Carnegie Mellon University
Software Engineering Institute

Semantic-Equivalence Checking to Determine Decompilation Fidelity at the Function Level

Will Klieber
David Svoboda

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

Copyright 2021 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.

DM21-0590

Overview

- Goal: Increase software assurance of components available only in binary form.
  - Decompile and perform static analysis on decompiled code.
  - Make localized repairs to functions of the binary.
- We will adapt an existing open-source decompiler (Ghidra) to produce decompiled code suitable for static analysis and/or repair.
  - 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.
- We aim to develop a tool for:
  - determining which individual functions have been correctly decompiled,
  - recombining repaired functions with the original binary files.
- This work, if successful, will enable DoD to find and fix potential vulnerabilities in binary code that might otherwise be cost-prohibitive to investigate or repair manually.

© 2021 Carnegie Mellon University
[Distribution Statement A] Approved for public release and unlimited distribution.

Overview (continued)

A perfect decompilation of the entire binary isn't necessarily required to get significant benefit, as long as enough relevant functions can be correctly decompiled.

Main contributions of our work:

- We will develop a semantic-equivalence checker to determine which individual functions are correctly decompiled.
- We will address low-hanging fruit for improving the syntactic and semantic correctness of decompiler output.

© 2021 Carnegie Mellon University
[Distribution Statement A] Approved for public release and unlimited distribution.

Example of function that doesn't decompile correctly

```
1.  int main() {
2.    char* opt_name[2] = { "Option A", "Option B" };
3.    puts("Enter 'A' or 'B' and press enter.");
4.    int input = getchar();
5.    if ((('B' - input) >> 1) != 0) {puts("Bad choice!"); exit(1);}
6.    puts(*(opt_name + (input - 'A'))); /* same as opt_name[input – 'A'] */
7.  }
```

The problem is that the compiler (with "-O1") changes "`opt_name + (input - 'A')`" to "`(opt_name - 'A') + input`" and replaces "`(opt_name - 'A')`" with a numeric constant. Ghidra decompiles this function to:

```
...
iVar1 = getchar();
...
puts(*(char **)((long)iVar1 * 8 + 0x600bf8)); /* the size of char* is 8 bytes */
```


© 2021 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Example of code that we will be able to handle

Original Code

```
void insertion_sort(unsigned int* A, size_t length) {
    for (size_t j = 1; j < length; ++j) {
        unsigned int key = A[j];
        /* insert A[j] into the sorted sequence A[0..j-1] */
        size_t i = j - 1;
        while (i >= 0 && A[i] > key) {
            A[i + 1] = A[i];
            --i;
        }
        A[i + 1] = key;
    }
}
```

Decompiled Code

```
void insertion_sort(long param_1, ulong param_2) {
  uint uVar1; ulong uVar2; ulong local_18; ulong local_10;
  local_18 = 1;
  while (local_18 < param_2) {
    uVar1 = *(uint *)(param_1 + local_18 * 4);
    uVar2 = local_18;
    while (local_10 = uVar2 - 1,
           uVar1 < *(uint *)(param_1 + local_10 * 4))
    {
      *(undefined4 *)(param_1 + uVar2 * 4) =
         *(undefined4 *)(param_1 + local_10 * 4);
      uVar2 = local_10;
    }
    *(uint *)(uVar2 * 4 + param_1) = uVar1;
    local_18 = local_18 + 1;
  }
}
```


© 2021 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

State of the Art – Recompilation of decompiled code

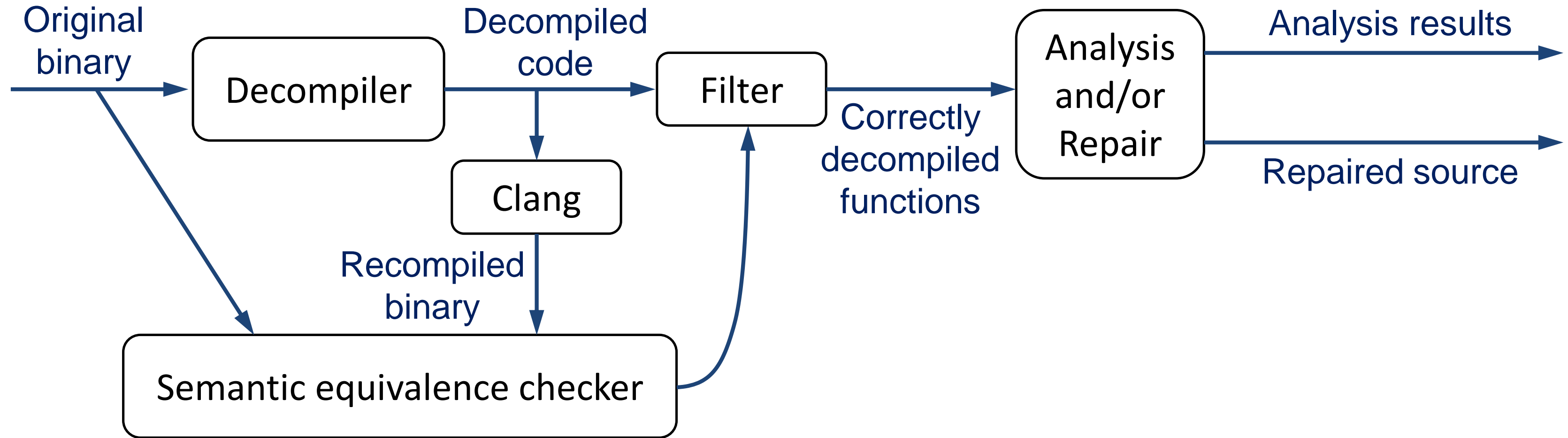
- Zhibo Liu and Shuai Wang. "How far we have come: testing decompilation correctness of C decompilers." *ACM Int'l Symposium on Software Testing & Analysis (ISSTA),* July 2020.
  - Tested **synthetic** code **without input or nondeterminism**.
  - Ghidra: out of 2504 test cases (averaging around 250 LoC), 93% were correctly decompiled.
  - Only **unoptimized** code. No structs, unions, arrays, or pointers.

Tool Name	# of EMI Variants	Decompil-ation Failures	Recompil-ation Failures	Decompilation Defects	
				Csmith Output	EMI Variant
IDA-Pro	3786	0	208	1	69
JEB3	2510	1	13	30	9
RetDec	907	0	187	346	380
Radare2/Ghidra	2504	0	0	53	126
Total	9707	1	408	430	584

© 2021 Carnegie Mellon University

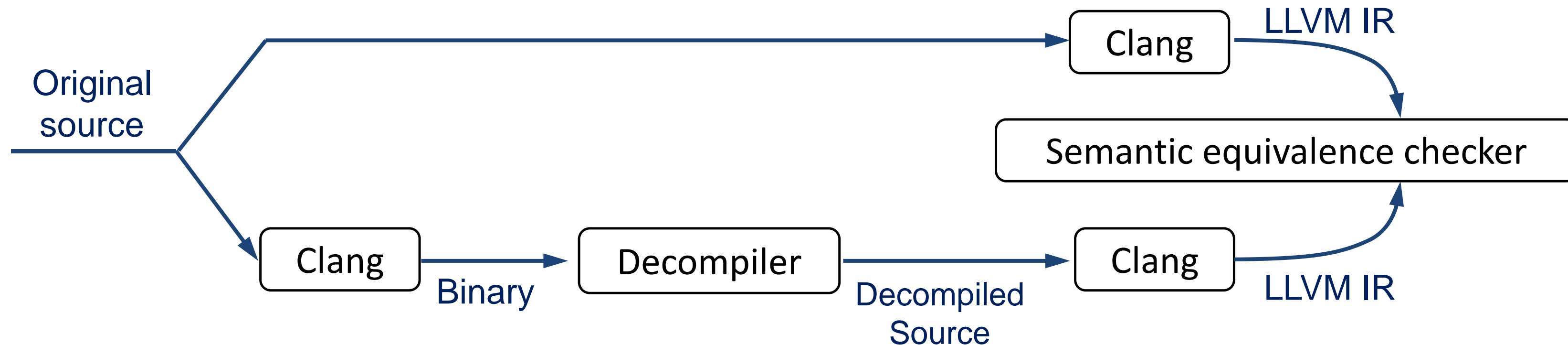
[Distribution Statement A] Approved for public release and unlimited distribution.

Ideal pipeline (for use on in-the-wild binaries)



© 2021 Carnegie Mellon University
[Distribution Statement A] Approved for public release and unlimited distribution.

FY21 pipeline (for measurement and evaluation)



© 2021 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Platform Information

- 64-bit Ubuntu 18.04
- Ghidra 9.1.2
- Java (openjdk 11.0.10)
- GCC 7.5

All running in a Docker container on a Mac laptop.

© 2021 Carnegie Mellon University
[Distribution Statement A] Approved for public release and unlimited distribution.

Decompilation Process

Ghidra takes an object (`.o`) file or library (`.a` or `.so`) file or executable (Windows/Mac/Linux).

If `${GHIDRA}` points to Ghidra installation, and `${BINARY}` points to binary file to decompile:

```
mkdir tmp
${GHIDRA}/support/analyzeHeadless  ./tmp  automatic \
    -import ${BINARY}  -postScript MyDecompile.java
```

produces three output files:

File	Purpose
`a.out.h`	Header file with all function declarations including all included declarations, like **`puts()`**
`a.out.c`	File with all function implementations
`a.out.sym`	File with all declared symbols

`gcc a.out.c`
should recompile everything (in theory)

© 2021 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Postprocessing

Python script, to be run after Ghidra:

- Splits **`a.out.c`** into many files, one per function
- All files go into a newly-created **`src`** directory
- Fixes simple errors
- Does not alter original input files
- Independent & Ignorant of Ghidra

© 2021 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Ghidra output affected by:

- Compiler Optimization level
  - We used **`-O2`**
- Compiler Debug level
  - We used **`-g0`**
- Executable platform (Windows/Linux/Mac)
  - We used Linux (and tested Mac & Windows)
  - Linux executable produces many "trivial" functions (that only call similarly-named functions)

© 2021 Carnegie Mellon University
[Distribution Statement A] Approved for public release and unlimited distribution.

Code Recompilation

The table shows the percentage of source-code functions that are extracted as recompilable (i.e., syntactically valid) C code.

SPEC 2006 Benchmarks

Project	Source Functions	Recomp Functions	Percent
dos2unix	40	17	43%
jasper	725	377	52%
lbm	21	13	62%
mcf	24	18	75%
libquantum	94	34	36%
bzip2	119	80	67%
sjeng	144	93	65%
milc	235	135	57%
sphinx3	369	183	50%
hmmer	552	274	50%
gobmk	2,684	853	32%
hexchat	2,281	1,106	48%
git	7,835	3,032	39%
ffmpeg	21,403	10,223	48%
Average			**52%**

Causes of failure:

- Function signature mismatch
- Struct definition mismatch
- Measurement error: inlined functions, unused functions eliminated as dead code, etc. (Significant for hexchat, but zero or small effect for others.)

© 2021 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Types of syntactic errors

Count	Error type
609	Request for member in something not a structure or union
706	Invalid operands to binary operator
910	Other
2,972	Use of undeclared identifier
1,224	void value not ignored as it ought to be
1,153	too many arguments to function
3,434	too few arguments to function
11,008	**Total**

© 2021 Carnegie Mellon University
[Distribution Statement A] Approved for public release and unlimited distribution.

Optimization Level: Mitigations: "trivial" functions

Linux executable produces many "trivial" functions (that only call similarly-named functions)

We suspect these functions are actually entries in the Global Offset Table (GOT), which is used on Linux for runtime linking.

For example, any call to **`puts(3)`** causes this function to be produced:

```
int puts(char *__s)
{
  int iVar1;

  iVar1 = puts(__s);
  return iVar1;
}
```

We prune such functions.

© 2021 Carnegie Mellon University
[Distribution Statement A] Approved for public release and unlimited distribution.

Ghidra Bugs

- Extra Typedefs
- Static Variable Declarations
- Structs

© 2021 Carnegie Mellon University
[Distribution Statement A] Approved for public release and unlimited distribution.

Ghidra Bugs: Extra Typedefs

When Ghidra creates a struct, it also adds this line:

```
typedef struct foo foo, *Pfoo;
```

But consider the POSIX **`stat(2)`** function:

```
int stat(const char *restrict pathname,
         struct stat *restrict statbuf);
```

When Ghidra decompiles any code that calls this function, it produces:

```
int stat(const char*,struct stat*);  /* stat is a function */
typedef struct stat stat, *Pstat;    /* stat is a typedef  */
```

The same problem occurs with the POSIX **`sigaction(2)`** and **`sysinfo(2)`** functions/structs.

© 2021 Carnegie Mellon University
[Distribution Statement A] Approved for public release and unlimited distribution.

Ghidra Bugs: Static Variable Declarations

Ghidra does not include static variables in **`a.out.h`** or **`a.out.c`**

- This was fixed by capturing symbols (in Java plugin):
  - Output in **`a.out.sym`**
- Python postprocessing code adds them all to **`a.out.h`**, as type **`undefined`**.

Ghidra's "Symbol Table" window does contain actual types for these variables.

- But we could not find any simple API for accessing those datatypes.
- Also no initial values

Example static variable declaration:

```
int x = 5;
```

becomes:

```
undefined x;
```


© 2021 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Plans

Study Ghidra, in the hopes of improving quality of decompiled code, and sending patches back to NSA:

- Find/build an API to access types of static variables. Add these types to `a.out.h`
- Function Signature Unification
- Type Unification
  - Identify pointer types
    - e.g., an integer that is only cast to a pointer & dereferenced should really be a pointer.
- Struct Unification
- Eliminate extra typedefs (e.g., `stat`/`Pstat`)
- Identify structs/arrays based on calls to `malloc()`

Question: *Do we try to recreate the code as originally written or do we try to make code as high-level (or readable) as possible? (while preserving semantics)*

© 2021 Carnegie Mellon University
[Distribution Statement A] Approved for public release and unlimited distribution.

Plans: Function Signature Unification

- Many function calls disagree with the number & types of arguments to a function.
  - Sometimes the function declaration also disagrees with the original source code!
- Make sure all calls to a function agree in argument number / types with each other & the function declaration.
  - When # of arguments disagree, assume greatest # of arguments is correct.
  - OR
  - When # of arguments disagree, assume call site is correct

© 2021 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Function signatures: Example problem

- Consider a function call chain: fn1 calls fn2, which calls fn3.
- Assume a calling convention in which arguments are passed via registers.
- Assume fn2 passes its 2nd argument to fn3 as its 2nd argument, e.g.:

```
int fn2(int arg1, int arg2) {return fn3(arg1, arg2);}
```

- Then determining the number of arguments of fn2 cannot be done by looking only at the binary code of fn1 and fn2; the binary code of fn3 is required.

To fix this we would need to start with leaf functions (i.e., functions that don’t call any other functions in our code).

- We’ll need something more complex for recursive functions.

© 2021 Carnegie Mellon University
[Distribution Statement A] Approved for public release and unlimited distribution.

State of the Art – Semantic Equivalence Checking

- Rahul Sharma, et al., “Data-driven equivalence checking” (OOPSLA 2013):
  - Tries to prove whether two loops in x86 machine code are semantically equivalent, by inferring simulation relations from test runs and using an SMT solver to prove the simulation relation.
- Berkeley Churchill, et al. “Semantic program alignment for equivalence checking” (PLDI 2019):
  - Tries to prove whether two functions in x86-64 are equivalent by constructing an aligned product program.
- Manjeet Dahiya and Sorav Bansal. “Black-Box Equivalence Checking Across Compiler Optimizations” (APLAS ’17).
- David Ramos and Dawson Engler. “Practical, Low-Effort Equivalence Verification of Real Code” (CAV ’11).

© 2021 Carnegie Mellon University
[Distribution Statement A] Approved for public release and unlimited distribution.

Using SeaHorn to prove equivalence

- We are using the SeaHorn verification framework as the backend for our semantic equivalence checker. SeaHorn in turn uses the Z3 SMT solver.
- We currently can check semantic equivalence on loop-free leaf functions with arbitrary pointer aliasing.
- A question we ask SeaHorn is: Does the decompiled function have the *same effect* on the memory as the original function? I.e., for an arbitrary given initial memory state, does the decompiled function transition to the same final memory state as the original function?
- Although conceptually we consider the entire memory space, the *representation* of memory can be rather small: we need only one symbolic memory address for each memory access (read or write) in the original and decompiled functions.
  - When reading via a symbolic memory address, the SMT solver must consider previous writes to different symbolic addresses that are potential aliases of the address that is being read.

© 2021 Carnegie Mellon University
[Distribution Statement A] Approved for public release and unlimited distribution.

Loops and recursion

Now we are working on handling loops and recursion.

Hypothesis:

- We can use information from Ghidra to establish the correspondence between:
  - variables in the binary (whether they're in CPU registers or on the stack),
  - and variables in the decompiled code.
- Likewise, we can map loops, basic blocks, and function call-sites in the original binary to the corresponding loops, basic blocks, and function call-sites in the decompiled code.
- We check semantic equivalence of functions by:
  1. identifying correspondence between basic blocks in the original binary and the decompiled code
  2. proving semantic equivalence of the straight-line code in the basic blocks

© 2021 Carnegie Mellon University

[Distribution Statement A] Approved for public release and unlimited distribution.

Cutpoints

- Complication: The correspondence between constructs (basic blocks, variables, etc.) in original code and constructs in the decompiled code is sometimes more complicated than a simple one-to-one equality mapping.
- Solution: Use the same approach taken by Rahul Sharma et al. in their paper "Data-driven equivalence checking" (OOPSLA 2013): Construct a *simulation relation* using *cutpoints* and linear equalities:
  - A *cutpoint* is a pair of program points, one in the original code and one in the decompiled code.
  - Cutpoints are chosen to divide the loops into loop-free segments.
  - We aim to prove semantic equivalence by showing that:
    - Executions of the original and decompiled functions move together from cutpoint to cutpoint.
    - At each cutpoint, the value of each variable and symbolic memory cell in the decompiled code can be written as a linear combination of those in the original code.
      - If inexpressible as a linear combination, report that equivalence checking failed.
    - At the final cutpoint(s) (i.e., the function exit point(s)), the return value and the memory contents are same in the original and decompiled functions.

© 2021 Carnegie Mellon University
[Distribution Statement A] Approved for public release and unlimited distribution.