Carnegie Mellon University
Software Engineering Institute

Semantic Fidelity of Decompilers

Will Klieber
David Svoboda

Overview

- Goal: Determine which functions in a binary are decompiled faithfully w.r.t. semantics.
- 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.
- 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
  - Formal verification with SeaHorn

Incorrect types don't always prevent semantic equivalence

Original Code

```c
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;
    }
}
```

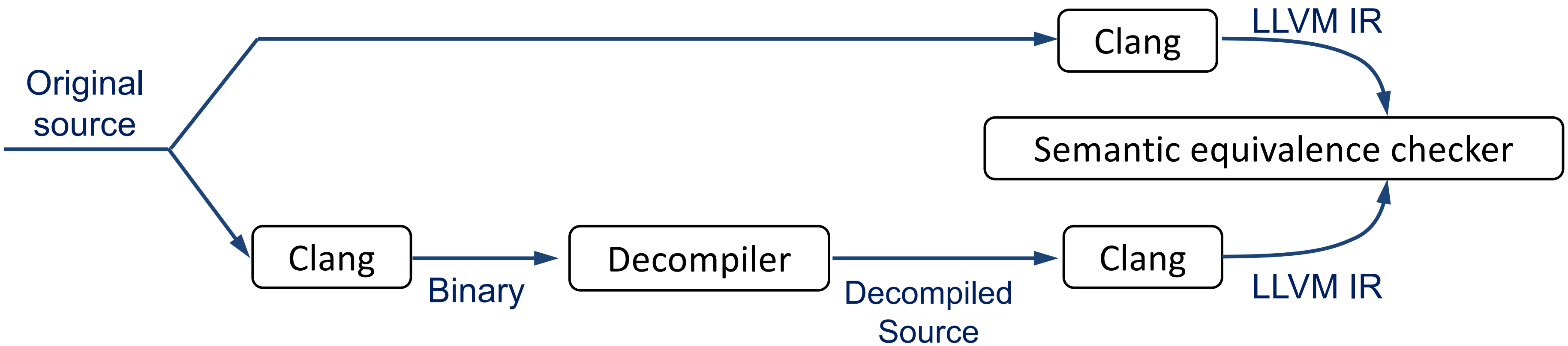
Decompiled Code

```c
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 by Ghidra.

Semantic equivalence pipeline



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

```
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);
}
```

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
```

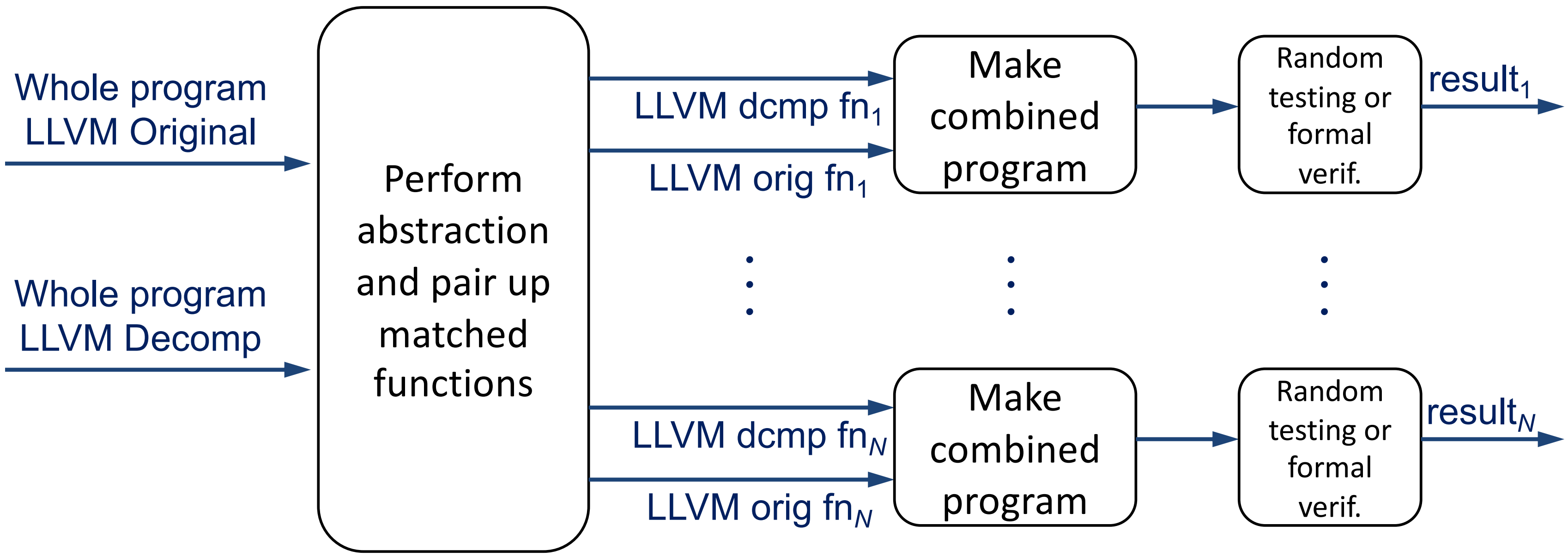
Bounded semantic equivalence checking with logging

- 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 ==> can give false counterexamples to equivalence.

Formal verification and randomized testing

- We are planning to use SeaHorn to formal verification of equivalence, but we don't have it fully working yet.
- 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.
  - Arguments to functions are also chosen randomly.

Details of semantic equivalence checker

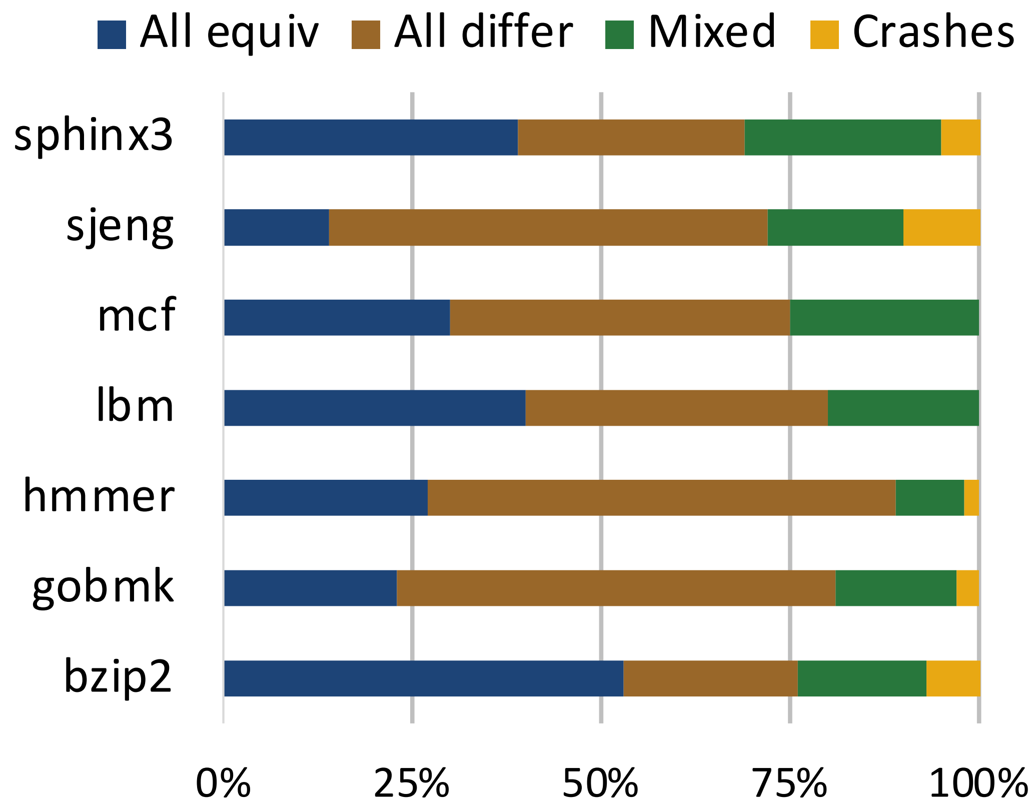


Results of semantic equivalence testing

- Tested 2650 functions from SPEC2006 that decompiled to syntactically valid code.
  - This excludes functions that were non-testable:
    - Multiple functions with the same name.
    - Issue with functions that return a large struct, compiled with "-g".
- 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.
  - On 5% functions, our tool crashed.
    - Bug in loop bounding
    - Bug in handling calls to functions such as `abort` that don't return

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%
hmmer	27%	62%	9%	2%
gobmk	23%	58%	16%	3%
bzip2	53%	23%	17%	9%



Some causes of non-equivalence

- Wrong type of global variable.
- Wrong number of arguments.
- Missing or extraneous return value.

Example of non-equivalence: bzip2: setExit

- Global variable `exit_value` is defined as a 32-bit integer type in the original source.
- Ghidra didn't define this global variable at all. Our postprocessing script added a definition of type `undefined` (an 8-bit integer type).
- The mismatch in bit-width causes non-equivalence when the value doesn't fit in 8 bits.

Original	Decompiled

`static void setExit ( Int32 v )`
`{`
`    if (v > exitValue) exitValue = v;`
`}`
	`void setExit(int param_1)`
	`{`
	`  if (exitValue < param_1) {`
	`    exitValue = param_1;`
	`  }`
	`  return;`
	`}`

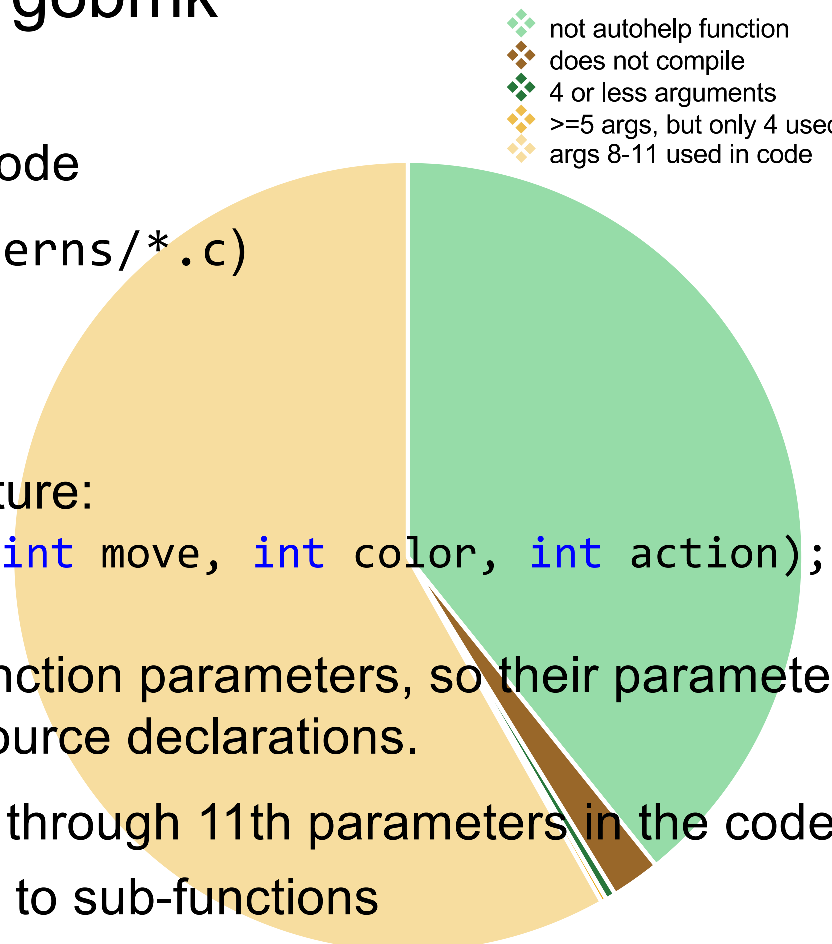
Example of non-equivalence: bzip2: spec_rewind

- Global variable `spec_fd` is defined as an **array of structs** in the original source.
- Ghidra didn't define this global variable at all. Our postprocessing script added a definition of type `undefined` (an 8-bit integer type).
- In the decompiled code, there is a memory read to get the value of spec_fd, but in the original source code, there is no corresponding memory read, since the address of the global array spec_fd is known at compile-time.

Original	Decompiled

`int spec_rewind(int fd) {`
`    spec_fd[fd].pos = 0;`
`    return 0;`
`}`
	`undefined8 spec_rewind(int param_1)`
	`{`
	`  *(undefined4 *)(spec_fd + (long)param_1 * 0x18 + 8) = 0;`
	`  return 0;`
	`}`

Semantic-Equivalence failures in gobmk



- 2693 unique functions in gobmk source code
- 1637 `autohelper` functions (in `src/patterns/*.c`)
- 1583 `autohelper` functions recompile,
  - but all fail semantic equivalence. Why?
- All `autohelper` functions have this signature:

```
static int autohelper…(int trans, int move, int color, int action);
```

- But 1572 of these files have 5 or more function parameters, so their parameter declarations do not match their original source declarations.
- And 1566 of these definitely use their 8th through 11th parameters in the code
  - E.g. not just by passing parameter lists to sub-functions

Platform Information

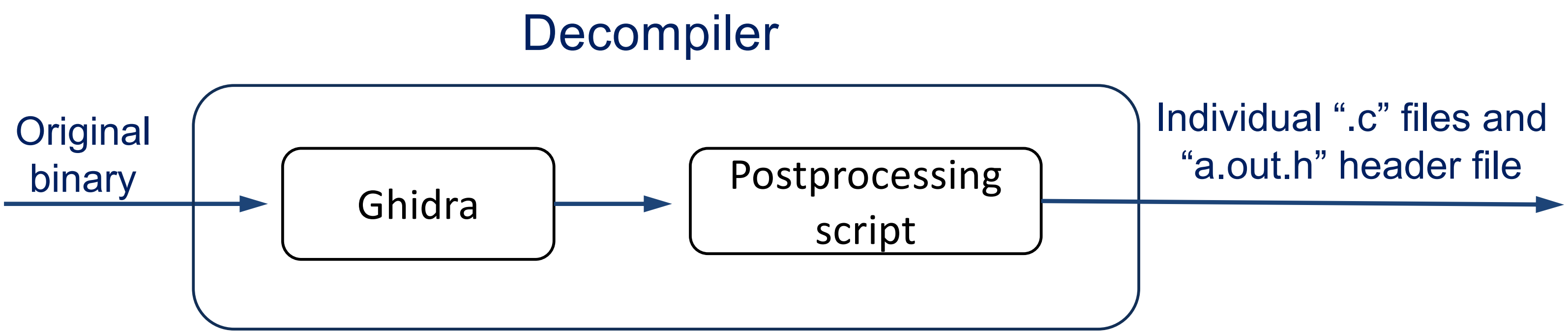
- 64-bit Ubuntu 18.04
- Ghidra ~~9.1.2~~ 10.1.4
- Java (openjdk 11.0.10)
- Clang 6.0 and 8.0

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

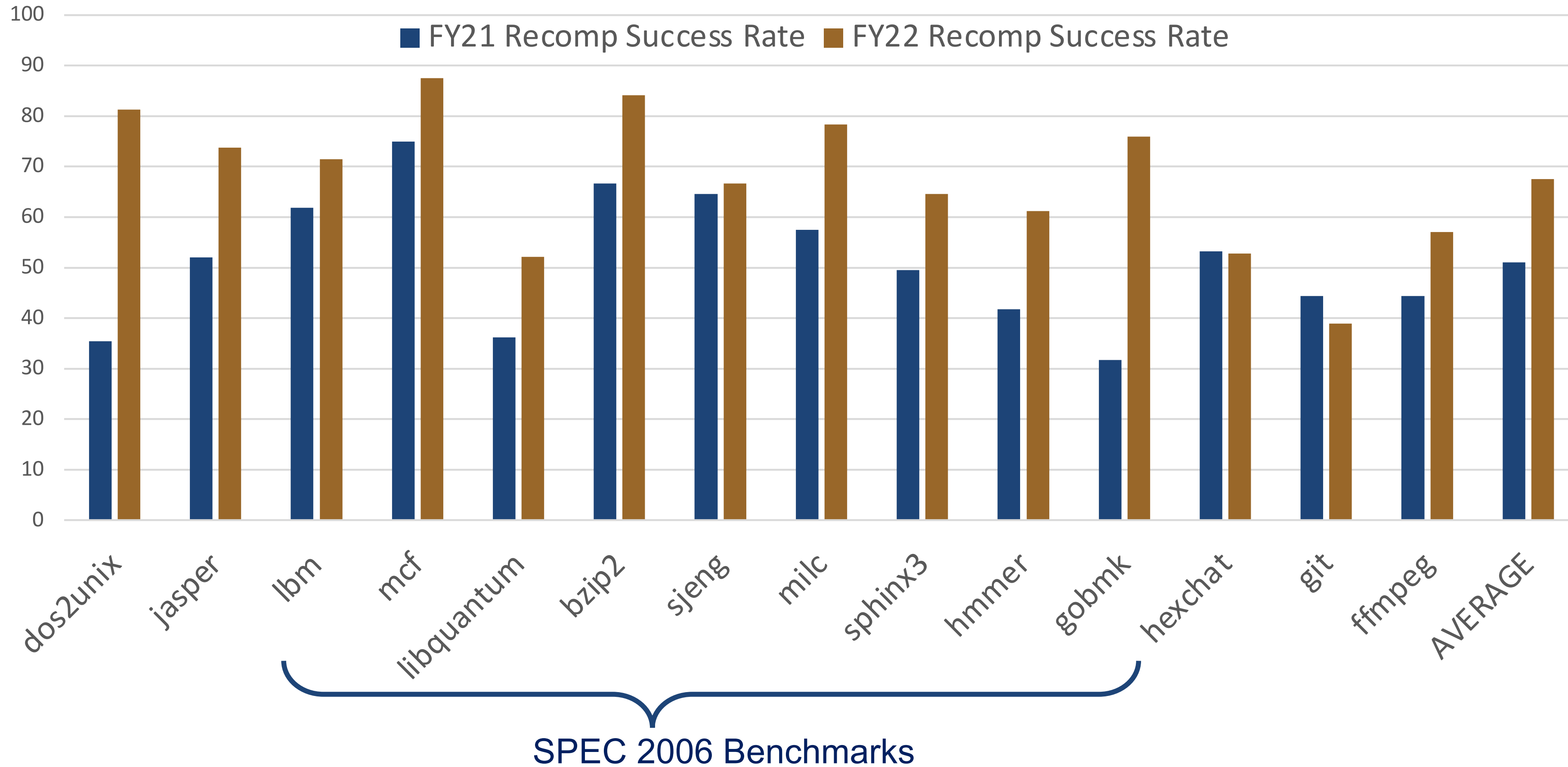
Code Recompilation

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

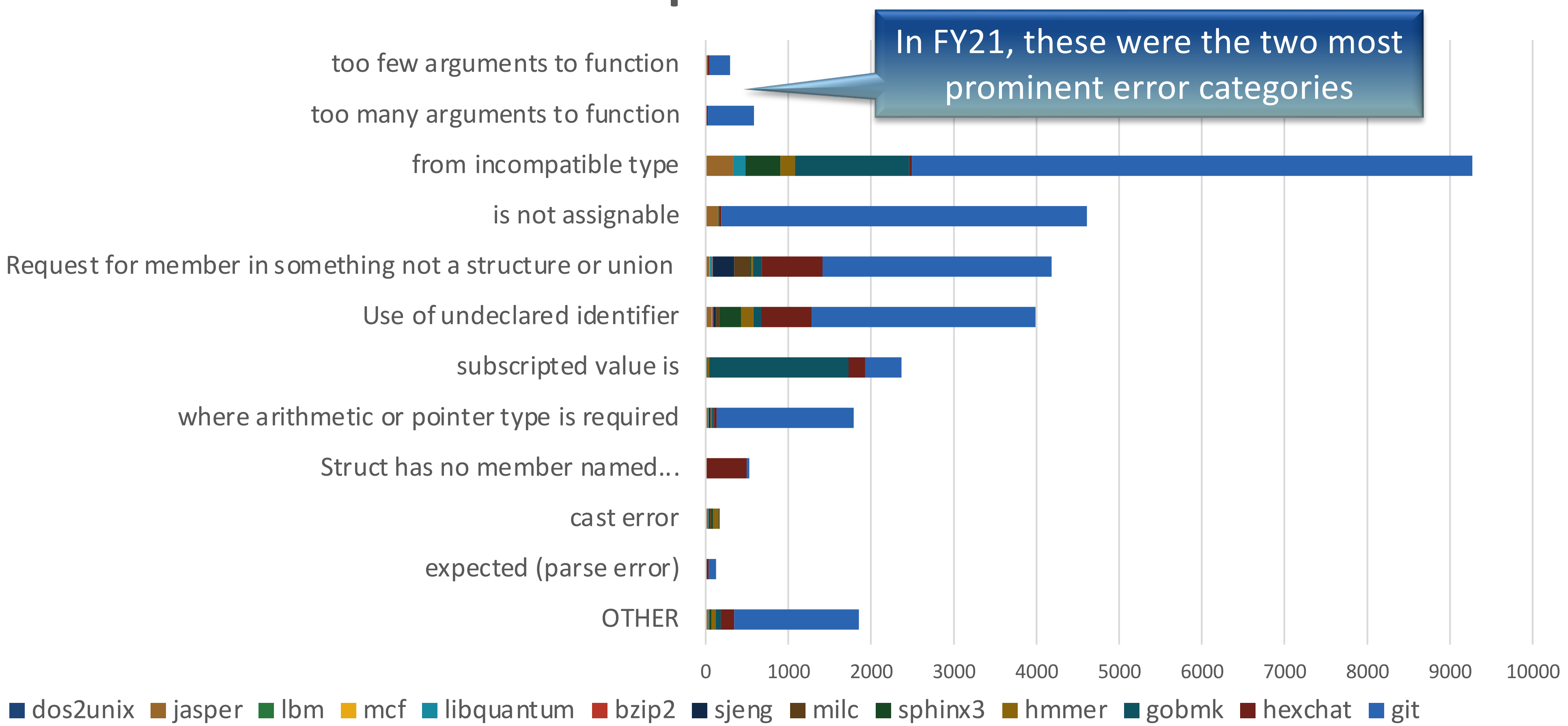
SPEC 2006 Benchmarks (lbm through gobmk)

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