Combined Analysis for Source Code and Binary Code for Software Assurance

November 2021

William Klieber
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.


DM21-0866
Overview

• 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

### 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;
    }
```
State of the Art – Recompilation of Decompiled Code

- Paper: “How far we have come: testing decompilation correctness of C decompilers.”

  - 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)

Original binary → Decompiler → Decompiled code → Filter → Analysis and/or Repair → Analysis results

Decompiler

Clang

Semantic equivalence checker

Recompiled binary

Correctly decompiled functions

Recompiled binary
FY21 Pipeline (for Measurement and Evaluation)

Original source

Clang

Binary

Decompiler

Decompiled Source

Clang

LLVM IR

Semantic equivalence checker

Clang

LLVM IR

Clang
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.
# Code Recompilation

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

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

**SPEC 2006 Benchmarks**
## Types of Syntactic Errors

<table>
<thead>
<tr>
<th>Count</th>
<th>Error Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>609</td>
<td>Request for member in something not a structure or union</td>
</tr>
<tr>
<td>706</td>
<td>Invalid operands to binary operator</td>
</tr>
<tr>
<td>910</td>
<td>Other</td>
</tr>
<tr>
<td>2,972</td>
<td>Use of undeclared identifier</td>
</tr>
<tr>
<td>1,224</td>
<td>Void value not ignored as it ought to be</td>
</tr>
<tr>
<td>1,153</td>
<td>Too many arguments to function</td>
</tr>
<tr>
<td>3,434</td>
<td>Too few arguments to function</td>
</tr>
<tr>
<td>11,008</td>
<td>Total</td>
</tr>
</tbody>
</table>
Increasing Accuracy of Function Arguments

• Consider a chain of function calls: \textit{fn1} calls \textit{fn2}, which calls \textit{fn3}.
• Calling convention: arguments are passed via CPU registers.
• Note that \textit{fn2} may forward some of its arguments to \textit{fn3} \textit{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

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)