Carnegie Mellon University Software Engineering Institute

Research Review 2021

Combined Analysis for Source Code and Binary Code for Software Assurance

November 2021

William Klieber

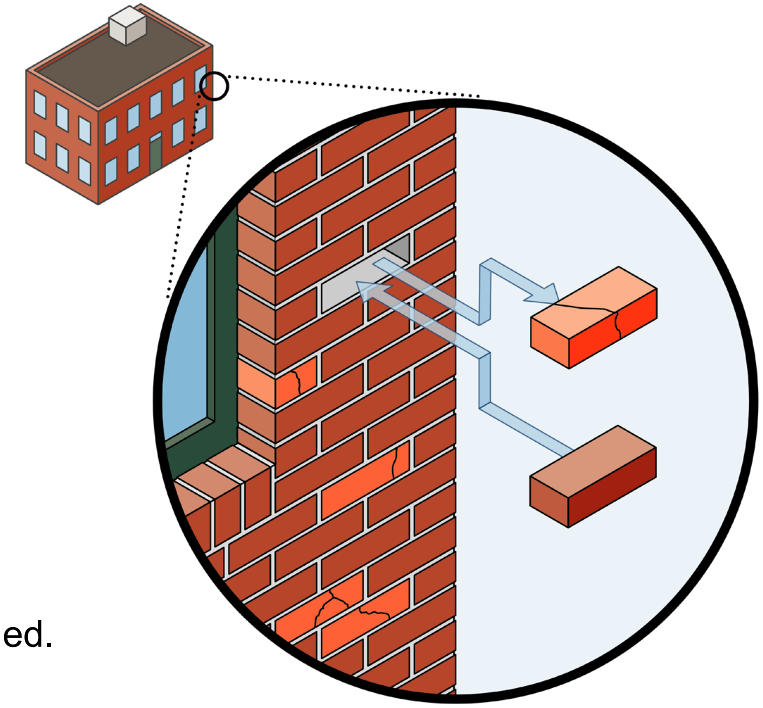
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

DM21-0866

Overview

Carnegie Mellon University Software Engineering Institute

- Goal: Increase assurance of binary components.
  - Decompile and perform static analysis.
  - Perform localized repairs.
  - Increase trustworthiness of software fielded by DoD.
- Adapt an existing open-source decompiler (Ghidra):
  - originally developed for manual reverse engineering
  - not designed to produce recompilable code
  - gap: semantic inaccuracies and syntactic errors
- Key technical steps:
  - Determine which functions have been correctly decompiled.
  - Run static analysis and localized repair.
  - Recombine (e.g., using DDisasm).



Overview (continued)

- A perfect decompilation of the entire binary isn’t necessary.
- Main contributions of our work
  - Develop semantic-equivalence checker.
  - Improve decompiler.
    - Submit to the mainline branch of Ghidra.
- This line of work is continuing this year (FY22).

DoD Impact

- Enable the DoD to find and fix potential vulnerabilities in binary code.

- Collaborators and interested transition partners at the DoD
  - have binaries for which software assurance is desired
  - evaluate and improve our tool
  - use the tool in practice when it is ready

Example of Original and Decompiled Code

Carnegie Mellon University Software Engineering Institute

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;
    }
}
```

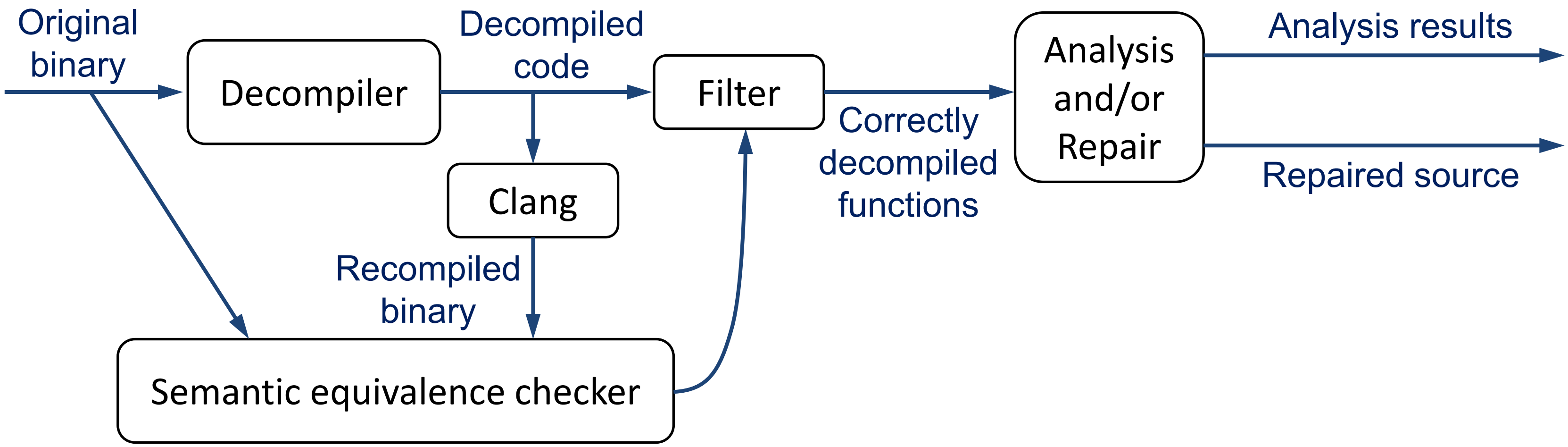
Decompiled Code

```
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;
  }
}
```

State of the Art – Recompilation of Decompiled Code

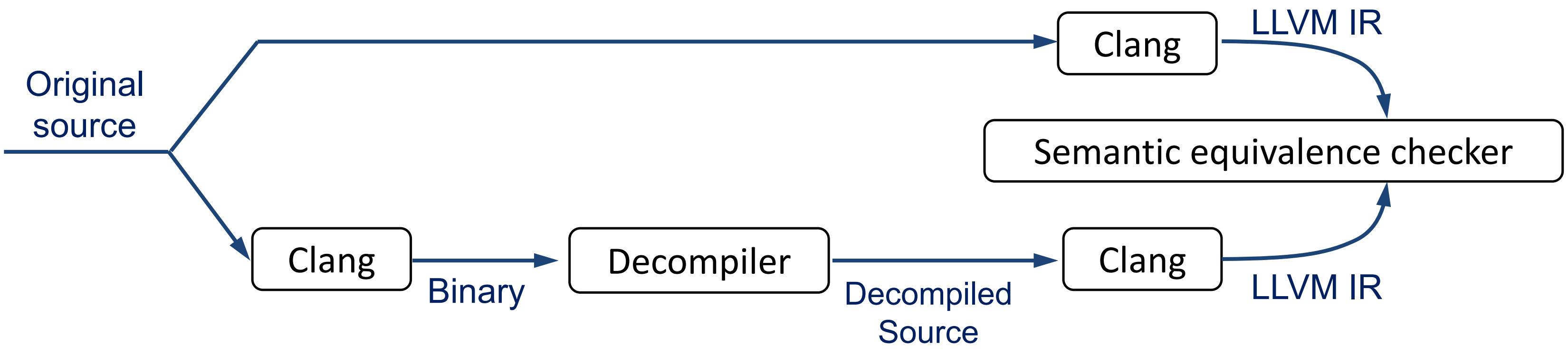
- Paper: “How far we have come: testing decompilation correctness of C decompilers.” *ACM Int’l Symposium on Software Testing & Analysis (ISSTA),* July 2020.
  - tested **synthetic** code **without input or nondeterminism**
  - Ghidra: out of 2504 test cases (averaging around 250 LoC), 93% were correctly decompiled
  - only **unoptimized** code
  - no structs, unions, arrays, or pointers

Ideal Pipeline (for Use On In-the-Wild Binaries)



FY21 Pipeline (for Measurement and Evaluation)

Carnegie Mellon University Software Engineering Institute



Proving Semantic Equivalence

- We use SeaHorn as the backend for our semantic equivalence checker.
  - SeaHorn in turn uses the Z3 SMT solver.
- Ask SeaHorn: Does the decompiled function have the *same effect* on the memory as the original function?
  - Conceptually, we consider the entire memory space.
  - The *representation* of memory is rather small.
  - There is one symbolic memory address for each memory access.
  - The SMT solver must consider aliasing between different symbolic addresses.
- This is work in progress and we expect to have results early in FY22.

Carnegie Mellon University Software Engineering Institute

Code Recompilation

This 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%**

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**

Increasing Accuracy of Function Arguments

- Consider a chain of function calls: *fn1* calls *fn2*, which calls *fn3*.
- Calling convention: arguments are passed via CPU registers.
- Note that *fn2* may forward some of its arguments to *fn3* **implicitly**.
- To determine the number of arguments, do a whole-program analysis:
  - Start by analyzing leaf functions and work upwards, asking Ghidra to redo analysis given new information about callees.
  - For recursive functions, use a fixed-point algorithm.

Conclusion

- We are adapting Ghidra.
- About half of decompiled functions successfully recompile (but semantic equivalence hasn't been assessed yet).
- We don't need a perfect decompilation of the entire binary.
- This line of work is continuing this year (FY22).
- If you are interested in collaborating or transitioning into practice, please get in touch with us.

Contact Information

Carnegie Mellon University Software Engineering Institute

Presenter / Point of Contact

Will Klieber

Software Security Researcher

Telephone: +1 412.268.9207

Email: info@sei.cmu.edu

Team

Will Klieber

David Svoboda

Ruben Martins (CMU SCS)