Carnegie Mellon University Software Engineering Institute

Semantic Equivalence Checking of Decompiled Binaries

NOVEMBER 14–16, 2022

Will Klieber
Software Security Researcher

©2022

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

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.

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.
  - Enable the DoD to find and fix potential vulnerabilities
- We estimate that the equivalent of at least 100 million LOC of binary-only software is in use by DoD.
  - Old legacy code
  - Code from contractors
- Protect against cyberattacks that hijack the build process (e.g., SolarWinds attack).
  - Analysis of the binary executable can find injected malware not present in the source code.
- It’s much easier to work with decompiled code than machine code.
- But can the decompilation be trusted? We investigate!

©2022
[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

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)

©2022

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

Previous State of the Art

Carnegie Mellon University Software Engineering Institute

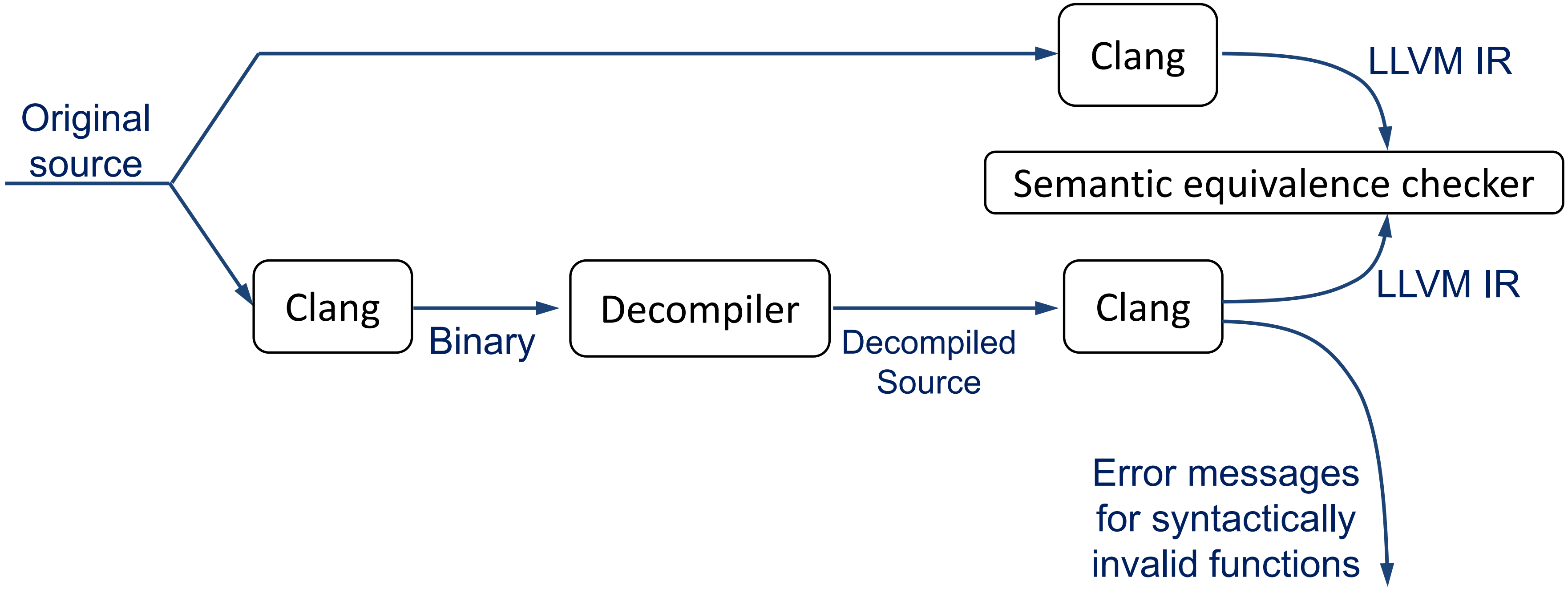
- Zhibo Liu and Shuai Wang. "How far we have come: testing decompilation correctness of C decompilers." *ACM Int'l Symposium on Software Testing & Analysis (ISSTA),* July 2020.
  - Out of 2504 test cases, 93% were correctly decompiled by Ghidra.
  - Tested **synthetic** test cases **without input or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

©2022

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

Pipeline for Measurement and Evaluation

Carnegie Mellon University Software Engineering Institute



©2022

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

Carnegie Mellon University Software Engineering Institute

Syntactic Validity of Decompiled Code – SPEC2006

Codebase	Source Functions	Recompilation Success Rate
lbm	21	71%
mcf	24	88%
libquantum	94	52%
bzip2	120	84%
sjeng	144	67%
milc	235	78%
sphinx3	370	65%
hmmer	657	61%
gobmk	2,693	76%
Average		**71%**

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

©2022

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

Semantic Equivalence Checking of Ghidra on SPEC2006

- Tested 1157 functions from SPEC2006 that decompiled to syntactically valid code.
  - Excludes 1500 autogenerated functions from gobmk
  - Excludes functions that were non-testable:
    - Multiple functions with the same name.
- Ran 1000 trials of each function.
- Results:
  - 35% of functions behaved equivalently on all runs.
  - 30% of functions behaved non-equivalently on all runs.
  - 31% 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 3% functions, our tool failed on at least one run.
    - Failure in loop bounding

©2022

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

Carnegie Mellon University Software Engineering Institute

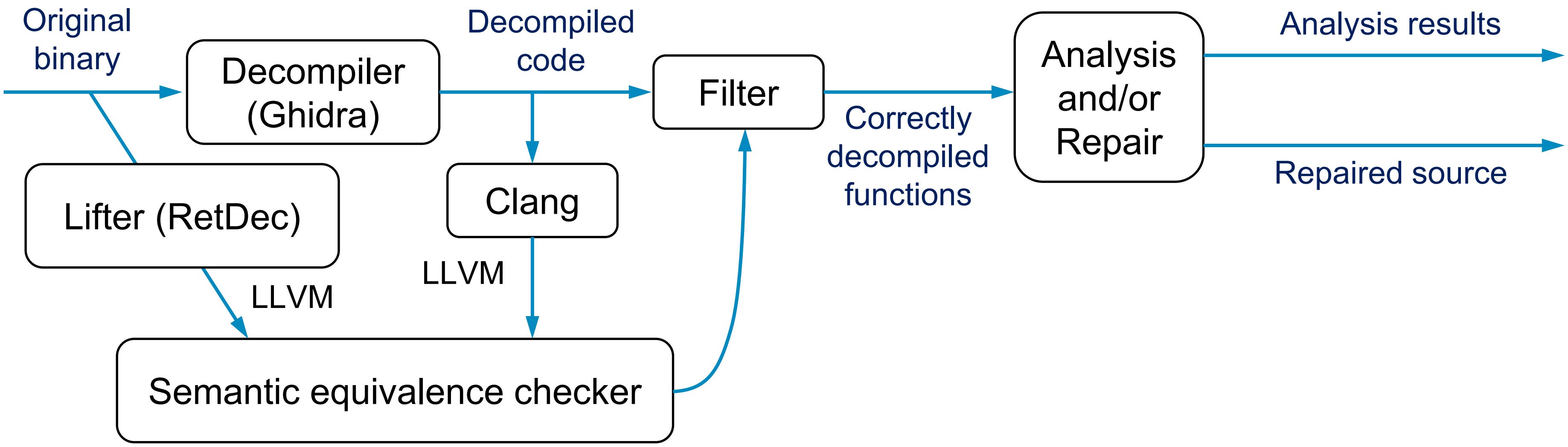
Semantic Equivalence – Results by Benchmark Suite

	All equiv	All differ	Mixed	Tool fail
libquantum	54%	34%	9%	3%
milc	49%	33%	16%	6%
sphinx3	48%	31%	19%	2%
bzip2	43%	30%	25%	4%
lbm	40%	47%	7%	7%
sjeng	29%	48%	14%	10%
mcf	26%	47%	21%	5%
gobmk	26%	15%	56%	1%
hmmer	22%	61%	13%	4%
OVERALL	**35%**	**30%**	**31%**	**3%**

©2022
[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

Pipeline for Use on Binaries without Original Source

Carnegie Mellon University Software Engineering Institute



©2022

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

Carnegie Mellon University Software Engineering Institute

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.

©2022
[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

Carnegie Mellon University Software Engineering Institute

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%
Probability that a checkable decompiled function is semantically equivalent to original source:	**23.8%**	**24.1%**

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
Probability that a checkable decompiled function is semantically equivalent to original source when Ghidra and RetDec agree on it:	**77%**

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

This analysis was performed on cFS git commit `753ed54` (Apr 25, 2022)

©2022

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

Semantic Fidelity of Decompilers

Details of Technical Approach

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

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);
    ...
}
```


©2022
[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

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.

©2022

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

Transformation to Test for Structural Equivalence


```
ulong lmclass_get_nclass(long *param_1) {
  long lVar1;
  ulong uVar2;

  lVar1 = *param_1;
  uVar2 = 0;
  while (lVar1 != 0) {
    uVar2 = (ulong)((int)uVar2 + 1);
    lVar1 = *(long *)(lVar1 + 0x10);
  }
  return uVar2;
}
```

```
ulong lmclass_get_nclass(long *param_1) {
  long lVar1;
  ulong uVar2;

  lVar1 = read_mem_long(param_1);
  uVar2 = 0;
  while (lVar1 != 0) {
    uVar2 = (ulong)((int)uVar2 + 1);
    lVar1 = read_mem_long((long *)(lVar1 + 0x10));
  }
  return retval_ul(uVar2);
}
```


©2022
[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

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
```


©2022

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

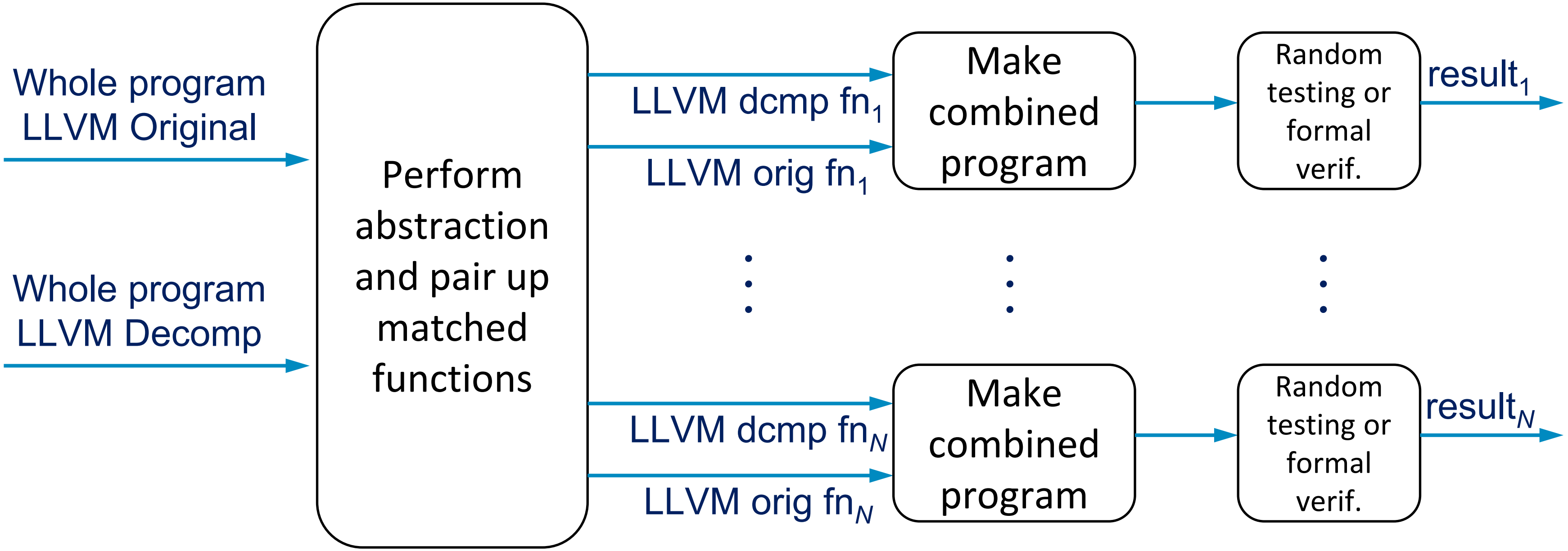
Bounded Semantic Equivalence Checking with Logging

- 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 ==> can give false counterexamples to equivalence.

©2022
[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

Details of Semantic Equivalence Checker

Carnegie Mellon University
Software Engineering Institute



©2022
[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

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.

©2022

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

Conclusion

Carnegie Mellon University Software Engineering Institute

- 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 (`info@sei.cmu.edu`).
  - 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.

©2022
[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

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

Contact us at `info@sei.cmu.edu`

©2022

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.