Carnegie Mellon University
Software Engineering Institute

Semantic Fidelity of Decompilers

Will Klieber
David Svoboda

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.

DM22-0583

Overview

- We adapt 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.
- Measurement of semantic fidelity: Percentage of decompiled functions that are semantically equivalent to the corresponding original functions.
- By “semantically equivalent”, we mean that, on all possible executions, if the two functions (original and decompiled) are given the same input, they produce the same output and side effects.
  - Randomized testing / fuzzing
  - Formal verification with SeaHorn

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 */
```

Incorrect types don't prevent semantic equivalence

Original Code

```
void insertion_sort(unsigned int* A, size_t len) {
    for (size_t j = 1; j < len; ++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;
    }
}
```

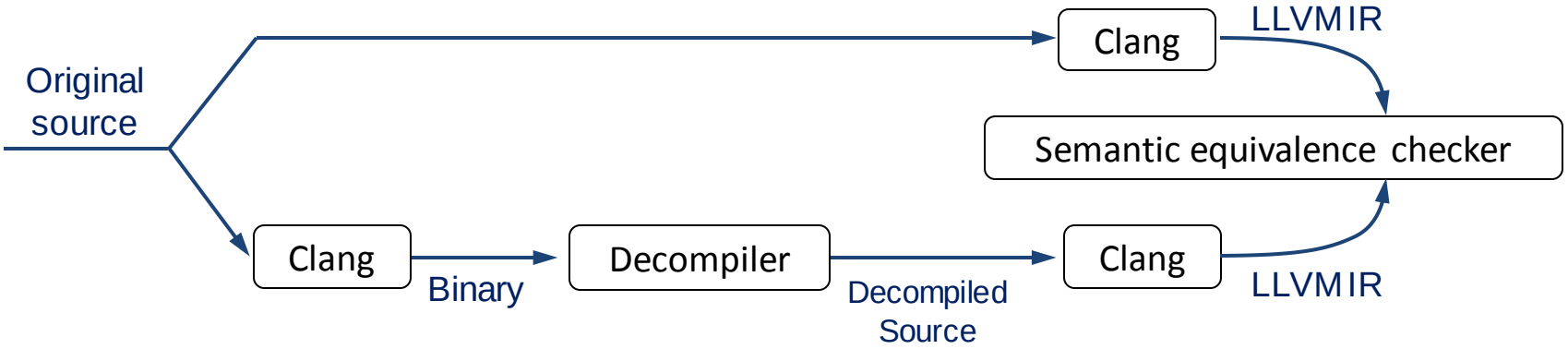
Decompiled Code

```
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;
  }
}
```

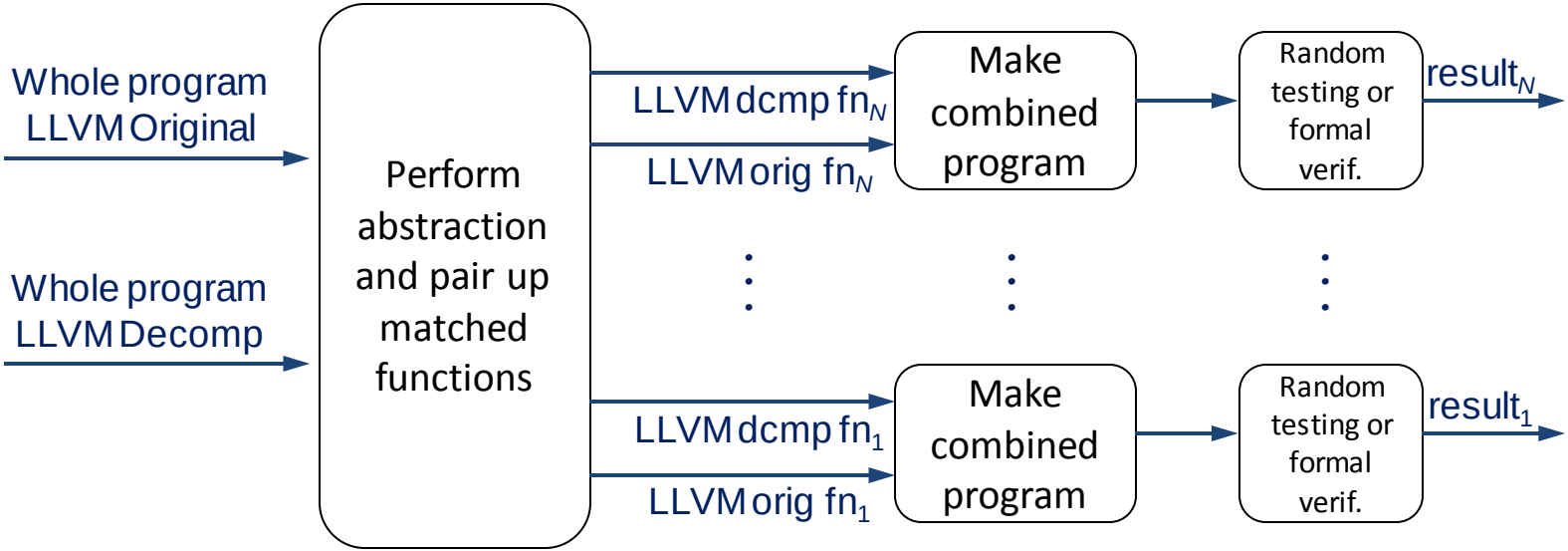
Previous State of the Art

- 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.
  - Tested **synthetic** test cases **without input or nondeterminism,** averaging 243 LoC each.
  - Only **unoptimized** code. No structs, unions, arrays, or pointers.
  - Out of 2504 test cases, 93% were correctly decompiled.
- Existing research in checking semantic equivalence at the machine-code level has focused on checking that compiler optimizations preserve semantics.

Semantic Equivalence Pipeline



Details of semantic equivalence checker



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.
  - Can produce false positives: some semantically equivalent pairs are flagged.
  - But no false negatives: every semantically non-equivalent pair is flagged.
- Replace mem reads, mem 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. }
```

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 are sometimes structured differently in decompiled vs the original ==> can give false counterexamples to equivalence.

Current status

- We don't yet have formal verification working.
- So, we are doing 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.
- Evaluation benchmark suite: SPEC2006.
  - Selected a sample of 629 functions from SPEC2006 that decompiled to syntactically valid code.
  - Ran 1000 trials of each function.
  - Results:
    - 28% of functions behaved equivalently on all runs.
    - 48% of functions behaved non-equivalently on all runs.
    - 19% of functions had some runs that behaved equivalently and some that didn't.
    - On 6% functions, our tool crashed (due to a bug in loop bounding).

Platform Information

- 64-bit Ubuntu 18.04
- Ghidra ~~9.1.2~~ 10.1.4
- Java (openjdk 11.0.10)
- Clang 6.0

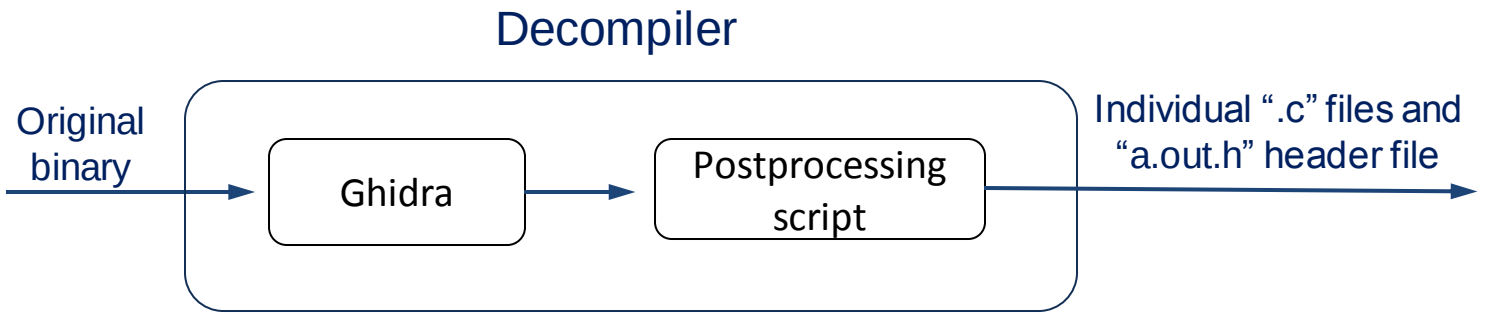
All running in a Docker container on a Mac laptop.

Postprocessing Ghidra Output

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

Postprocessing Ghidra Output (cont.)



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

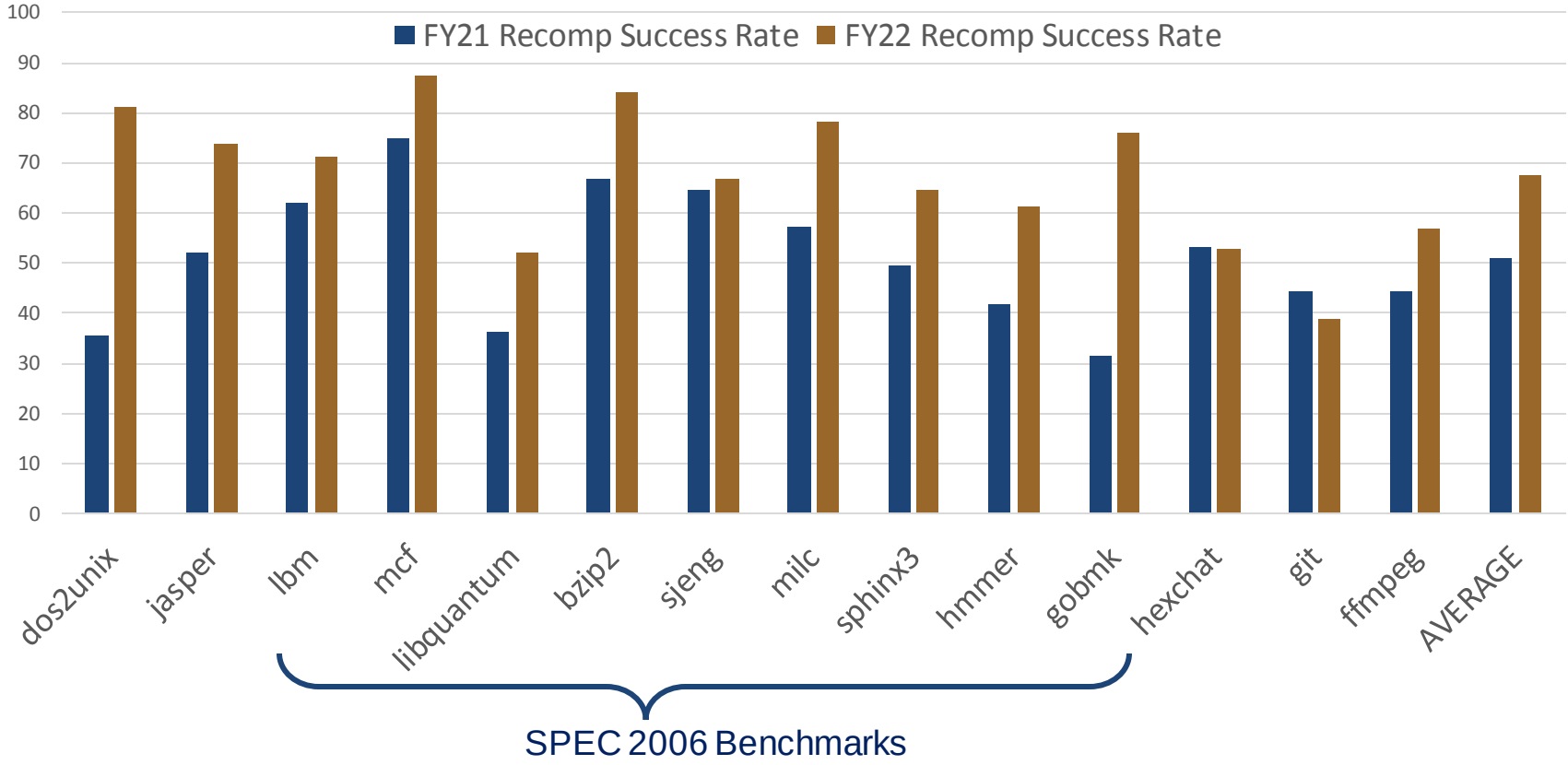
Code Recompilation

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

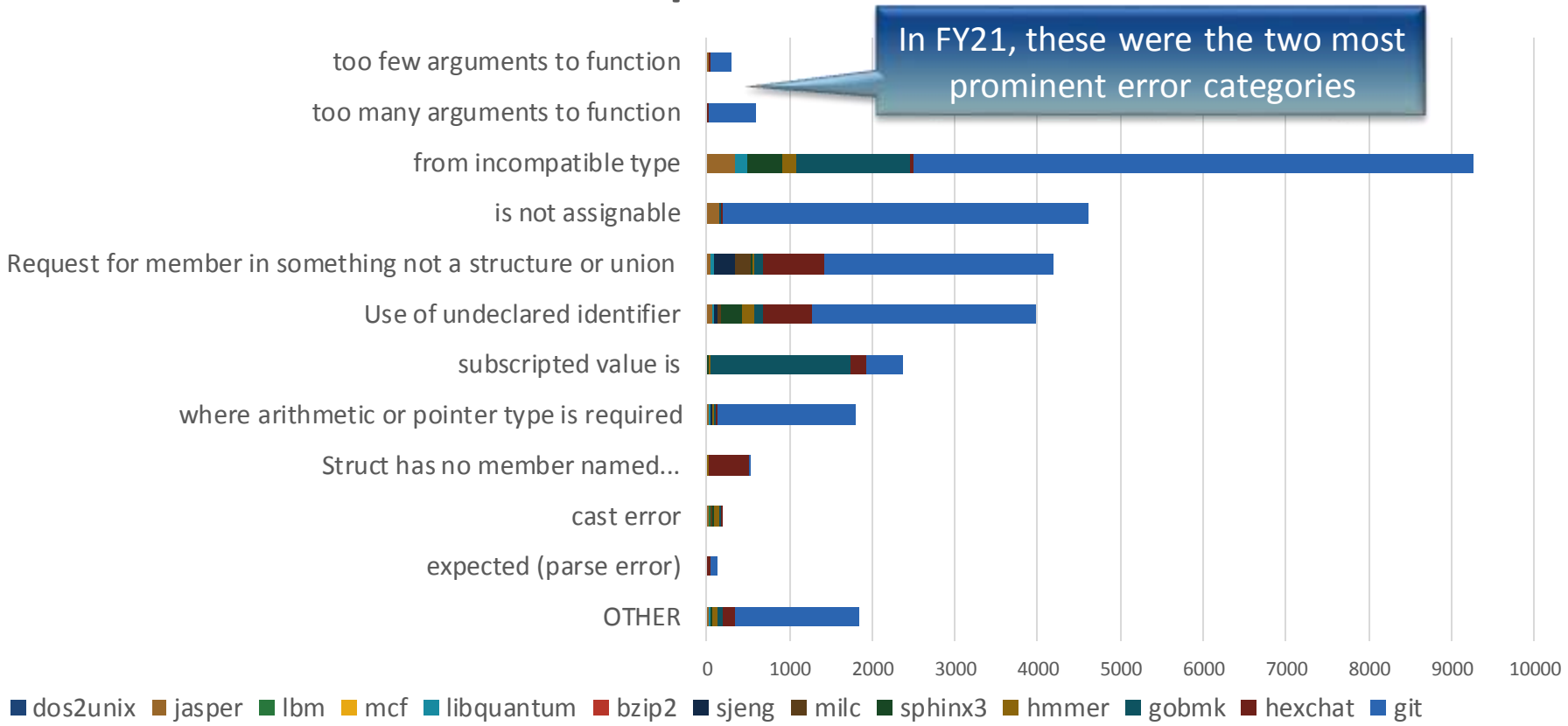
SPEC 2006 Benchmarks

Project	Source Functions	FY21 Recomp Success Rate	FY22 Recomp Success Rate
dos2unix	48	35%	81%
jasper	725	52%	74%
lbm	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%
hmmer	657	42%	61%
gobmk	2,693	32%	76%
hexchat	2,076	53%	53%
git	6,832	44%	39%
ffmpeg	23,053	44%	57%
Average		**51%**	**68%**

Recompilation Improvement over Last Year



FY22 Recompilation Error Partition



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  */
```

FY22: The same problem occurs with the POSIX **sigaction(2)** and **sysinfo(2)** functions/structs.

Other FY22 Postprocessor Improvements

- Turn on Ghidra's `Decompiler Parameter ID` feature
  - This fixed most of the `too few/many arguments` errors

- Force correct declaration of main():
  - `int main(int, char**, char**);`

- Ghidra produces C function names that start with digits (not valid in C)
  - Our fix: Prepend function name with `FN_`

- Remove duplicate enumeration constants