255` is interpreted as a `signed char`, which gives it a value of `-128`. So, the comparison returns a positive value.

```
const char src[] = {64, 1, 0};
const char dst[] = {64, 255, 0};
int ret = strcmp(src, dst);
if (ret >= 0) {
    printf("BUG!\n");
}
```

`strcasecmp` **and** `strncasecmp`

We found a bug in the `strcasecmp` and the `strncasecmp` implementations of `dietlibc`. The Linux manpage for `strcasecmp()` and `strncasecmp()` [21] states the following:

> `int strcasecmp(const char *s1, const char *s2); int strncasecmp(const char *s1, const char *s2, size_t n);`
>
> The `strcasecmp()` function shall compare, while ignoring differences in case, the string pointed to by `s1` to the string pointed to by `s2`. The `strncasecmp()` function shall compare, while ignoring differences in case, not more than `n` bytes from the string pointed to by `s1` to the string pointed to by `s2`.
>
> In the `POSIX` locale, `strcasecmp()` and `strncasecmp()` shall behave as if the strings had been converted to lowercase and then a byte comparison performed. The results are unspecified in other locales.

The implementations do not handle the case when the characters belong to the extended `ASCII` set.

An example input is shown below.

```
const char src[] = {'A', 1, 0};
const char dst[] = {'a', 255, 0};
int ret = strcasecmp(src, dst);
if (ret >= 0) {
    printf("BUG!\n");
}

int ret = strncasecmp(src, dst, 3);
if (ret >= 0) {
    printf("BUG!\n");
}
```

`wcschr` and `wcsrchr`

We found a bug in the `wcschr` and the `wcsrchr` implementations of `dietlibc`. According to the the `wcschr` spec [13] and `wcsrchr` spec [22], if the input `c` is the null wide character, then the functions locate the terminating null wide character. `dietlibc's` implementation does not follow that and returns `NULL` instead.

An example input is shown below.

```
const wchar_t src[] = {128, 64, 0};
wchar_t* ret = wcschr(src, 0);
if (!ret) {
    printf("BUG!\n");
}

wchar_t* ret = wcsrchr(src, 0);
if (!ret) {
    printf("BUG!\n");
}
```

Chapter 5

Comparison with Differential Fuzzing

As discussed in Chapter 1, one could use the inequivalence checker in a differential testing setting as well. To compare our approach with existing differential testing techniques, we used `afl-fuzz` [23], which is a security oriented fuzzer. We considered all the program pairs for which our tool is able to prove inequivalence, and run `afl-fuzz` on all those pairs.

5.1 American Fuzzy Lop

American **F**uzzy **L**op (`afl-fuzz`) is a security oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary. It is a coverage-guided mutation based fuzzer. This substantially improves the functional coverage for the fuzzed code. You provide a set of valid inputs and it will search for crashes and hangs in the vicinity of these inputs. It is not suited to search for logical errors in the input programs, unless we explicitly instrument the programs to crash/hang in the event of these errors.

A typical workflow to test a program using `afl-fuzz` is shown below. `afl-fuzz` feeds the input via standard input or an input file -- so the input is a series of characters. We need to implement a *fuzzing harness* to appropriately parse the input so that it can be sent to our test program. The harness implementation decides the input space that would be explored. The fuzzer then monitors the program for crashes and hangs.

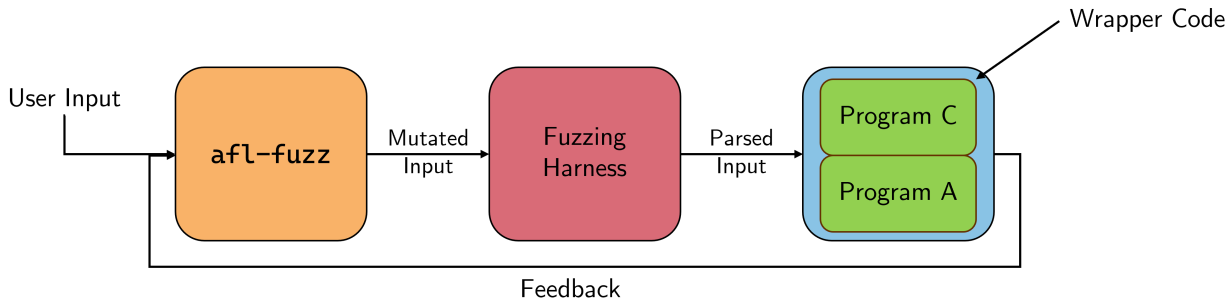


Figure 5.1: An example `afl-fuzz` workflow

In the case of differential testing, we have two input programs instead of a single program. To use `afl-fuzz` for differential fuzzing, we need to some wrapper code around the input

programs, which passes the same input to both the implementations, and checks for unequal return values or unequal heap states. If it finds a bug, then the wrapper code will crash, and `afl-fuzz` will look for this crash.

5.2 Designing Harnesses

Assume that we have *(func, src, dst)* triplets for which our tool found a distinguishing input. We want to then use `afl-fuzz` to try and detect a bug for all such triplets. We need to implement fuzzing harnesses for each function *func* to be tested. The fuzzing harness would pass the same input to both the implementations, and would abort if their outputs (return values and memory regions) differ. The fuzzer will try to find an input which triggers a crash (which will be a bug in our case).

We may have multiple harness designs for a given *func*. In such a case, we use **macros** to toggle the implementations. For instance, we could have the following two designs for `memcpy(dst,src,n)`:

- Non-overlapping memories: both `dst` and `src` memories would be non-overlapping, with an allocated size of `MAX_SIZE` each. This would lead to some part of the input space being unexplored, since the inputs can never overlap.
- Overlapping memories: both `dst` and `src` lie at some offset in a "virtual memory" of size `VMEM_SIZE`, allowing for memory overlap to occur in the input. This harness can be enabled by defining the macro harness `OVERLAP`

If there are n such (mutually independent) macros, then there would be 2^n total harness designs. For a given program pair, we run `afl-fuzz` for all such harnesses to see the effect of harness design on fuzzer capability. Each experiment can be characterized by a 4-tuple *(func, src, dst, config)*, where *config* is a set of macros which are enabled for the harness.

The following macros were used to design harnesses for the given functions:

- `OVERLAP`: this pertains to functions which modify memory. When enabled, the input memories can be overlapping. The fuzzer can now explore a larger input space.
- `CHARC`: this pertains to functions which take a character (usually named c, having type `int`) as one of its input. `memccpy` and `memchr` are prominent examples.
  The difference lies in how the input character is parsed. The harness reads the input file line-by-line and stores each line in a character array. When `CHARC` is enabled, c is read as follows:

  ```
  *c = cstr[0];
  ```

  This restricts the value of c to be in the interval $[-128, 127]$. When `CHARC` is disabled, c is read as:

```
*c = atoi(cstr);
```

The value of c can now span the entire space of 32-bit signed integers. However, if `atoi` cannot extract an integer from `cstr`, then it will return 0.

5.3 Evaluation

We have a total of 148 unique *(func, src, dst)* triplets. Let `macros` be a map from function name to a set of macros that can be enabled for the harnesses for that function. Following are some examples:

$$\begin{aligned}
\texttt{macros}(\texttt{memccpy}) &= \{\texttt{OVERLAP}, \texttt{CHARC}\} \\
\texttt{macros}(\texttt{memchr}) &= \{\texttt{CHARC}\} \\
\texttt{macros}(\texttt{memcpy}) &= \{\texttt{OVERLAP}\} \\
\texttt{macros}(\texttt{strcmp}) &= \{\} \\
\texttt{macros}(\texttt{swab}) &= \{\texttt{OVERLAP}\}
\end{aligned}$$

For a given function f, the number of possible harness configurations would be

$$|\texttt{PowerSet}(\texttt{macros}(f))|$$

For convenience, denote the {} harness where no macro is enabled, as `DEFAULT`. The number of unique *(func, src, dst, config)* tuples is 258. The timeout for each fuzzing experiment was 2hrs, with execution timeout for each exec being 30ms.

For each inequivalent pair, we could check how many harnesses found a bug, or if any harness found a bug. The following table shows the number of bugs that `afl-fuzz` was able to find:

Fuzzer Success Rate	
(func, src, dst)	140/148
(func, src, dst, config)	179/258

Table 5.1: `afl-fuzz` success rate

We discuss more about these cases in detail in the next section.

5.4 Case Studies

Fuzzer did not find a bug

This section summarizes the cases where none of the harness designs for a given *func* found a bug:

- (`stpncpy`, `freebsd`, `newlib_small`): There were two possible harness designs for `stpncpy`: `DEFAULT` and {`OVERLAP`}. The counterexample generated by our tool does not correspond to a valid input (invalid pointer value). This is the reason why both the harnesses did not find a bug
- (`strcmp`, `dietlibc_small_patched`, `dietlibc_fast`): `strcmp` takes two strings s_1, s_2 as input. The behaviour of these programs differs in two situations:
  - When s_1 and s_2 are not aligned with respect to each other
  - When s_1 and s_2 are not aligned to word boundaries

  For the harness to find a bug, it needs to generate input pointers that satisfy the criteria above. Our harness cannot do it because the pointers are word-aligned by default. This leads to some part of the input space being unexplored.
  A similar case occurred with (`strcmp`, `dietlibc_fast`, `dietlibc_fast_patched`)
- (`strlen`, `dietlibc_small`, `glibc`): `dietlib_small` returns 0 if the input pointer was `NULL`. Hence, the counterexample generated by our tool does not correspond to a valid input (invalid pointer value). This is the reason why the fuzzer did not find any bug.
  A similar case occurred with (`strlen`, `dietlibc_small`, `klibc`) and (`strlen`, `dietlibc_small`, `newlib`)
- (`swab`, `musl`, `musl_patched`): A similar case occurred with (`swab`, `musl`, `openbsd`). This case is discussed in Section-5.4

Fuzzer did not find a bug in `swab`

The man page for `swab` states the following specification:

> `void swab(const void *src, void *dest, ssize_t n);`
>
> The `swab()` function copies `n` bytes from the array pointed to by `src` to the array pointed to by `dest`, exchanging adjacent even and odd bytes.

We have 2 different implementations of `swab`:

- `dietlibc`: This implementation swaps the bytes between `src` and `dest` without a temporary variable:

```
...
for (i=0; i<nbytes; i+=2) {
    d[i]=s[i+1];
    d[i+1]=s[i];
}
...
```

When `src` and `dest` memories coincide, this does not swap the bytes in-place, causing a bug.
An easy fix for this bug would be to load the values in `src` into temporary variables (call this implementation `dietlibc_patched`):

```
...
for (i=0; i<nbytes; i+=2) {
    char s0 = s[i], s1 = s[i+1];
    d[i]=s1;
    d[i+1]=s0;
}
...
```

- `musl`: This implementation has the same bug as `dietlibc`. On the same lines, we have another fixed implementation, namely `musl_patched`. The only difference this has from `dietlibc` is that the input pointer arguments have a `restrict` type qualifier:

```
void swab_musl(const void *restrict _src,
    void *restrict _dest, ssize_t n);
```

We ran our tool and the fuzzer between the base and the patched implementations. The results are shown in the figure below:

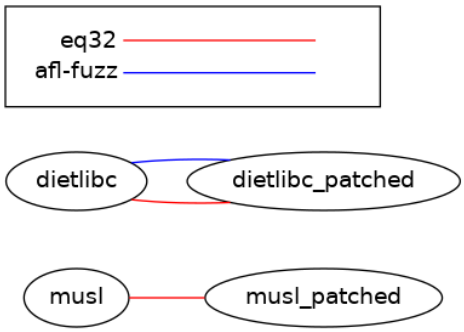


Figure 5.2: `swab`: An edge indicates that we were able to find a distinguishing input

`eq32` was able to find a distinguishing input between the base and the patched implementations. These inputs correspond to coinciding memories.

`swab` has two possible harness designs: `DEFAULT` and `{OVERLAP}`. The `DEFAULT` harness will not be able to find a bug because it cannot explore overlapping (and coinciding inputs). The

{OVERLAP} harness is only able to find a bug in dietlibc, and not in musl (despite both the implementations being almost the same in structure).

Here are some observations regarding the musl implementations:

- On removing the restrict qualifier from musl, the fuzzer is able to find a bug between musl and musl_patched
- By default, the afl-gcc appends -O3 to optimize fuzzing binary builds. The fuzzer is able to find a bug if we set AFL_DONT_OPTIMIZE (this would disable optimizations)
- On analyzing the assembly code generated for the musl implementation at -O3, we found that gcc does some trivial vectorization. It assumes that the input memories cannot alias (due to the restrict qualifier). Multiple memory locations from src are first loaded, and then are copied to dest while swapping adjacent bytes.
  However, on giving an input in which the memories coincide, since the loads occurs first, the in-place swab occurs correctly. Hence, the fuzzer is not able to find a bug in this case.
- On removing the restrict qualifier from musl and then analyzing the generated -O3 assembly code, it was observed that the compilers inserts from run-time checks that verify the information that restrict provides (so that it can branch to the vectorized version, else it would go to the fallback scalar implementation).
  On providing an input in which the memories coincide, we use the fallback implementation. This leads the fuzzer to find a bug

CHARC Bugs

As described above, this macro is used for benchmarks which take a character as input. I will motivate multiple harness designs with the help of the memrchr benchmarks. The man page for memrchr states the following specification:

> The memchr() function scans the initial n bytes of the memory area pointed to by s for the first instance of c. Both c and the bytes of the memory area pointed to by s are interpreted as unsigned char.
>
> The memrchr() function is like the memchr() function, except that it searches backward from the end of the n bytes pointed to by s instead of forward from the beginning.

We have 3 different implementations of memrchr:

- openbsd: This implementation follows the specification and interprets both the character c and the memory bytes as unsigned char (it involves a unsigned char vs unsigned char comparison)

```
const unsigned char *cp;
...
    cp = (unsigned char *)s + n;
    do {
        if (*(--cp) == (unsigned char)c)
...
```

- `dietlibc`: This implementation interprets `c` as `signed int`, while it interprets the memory bytes as `signed char` (it involves a `signed char` vs `signed int` comparison)

```
register const char* t=s;
...
for (i=n; i; --i) {
    if (*t==c)
...
```

- `glibc`: This implementation interprets `c` as `signed int`, while it interprets the memory bytes as `unsigned char` (it involves a `unsigned char` vs `signed int` comparison, where the former would get sign-extended to a `signed integer` in the range $[0, 255]$)

```
...
char_ptr = (unsigned char*)s + n;
...
while (n-- > 0) {
    if (*--char_ptr == c)
...
```

`memrchr` has two possible harness designs, `DEFAULT` and `{CHARC}`. `afl-fuzz` was run on all 3 pairs, with both the harnesses. The results are shown in the figure below -- an edge between two implementations indicates that the harness found a bug.

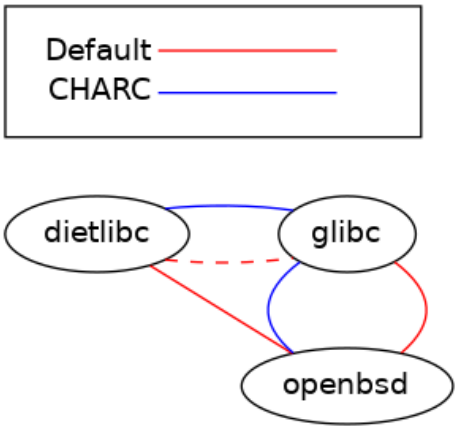


Figure 5.3: `CHARC` bugs in `memrchr` -- red edge indicates that the `DEFAULT` harness found a bug, while a blue edge indicates that the `{CHARC}` harness found a bug. The dotted case is discussed below

The section below explains why one harness is able to distinguish between some implementations, while the other is not.

- `dietlibc-openbsd`: A bug is triggered when `c` has a value outside the range of `signed char`, in which case the comparison in `dietlibc` can never succeed
  - `DEFAULT`: with this design, `c` can potentially span the entire range of `signed int`, so the fuzzer is able to find a bug when the following condition holds:

    $$c < -128 \ \vee \ c > 127$$

  - `{CHARC}`: with this design, `c` is always an integer in the range $[-128, 127]$, so the fuzzer cannot find a bug

- `openbsd-glibc`: A bug is triggered when `c` has a value outside the range of `unsigned char`, in which case the comparison in `glibc` can never succeed
  - `DEFAULT`: with this design, `c` can potentially span the entire range of `signed int`, so the fuzzer is able to find a bug when the following condition holds:

    $$c < 0 \ \vee \ c > 255$$

  - `{CHARC}`: with this design, `c` is always an integer in the range $[-128, 127]$, so the fuzzer is able to find a bug when $c \in [-128, 0)$

- `dietlibc-glibc`: In this case, both of the implementations are buggy with respect to the actual specification, but we are concerned about an input that distinguishes *these two* implementations. It is easy to see that these implementations can be differentiated when the following condition holds:

  $$c \in [-128, 0) \ \cup \ [128, 256)$$

  - `DEFAULT`: with this design, `c` can potentially span the entire range of `signed int`. The space of buggy inputs is small, so the probability that the fuzzer will find a bug is very low.
    Assuming that the fuzzer would generate uniformly random inputs, and `atoi` preserves the input distribution, then the probability of finding a bug is roughly $\frac{256}{2^{32}} = \frac{1}{2^{24}}$.
    It took `afl-fuzz` 40 minutes and 12.1M executions to find the first crash, which is a lot as compared to other benchmarks.
  - `{CHARC}`: with this design, `c` is always an integer in the range $[-128, 127]$, so the fuzzer is able to find a bug quickly when $c \in [-128, 0)$

Similar cases arose in other benchmarks like `memccpy` and `memchr`, where the `{CHARC}` harness did not find a bug, but the `DEFAULT` harness did. So, whether or not the fuzzer will find a bug heavily depends on the input space that can be explored by the harness and the size of the *buggy* input space.

Memory overlap bugs

For functions that modify memory, it should be sufficient to have non-overlapping inputs and compare the return values from both the implementations. For functions that do modify memory (`memccpy`, `memcpy`), a large part of the input space will remain unexplored with non-overlapping inputs.

- Defining `OVERLAP` would enable a harness which allows for overlapping memories. It reads both the input strings and populates them in a "virtual memory" of a given size
- The same initial memory is fed to both the implementations and the harness checks if the output memory is same for both
- There were 49 *(func, src, dst)* triplets where `eq32` found a bug with overlapping memories
- The default fuzzing harness (which gives non-overlapping inputs) did not find any bug in all of these cases, while the `OVERLAP` harness is able to find a bug
- One thing to note is that most of the bugs are not actually valid, because the function specification declares input memory overlap as undefined behaviour. But these bugs may be of interest to us in some cases

5.5 Conclusion

As seen in the previous sections, the capability of the fuzzer to find a bug depends on a number of factors. One of those factors is the Designing Harnesses. For each pair that has to be tested with the fuzzer, we need to design a fuzzing harness for that function. As discussed in Section-5.2, we may have multiple harness designs for some functions. The harnesses depend on the programs themselves. This increases a lot of manual work as the number of benchmarks increase. Our tool does not have this limitation.

The harness design itself may limit the effectiveness of the fuzzing technique. Whether or not the fuzzer will find a bug depends on the input space that the harness can explore. A very simple fuzzer which just compares the return values may not be sufficient for heap manipulating programs. This requires a non-trivial harness. This directly depends on the harness design. This is not an easy task because we don't know the space of *buggy* inputs beforehand.

In most cases, the fuzzer is able to find a bug very quickly (in less than 10 seconds), while it usually takes around 100s for our tool for the same. But as discussed in Section-5.4, it takes a lot of time to find a bug in (`memrchr`, `dietlibc`, `glibc`) with the `DEFAULT` harness. However, our tool can find a bug under a couple of minutes. If the space of *buggy* inputs is small as compared to the space of all the inputs that the harness can explore, then the time

taken to find a bug by the fuzzer increases. This happens because fuzzers usually apply brute-force techniques (with some guided genetic algorithms), but our symbolic analysis is highly guided.

Chapter 6

An Alternate Approach to Inequivalence Checking

As discussed in Section-3.5, we are essentially collecting paths in the Data-Flow Analysis. Based on this observation, an alternative approach to inequivalence checking would to run the Data-Flow Analysis on the individual programs and then try to find a bug.

6.1 Implementation Overview

Let the input programs be denoted by *src* and *dst*. The approach is simple, we start from *failconds* that correspond to observational inequivalence. So, $f \in \{\texttt{ret}_{\texttt{src}} \neq \texttt{ret}_{\texttt{dst}},\ \texttt{H}_{\texttt{src}} \neq \texttt{H}_{\texttt{dst}}\}$. For each failure condition f, we run the DFA on both *src* and *dst* and collect sets of paths in both the programs.

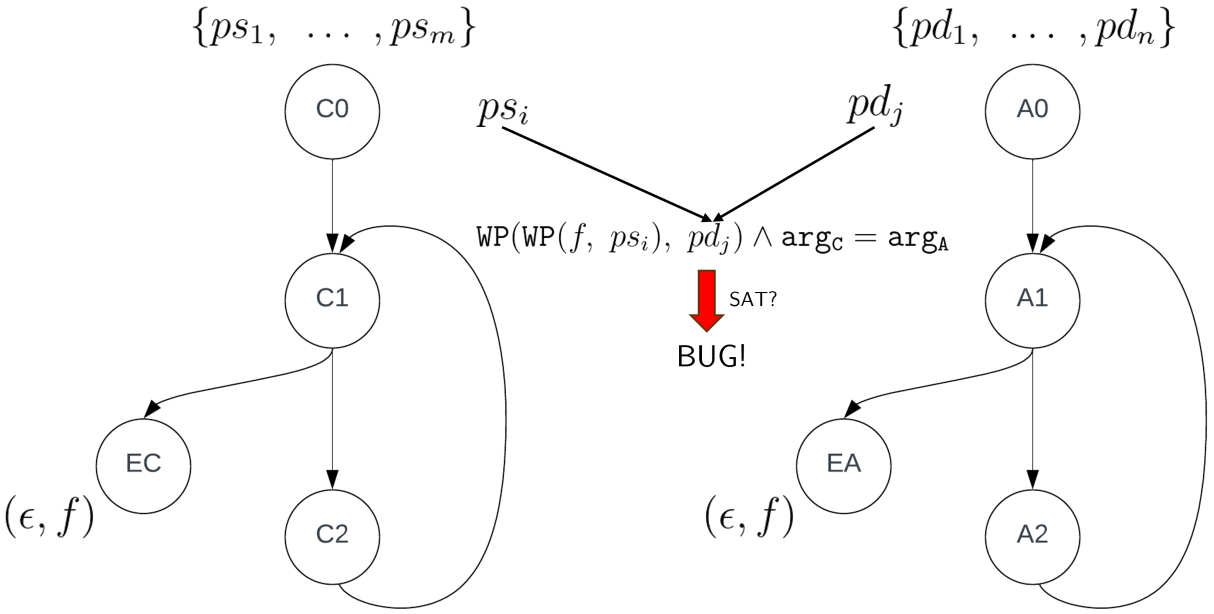


Figure 6.1: Alternate Inequivalence Checking Approach

Assume that we collect m paths $\{ps_1,\ ps_2,\ \ldots,\ ps_m\}$ in *src* and collect n paths $\{ps_1,\ ps_2,\ \ldots,\ ps_m\}$ in *dst*. Then for each $i \in \{1,\ \ldots,\ m\}$, $j \in \{1,\ \ldots,\ n\}$, discharge the following condition to an SMT solver

$$\texttt{WP}(\texttt{WP}(f,\ ps_i),\ pd_j) \wedge \texttt{arg}_{\texttt{src}} = \texttt{arg}_{\texttt{dst}} \wedge \texttt{H}_{\texttt{src}} = \texttt{H}_{\texttt{dst}}$$

If any of the $m * n$ conditions is satisfiable, then we have found a bug. This alternate implementation is also implemented as a part of the `eq32` tool, and can be enabled by the `--ineq-individual-tfgs` flag.

6.2 Comparing the two approaches

In the discussion going forward, we refer to the previous approach as `CFG-Approach`, while the new approach is referred to as `TFG-Approach`. We first discuss the fundamental differences between the approaches.

Graph on which the DFA is run

This difference stems from the methodologies itself. In `CFG-Approach`, we collect failed-CFGs (which could be partial or complete) and then run the DFA on each of them, while in `TFG-Approach`, we run the DFA on both the input TFGs.

Dependency on the equivalence checker

The `CFG-Approach` directly depends on the number of failed CFGs that we collect and the unroll-factor μ (both of which depend on the equivalence checker), while the `TFG-Approach` is completely agnostic to the equivalence checker

Validity of counter-examples

Due to soundness of our DFA, any counterexamples that we get at the start-node are guaranteed to trigger the *failcond* at the *failcond-pc*. For `CFG-Approach`, the *failconds* can be of different types. As discussed in Section-3.6, we have to verify the validity of any counterexamples that we get.

While for `TFG-Approach`, the only *failconds* that we have in this approach are the following, at the exit-pc:

$$\texttt{ret}_{\texttt{src}} \neq \texttt{ret}_{\texttt{dst}}$$
$$\texttt{H}_{\texttt{src}} \neq \texttt{H}_{\texttt{dst}}$$

The counterexamples (if any) will definitely trigger these failure conditions, so they will be valid distinguishing inputs (as these conditions correspond to observational inequivalence). There is no need to check their validity.

Counterexamples execution traces

As discussed in Section-3.6, we use a heuristic in `CFG-Approach` to get counterexamples with smaller execution traces. This is only a heuristic and does not guarantee bounded execution traces. While for the `TFG-Approach`, since the paths that we collect in this approach would have bounded length, the execution traces of the counterexamples are guaranteed to be bounded (by some function of ν).

Algorithm to get counterexamples

The algorithm for `CFG-Approach` is discussed in 1. The algorithm for `TFG-Approach` is described below.

Algorithm 3: Alternative Inequivalence Checking Approach

```
1  Function altIneqChecking(src, dst, μ, ν_max):
2      π, _ ← bestFirstSearch(src, dst, μ);
3      if π ≠ null then
4          return EQUIV, {};
5      end
6      ν ← 1;
7      while ν ≤ ν_max do
8          Γ_ce ← computeIneqAlt(cg, ν);
9          if Γ_ce ≠ ϕ then
10             return INEQ, Γ_ce;
11         end
12         ν ← ν * 2;
13     end
14     return FAIL, {};
```

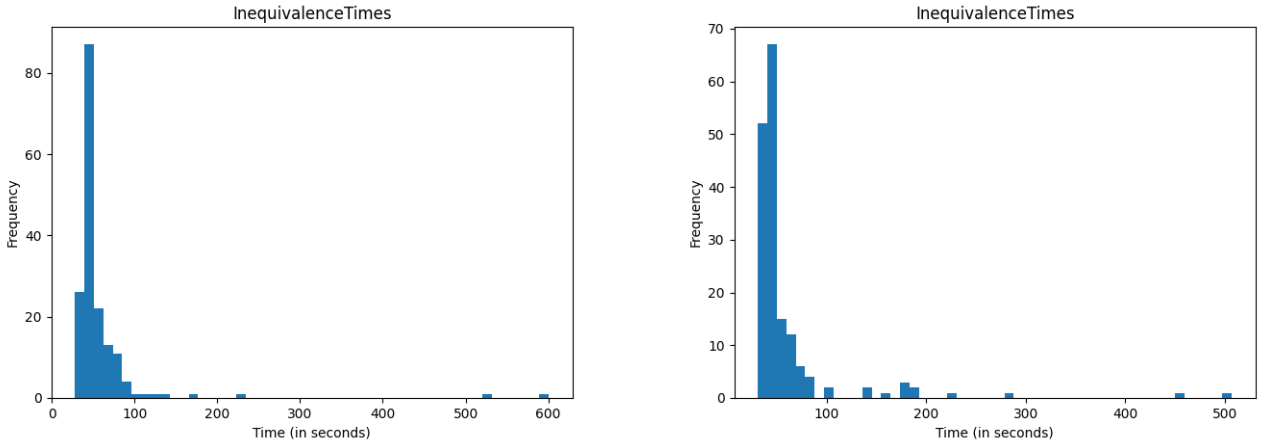
This algorithm is mostly the same as alg-1. It has a call to `computeIneqAlt()` (instead of `computeIneq()`), which runs the DFA on both the input programs and returns a set of counterexamples. It is based on the algorithm described in Section-6.1.

6.3 Experimental Comparison

We have 171 *(func, src, dst)* triplets which were found to be inequivalent through `CFG-Approach`. We tested the same pairs with `TFG-Approach` and compared the two approaches.

- Number of Inequivalences: `CFG-Approach` is able to find all inequivalences, while `TFG-Approach` can find distinguishing inputs for all triplets except one (`TIMEOUT` occured for (`memcpy`, `dietlibc_small`, `musl`) -- discussed in Section-6.4.1)

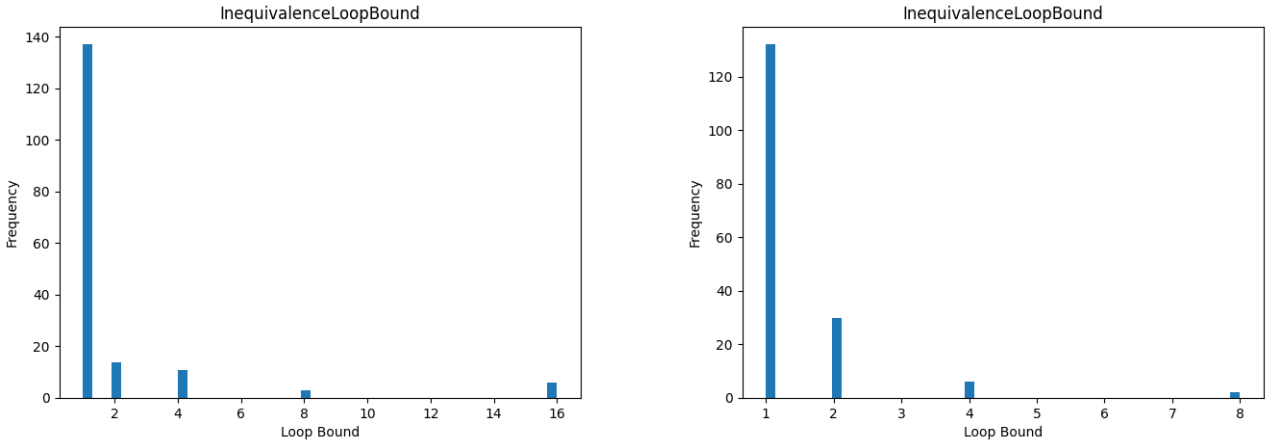
- Time taken: As it can be seen below, both the approaches are able to find counterexamples very quickly. For `CFG-Approach`, number of inequivalences within 100 seconds was 163, while it was 156 for `TFG-Approach`.



`CFG-Approach` `TFG-Approach`

Figure 6.2: Time Taken to find Inequivalences

- Inequivalence Loop Bound frequency: As it can be seen below, most of the counterexamples were found at very low values of loop-bound (ν)



`CFG-Approach` `TFG-Approach`

Figure 6.3: Loop Bounds at which Inequivalences were found

- Inequivalence Unroll Factor frequency and Ranking-Ratio: Since `TFG-Approach` is agnostic to the unroll-factor used for the equivalence checker, we observe unroll-factor values only for the first approach. Majority of the counterexamples were found at very low values of the unroll-factor

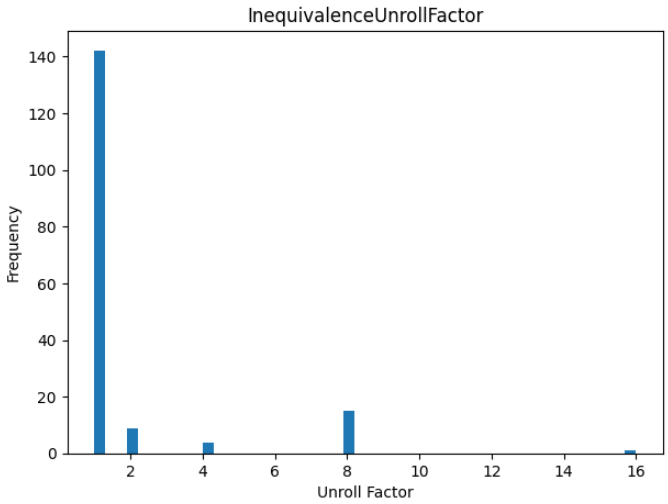


Figure 6.4: `CFG-Approach`: Unroll Factors at which we found inequivalence

We also observe that in `CFG-Approach`, 74 inequivalences were found with the highest ranked *failed-CFG*

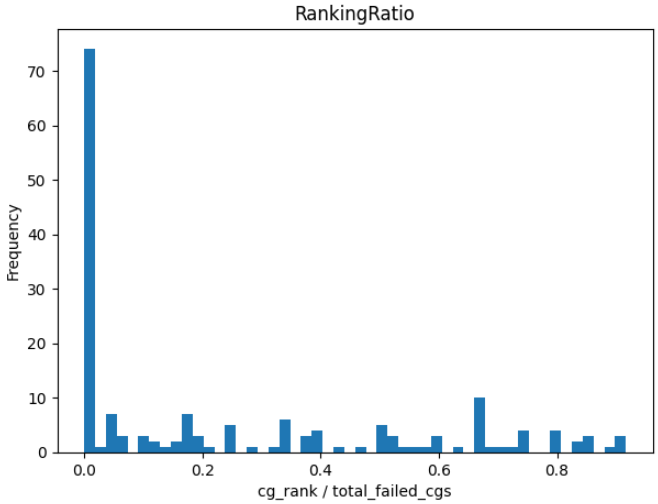


Figure 6.5: `CFG-Approach:` Performance of the Ranking Strategy used

The performance of both the approaches is almost the same on these benchmarks, because most of them were simple, single loop programs. As discussed in the next section, the approaches deviate when we have multiple loops or nested loops.

6.4 Case Studies

6.4.1 Path Explosion in `musl::memcpy`

The `man` page for `memcpy` states the following specification:

> `void *memcpy(void *dest, const void *src, size_t n);`
>
> The `memcpy()` function copies `n` bytes from memory area `src` to memory area `dest`. The memory areas must not overlap.

Depending on the implementation, memory overlap can lead to different behaviours. For instance, if the memories overlap, `src` $<$ `dst`, and the implementation does a forward copy, then the source memory would overwrite itself during the copy.

The `dietlibc_small` implementation consists of a simple loop that does a forward copy from `src` to `dst`, while the `musl` implementation copies multiple bytes chunks at once and has multiple loops. The performance of both the approaches on this benchmark:

- `CFG-Approach`: This approach found a bug at the following parameters:

`unroll-factor`	1
`loop-bound`	1
`ranking-ratio`	2/10
Time Taken	174s

- `TFG-Approach`: This approach timed out at 90 minutes. We start from a low value of ν and double it iteratively. The number of paths collected in the `src-tfg` and `dst-tfg` is tabulated below:

`loop-bound`	`src-tfg` paths	`dst-tfg` paths
1	2	352
2	3	1728
4	5	15360

  As we can see, the number of paths collected in `dst-tfg` increases exponentially, and we still don't have a good coverage of paths. The run timed out while doing a pairwise satisfiability check with these paths.

6.4.2 `CFG-Approach` generally requires a low value of loop-bound

Consider the following input programs:

```
int cyclic_23_src(int n, int m){
    int sum = 0;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            sum++;
        }
    }
    return sum;
}

int cyclic_23_dst(int n, int m){
    int sum = 0;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            if (i == 1 && j == 1) {
                sum += 2;
            } else {
                sum++;
            }
        }
    }
    return sum;
}
```

It is easy to see that $n \geq 2 \wedge m \geq 2$ is a sufficient condition for inequivalence. Below, we try to figure out the minimum value of ν required for both the approaches:

- `TFG-Approach`: The `src-tfg` is shown below.

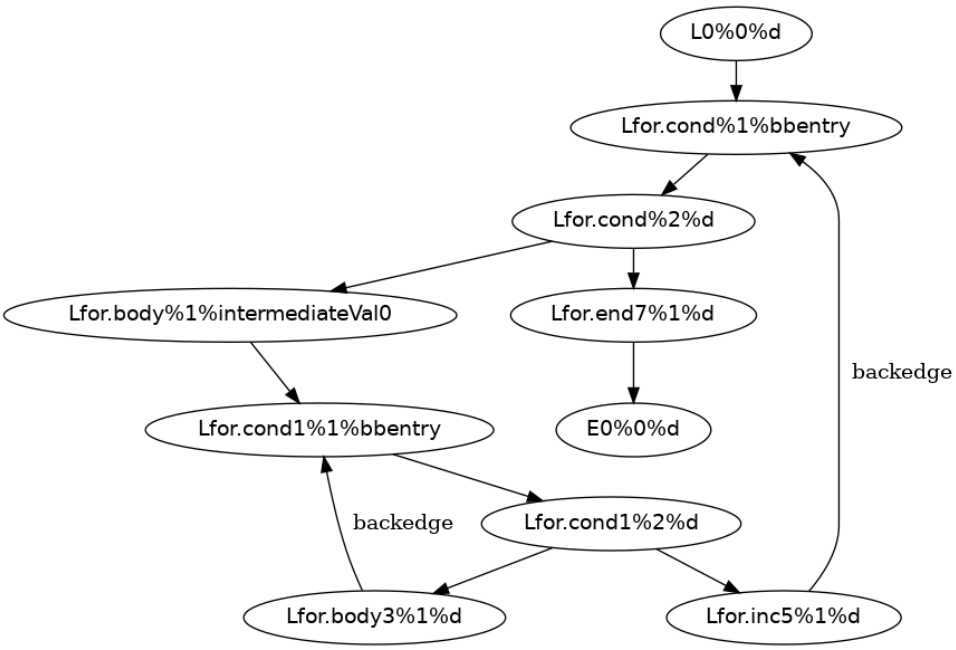


Figure 6.6: `cyclic_23_src`. `dst-tfg` is similar in structure, but has extra branches in the inner loop body

The smallest execution path $\mathtt{p}_{\mathtt{min}}$ in the `src-tfg` that will trigger distinguishing behaviour in both the programs would occur at $n = 2 \wedge m = 2$. This means that the outer loop would be executed twice, and in both the iterations of the outer loop, the inner loop will also be executed twice. The iteration space consists of the points (i, j):

$$\{(0,0),\ (0,1),\ (1,0),\ (1,1)\}$$

The following edge checks the condition for the inner loop:

$$\texttt{Lfor.cond1\%1\%bbentry} \Rightarrow \texttt{Lfor.cond1\%2\%d}$$

This edge will occur 6 times in $\mathtt{p}_{\mathtt{min}}$ (4 times for entering the inner loop, 2 times for exiting it), which is why we need $\nu \geq 6$.

- `CFG-Approach`: The product-CFG for these two programs is shown below.

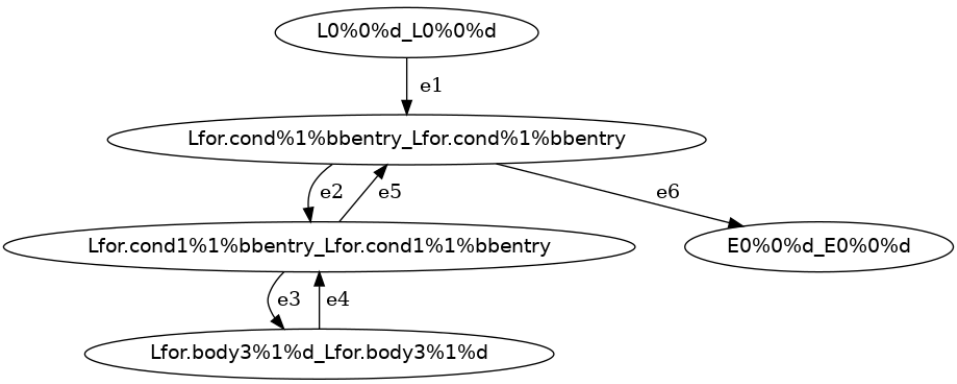


Figure 6.7: The product program has a more compact structure than the individual programs

The smallest execution path $\mathtt{p_{min}}$ in this product-CFG that will trigger distinguishing behaviour is shown below:

$$e_1 \cdot e_2 \cdot (e_3 \cdot e_4)^2 \cdot e_5 \cdot e_2 \cdot (e_3 \cdot e_4)^2 \cdot e_5 \cdot e_6$$

The edges e_3, e_4 have the maximum frequency of 4, which is why we need a $\nu \geq 4$.

Performance of both the approaches on `cyclic_23`:

- `CFG-Approach`: This approach found a bug at the following parameters:

`unroll-factor`	1
`loop-bound`	1
`ranking-ratio`	7/10
Time Taken	38.28s

We are able to find a bug at $\nu = 1$, and not 4, because of failure conditions other than those of observational inequivalence.

- `TFG-Approach`: This approach did not find any distinguishing input at $\nu = 4$, because it did not cover the buggy paths. But, it was able to find a bug at $\nu = 6$ (time taken = 300s). The number of paths collected in the `src-tfg` and `dst-tfg` is tabulated below:

`loop-bound`	`src-tfg` paths	`dst-tfg` paths
1	2	2
2	4	6
4	16	86
6	64	1364

We make the following small modification to `cyclic_23_dst`:

```
int cyclic_23_dst(int n, int m){
    int sum = 0;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            if (i == 1 && j == 2) { // Change here
                sum += 2;
            } else {
                sum++;
            }
        }
    }
    return sum;
}
```

It can be seen that the minimum value of ν required by `CFG-Approach` is 6, while `TFG-Approach` needs a minimum value of 8.

Performance of both the approaches after this modification:

- `CFG-Approach`: This approach found a bug at the following parameters:

`unroll-factor`	1
`loop-bound`	2
`ranking-ratio`	16/35
Time Taken	40.68s

  We are able to find a bug at $\nu = 2$, and not 6, because of failure conditions other than those of observational inequivalence.

- `TFG-Approach`: This approach did not find any distinguishing input at $\nu = 4, 6$ because it did not cover the buggy paths. It was able to find a bug at $\nu = 8$, but it took 242 minutes for the same.
  The number of paths collected in the `src-tfg` and `dst-tfg` is tabulated below:

`loop-bound`	`src-tfg` paths	`dst-tfg` paths
1	2	2
2	4	6
4	16	86
6	64	1364
8	256	21846

6.5 Conclusion

Both the approaches perform equally well on small programs, typically with at most one loop. The number of paths that are collected are also less. In most cases, both the approaches find counterexamples at low values of unroll-factor and loop-bound. On larger programs with multiple loops, `CFG-Approach` performs better than `TFG-Approach`.

`TFG-Approach` is like a brute-force approach which explores all the paths in the individual programs up to a certain bound. This may lead to an exponential blowup in the number of paths collected. Moreover, there could be another quadratic blowup on top of this because we have to check for pairwise satisfiability of paths in *src* and *dst*. The fundamental problem in `TFG-Approach` is that for some of the paths that we collect, their path conditions could be `false`. Moreover, when we do a pairwise checking of paths, not all pair of paths can be taken together. `CFG-Approach` mitigates this problems to some extent because each *product-edge* contains path-sets that execute in a lockstep, which makes this approach more guided.

One situation where `TFG-Approach` could be useful is when the input programs have very poor correlations between them, in which case we might not get any *failed-CFGs*. But since this approach is agnostic to the equivalence checker, it will still collect all the paths.

`CFG-Approach` typically requires a lower value of loop-bound because a product-CFG is a more compact representation of both the input programs. We believe that `CFG-Approach` is heavily dependent on the equivalence checker. I believe that the better the equivalence checker is, the better correlations we can find, the better this approach will be.

Chapter 7

Conclusion, Limitations and Future Work

We have created and described an automatic and sound approach for Inequivalence Checking, based on the Equivalence Checker tool `Counter`. We evaluated our tool on functions from 9 different C library implementations and found a number of inequivalences. We also discussed that some of the bugs could be invalid due to Undefined Behaviour based on function specifications, but those bugs may still be of importance to us. So far, bugs in 11 functions have been reported across 4 different C libraries.

We discussed an alternative approach to inequivalence checking, which was agnostic to the Equivalence Checker and observed that it does not scale to larger and more complex programs. We believe that the problems of Equivalence and Inequivalence Checking are tightly bound, and can help each other. Our approach would improve more and more as the Equivalence Checker improves.

We also compared our approach against state of the art fuzzing techniques (`afl-fuzz`) and found that it requires a lot of manual work to design fuzzing harnesses. Designing harnesses is a tricky task because some part of the input space might remain unexplored, which could lead to a bug being unidentified. Our approach does not have this limitation and is heavily guided.

However, our approach is not without limitations. Our tool does not support Inter-Procedural Inequivalence Checking and has very limited support for Equivalence Checking with function calls (the functions are assumed to be uninterpreted functions). This limits our capability to test more complex and large scale programs. Differential Fuzzing Techniques do not have this limitation. Another limitation is that we still may have a blowup in number of paths as the loop-bound increases. So, choosing an appropriate value of ν automatically is important – a low value may not ensure enough coverage of paths, while a large value may increase the run-time. Currently, we start from low values of ν and increase it iteratively.

Another limitation is that we currently run the equivalence checker to completion first and then run our DFA on the collected failed-CFGs, but this may take a lot of time on larger benchmarks, at a high value of the unroll-factor. A solution for this could be to switch back-and-forth between equivalence and inequivalence checking. It would be crucial to strike a balance between the two.

As for our Future Work, we believe that it is possible to design a more sophisticated and precise transfer function for product-CFG loop body using Region Based Analysis. `Counter`

already infer some affine invariants at each of the product-CFG nodes, and it might be possible to use them to our advantage in the Backward Analysis.

REFERENCES

[1] Shubhani Gupta, Abhishek Rose, and Sorav Bansal. Counterexample-guided correlation algorithm for translation validation. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

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

[3] The GNU C library. https://www.gnu.org/software/libc/.

[4] diet libc - a libc optimized for small size. https://www.fefe.de/dietlibc/.

[5] FreeBSD libc sources. https://github.com/freebsd/freebsd-src.

[6] NetBSD libc sources. https://github.com/NetBSD/src.

[7] OpenBSD libc sources. https://github.com/openbsd/src.

[8] klibc sources. https://mirrors.edge.kernel.org/pub/linux/libs/klibc/2.0/.

[9] musl webpage. https://musl.libc.org/.

[10] Newlib: a C standard library for embedded systems. https://sourceware.org/newlib/.

[11] μClibc: an embedded C library. https://uclibc-ng.org/.

[12] `memccpy` specification. https://pubs.opengroup.org/onlinepubs/9699919799/functions/memccpy.html.

[13] `wcschr` linux man page. https://man.openbsd.org/wcschr.3.

[14] `swab` specification. https://man7.org/linux/man-pages/man3/swab.3.html.

[15] C standard iso/iec 9899. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf.

[16] `memcpy(3)` linux manual page. https://man7.org/linux/man-pages/man3/memcpy.3.html.

[17] `strrchr` linux man page. https://man7.org/linux/man-pages/man3/strchr.3.html.

[18] NetBSD::swab fixed implementation. https://github.com/NetBSD/src/blob/trunk/lib/libc/string/swab.c.

[19] `memchr` linux man page. https://man7.org/linux/man-pages/man3/memchr.3.html.

[20] `strcmp` linux man page. https://man7.org/linux/man-pages/man3/strcmp.3.html.

[21] `strcasecmp` and `strncasecmp` specification. https://pubs.opengroup.org/onlinepubs/009696799/functions/strcasecmp.html.

[22] `wcsrchr` linux man page. https://man.openbsd.org/wcsrchr.3.

[23] American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/.