

RESEARCH REVIEW 2022

**Carnegie  
Mellon  
University**  
Software  
Engineering  
Institute

# Semantic Fidelity of Decompilers

**NOVEMBER 14–16, 2022**

Will Klieber  
Software Security Researcher

# Document Markings

Copyright 2022 Carnegie Mellon University.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other use. Requests for permission should be directed to the Software Engineering Institute at [permission@sei.cmu.edu](mailto:permission@sei.cmu.edu).

Carnegie Mellon® and CERT® are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

DM22-0822

# Introduction

- Goal: Increase software assurance of binary components.
  - Detect and/or repair vulnerabilities.
- Some DoD software doesn't have available source code:
  - Old legacy code
  - Code from contractors
- Protect against cyberattacks that hijack the build process (e.g., SolarWinds attack).
- It's much easier to work with decompiled code than machine code.
- But can the decompilation be trusted? We investigate!

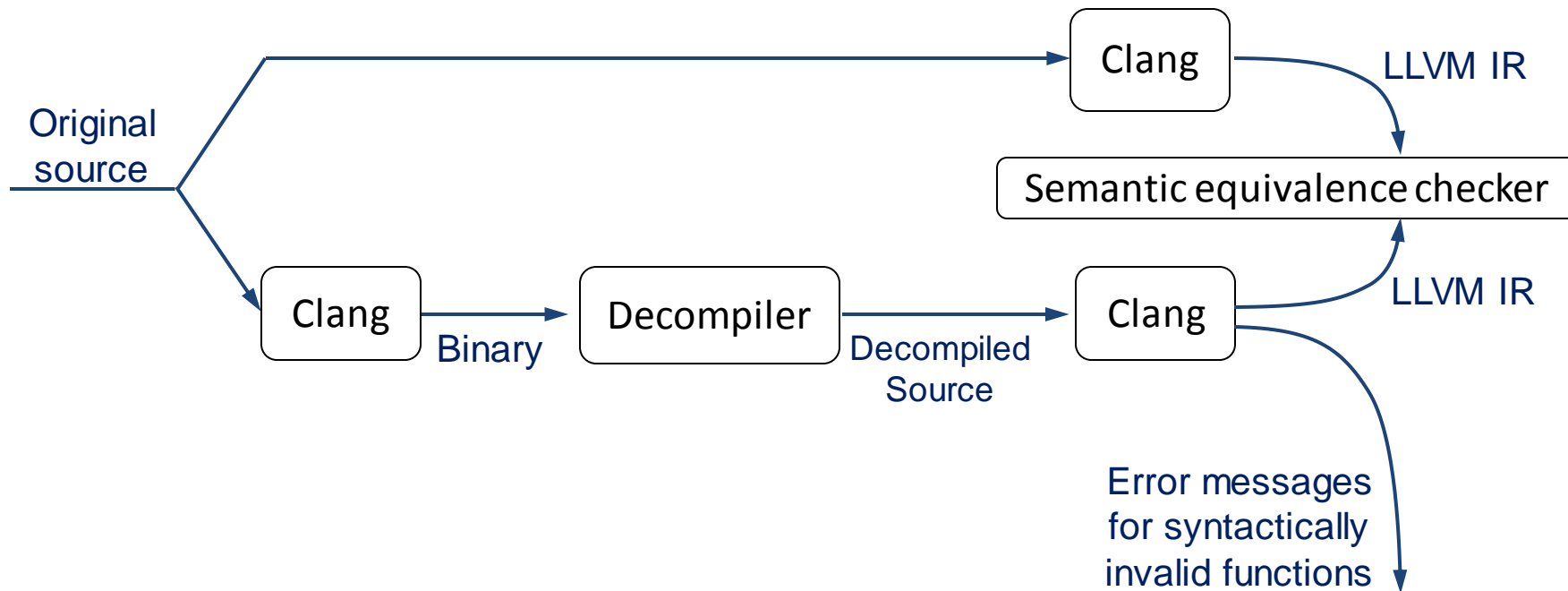
# Overview

- Main technical challenge: Determine which functions in a binary are decompiled to a semantically equivalent form.
- We work with an existing open-source decompiler (Ghidra):
  - Existing decompilers were developed for aiding manual reverse engineering.
  - They were not designed to produce recompilable code.
  - **Gap**: Decompiled code often has semantic inaccuracies and syntactic errors.
- By “semantically equivalent”, we mean: On all possible executions, if the two functions (original and decompiled) are given the same input, they produce the same output and side effects.
- Two ways of evaluating semantic equivalence:
  - Randomized testing (works for all functions, but can miss counterexamples)
  - Formal verification with SeaHorn (cannot handle certain constructs, e.g., floating-point comparisons)

# Previous State of the Art

- Zhibo Liu and Shuai Wang. “How far we have come: testing decompilation correctness of C decompilers.” *ACM Int’l Symposium on Software Testing & Analysis (ISSTA)*, July 2020.
  - Out of 2504 test cases, 93% were correctly decompiled by Ghidra.
  - Tested **synthetic** test cases **without input or nondeterminism**, averaging 243 LoC each.
  - Only **unoptimized** code. No structs, unions, arrays, or pointers.

# Pipeline for Measurement and Evaluation



# Syntactic Validity of Decompiled Code – SPEC2006

This table shows the percentage of decompiled functions that are recompilable (i.e., syntactically valid) C code.

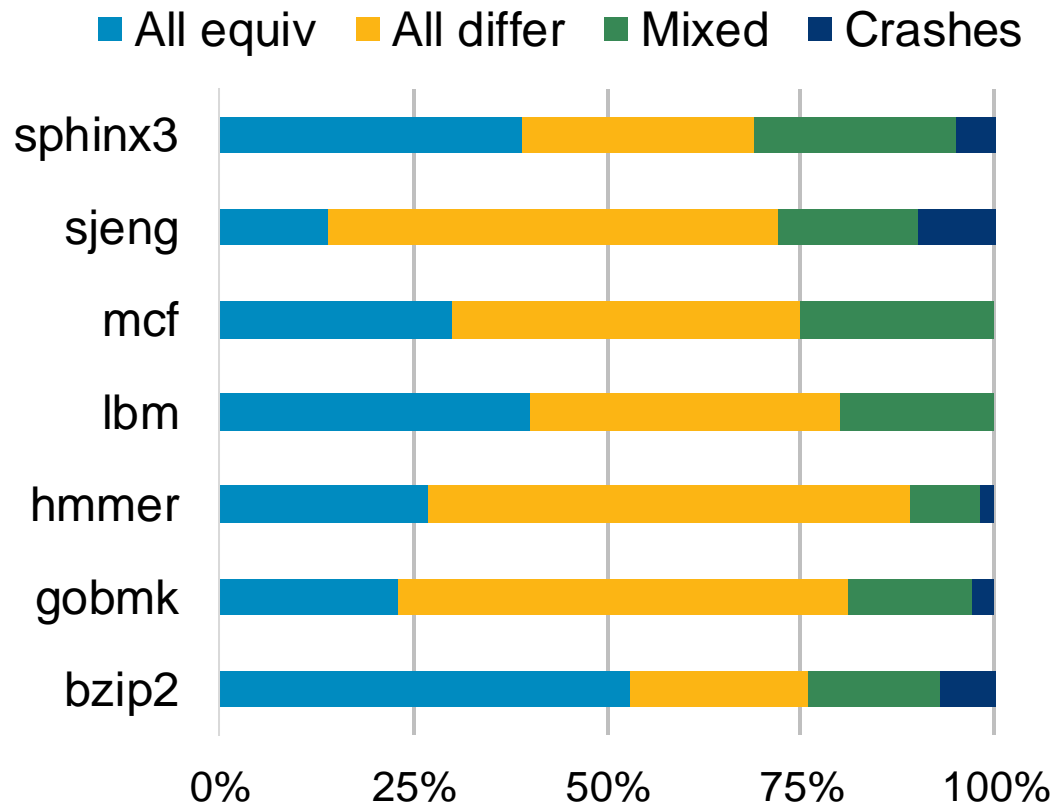
Codebase	Source Functions	FY21 Recomp Success Rate	FY22 Recomp Success Rate
lbn	21	62%	71%
mcf	24	75%	88%
libquantum	94	36%	52%
bzip2	120	67%	84%
sjeng	144	65%	67%
milc	235	57%	78%
sphinx3	370	49%	65%
hmm	657	42%	61%
gobmk	2,693	32%	76%
<b>Average</b>		<b>54%</b>	<b>71%</b>

# Semantic Equivalence Checking of Ghidra on SPEC2006

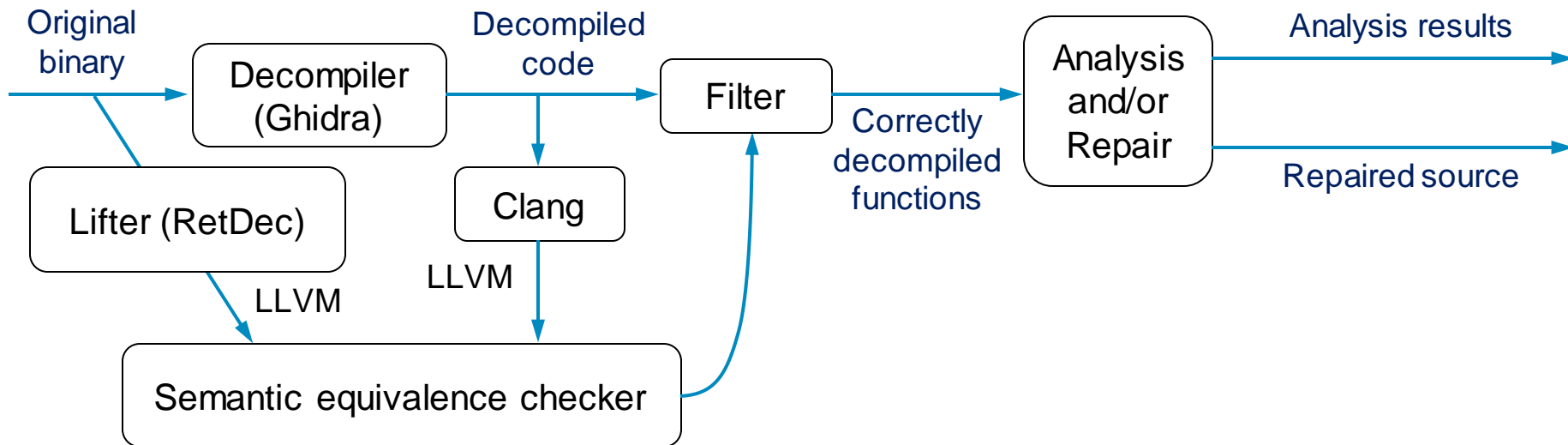
- Tested 2650 functions from SPEC2006 that decompiled to syntactically valid code.
  - This excludes functions that were non-testable:
    - Multiple functions with the same name.
- Ran 1000 trials of each function.
- Over 1500 “autohelper” functions from gobmk – all behaved non-equivalently.
- Of the remaining 1067 functions:
  - 29% of functions behaved equivalently on all runs.
  - 49% of functions behaved non-equivalently on all runs.
  - 18% of functions had some runs that behaved equivalently and some that didn't.  
(Of course, a single non-equivalent run suffices to prove that the functions aren't equivalent.)
  - On 5% functions, our tool failed on at least one run.
    - Failure in loop bounding

# Results by Benchmark Suite

	All equiv	All differ	Mixed	Crashes
sphinx3	39%	30%	26%	6%
sjeng	14%	58%	18%	14%
mcf	30%	45%	25%	0%
lbm	40%	40%	20%	0%
hmmmer	27%	62%	9%	2%
gobmk	23%	58%	16%	3%
bzip2	53%	23%	17%	9%



# Pipeline for Use on “In the Wild” Binaries without Original Source



# Combining Ghidra and RetDec

- **Original hypothesis:** We were expecting that a binary lifter such as RetDec would be able to serve as a reasonably good proxy for semantic ground truth.
- However, it turns out that RetDec isn't any better than Ghidra at semantic fidelity.
- **New hypothesis:** When Ghidra and RetDec agree with each other on the semantics of a function, they are more likely to also agree with the original source.
- We successfully tested this hypothesis on the NASA Core Flight System (cFS) (<https://github.com/nasa/cFS>).
- Technical note: Although we use the term “equivalence,” the relation that our implementation computes actually is not symmetric:
  - If the function from RetDec returns a value but the original function does not, we still count the RetDec function as equivalent to the original source.
  - But if the original-source function returns a value, then for equivalence we require that RetDec also return the same value.

# Results on NASA cFS (total source functions: 1268)

	Ghidra	RetDec
Number of decompiled functions checkable for semantic equivalence:	520	952
Number of functions semantically equivalent to source:	124	229
Percentage of original source functions for which decompiled function is semantically equivalent:	9.8%	18.1%
<b>Probability that a checkable decompiled function is semantically equivalent to original source:</b>	<b>23.8%</b>	<b>24.1%</b>

“Checkable for semantic equivalence” means: the decompiled function is syntactically valid and there is a matched function from the original source.

Number of source functions for which both Ghidra and RetDec produce checkable decompiled functions:	519
Number of functions on which Ghidra and RetDec agree with each other:	115
Number of functions on which Ghidra and RetDec agree with each other and with the original source:	88
<b>Probability that a checkable decompiled function is semantically equivalent to original source when Ghidra and RetDec agree on it:</b>	<b>77%</b>

This analysis was performed on cFS git commit 753ed544d8606dce598e2aabdbc5bd7548ce9138 (Apr 25 12:17:44 2022 -0400)

RESEARCH REVIEW 2022

Semantic Fidelity of Decompilers

# Details of Technical Approach

**Carnegie  
Mellon  
University**  
Software  
Engineering  
Institute

# Problem: Semantic Equivalence with Unavailable Callees

- In the decompiled code, there might be a function call where:
  - the callee is unavailable, and
  - the callee might write to memory
- This complicates our attempts to establish an equivalence between the memories.

## Example:

```
void vithist_frame_windup (vithist_t *vh, int32 frm, ...) {  
    ...  
    vh->frame_start[vh->n_frm] = vh->n_entry;  
    ...  
    vithist_lmstate_reset(vh);  
    ...  
}
```

# Solution: Stricter Notion of Equivalence

- Look for a *structural* equivalence:
  - Check that the sequence of **operations with side effects** is the same.
    - Memory reads, memory writes, function calls
  - Some semantically equivalent pairs are flagged.
  - But every semantically non-equivalent pair is flagged.
- Replace memory reads, memory writes, and function calls with logging.
  - Reads and function calls return a nondeterministic value.  
(Same order of nondeterministic values for original and decompiled)
  - Also log the return value of the original and decompiled functions.
- Execute original and decompiled functions and compare their logs for equivalence.

# Transformation to Test for Structural Equivalence

```
1.  ulong lmclass_get_nclass(long *param_1) {
2.    long lVar1;
3.    ulong uVar2;
4.
5.    lVar1 = *param_1;
6.    uVar2 = 0;
7.    while (lVar1 != 0) {
8.        uVar2 = (ulong)((int)uVar2 + 1);
9.        lVar1 = *(long *)(lVar1 + 0x10);
10.   }
11.   return uVar2;
12. }
```

```
1.  ulong lmclass_get_nclass(long *param_1) {
2.    long lVar1;
3.    ulong uVar2;
4.
5.    lVar1 = read_mem_long(param_1);
6.    uVar2 = 0;
7.    while (lVar1 != 0) {
8.        uVar2 = (ulong)((int)uVar2 + 1);
9.        lVar1 = read_mem_long((long *)(lVar1 + 0x10));
10.   }
11.   return retval_ul(uVar2);
12. }
```

# Example of Log

## Original

```
static void setExit ( Int32 v )
{
    if (v > exitValue) exitValue = v;
}
```

## Decompiled

```
void setExit(int param_1)
{
    if (exitValue < param_1) {
        exitValue = param_1;
    }
    return;
}
```

**ORIGINAL****DECOMPILED**

READ ADDR 0000270f

| READ ADDR 0000270f

WRITE ADDR 0000270f

| WRITE ADDR 0000270f

WRITE VALUE 0000008d

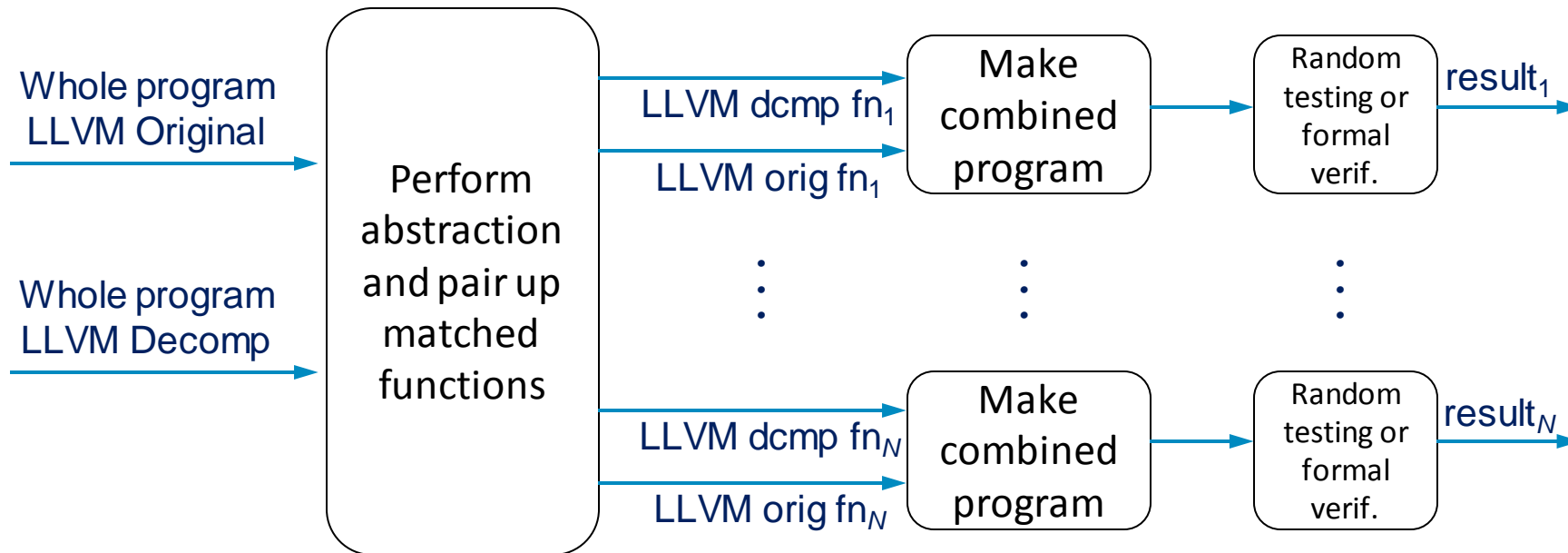
| WRITE VALUE 0000008d

PASS

# Bounded Semantic Equivalence Checking with Logging

- Comparing the logs is impractical for existing verification tools in the unbounded case.
  - (at least for the straightforward approach of non-interleaved execution)
- Bound the number of execution steps:
  - Unroll loops for a fixed number of iterations.
  - Problem: Loops can potentially be structured differently in decompiled vs. the original  $\implies$  can give false counterexamples to equivalence.

# Details of Semantic Equivalence Checker



# Formal Verification and Randomized Testing

- SeaHorn can sometimes formally verify equivalence, but it can't handle some common constructs (e.g., branching on result of floating-point comparison).
- Our experiments in this project have mostly used randomized testing instead.
  - We initialize an array of random values (biased toward small values) and run both the original function and the decompiled function with this array.
  - Arguments to functions are also chosen randomly.

# Conclusion

- Decompilers have potential to greatly help with software assurance for binary code.
- But existing decompilers often aren't semantically faithful.
- Requiring that two decompilers agree on semantics can greatly increase confidence.
  - (E.g., requiring RetDec and Ghidra to agree raises success rate from 24% to 77% on NASA cFS.)
- Our tool can also help measure improvements to decompiler semantic accuracy.
- If you are interested in trying our tool, please contact us.
  - Currently the tool is Distro D — it can be distributed only to DoD and contractors. But we are seeking approval to distribute it more widely.

# Team Photos



**Will Klieber**  
Software Security Engineer



**David Svoboda**  
Software Security Engineer



**Mike McCall**  
Software Security Engineer



**Lori Flynn**  
Software Security Engineer



**Ruben Martins**  
Assistant Research  
Professor, CMU