Combined analysis for source code and binary code for software assurance

Quarterly Review

Aug 9, 2021

William Klieber

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

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.

DM21-0704

FY21 LENS	Combined analysis for source code and binary code	R&E Category Cyber

Problem: Many DoD entities need software assurance for both source code and binary, as well as mixed systems (e.g., source code plus binary libraries).

Solution: A technique of decompiling binary code for the purpose of analysis and localized repair.

Approach: We will adapt an existing decompiler (e.g., Ghidra) to produce code suitable for static analysis/repair, and we'll evaluate it with real-world (optimized) binary files. A perfect decompilation of the entire binary isn't required to get significant benefit, as long as enough relevant functions are correctly decompiled.

Outcome & Anticipated DoD Impact:

- This project will demonstrate the feasibility of decompiling binary libraries for the purpose of (1) joint static analysis with source code and (2) localized repairs to functions of the library.
- This LENS will lay the groundwork for the follow-on MTP for further developing this technology (so that it can be transitioned to DoD):
  - Enable DoD to increase software assurance for projects that include binary components.
  - Develop framework for making localized repairs (either manual or automated) to functions of a binary library or executable.

Who is (or should be) interested:

This work should be of interest to DoD organizations involved in software assurance for systems that include binary-only software components.

Programmatic

Major Milestones		FY21			
		Q1	Q2	Q3	Q4
1	Automate extraction of decompiled code from Ghidra. Measure the baseline extraction success percentage	X			
2	Address "low-hanging fruit" to improve decompilation; goal: at least 75% of functions can be extracted (as valid code)		X		
3	Implement semantic equivalence checker; goal: checker can determine equivalence for at least 66% of extracted functions			O	
4	Run static analysis on full source, decompiled, and black-box; goal: decompiled is closer to full-source than to black-box.				O
5	Write paper and submit to conference				O

LENS Project Review Status

Project Title	**Combined analysis for source code and binary code for software assurance**
Principal Investigator	William Klieber
Awarded	FY start 2021
Review Date	Aug 9, 2021
TRL at start	3
TRL currently	3—4
Project end date	Sep 30, 2021
Intended project impact	Demonstrate feasibility of decompiling binaries for software assurance. Develop knowledge necessary for successful execution of follow-on MTP project: • Develop and experimentally validate techniques for determining semantic equivalence of decompiled code The MTP project aims to increase software assurance by enabling DoD to find and fix vulnerabilities in binaries that currently are cost-prohibitive to investigate or repair.
Transition partner(s) / status	We have talked with the planned DoD collaborators and plan to share our tool in Q4 and to collaborate more extensively during the follow-on MTP.

Milestone Update

		FY21			
		Q1	Q2	Q3	Q4
1	Automate extraction of decompiled code from Ghidra. Measure the baseline extraction success percentage	X			
2	Address "low-hanging fruit" to improve decompilation; goal: at least 75% of functions can be extracted (as valid code)		X		
3	Implement semantic equivalence checker; goal: checker can determine equivalence for at least 66% of extracted functions			O	
4	Run static analysis on original source code and decompiled code; goal: Similarity of static analysis results (original source vs decompiled) is at least 50%.				O
5	Write paper and submit to conference				O

Task is complete but the goal of 75% wasn't met; the average extraction rate is currently 54% (range: 32%--80%).

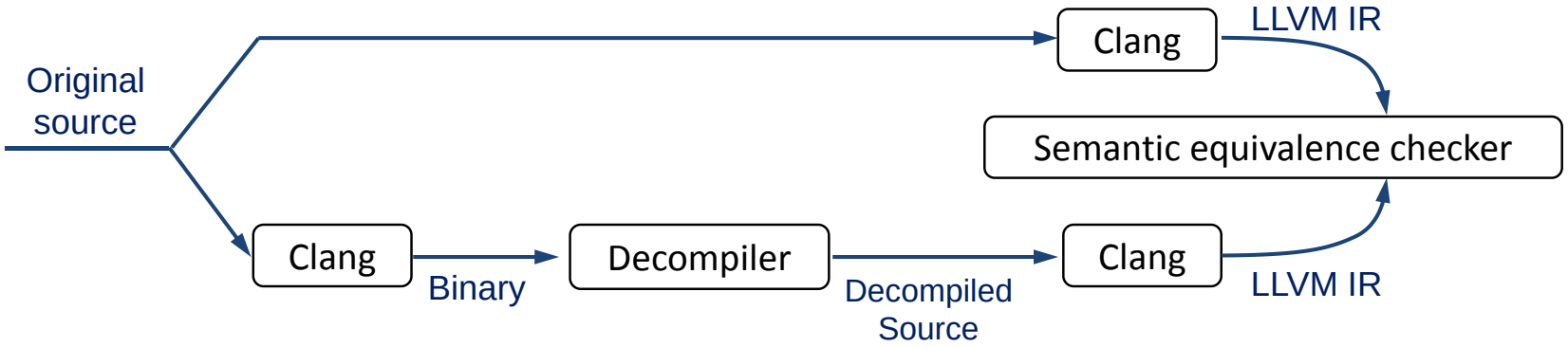
Task is taking more effort than originally anticipated. New ETA: Mid-September.

Tech Transition / Impact

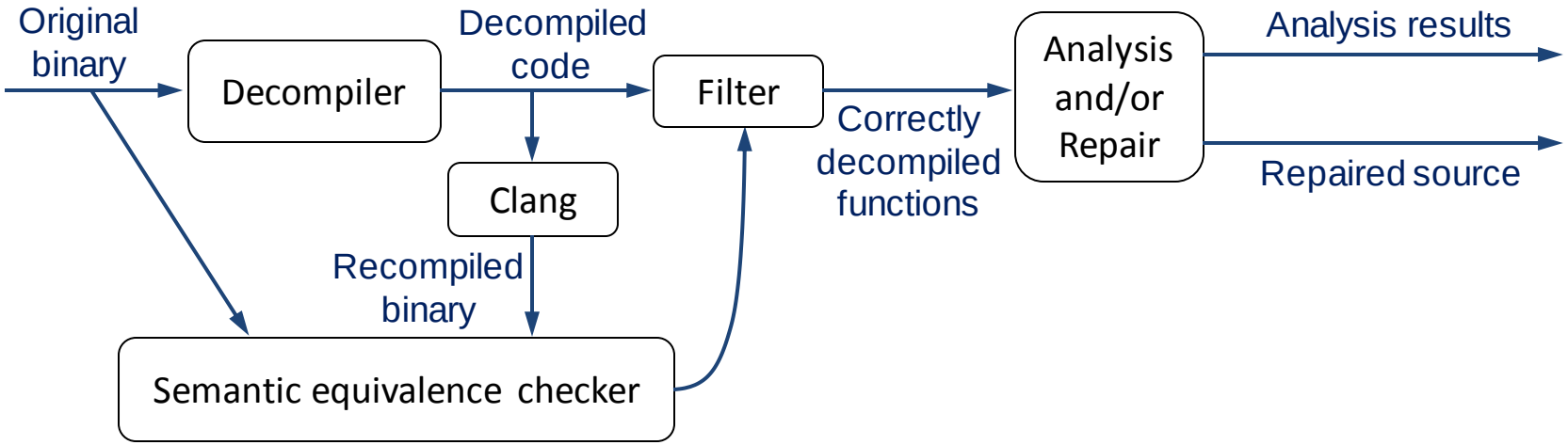
- We presented at MTEM this July.
  - (Note: in keeping with the 6.2 / FR designation, we didn't analyze any malware. However, the techniques are definitely applicable for malware too, and our results indicate which decompilation tasks are hard for decompilers even in the "easier" case of non-malicious code.)
- In Q4, we will share an update and our tool with our DoD partners and ask if they have any additional feedback
- Aiming to present at SwA CoP in FY22.
- In the follow-on MTP:
  - In Q2-Q3 of the MTP, we will share an initial version of the tool with our DoD partners for preliminary evaluation and feedback.
  - In Q4 of the MTP, we will work with our DoD partners to evaluate the final version of the tool on representative DoD binaries.

Brief Review

LENS pipeline (for measurement and evaluation)



MTP pipeline (for use by DoD on in-the-wild binaries)



Example of problematic decompiled code

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;
  }
}
```

Syntactic Validity of Decompiled Code in Open-Source Projects

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
toy	10	8	80%
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			**54%**

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**

Semantic Equivalence for Loop-Free Leaf Functions

- We are using the SeaHorn verification framework as the backend for our semantic equivalence checker. SeaHorn in turn uses the Z3 SMT solver.
- 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.

Update on Progress

Wins, Challenges, and Risks

Wins

- Building on Ghidra, we have been successfully decompiled several real-world codebases such that a significant percentage of functions can be recompiled.
- Using SeaHorn, we have successfully checked loop-free code (even with complex pointer aliasing) for semantic equivalence.

Challenges

- Semantic equivalence checking for non-leaf functions with loops is harder than anticipated.
- Ghidra performs worse than expected on identifying the number of function arguments.
- Ghidra performs worse than expected at producing recompilable code in the presence of `structs`.

Risks

- Schedule is delayed due to difficulty of semantic equivalence checking implementation.
- We might not be able to get Ghidra to extract (as valid C code) the desired percentage of functions.
- Static analysis tools might do poorly on the kind of code that Ghidra produces.

Cutpoints and Variable Mappings

- We are following the approach taken by Rahul Sharma et al. in their paper "Data-driven equivalence checking" (OOPSLA 2013)
- We 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 one cutpoint to the next.
    - At each cutpoint, the value of each variable 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) (the function exit point(s)), the return value is the same in the original and decompiled functions, and the sequence of externally visible actions (memory writes, unmodeled function calls) is the same.

Example: Insertion Sort: Original and Decompiled

```
void insertion_sort_orig(unsigned int* A,
                         size_t length) {
  for (size_t j = 1; j < length; ++j) {
    unsigned int key = A[j];
    ssize_t i = j - 1;
    while (i >= 0 && A[i] > key) {
      A[i + 1] = A[i];
      --i;
    }
    A[i + 1] = key;
  }
}
```

```
void insertion_sort_decomp(uint *A, size_t length) {
  uint uVar1, key; size_t sVar2, j; ssize_t i;
  j = 1;
  while (j < length) {
    uVar1 = A[j];
    sVar2 = j;
    while ((i = sVar2 - 1, /* comma operator */
            -1 < i && (uVar1 < A[i]))) {
      A[sVar2] = A[i];
      sVar2 = i;
    }
    A[sVar2] = uVar1;
    j = j + 1;
  }
}
```

Example: Insertion Sort: Mappings

(shown in C source code rather than LLVM IR for ease of understanding)

```
void insertion_sort_orig(unsigned int* A,
                         size_t length) {
  for (size_t j = 1; j < length; ++j) {
    unsigned int key = A[j];
    ssize_t i = j - 1;
    while (i >= 0 && A[i] > key) {
      A[i + 1] = A[i];
      --i;
    }
    A[i + 1] = key;
  }
}
```

```
void insertion_sort_decomp(uint *A, size_t length) {
  uint uVar1, key; size_t sVar2, j; ssize_t i;
  j = 1;
  while (j < length) {
    uVar1 = A[j];          // key_orig = uVar1
    sVar2 = j;             // i_orig = sVar2 - 1
    while ((i = sVar2 - 1,
            -1 < i && (uVar1 < A[i]))) {
      A[sVar2] = A[i];     // i_orig = sVar2 - 1
      sVar2 = i;           // i_orig = sVar2
    }
    A[sVar2] = uVar1;      // i_orig = sVar2 - 1
    j = j + 1;
  }
}
```

Algorithm

1. Generate cutpoints
   - More difficult than in the Sharma paper, due to nested loops and multiple loops.
   - Currently done manually
2. Gather zero or more pairs of traces for the original and decompiled code.
3. Using basic linear algebra to find a linear combination that maps between original variables and decompiled variables for the set of gathered traces.
   - If system is overdetermined, fail.
4. Use SeaHorn to check whether the mapping (from the above step) holds true in general. If it doesn't, add the counterexample trace to our set of gathered traces and go back to step 3.

Example: cmp_name (from sphinx3)

Original:

```
static int32 cmp_name(const void *a, const void *b) {
  return (strcmp_nocase(tmp_defn[*((int32 *)a)].name,
                        tmp_defn[*((int32 *)b)].name));
}
```

Decompiled:

```
void cmp_name(int *param_1, int *param_2) {
  strcmp_nocase(*(undefined8 *)(tmp_defn + (long)(*param_1) * 0x20),
                *(undefined8 *)(tmp_defn + (long)(*param_2) * 0x20));
  return;
}
```

Example: replace_weaker_arc (from mcf)

Original:

```
...
new[0].tail     = tail;
new[0].head     = head;
new[0].org_cost = cost;
new[0].cost     = cost;
new[0].flow     = (flow_t)red_cost;
...
```

Decompiled:

```
...
param_2[1] = param_3;
param_2[2] = param_4;
param_2[7] = param_5;
*param_2   = param_5;
param_2[6] = param_6;
...
```

Example original: vithist_frame_windup (from sphinx3)

```
void vithist_frame_windup (vithist_t *vh, int32 frm,
                           FILE *fp, kbcore_t *kbc)
{
    assert (vh->n_frm == frm);
    vh->n_frm++;
    vh->frame_start[vh->n_frm] = vh->n_entry;
    if (fp) {vithist_dump (vh, frm, kbc, fp);}
    vithist_lmstate_reset (vh);
    vh->bestscore[vh->n_frm] = MAX_NEG_INT32;
    vh->bestvh[vh->n_frm] = -1;
}
```

Example decompiled: vithist_frame_windup (from sphinx3)

```
int vithist_frame_windup (long param1, undefined8 param2, long param3, undefined8 param4) {
  int iVar1 = *(int *)(param1 + 0x14);
  if (iVar1 == (int)param2) {
    *(int *)(param1 + 0x14) = iVar1 + 1;
    *(undefined4 *)(*(long *)(param1 + 8) + 4 + (long)iVar1 * 4) = *(undefined4 *)(param1 + 0x10);
    if (param3 != 0) {
      vithist_dump(param1,param2,param4,param3);
    }
    vithist_lmstate_reset(param1);
    *(undefined4 *)(*(long *)(param1 + 0x20) + (long)*(int *)(param1 + 0x14) * 4) = 0x80000000;
    *(undefined4 *)(*(long *)(param1 + 0x28) + (long)*(int *)(param1 + 0x14) * 4) = 0xffffffff;
    return 1;
  }
  __assert_fail("vh->n_frm == frm", "vithist.c", 0x1e3,
                "void vithist_frame_windup(vithist_t *, int32, FILE *, kbcore_t *)");
}
```

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.

Solution: Delete the extraneous typedefs.

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 start with leaf functions (i.e., functions that don't call any other functions in our code) and work upwards, asking Ghidra to redo analysis given new information about callees.
  - We'll need something slightly more complex for recursive functions.
- This technique should also address incorrect return types.

Remaining technical work

- Finish implementing semantic equivalence checker.
- Run semantic-equivalence experiments.
- Write paper.
- Implement static-analysis testing framework.
- Run static-analysis experiments.